UNIVERSITY OF CALIFORNIA,
IRVINE

Extending the Reach of AND/OR Search for Optimization over Graphical Models

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Lars P. Otten

Dissertation Committee:
Professor Rina Dechter, Chair
Professor Alexander Ihler
Professor Padhraic Smyth

2013

DEDICATION

To my parents, for their unwavering support.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

This thesis would not have been possible without the help and contributions, directly and indirectly, of many others. Most significantly, I am indebted to my advisor, Prof. Rina Dechter, for her guidance and support throughout my time in graduate school. Rina let me explore and pursue the questions I found of interest, but also provided direction when I needed it. Her high standards have made me a better researcher and helped me grow as a person.

I am also grateful to my committee members Profs. Alexander Ihler and Padhraic Smyth, for their comments on earlier versions of this thesis and fruitful discussion and suggestions in general.

I would like to thank the other members of our research group over the years, Kalev Kask, Bozhena Bidyuk, Robert Mateescu, Radu Marinescu, Vibhav Gogate, Tuan Nguyen, Natalia Flerova, Andrew Gelfand, and William Lam. They have provided a cooperative and stimulating environment to work in, with many insightful discussions but also a good amount of humor.

My graduate school friends, many of them fellow doctoral students, should also not go unmentioned. Be it through intramural sports (with our "Bayes All-Stars" team), hiking and backpacking trips, board game nights, beer tastings, Summer BBQs, Halloween parties, and more, they deserve thanks for many unforgettable experiences – and welcome relief from hard work – over the years.

I want to thank Prof. Dan Geiger, Mark Silberstein, and their research group at the Technion University in Haifa, Israel, for hosting me twice as a visiting researcher, for their valuable comments on various early paper drafts, and for inviting me to collaborate on the Superlink-Online SNP system.

Last but not least, I couldn't have done this without the incredible support of my family, who have always believed in me. Most of all, I am indebted to my wife Teiko, for her understanding and encouragement, in particular throughout the strenuous last months of the thesis writing process.

I am grateful for the assistance I have received for my graduate studies from NSF awards IIS-0713118, IIS-1065618, NIH grant 5R01HG004175-03, and a Dean's Fellowship at the Donald Bren School of Information and Computer Sciences.

CURRICULUM VITAE

Lars P. Otten

EDUCATION

Doctor of Philosophy in Computer Science **2013**
University of California, Irvine *Irvine, California*

Master of Science in Dependable Computer Systems **2006**
Chalmers University of Technology *Gothenburg, Sweden*

Intermediate Diploma in Computer Science **2004**
RWTH Aachen University *Aachen, Germany*

Intermediate Diploma in Mathematics **2004**
RWTH Aachen University *Aachen, Germany*

RESEARCH EXPERIENCE

Graduate Research Assistant **2006–2013**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2008–2009**
University of California, Irvine *Irvine, California*

SELECTED HONORS AND AWARDS

Dean's Fellowship, Information and Computer Sciences **2006–2010**
University of California, Irvine

REFEREED JOURNAL PUBLICATIONS

A System for Exact and Approximate Genetic Linkage Analysis of SNP Data in Large Pedigrees — **Jan 2013**
Bioinformatics, Vol. 29(2)

Anytime AND/OR Depth-first Search for Combinatorial Optimization — **Aug 2012**
AI Communications, Vol. 25(3)

REFEREED CONFERENCE PUBLICATIONS

A Case Study in Complexity Estimation: Towards Parallel Branch-and-Bound over Graphical Models — **Aug 2012**
28th Conference on Uncertainty in Artificial Intelligence

Join-Graph-Based Cost-Shifting Schemes — **Aug 2012**
28th Conference on Uncertainty in Artificial Intelligence

Advances in Distributed Branch and Bound — **Aug 2012**
20th European Conference on Artificial Intelligence

Anytime Depth-first Search for Combinatorial Optimization — **Jul 2012**
4th Annual Symposium on Combinatorial Search

Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models — **Aug 2011**
25th AAAI Conference in Artificial Intelligence

Finding Most Likely Haplotypes in General Pedigrees through Parallel Search with Dynamic Load Balancing — **Jan 2011**
16th Pacific Symposium on Biocomputing

Towards Parallel Search for Optimization in Graphical Models — **Jan 2010**
11th International Symposium on Artificial Intelligence and Mathematics

Maximum Likelihood Haplotying through Search on a Grid of Computers — **May 2009**
13th International Conference on Research in Computational Molecular Biology

Refined Bounds for Instance-Based Search-Complexity of Counting and Other #P Problems — **Sep 2008**
14th International Conference on Principles and Practice of Constraint Programming

On the Practical Significance of Hypertree vs. Tree Width — **Jul 2008**
18th European Conference on Artificial Intelligence

Bounding Search Space Size via (Hyper)tree Decompositions **Jul 2008**
24th Conference on Uncertainty in Artificial Intelligence

Randomization in Constraint Programming for Airline Planning **Sep 2006**
12th International Conference on Principles and Practice of Constraint Programming

SOFTWARE

DAOOPT `github.com/lotten/daoopt`
Implementation of standard, breadth-rotating and parallel AND/OR Branch-and-Bound under open-source GPL license.

Superlink-Online SNP `cbl-hap.cs.technion.ac.il/superlink-snp/`
Uses parallel AND/OR Branch-and-Bound for maximum likelihood haplotyping as part of an online system for genetic pedigree analysis.

ABSTRACT OF THE DISSERTATION

Extending the Reach of AND/OR Search
for Optimization over Graphical Models

By

Lars P. Otten

Doctor of Philosophy in Computer Science

University of California, Irvine, 2013

Professor Rina Dechter, Chair

This thesis presents substantial enhancements to the state of the art in combinatorial optimization over graphical models. Our contributions are relevant in the context of both exact and approximate reasoning over Bayesian and Markov networks, weighted constraint satisfaction problems, and other related queries. While the focus of this work is on probabilistic and constraint inference, we also draw from the areas of distributed computing and statistical learning. Relevant practical applications we consider include genetic linkage analysis, protein side-chain prediction, medical diagnosis, resource scheduling, and signal processing.

We extend AND/OR Branch-and-Bound (AOBB), a leading algorithm for optimization queries over graphical models. AOBB applies the principle of depth-first branch-and-bound to AND/OR search spaces, which exploit conditional independencies via problem decomposition and merge unifiable subproblems through caching of partial solutions. This thesis presents fundamental extensions to AOBB in three regards.

First, we significantly improve the applicability of AOBB as an approximation scheme. We analyze and demonstrate the inherent conflict between problem decomposition (through AND/OR search spaces) and the anytime behavior of AOBB and depth-first search in general. We introduce a new algorithm, Breadth-Rotating AND/OR Branch-and-Bound

(BRAOBB), which drastically improves upon AOBB with respect to its anytime performance while maintaining desirable depth-first complexity guarantees. Comprehensive analysis and experimental evaluation demonstrate the scheme's effectiveness. Furthermore, our entry based on BRAOBB placed first in all three optimization tracks of the PASCAL 2012 Probabilistic Inference Challenge.

Second, we investigate the instance-based run-time complexity of AOBB. The asymptotic worst-case bounds are both time and space exponential in the problem's induced width, but often prove to be very loose due to the algorithm's powerful pruning, as we show empirically. We identify a range of (sub)problem features and develop learning schemes to estimate run-time complexity based on statistical regression analysis. We conduct extensive experimental evaluation within and across various problem classes and demonstrate convincing predictive performance.

Third, we describe a parallel AND/OR Branch-and-Bound scheme that pushes the boundaries of feasibility for exact reasoning by orders of magnitude. We adapt the paradigm of parallel tree search to AND/OR search spaces; our implementation distributes conditioned subproblems on a grid of independent computers. In this context, we show how the pruning power of AOBB can cause large variance in subproblem complexity, which makes load balancing extremely elusive and impairs parallel performance. We thus propose load balancing based on the run-time estimation scheme presented earlier in the thesis, learning a complexity model offline from previously solved subproblems. Through experimental results using hundreds of computers on problem instances from a variety of classes we show convincing parallel performance with several orders of magnitude speedup over sequential AOBB, but we also highlight and analyze some inherent limitations.

Our implementations of AOBB, BRAOBB and parallel AOBB are available online under an open-source license.

Chapter 1

Introduction

Optimization problems over graphical models come in various forms with many applications of practical significance, ranging from computational biology and genetics to scheduling tasks and coding networks. Conceptually one can distinguish two principal classes of problems: probabilistic inference, where optimization queries typically refer to maximizing a product over conditional probabilities, and weighted constraint networks, where one wishes to minimize a sum of local cost functions. Both of these tasks are known to be NP-hard. Central to their solution process is the usage of the underlying graph structure to capture and exploit the interactions between variables.

One established and efficient class of algorithms for solving these problems exactly is depth-first Branch and Bound over AND/OR search spaces. Developed in the past decade within both the probabilistic reasoning and constraint communities, these methods are effective (a) because they use sophisticated lower bound schemes such as soft arc-consistency [75] or the mini-bucket heuristic [28, 82], (b) because they avoid redundant computation using caching schemes, and most significantly, (c) Because they take advantage of problem decomposition by exploring an AND/OR search space [87] or an equivalent representation. The efficiency of

these algorithms was established in several evaluations, including recent UAI competitions [35], and their properties when used for exact computation are well documented [62, 82, 83].

1.1 Dissertation Outline and Contributions

This thesis extends the cited previous work to build upon and improve AND/OR Branch-and-Bound in several dimensions. On the one hand, we substantially widen the applicability of AOBB for approximate reasoning by improving and, in some cases, restoring its anytime performance characteristics. On the other hand, we push the boundary of feasibility with regards to exact inference by proposing a distributed implementation of AOBB that runs on hundreds of computers. The following paragraphs elaborate.

1.1.1 Breadth-Rotating AND/OR Branch-And-Bound

Chapter 2 is concerned with AOBB in sequential execution, on a single processor, and its *anytime behavior* [119]. As a depth-first branch-and-bound scheme, AOBB should be able to generate solutions that get better and better over time, until it eventually discovers an optimal solution and finally proves its optimality. This behavior is very useful in practice since in many cases finding a feasible solution is easy but an optimal one is hard to attain. In fact, this anytime property allows a branch-and-bound algorithm to function as an approximate reasoning scheme for otherwise infeasible problems or when time is limited. AOBB as developed in the past [82], however, does generally not have this property, which serves as the motivation for this chapter as follows:

Contributions

- We demonstrate analytically and empirically how problem decomposition in AND/OR search spaces can conflict with the anytime characteristics of AOBB. In particular, when traversed in a depth-first manner, all but one decomposed subproblem will be fully solved before a single overall solution can be returned.

- We introduce a simple modification that can mitigate this issue under certain conditions. Namely, if only one of the decomposed subproblems is "hard," processing subproblems in a suitable order can restore the anytime performance to some extent, which we also demonstrate empirically.

- The main contribution of this chapter, however, is a new branch-and-bound scheme called *Breadth-Rotating AND/OR Branch-and-Bound* (BRAOBB) that tackles the issue in a more principled way. The algorithm combines depth-first and breadth-first exploration by periodically "rotating" over the different subproblems, each of which is processed depth-first as before.

- Despite the breadth-first characteristics, we show that BRAOBB retains the favorable complexity guarantees of ordinary depth-first search (in particular, the OPEN list of nodes grows linearly).

- We conduct large-scale empirical evaluation of BRAOBB spanning tens of thousands of hours of CPU time over multiple problem classes and various time limits. Results demonstrate superior anytime performance of BRAOBB, especially for cases where standard AOBB fails entirely.

- We also perform a comparison against a state-of-the-art stochastic local search solver (SLS, [60]) and outline the varying strengths of the schemes. We then show how BRAOBB can be joined with SLS to combine the benefits of both approaches.

The strength of this approach was further demonstrated in the PASCAL 2011 Inference Challenge, where an entry based on Breadth-Rotating AND/OR Branch-and-Bound placed first in all three categories of the optimization track [36].

1.1.2 Complexity Prediction of AND/OR Branch-and-Bound

Chapter 3 investigates the runtime complexity of AOBB. It is well known that its asymptotic complexity is exponential in the problem instance's induced width, as a result of the bound on the size of the context-minimal AND/OR search graph that the algorithm explores [82]. Together with other structural parameters (e.g., number of variables and maximum domain size) this is often used to judge a problem's hardness.

In the context of optimization problems, however, the aforementioned asymptotic bounds are generally very loose, due to the pruning power of the AOBB algorithm. This discrepancy is the starting point for the contributions in this chapter, where we propose a learning approach to better predict the runtime of AOBB, as outlined in the following.

Contributions

- We develop a finer-grained upper bound on the size of the search space explored by AOBB, based on a problem's structural parameters. However, we show empirically that it is still very loose in practice, which suggests that one needs to go beyond the problem's graph structure to get a better handle on problem complexity.
- We identify an extended set of 35 problem features and parameters to form the basis for complexity estimation. These features capture structural properties of the problem or subproblem at hand, but also aim to incorporate more dynamic information based on

the problem's cost function, such as upper and lower bounds and the pruning behavior on a small search space probe.

- Based on these features, we then propose to employ a statistical learning approach to more accurately predict the size of the AOBB search space size. In particular, we propose an expression that is exponential in a linear combination of the 35 features and apply a logarithmic transformation to obtain a linear regression model, for which a variety of established learning algorithms exist [59].

- Extensive experimental evaluation of the quality of the learned complexity model is conducted. Starting from thousands of sample subproblems from four different problem classes, we investigate the estimation performance at different, increasingly more general levels, starting from just subproblems within a given problem instance, to learning per problem class and across classes, to learning from one problem class applied to another other, a setup resembling transfer learning.

- The results show generally good predictive performance in all evaluated learning scenarios, except for some cases of transfer learning. Estimates and actual complexities exhibit a relatively high degree of correlation.

- Closer analysis of the experimental results indicates that the most informative features for complexity prediction are indeed dependent on the problem's cost function tables, reflecting the earlier observation that only structural parameters like the induced width are not sufficiently informative in practice.

1.1.3 Parallelizing AND/OR Branch-and-Bound

In Chapter 4 we move to parallel execution of AND/OR Branch-and-Bound, with the aim of pushing the boundaries of feasibility for exact inference. This shift is quite obvious given the

pervasiveness of multi-core CPUs in commodity computers connected over local networks or the Internet.

Within the general field of parallel and distributed computing there exists a whole range of parallelization paradigms, which can be classified along several axes. Most crucial among those is the question of how the parallel processes exchange information. Multi-core or multi-CPU systems have shared, fast main memory that can be read and written by all processes [51, 81]. On the other end of the spectrum are cluster and grid computing, with independent hosts, each with their own, private main memory, that can exchange messages in different ways and to a varying degree [43].

The contribution of this chapter lies in putting optimization over graphical models in the parallelization context. Specifically we focus on a grid computing approach, operating on a large set of autonomous, loosely connected systems. A more specific outline is given in the following.

Contributions

- We adapt and extend the concept of parallel tree search [51, 53, 74] for the graphical model context and apply it to AND/OR Branch-and-Bound, a highly advanced, state-of-the-art sequential algorithm. The result is *parallel AND/OR Branch-and-Bound*, in which search is performed centrally up to a certain point by a "master" host and the remaining conditioned subproblems are processed separately and in parallel by "workers."

- We argue that, as a distributed algorithm, effective load balancing is the key to good parallel performance. To that end we propose two variants of parallel AOBB:

  - A baseline version that generates parallel subproblems at a fixed depth. As a consequence, these subproblems generally all have identical structure and the same upper bound on their search space size (cf. analysis in Chapter 3).
  - A more dynamic version that uses the runtime estimation model proposed in Chapter 3 to generate parallel subproblems of varying structure at different depths, in an attempt to account for the pruning power of AOBB, which can have vastly divergent effects in different parts of the state space.

- We provide analysis of parallel AOBB from a distributed system standpoint, highlighting the execution environment, its communication patterns and the resulting potential for overhead.

- We furthermore analyze the parallel scheme from a graphical model reasoning perspective to demonstrate that our problem setting is far from *embarrassingly parallel*. We work out the two sources of redundancy inherent to the parallelization process, both of which have their cause in the communication limitations of the grid environment. Specifically, (1) solutions cannot be propagated as bounds across subproblems; (2) caching of unifiable subproblems, a core feature of AND/OR search spaces, is compromised in the same way, the negative effect of which we bound analytically using the graphical model structure.

- We conduct extensive empirical evaluation across instances from four problem classes, running parallel experiments on 20, 100, and 500 CPUs. Performance is assessed through a variety of metrics, from parallel runtime and speedup, to average resource utilization, to parallel overhead and redundancy. Some of the central results include:
  - Overall parallel performance is good, with substantial speedups in several cases. With 20 and 100 CPUs in particular we find speedups fairly close to the optimum, equal to the number of CPUs. We also see that variable-depth parallelization is indeed often superior to the fixed-depth variant by virtue of its ability to detect

and avoid potential bottlenecks. It falls behind, however, in some cases where the underlying complexity predictions are sufficiently inaccurate and, for instance, drastically underestimate one subproblem's complexity.

– Our analysis shows the practical effect of the structural redundancies to be far less pronounced than the earlier worst-case analysis suggested; instead of exponential in the depth of the subproblem frontier, our results show a more linear behavior, often with a small slope of less than 2. However, in some cases, from one problem class in particular, we observe a faster increase of redundancies, which, while still growing linearly, inhibits scalability of the scheme to large numbers of CPUs (since that requires more subproblems and thus a deeper cutoff).

To the best of our knowledge our contribution is the first of its kind, i.e., it constitutes the first general-purpose parallel implementation of an advanced algorithm for optimization over graphical models, running on a computational grid. Its viability has been further confirmed by its successful deployment for haplotype analysis within Superlink-Online SNP, an online system for genetic analysis of linkage data used by researchers worldwide.

1.2 Preliminaries

The remainder of this chapter introduces the necessary concepts and notation that this work builds upon. We start by formally defining the concept of a graphical model and a number of relevant properties.

1.2.1 Definition of Graphical Models

Graphical models present a powerful formalism for capturing problem structure in probabilistic and constraint reasoning, which is applicable in many practical problem domains. We first present the formal definition of a graphical model (over discrete variables) and then show a number of more concrete instantiations.

DEFINITION **1.1** (graphical model)**.** *A* graphical model $\mathcal{R}$ *is a 4-tuple* $(X, D, F, \otimes)$*, where:*

- $X = \{X_1, \ldots, X_n\}$ *is a set of discrete variables,*
- $D = \{D_1, \ldots, D_n\}$ *is the set of respective finite variable domains,*
- $F = \{f_1, \ldots, f_m\}$ *is a set of real-valued functions, each with scope*$(f_i) \subseteq X$*,*
- $\otimes$ *is a combination operator, applied over the set of functions* $\otimes_i f_i$*.*

We point out that graphical models can also be defined over continuous variables, but that is outside of the scope of this thesis. Given a graphical model we can then define a an optimization problem over it as follows.

DEFINITION **1.2** (optimization problem)**.** *A* graphical model optimization problem $\mathcal{P}$ *is a pair* $(\mathcal{R}, \Downarrow)$ *of a graphical model* $\mathcal{R} = (X, D, F, \otimes)$ *as defined in Definition 1.1 and a marginalization operator* $\Downarrow \in \{\min, \max\}$*, where the goal is to compute* $\Downarrow \otimes_i f_i$*.*

Finally, the set of function scopes of a graphical model lets us reason about its underlying graph structure.

DEFINITION **1.3** (primal graph)**.** *The* primal graph *of a graphical model* $\mathcal{R} = (X, D, F, \otimes)$ *is a graph* (V, E) *that has the variables as its nodes,* $V = X$*, and edges connecting any two variables that appear in the scope of a function, i.e.* $(X, Y) \in E \Leftrightarrow \exists f \in F \colon \{X, Y\} \subseteq$ *scope*(f)*.*

1.2.2 Applications of Graphical Models

Some common incarnations of the graphical model formalism include:

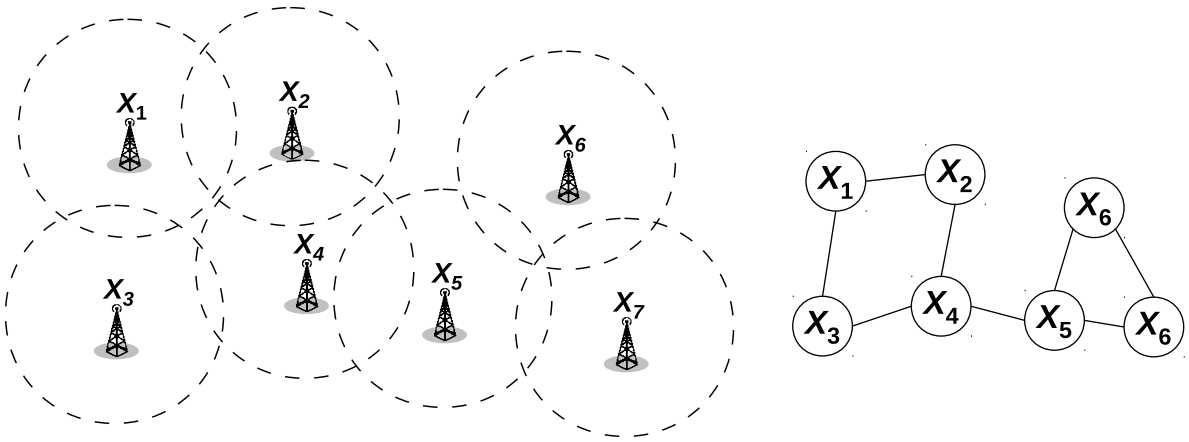
- *Weighted constraint networks*, where the cost functions are real-valued constraints over subset of variables and the combination operator is $\sum$ [28]. This is a generalization of *constraint satisfaction problems* (CSPs), where assignments don't have weights but are either valid or invalid.
- *Bayesian* or *belief networks*, where the functions are conditional probability tables of a variable distribution, $P(X_i \mid par(X_i))$ conditioned on the values of a variable's parents, and the combination operator is $\prod$ [96].
- *Markov networks* or general *factor graphs*, where the functions are called factors or local potentials (not necessarily normalized) with a combination operator $\prod$ [96].

In particular, we can define the following two closely related general optimization problems in the contexts of constraint reasoning and probabilistic inference, respectively.

DEFINITION **1.4** (constraint optimization problem)**.** *A* constraint optimization problem *(also called weighted or soft constraint problem) is a graphical model* $(X, D, F, \sum)$ *with the goal to compute* $\min_X \sum_i f_i$*. The set of cost functions* F *can be seen as penalty terms, and we try to minimize the sum of all such penalties.*

DEFINITION **1.5** (most probable explanation)**.** *The problem of finding the* most probable explanation (MPE) *over a Bayesian network* $(X, D, F, \prod)$ *entails computing* $\max_X \prod_i f_i = \max_X \prod_i P(X_i \mid par(X_i))$ *and the corresponding assignment* $\arg\max_X \prod_i f_i$ *that maximizes the joint probability.*

Definition 1.4 can actually also capture "traditional" constraint satisfaction problems (in a non-optimization context) by expressing constraint relations as functions with costs 0 and



(a) Radio towers, range indicated by dashed circles. **(b)** Underlying primal graph.

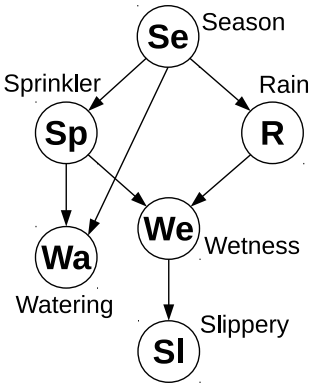
Figure 1.1: Example constraint problem.

"$+\infty$" (or a very large constant), respectively. Definition 1.5 is often equally applied to Markov networks, where the functions are not normalized probability tables but general factors in a multiplicative setting. Finally, we point out that in the context of this thesis these problems are actually interchangeable, through conversion to and from the logarithmic domain or inversion of cost values.

Example **1.1.** *Figure 1.1a depicts an example problem with seven radio towers denoted X_1 through X_7, for instance as part of a cellular phone network, whose geographical range of transmission partially overlaps. We are asked to assign, from a set of possible choices, a transmission frequency window to each tower with the objective to minimize interference between towers whose ranges overlap.*

It is straightforward to formalize this problem as a constraint optimization problem (cg. Definition 1.4) with the seven towers as variables $X = \{X_1, \ldots, X_7\}$ and the set of possible frequency assignments as their respective domains. Furthermore we can define local cost functions for each pair of towers with overlapping ranges, where interference is penalized according to a given practical metric. The resulting primal graph is shown in Figure 1.1b.

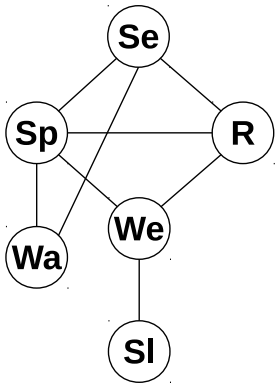
Example **1.2.** *(after [96]) Figure 1.2a shows an example Bayesian network with seven variables, capturing the causal relationships between the season of the year (Se), the configuration*



(a) Directed acyclic graph.

Se	R	$P(R\|Se)$
Spring	Yes	0.3
Spring	No	0.7
Summer	Yes	0.1
Summer	No	0.9
Fall	Yes	0.5
Fall	No	0.5
Winter	Yes	0.8
Winter	No	0.2

(b) Example probability table.



(c) Primal graph.

Figure 1.2: Example Bayesian network with seven variables.

of the sprinkler system (Sp), whether it is raining (R), if the ground is wet (We) and slippery (Sl), and whether additional watering is necessary (Wa). According to the Bayesian network the joint probability factors into $P(Se) \cdot P(Sp|Se) \cdot P(R|Se) \cdot P(We|Sp, R) \cdot P(Sl|We) \cdot P(Wa|Se, Sp)$. *An example of the conditional probability table for* $P(R|SE)$ *is given in Figure 1.2b, while Figure 1.2c shows the resulting primal graph (also known as moral graph in the context of Bayesian networks).*

1.2.3 Properties of Graphical Models

This section introduces a number of graph concepts that are central in the context of graphical model inference.

Definition **1.6** (induced graph, induced width)**.** *Given a primal graph* $G = (V, E)$ *of a graphical model and an ordering* $d = X_1, \ldots, X_n$ *of its nodes, the* induced graph *of* G *is obtained as follows: from last to first in* d*, each node's preceding neighbors are connected to form a clique. The* width *of a node is the number of neighbors that precede it, the* induced width w *is the maximum width over all nodes in the induced graph along ordering* d.

Finding an ordering of minimal induced width w^*, over all possible variable orderings, is known to be NP-complete [7, 12, 44], in practice ordering heuristics like *min-degree* or *min-fill* are used [69].

DEFINITION **1.7** (tree decomposition, tree width)**.** *[100] A* tree decomposition *of a graphical model* $(X, D, F, \otimes)$ *is tree* $T = (V, E)$, *where* V *is a set of nodes, also called "clusters," and* E *is a set of edges, together with a labeling function* χ *that labels each* $v \in V$ *with a set* $\chi(v) \subset X$ *such that:*

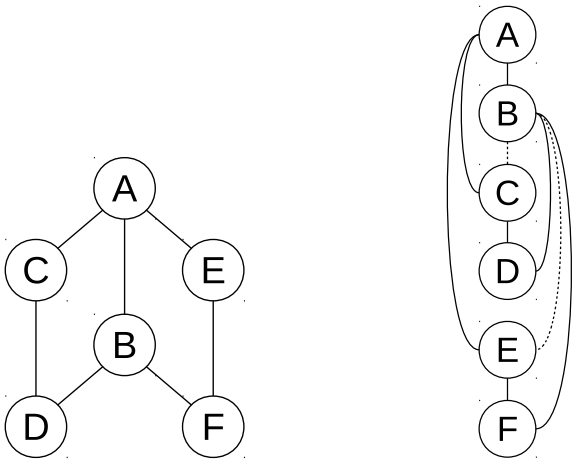
1. *For each* $f_i \in F$ *there exists* $v \in V$ *such that* $scope(f_i) \in \chi(v)$, *i.e. each function's scope is contained in at least one cluster.*
2. *For each* $X_i \in X$ *the set* $\{v' \in V \mid X_i \in \chi(v)\}$ *forms a connected subtree of* T*; this is also called the "running intersection" or "connectedness" property.*

The tree width *of a tree decomposition is defined as* $w := \max_v |\chi(v)| - 1$, *i.e. the size of the largest cluster minus 1. The tree width* w^* *of a graphical model is the minimum tree width over all its tree decompositions.*

There is an obvious connection between the clusters of a tree decomposition and the cliques of a induced graph, as defined above. In particular, one can easily arrange the induced graph's cliques into a tree structure, yielding a tree decomposition. It is therefore well known that the induced width of a graphical model and its underlying graph structure is identical to its tree width, capturing the same thing from two different, but similar perspectives [5, 11, 31].

EXAMPLE **1.3.** *Figure 1.3a shows the primal graph of an example problem with six variables. Figure 1.3b depicts the induced graph along ordering* A, B, C, D, E, F *with two additional edges* (B, C) *and* (B, E) *and width* $w = 2$.

Two possible tree decompositions are provided in Figure 1.4. We note that the tree decomposition in Figure 1.4a in particular corresponds directly to the induced graph in Figure 1.3b,



(a) Primal graph **(b)** Induced graph

Figure 1.3: Example primal graph over six variables and induced graph along ordering A, B, C, D, E, F with induced width $w = 2$.

(a) Possible tree decomposition with six clusters.

(b) Possible tree decomposition with four clusters.

Figure 1.4: Two possible tree decompositions of the example problem from Figure 1.3, both with tree width 2.

while Figure 1.4b merges the two clusters at the top into their neighbors – in that sense these two clusters are redundant. While varying in their number of clusters (six and four, respectively), the two tree decompositions in Figure 1.4 have the same tree width of $w = 2$.

1.3 Solving Graphical Model Problems through Search

Search algorithms present a common approach to systematically enumerate all of the combinatorially many possible assignments of a given graphical model.

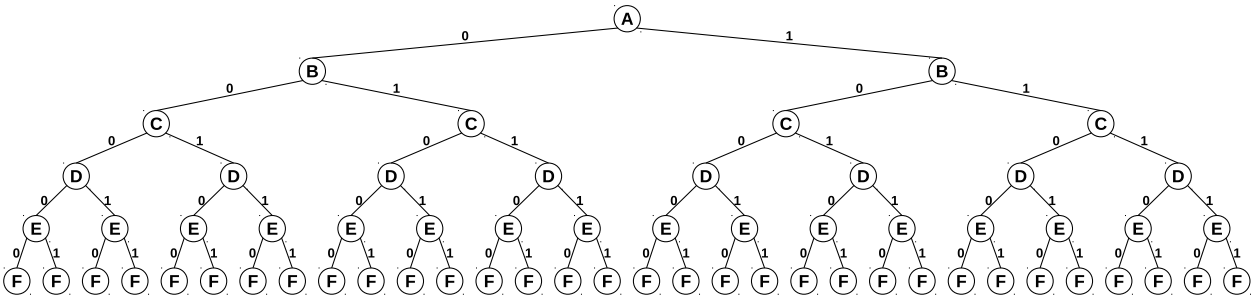


Figure 1.5: Example OR search tree along ordering A, B, C, E, F, G (first to last) for the example problem in Figure 1.3.

In its simplest incarnation, the algorithm instantiates one variable after the other, trying different values in a depth-first manner. As a consequence the number of search nodes to be explored is $O(k^n)$, where n is the number of problem variables and k the maximum domain size. This approach is often referred to as *OR search*. A solution is represented by a path of length n from the root to a leaf, and each node on the path (except the leaf) has exactly one of its children selected for the solution path (hence the "or" moniker).

Example **1.4.** *Figure 1.5 displays the full OR search tree for the six-variable example problem Figure 1.3 with variables instantiated in the order A, B, C, D, E, F (first to last). Every node except the leaves has exactly two children, corresponding to a value assignment of 0 or 1 to the respective variable. Consequentially, the number of nodes grows by a factor of 2 from one level to the next, implying overall size exponential in the number of problem variables. A full assignment corresponds to a path of length 6 from the root to a leave.*

Already at this point it is worth noting, however, that these search space sizes are typically upper bounds, depending on the problem instance at hand. In particular, if a function's scope is fully instantiated by a partial assignment along a given search space path, its value can be looked up. If it is inconsistent (e.g., a probability of 0 in a Bayesian network) we can conclude that the partial assignment cannot possibly be extended to a solution – exploration of the current branch is thus halted and the algorithm backtracks.

The following sections introduce the enhanced concept of AND/OR search spaces, which can yield exponential time savings by exploiting problem structure.

1.3.1 AND/OR Search Spaces

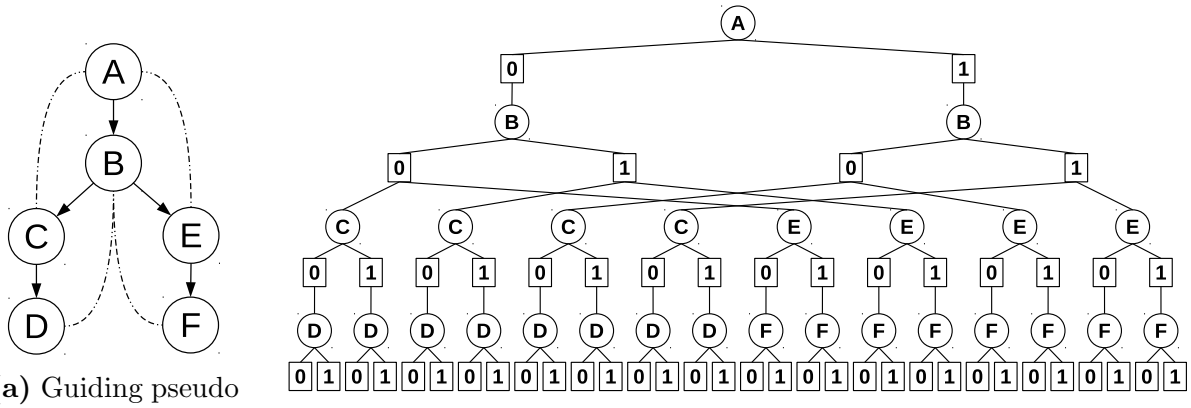
The concept of *AND/OR search spaces* has recently been introduced to graphical models to better capture the structure of the underlying graph during search [29]. The search space is defined using a *pseudo tree* of the graph, which captures problem decomposition as follows:

DEFINITION **1.8** (pseudo tree)**.** *[44] Given the primal graph $G = (V, E)$ of a graphical model, a* pseudo tree *is a directed, rooted tree $T = (V, E')$ such that every arc of G not included in E' is a backarc in T, namely it connects a node in T to an ancestor in T. The arcs in E' may not all be included in E.*

EXAMPLE **1.5.** *A pseudo tree for the example in Figure 1.3a is shown in Figure 1.6a, corresponding to the induced graph in Figure 1.3b along ordering A, B, C, D, E, F – recall that the induced graph is constructed last to first in the ordering, while the pseudo tree proceeds last to first. We note that B has two children in the pseudo tree, capturing the fact that the two subproblems over C, D and E, F, respectively, are independent once A and B have been instantiated.*

1.3.1.1 AND/OR Search Trees

Given a graphical model instance with variables X and functions F, its primal graph (V, E), and a pseudo tree $\mathcal{T}$, the associated *AND/OR search tree* consists of alternating levels of OR and AND nodes [29]. Its structure is based on the underlying pseudo tree $\mathcal{T}$: the root of the AND/OR search tree is an *OR node* labeled with the root of $\mathcal{T}$. The children of an OR node $\langle X_i \rangle$ are *AND nodes* labeled with assignments $\langle X_i, x_j \rangle$ that are consistent



(a) Guiding pseudo tree.

(b) AND/OR search tree with 27 OR and 54 AND nodes.

Figure 1.6: Example pseudo tree and AND/OR search tree along ordering A, B, C, D, E, F (first to last) for the problem in Figure 1.3.

with the assignments along the path from the root; the children of an AND node $\langle X_i, x_j \rangle$ are OR nodes labeled with the children of X_i in $\mathcal{T}$, representing conditionally independent subproblems.

EXAMPLE **1.6.** *Figure 1.6b shows the AND/OR search tree resulting from the primal graph in Figure 1.3a when guided by the pseudo tree in Figure 1.6a. Note that the AND nodes for B have two children each, representing independent subtrees rooted at C and E, respectively, thereby capturing problem decomposition. Also note that the depth of the search tree is only 4 (as opposed to 6 for standard OR search in Figure 1.5).*

THEOREM **1.1.** *[29] Given a pseudo tree $\mathcal{T}$ of a graphical modal with height h, the size of the AND/OR search tree based on $\mathcal{T}$ and the time complexity of an algorithm exploring it is $O(n \cdot k^h)$, where k bounds the domain size of variables. The space complexity of an algorithm exploring the AND/OR search tree in a depth-first manner is $O(h)$.*

1.3.1.2 AND/OR Search Graphs

AND/OR Search Trees can offer exponential savings in the number of explored nodes over the standard OR search approach, thus dramatically reducing computation time [29]. Additional

improvements in time complexity can be achieved by detecting and unifying redundant subproblems based on their *context*:

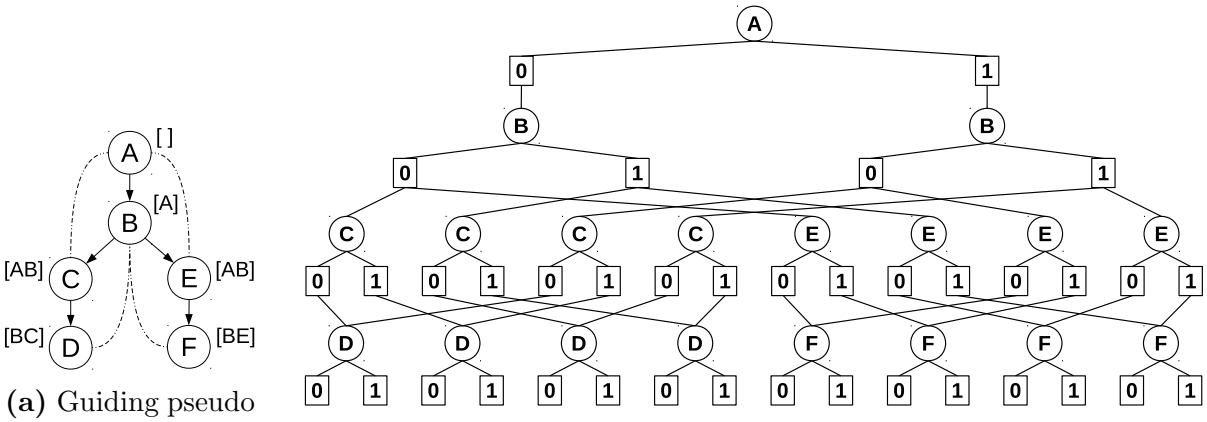
DEFINITION **1.9** (OR context)**.** *[29] Given the primal graph $G = (V, E)$ of a graphical model and a corresponding pseudo tree $\mathcal{T}$, the* OR context *of a node X_i in $\mathcal{T}$ are the parents of X_i in $\mathcal{T}$ that have connections in G to X_i or its descendants.*

In other words, the context of a variable X_i is the partial instantiation that separates the subproblem rooted at X_i from the rest of the network.

EXAMPLE **1.7.** *Figure 1.7a shows the same pseudo tree as before (cf. Figure 1.6a) but with added context information for each variable. We see that A, B, C, and E each have all their respective ancestors as their context. Notably, however, neither D or F have A in their context – this means the solution to any subproblem rooted at D or F will be independent of the value assigned to A on the respective path. This independence relation should also become intuitively clear when looking at the primal graph in Figure 1.3a, where D and F are conditionally independent of A given B, C and B, E, respectively.*

In the context of AND/OR search, identical subproblems identified by their context can be merged, yielding an *AND/OR search graph* [29]. Merging all context-mergeable nodes yields the *context-minimal* AND/OR search graph. This concept is often also referred to as *caching*, based on its typical implementation: Once the depth-first exploration has fully solved a subproblem, its optimal solution cost is stored into a cache table indexed by the context assignment. When a subproblem with the same context assignment is encountered later on, the solution is retrieved from the cache table and reused.

EXAMPLE **1.8.** *Figure 1.7b shows the context-minimal AND/OR search graph for the example problem from Figure 1.3a when using the pseudo tree from Figure 1.7a. In contrast to the AND/OR tree in Figure 1.6b, the OR nodes for D (with context $\{B, C\}$) and F (con-*



(a) Guiding pseudo tree with contexts.

(b) AND/OR search graph with 19 OR and 38 AND nodes.

Figure 1.7: Example AND/OR search graph for problem in Figure 1.3.

text $\{B, E\}$*) have two edges converging from the AND level above them, signifying caching (namely, the assignment of* A *does not matter).*

THEOREM **1.2.** *[29] Given a pseudo tree* $\mathcal{T}$ *of a graphical model with induced width (or tree width)* w*, the size of the context-minimal AND/OR search graph based on* $\mathcal{T}$ *and the time complexity of an algorithm exploring it is* $O(n \cdot k^w)$*, where* k *bounds the domain size of variables. The space complexity of a depth-first algorithm exploring the context-minimal AND/OR graph is* $O(n \cdot k^w)$*.*

The increased asymptotic space complexity is due to the additional memory required for storing the cache tables. Caching can be therefore be seen as a way of trading shorter computation time for increased space requirements.

Given an AND/OR search space $S_{\mathcal{T}}$, a *solution subtree* $Sol_{S_{\mathcal{T}}}$ is a tree such that (1) it contains the root of $S_{\mathcal{T}}$; (2) if a nonterminal AND node $n \in S_{\mathcal{T}}$ is in $Sol_{S_{\mathcal{T}}}$ then all its children are in $Sol_{S_{\mathcal{T}}}$; (3) if a nonterminal OR node $n \in S_{\mathcal{T}}$ is in $Sol_{S_{\mathcal{T}}}$ then exactly one of its children is in $Sol_{S_{\mathcal{T}}}$.

1.3.1.3 Weighted AND/OR Search Spaces

Given an AND/OR search graph, each edge from an OR node X_i to an AND node x_i can be annotated by *weights* derived from the set of cost functions F in the graphical model: the weight $l(X_i, x_i)$ is the combination of all cost functions whose scope includes X_i and is fully assigned along the path from the root to x_i, evaluated at the values along this path. Furthermore, each node n in the AND/OR search graph can be associated with a *value* $v(n)$, capturing the optimal solution cost to the subproblem rooted at n, subject to the current variable instantiation along the path from the root to n.

The value $v(n)$ of a node n can be computed recursively using the values of n's successors [29]. In a max-product scenario like MPE over a Bayesian network, for instance, we have the following:

- If n is an AND node, $v(n)$ is the product of the values of n's successors.
- If n is an OR node, $v(n)$ is the maximum over all its childrens' values.

1.3.2 AND/OR Branch-and-Bound

AND/OR Branch and Bound (AOBB) is a state-of-the-art algorithm for solving optimization problems such as max-product over graphical models [82, 83]. Assuming a maximization query, AOBB traverses the weighted context-minimal AND/OR graph in a depth-first manner while keeping track of the current lower bound on the maximal solution cost. A node n will be pruned if this lower bound exceeds a heuristic upper bound on the solution to the subproblem below n (cf. Section 1.3.3). The algorithm interleaves forward node expansion with a backward cost revision or propagation step that updates node values (capturing the current best solution to the subproblem rooted at each node), until search terminates and the optimal solution has been found [82].

Algorithm 1.1 AND/OR Branch-and-Bound (AOBB)

Given: Graphical model optimization problem $(X, D, F, \otimes, \Downarrow)$ and pseudo tree $\mathcal{T}$ with root X_o, heuristic evaluation function h.
Output: cost of optimal solution

```
1: OPEN ← {⟨X_0⟩}
2: while OPEN ≠ ∅:
3:    n ← top(OPEN)  // top node from stack, depth-first
4:    if checkpruning(n, h(n)) = true:
5:       prune(n)  // perform pruning
6:    else if cachelookup(n) ≠ NULL:
7:       value(n) ← cachelookup(n)  // retrieve cached value
8:    else if n = ⟨X_i⟩ is OR node:
9:       for x_j ∈ D_i:
10:         create AND child ⟨X_i, x_j⟩
11:         add ⟨X_i, x_j⟩ to top of OPEN
12:   else if n = ⟨X_i, x_j⟩ is AND node:
13:      for Y_r ∈ children_T(X_i):
14:         generate OR node ⟨Y_r⟩
15:         add ⟨Y_r⟩ to top of OPEN
16:   if children(n)= ∅:   // n is leaf
17:      propagate(n)  // upwards in search space
18: return value(⟨X_0⟩)  // root node has optimal solution
```

Algorithm 1.1 shows pseudo code for AOBB: Starting with just the root node $\langle X_0 \rangle$ on the stack, the algorithm iteratively takes the top node n from the stack (line 3), thereby implementing depth-first exploration. Lines 4–7 try to prune the subproblem below n (by comparing a heuristic estimate of n against the current lower bound) and check the cache to see if the subproblem below n has previously been solved (full details were developed in [82]). If neither of these is successful, the algorithm generates the children of n (if any) and pushes them back onto the stack (8–15). If n is a terminal node in the search space (it was pruned, its solution retrieved from cache, or the corresponding X_i is a leaf in $\mathcal{T}$) its value is propagated upwards in the search space, towards the root node (16–17); cache entries and memory cleanup of fully solved subproblems along the way are applied where appropriate (see [82] for details). When the stack eventually becomes empty and the outer while loop exits, the value of the root node $\langle X_0 \rangle$ is returned as the solution to the problem (18).

THEOREM **1.3.** *[82] The time and space complexity of AND/OR Branch-and-Bound on a graphical model optimization problem with n variables, maximum domain size k, and using a variable ordering yielding induced width w is $O(n \cdot k^w)$, since it explores the context-minimal AND/OR search graph.*

Note that Corollary 1.3 captures only the asymptotic worst-case complexity. In practice we note that these bounds are known to be very loose, due to determinism in the problem specification and the powerful pruning of AOBB, both of which allow large portions of the search space to be ignored. This discrepancy will be the premise for the work presented in Chapter 3, where we aim to predict the actual amount of work (in number of node expansions) needed by AOBB to solve a given problem instance.

1.3.3 Mini-bucket Heuristic

The heuristic $h(n)$ that we deploy within our implementation of AOBB throughout this thesis is the mini-bucket heuristic. It is based on mini-bucket elimination, which is an approximate variant of variable elimination and computes approximations to reasoning problems over graphical models [28, 32]. The mini-bucket heuristic has been shown to be *admissible*, i.e., it never underestimates the true cost of a subproblem in a maximization setting (or never overestimates in a minimization context, respectively) [65, 87].

Algorithm 1.2 shows pseudo code for the mini bucket algorithm for application to a max-product problem like MPE. It receives as input the optimization problem, a variable ordering, as well as a control parameter i . It first partitions the functions into buckets according to the highest-indexed (wrt. to the ordering) variable in their scope (line 1). Subsequently, buckets are processed last to first by partitioning into mini-buckets that satisfy the specified i-bound, i.e., the union of the contained functions' scopes doesn't have more than i variables (line 3). Each mini-bucket is then processed separately by applying first the combination and then

Algorithm 1.2 Mini-bucket elimination for *max-product* (adapted from [82])

Given: Graphical model optimization problem $(X, D, F, \prod, \max)$, variable ordering $d = X_1, \ldots, X_n$, and parameter i ("i-bound").
Output: Upper bound on optimal solution cost.

1: Distribute functions $f_j \in F$ into buckets $B_1, \ldots, B_n$, where each function f_j goes into the bucket B_k of its highest variable X_k (wrt. ordering d), or bucket B_o for constant functions.
2: **for** $k \leftarrow n$ down to 1:
3: Partition bucket B_k into mini-buckets $B_k^{(1)}, \ldots, B_k^{(m)}$ such that $|vars(B_k^{(j)})| \leq i \ \forall j$.
4: **for** $j \leftarrow 1$ to m:
5: Let $f_1^{(j)}, \ldots, f_l^{(j)}$ be the set of functions in mini-bucket $B_k^{(j)}$.
6: Generate function $g_k^{(j)} := \max_{X_k}(\prod_{p=1}^{l} f_p^{(j)})$.
7: Add $g_k^{(j)}$ to the bucket of the highest variable in its scope (B_o if empty scope).
8: **return** $\prod_{f_j \in B_o} f_j$, mini-bucket upper bound.

the elimination operator (product and maximum in the case of a max-product problem, line 6). The resulting function is again placed into the bucket of its highest-indexed variable, where it will subsequently be processed with the other functions in that bucket (line 7). After processing all buckets, the combination of all constant functions generated along the way (collected in the "0-th" bucket) yields the overall bound on the solution cost (line 8).

The i-bound allows a trade-off between accuracy of the algorithm (and the resulting heuristic) on the one hand, and its time and space requirements on the other hand, as follows:

THEOREM **1.4.** *[32, 65] Given a graphical model with variable ordering having induced width w and an i-bound parameter, the time and space complexity of the mini-bucket algorithm MBE(i) is $O(n \cdot k^{\min(i,w)})$, where n and k are the number of problem variables as well as the maximum domains size, respectively.*

Higher values of i take more computational resources but yield more accurate bounds. We note that typically $i < w$ is chosen, which is why we generalize to say that MBE(i) is exponential in i. In the case where $i > w$ we point out that MBE(i) in fact corresponds to full variable/bucket elimination [27] and computes the exact solution.

It has been shown that the intermediate functions generated by MBE(i) overestimate the optimal solution cost to subproblems in the AND/OR search graph (assuming a maximization query), just as the overall bound returned by MBE(i) overestimates the overall solution [65]. These intermediate functions can therefore be used to derive a heuristic function that is admissible, as defined above.

The application of the mini-bucket heuristic in AOBB is twofold. First, it is used to determine the value ordering, i.e., it guides the order in which the children of an OR node (different instantiations of a given variable) are considered. Second, it is at the core of the powerful pruning in the branch-and-bound scheme, in which the current best solution is compared against the heuristic estimate of the optimal solution below a given node. Because of the heuristic's admissibility, if the current best solution exceeds this estimate (again assuming a maximization setting), the subproblem in question cannot possibly yield an improvement and can hence be safely pruned.

1.3.4 Other Related Work

This section will briefly review two related algorithms, Limited Discrepancy Search and Stochastic Local Search, which we will employ within our own work later in this thesis.

1.3.4.1 Limited Discrepancy Search

Limited Discrepancy Search (LDS) [58, 71, 99] is a systematic, but generally incomplete heuristic search algorithm. It was originally formulated for Boolean satisfiability (SAT) problems or more generally binary constraint satisfaction problem, i.e., not in an optimization context. It explores an OR search tree that is assumed to be "heuristically ordered." Namely,

Algorithm 1.3 LDS(n, l, h)

Given: Search node n, discrepancy limit along current path l, heuristic function h

```
1: if n is a goal node:
2:    return n
3: C ← children(n)
4: if C = ∅:
5:    return NULL
6: if l = 0:
7:    return LDS(first_h(C), 0, h)  // No further discrepancy on current path
8: result ← LDS(second_h(C), l − 1, h)  // Commit a discrepancy
9: if result ≠ NULL:
10:   return result
11: else
12:   return LDS(first_h(C), l, h)
```

LDS uses a heuristic function that imposes an order over each node's children, from most to least promising.

The intuition behind LDS is then that the guiding heuristic can be mostly relied upon, but not entirely. To formalize this the notion of a *discrepancy* is introduced, which denotes a pruning decision where the chosen path does not follow the most promising heuristic value. The discrepancy of a path in the search tree is then simply the number of discrepancies on it. Given a discrepancy limit l and a heuristic function h LDS explores all paths in the search tree with discrepancy less than or equal to l, according to h.

Pseudo code for the case of a binary search space is given in Algorithm 1.3. Taking after [58], it is presented as a simple recursive procedure LDS(n, l, h), which is called with a search node n (the root node in the initial call), the discrepancy limit l along the current path, and the heuristic function h. After checking for a goal node (lines 1-2) or dead end (lines 3-5), the actual discrepancy logic is applied:

1. If $l = 0$, no more discrepancies can be inserted on the current path and LDS only explores the (according to h) more promising child $first_h(C)$ (line 7).

2. Otherwise the less promising child $second_h(C)$ is explored first, with the discrepancy limit reduced by one in the recursive call (line 8).

3. If this doesn't yield a solution, the more promising child is considered, with an unchanged discrepancy limit (line 12).

It is straightforward to extend Algorithm 1.3 to non-binary variable domains, for instance by discounting at each step the 2^{nd} most promising child as discrepancy 1 (as before), the 3^{rd} most promising as discrepancy 2, etc. Similarly, branch-and-bound-style pruning logic can applied over the subtree implied by LDS with a given discrepancy limit, enabling the application to optimization problems.

It should also be obvious the LDS can be turned into an complete algorithm by running it iteratively with increasing discrepancy limits – eventually the limit will be high enough to permit exploring the entire search space. This is also how the algorithm was presented in [58], however it is not very efficient, since many nodes are re-expanded on every iteration. In our context, we have thus used LDS only for preprocessing purposes, to find an initial lower/upper cost bound (cf. Section 2.6.1).

1.3.4.2 Stochastic Local Search

All search algorithms described so far explore a search tree or search graph, where inner nodes represent partial assignments and leaf nodes capture full assignments (and potential solutions); the structure of the tree or graph allows for systematically enumerating the entire solution space or, in the case of LDS (Section 1.3.4.1), a well-defined part of it.

In contrast, *local search* operates on the space of all complete assignment. The algorithm moves from one assignment to the next by considering, at each step, a local neighborhood

of assignments, often defined through one variable assignment at a time. Local search is inherently incomplete and cannot prove optimality of a solution.

In its simplest form, local search starts out with an initial random full assignment and then at each step greedily moves to the neighboring assignment that improves the global solution cost the most. Clearly this simple greedy scheme, also called *hill climbing*, is prone to "getting stuck" in local optima, leading to potentially bad performance.

To mitigate this, algorithms like *Greedy+Stochastic Simulation* (G+StS) [64] add a random element, which probabilistically chooses between the greedy step and a stochastic step like sampling. In addition a restart mechanism can be applied, which regularly resets the local search procedure to a different random starting point while keeping track of the overall best solution found.

Orthogonal to these enhancements, *dynamic local search* generalizes the evaluation function that guides the local moves, making it independent of the actual problem cost function and allowing it to to change over time. In *Guided Local Search* (GLS) [94] in particular, the evaluation function consists of a sum of penalty terms over so-called "solution components," which are partial assignments for each cost function scope. Whenever a local maximum is reached, the contribution of each cost function to the overall solution cost is evaluated and those that contribute the least have the penalty for their respective assignment increased, to steer subsequent exploration away from this particular solution component.

One of the current state-of-the-art local search algorithms in the context of MPE queries over Bayesian networks is GLS^+ [60], which combines and extends the schemes outlined above. In particular, its core features are the following:

- It extends the evaluation function that guides the local moves to include not only the penalty values of GLS but also the logarithm of the actual global assignment cost, thereby reintroducing a greedy component.

- It regularly smoothes the penalty terms by a constant factor $\rho < 1$ to keep them from growing too large and effectively "blocking" certain parts of the search space.

- It frequently restarts the local search procedure, where initialization is not purely random but based on a pass of mini-bucket elimination with a relatively low i-bound, usually yielding a better starting from which the algorithm can improve rapidly.

- It has been subject to extensive parameter tuning and utilizes a highly efficient implementation of a caching mechanism for the local neighborhood evaluation at each step to significantly speed up computation time.

GLS$^+$ was entered into the approximate reasoning track of the UAI 2008 Probabilistic Inference Evaluation and proved competitive in a number of problem categories [23]. We will also include it as part of our evaluation of anytime performance in Chapter 2.

Chapter 2

Breadth-Rotating AND/OR Branch-and-Bound

2.1 Introduction

As outlined in Chapter 1, depth-first Branch-and-Bound is an established and efficient class of algorithms for exactly solving combinatorial optimization problems over problems. One property that is particularly valuable in practice is its *anytime behavior*.

Namely, when finding a feasible solution is easy but finding an optimal one is hard, depth-first Branch-and-Bound generates solutions that get better and better over time, until it eventually discovers an optimal one. Thus, it can function also as an approximation scheme for otherwise infeasible problems or when time is limited [56, 119].

Indeed, in the 2010 UAI Approximate Inference Challenge Branch-and-Bound solvers performed competitively with respect to approximation (placing 1^{st} and 3^{rd} in some categories) [35]. But we also observed an inability of AND/OR Branch-and-Bound (cf. Section 1.3.2) to

produce even a single solution on some instances, especially when the time bound was small. Thus motivated, this chapter will demonstrate that the issue is rooted in the underlying AND/OR search space.

These search spaces were originally introduced to graphical models to facilitate problem decomposition during search (e.g., [29]) and can be explored by any search strategy. When traversed depth-first, however, all but one decomposed subproblem will be *fully solved* before a single overall solution can be composed, voiding the algorithm's anytime characteristics. Mitigating and repairing this deficiency will be at the core of this chapter.

2.1.1 Contributions

The following outlines the central contributions of this chapter:

- We analyze and demonstrate empirically the conflict between the anytime behavior of depth-first branch-and-bound and the problem decomposition of AND/OR search spaces. In particular, in depth-first exploration all but one subproblem will be solved to completion before any overall solution can be constructed.
- We observe that under certain conditions, namely if only one of the decomposed subproblems is "hard," this adverse effect can be mitigated by processing subproblems in a suitable order. Specifically, we might be able to quickly solve the "easy" subproblems and combine their exact solutions with the gradually increasing solutions of the remaining "hard" subproblem. We demonstrate the merit and the limitations of this approach empirically.
- This chapter's main contribution is then a new Branch-and-Bound scheme over AND/OR search spaces, called *Breadth-Rotating AND/OR Branch-and-Bound (BRAOBB)* that addresses the anytime issue in a principled way, while maintaining the favorable

complexity guarantees of depth-first search. The algorithm performs breadth-first exploration of the different subproblems (by "rotating" through them), each of which is processed depth-first.

- Experimental evaluation is conducted on a variety of benchmark classes, including haplotype computation problems in genetic pedigrees, random grid networks, and protein side-chain prediction instances. We compare BRAOBB against one of the best variants of (standard) AND/OR Branch-and-Bound search, AOBB [82], and against an "ad hoc" fix that we suggest – the latter algorithm relies on a heuristic to quickly find a solution to each subproblem before reverting to depth-first search. We furthermore compare against a state-of-the-art stochastic local search (SLS) solver, which is specifically targeted at anytime performance but cannot provide any proof of optimality [60].

- Empirical results demonstrate superior anytime behavior of BRAOBB, meaning it generally finds better solutions sooner, especially over problematic cases where standard AOBB and its ad hoc fix fail. This includes several very hard instances from the 2010 UAI Approximate Inference Challenge and three weighted constraint satisfaction problem instances that are known to be very complex. We further observe many cases where BRAOBB outperforms SLS, but also find some evidence of the strengths of the latter – in particular where large domain sizes limit the accuracy of the mini-bucket heuristic used within BRAOBB. Based on this we show how local search and exhaustive AND/OR search can be combined to let us enjoy the benefits of both approaches.

Notably, a solver based on this concept recently won all three categories (20 seconds, 20 minutes, and 1 hour) in the MPE track of the PASCAL 2011 Inference Challenge [36], the successor to the 2010 UAI Challenge.

2.1.2 Chapter Outline

The remainder of this chapter is structured as follows: Section 2.2 surveys and contrasts with related work. Section 2.3 identifies the central conflict between problem decomposition and anytime performance and provides empirical results where the latter is compromised. The new algorithm Breadth-Rotating AOBB is proposed in Section 2.4 and its theoretical properties are analyzed.

Section 2.5 presents exhaustive experimental results and analysis using a wide range of example problems as well as summary statistics across more than 500 instances. Section 2.6 concludes and briefly describes out winning entry into the PASCAL 2011 Inference Challenge.

2.2 Related Work

The work presented here is focused on optimization problems defined over graphical models. As such our results are also relevant for a number of related algorithms that exploit the conditional independence relations captured by the graphical model structure:

- *Recursive Conditioning* [22] is guided by a *dtree* structure, a full binary tree with the problem's cost functions at its leaves. It instantiates variables like a cycle cutset scheme [26], but with the intention of breaking the problem into independent subproblems, on which it is then applied recursively. Its execution was shown to correspond to a particular AND/OR search space [29].

- Similarly, *Value Elimination* [6] is a scheme for probabilistic inference. It can accommodate dynamic variable orderings, but for fixed ones it was also shown to explore an AND/OR search space guided by a particular pseudo tree [29]. The focus, however,

is on marginalization problems like probability of evidence or finding the partition function.

- Finally, the *BTD* algorithm ("Backtracking with Tree Decomposition" [62]) is a depth-first search scheme for constraint optimization problems combined with tree decomposition-based inference methods and soft-consistency heuristics. Again, it can be viewed as AND/OR search along a specific pseudo tree that is compatible with the tree decomposition used by BTD.

All three schemes cited above, as well as any other scheme that can be seen as exploring an combinatorial AND/OR search space in a depth-first manner, are prone to degraded anytime performance and can in principle benefit from the ideas presented in this work.

We further note *Interleaved Depth-First Search* (IDFS) [85], whose underlying concept shares some similarities with our work. Namely, it performs interleaved processing of different branches in an otherwise depth-first search space. In this case, however, the intention is to mitigate branching mistakes made higher up in the search space with respect to successor orderings. IDFS was also only presented in a standard OR search framework and solely for general constraint satisfaction problems, i.e., not in an optimization context.

In the area of heuristic state-space search, with applications to path-finding problems such as the Towers of Hanoi, sliding tile puzzles, or general planning problems, some effort has gone into anytime search [10, 109]. Prominent algorithms in this field are based on the classic A^* best-first search algorithm [30, 57], more specifically its *weighted* A^* approximate variant, where the heuristic value in the node evaluation function is multiplied by a constant [95]. *Anytime Weighted* A^* [55] as well as the closely related *Anytime Repairing* A^* (ARA*) [79] extend this by iteratively running weighted A* with different sequences of such weights, eventually falling back to standard, exact A* search. The application of these concepts to

combinatorial optimization over graphical models and AND/OR search spaces in particular is the subject of ongoing work [42].

A slightly less obvious connection can also be made to recent contributions in the area of distributed, multi-agent constraint optimization. Algorithms like NCBB [17] and BnB-ADOPT [116] organize agents along a pseudo tree-like structure, thereby obtaining solutions to independent subproblems in parallel. ADOPT [86] uses a similar approach for agent-based best-first search. However, neither of these algorithms is concerned with anytime performance – in fact, in the multi-agent setting entirely different metrics are typically applied (such as number of messages exchanged between agents).

Finally, most directly related to the objective of this work is the well-known concept of local search, which explores local "neighborhoods" of assignments, either at random or guided by a heuristic, and which can be seen as specifically targeting anytime performance [84]. However, it differs from AOBB and the proposed BRAOBB in that it cannot prove optimality of the solutions it returns. Nevertheless we include the state-of-the-art stochastic local search solver GLS^+ [60] (cf. Section 1.3.4.2) in our empirical evaluation.

2.3 Anytime Behavior versus Problem Decomposition in AND/OR Search

As a depth-first branch-and-bound scheme one would expect AOBB to quickly produce a non-optimal solution and then gradually improve upon it, maintaining the current best one throughout the search. However this ability is compromised in the context of AND/OR search, as we will show in the following.

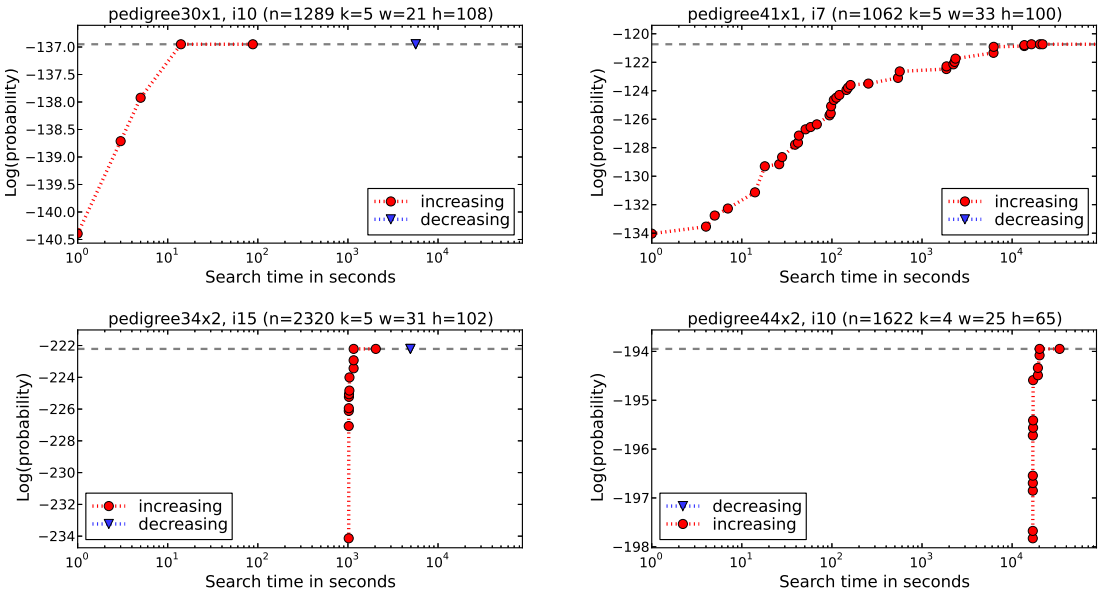


Figure 2.1: Impact of subproblem ordering on AOBB. Specified for each network: number of variables n, maximum domain size k, induced width w along the chosen ordering, height of the corresponding pseudo tree h. The dashed gray line indicates the optimal solution value.

Specifically, in AND/OR search spaces depth-first traversal of a set of independent subproblems will solve to completion all but one subproblem before the last one is even considered. As a consequence, the first generated overall non-optimal solution contains conditionally optimal solutions to all subproblems but the last one. Furthermore, depending on the problem structure and the complexity of the independent subproblems, the time to return even this first non-optimal overall solution can be significant, practically negating the anytime behavior of depth-first search (DFS).

2.3.1 Subproblem Ordering

In certain cases, the above suggests a simple remedy: if decomposition yields only one large subproblem and several smaller ones, the latter can be solved depth-first in relatively little time, to be then combined with the incrementally improving solutions of the larger subprob-

lem. Thus for anytime behavior an AOBB algorithm would need to process independent subproblems from "easy" to "hard,"

To demonstrate the practical impact of subproblem orderings, we use a simple heuristic that takes the induced width as a measure of subproblem hardness (motivated by its exponential role in the asymptotic complexity), i.e. we modify AOBB such that subproblems with smaller induced width will be processed first (in the general description of AOBB the subproblem ordering is left unspecified).

Figure 2.1 contrasts the anytime behavior of AOBB using this "increasing" subproblem order against the inverse one ("decreasing") by plotting the solution cost generated as a function of time on two example problems (the dashed horizontal line is the optimum cost); all other aspects of the algorithm remain constant. In particular, pedigree30x1 features exactly one single complex subproblem and a number of relatively simple ones. In this case processing subproblems by increasing induced width right away produces a non-optimal solution that improves rapidly. The inverse order yields the first solution only after about 90 minutes – the one complex subproblem has been fully solved and the overall solution is already optimal. Pedigree41x1 has a similarly advantageous structure and thus yields similar results – with the distinction that the inverse subproblem order does not produce any solution at all within 24 hours.

In case of pedigree34x2 and pedigree44x2, however, decomposition yields two complex subproblems: the increasing subproblem order still outperforms its inverse, yet it returns the initial solution only after about 1,000 and 17,000 seconds, respectively. In fact, no possible subproblem ordering can lead to acceptable anytime behavior in these cases due to the structure of subproblems, clearly highlighting the limits of this approach.

Independent of anytime behavior, we point out that incorporating different subproblem orderings impacts the algorithm's overall efficiency (i.e., the time to find and prove an optimal

solution): knowing the solution to one subproblem can aid the pruning of Branch-and-Bound in the next one to varying degrees. However, this issue has not been treated systematically in the literature for graphical models, with sporadic experiments also suggesting an easy-to-hard order, using some heuristic to determine subproblem complexity [82]. This general problem is outside the scope of this thesis, however.

Value Ordering

In this context it is worth point out the connection to the choice of value ordering in branch-and-bound search. Both for traditional OR spaces as well as AND/OR ones, it is known that the order in which different instantiations of a given variable (i.e., the children of an OR node in AND/OR spaces) are considered can have an impact on the overall number of node expansions [75, 82]. Specifically, discovering a better (or worse) optimal solution for one value instantiation can lead to stronger (or weaker) pruning for subsequent ones – which is quite similar to our observation regarding the order of subproblems. As noted in Chapter 1, in the context of AOBB the mini-bucket heuristic is used to determine the order in which value instantiations are explored, from most to least promising [82].

2.3.2 Greedy Subproblem Dive

Another relatively straightforward remedy for the compromised anytime behavior of AOBB is the following "ad hoc" fix: Every time that decomposition is encountered within the search space, we will try to greedily find a single initial solution to each independent subproblem before successively solving each of them to completion depth-first, through normal AOBB. To obtain this initial solution the algorithm can perform a greedy "dive" into each subproblem by only considering one value for each variable along the path (in case of the mini-bucket

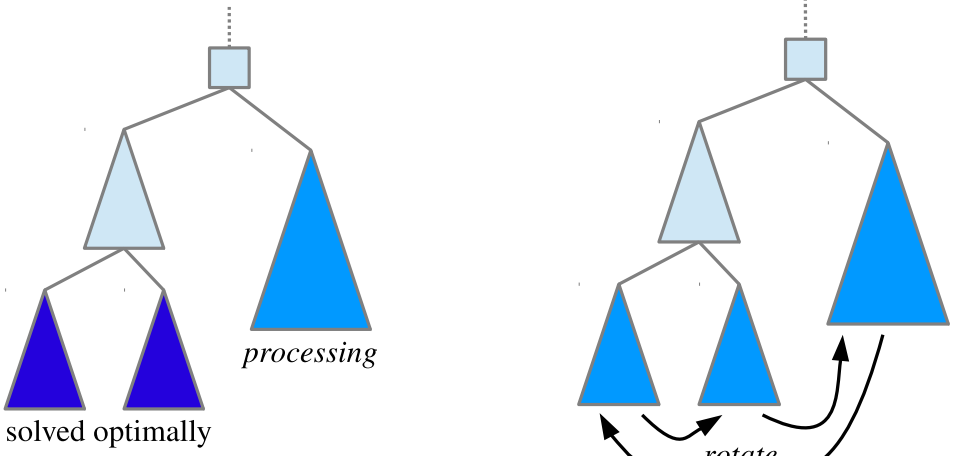
heuristic, it is easy to see that this is equivalent to a forward pass over the bucket structure [65]).

Clearly, the choice of the dive path is crucial for the algorithm's performance. Namely, if the chosen path leads to a dead end (zero probability), the dive will be futile and not yield a subproblem solution. This again negates the desired anytime behavior, since the subproblem for which the dive failed will not be reconsidered until the normal depth-first AOBB phase. And in fact experiments in Section 2.5 will demonstrate that the resulting performance depends heavily on the quality of the heuristic, which often prevents satisfactory anytime behavior. In the next section we will therefore propose a new search strategy that addresses the anytime issue over AND/OR search spaces in a principled manner.

2.4 Breadth-Rotating AOBB

In the following we develop a new search scheme called *Breadth-Rotating AND/OR Branch-and-Bound (BRAOBB)* that addresses the issue of anytime performance over AND/OR search spaces. It combines depth-first exploration with the notion of "rotating" through different subproblems in a breadth-first manner. Namely, node expansion still occurs depth-first as in standard AOBB, but the algorithm takes turns in processing subproblems, each up to a given number of operations at a time, round-robin style.

To motivate this approach, consider again that a solution is represented by a *solution tree* over an AND/OR search space, guided by a pseudo tree. A pure DFS scheme will construct the different branches of a solution tree one by one, ensuring optimality for each branch before moving to the next. To restore anytime behavior, we instead aim to develop all branches of the solution tree "simultaneously," which we achieve by rotating through them.



(a) Default depth-first subproblem processing.

(b) BRAOBB subproblem rotation.

Figure 2.2: Illustration of subproblem rotation in Breadth-Rotating AOBB.

Figure 2.2 illustrates this concept: In Figure 2.2a the two subproblems on the left have been solved to completion before the third subproblem is considered at all. Using BRAOBB, on the other hand, the three independent subproblems in Figure 2.2b contribute to the overall solution simultaneously.

2.4.1 Subproblem Rotation

More systematically, the algorithm maintains a list of currently open subproblems and repeats the following high-level steps until completion:

1. Move to next open subproblem P in a breadth-first fashion.
2. Process P depth-first, until either:
   (a) P is solved optimally,
   (b) P decomposes into child subproblems, or
   (c) a predefined threshold number of operations is reached.

Algorithm 2.1 Breadth-Rotating AOBB

Given: Graphical model $(X, F, D, \max, \prod)$ and pseudo tree $\mathcal{T}$ with root X_o, rotation threshold Z
Output: cost of optimal solution

```
1: ROOT ← {⟨X_0⟩}  // generate root subproblem
2: GLOBAL ← [ROOT]  // put it into queue
3: while GLOBAL≠ ∅
4:    LOCAL ← front(GLOBAL)  // next subproblem
5:    for z ← 1 to Z or until LOCAL = ∅
          or until childSubprob(LOCAL) ≠ ∅
6:       n ← top(LOCAL)  // next node in subproblem
7:       ...  // caching and pruning as in AOBB
8:       if n = ⟨X_i⟩ is OR node
9:          for x_j ∈ D_i
10:            create AND child ⟨X_i, x_j⟩
11:            add ⟨X_i, x_j⟩ to top of LOCAL
12:      else if n = ⟨X_i, x_j⟩ is AND node
13:         Y_1, ..., Y_m ← children_T(X_i)
14:         generate OR children ⟨Y_1⟩, ..., ⟨Y_m⟩
15:         if m=1  // no decomposition
16:            push ⟨Y_1⟩ to top of LOCAL
17:         else if m > 1  // problem decomposition
18:            for r ← 1 to m
19:               NEW ← {⟨Y_r⟩}  // new child subproblem
20:               push NEW to back of GLOBAL
21:      if children(n)= ∅  // n is leaf
22:         propagate(n)  // upwards in search space
23:   if LOCAL ≠ ∅  // subproblem not yet solved
24:      push LOCAL to end of GLOBAL
25: return value(⟨X_0⟩)  // root node has optimal solution
```

The threshold in (c) is needed to ensure the algorithm does not get stuck in one large subproblem where the other two conditions, (a) and (b), do not occur for a long time. Furthermore, in order to focus on a single solution tree at a time, a subproblem is only considered "open" if it does not currently have any child subproblems, as illustrated below.

2.4.2 Algorithm Pseudo Code

Algorithm 2.1 gives more detailed pseudo code for Breadth-Rotating AND/OR Branch-and-Bound as an extension of AOBB, which was described in Section 1.3.2. As stated, the

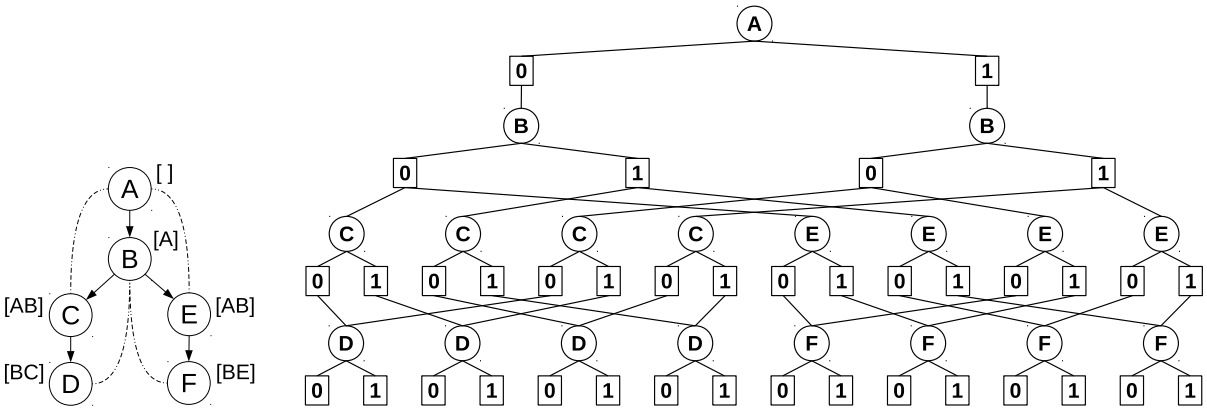
central element of BRAOBB lies in rotating over the different subproblems of the search space. To that end subproblems are organized into a global queue ($GLOBAL$); its first-in-first-out property ensures the desired breadth-first exploration across different branches of the solution tree. Each subproblem is itself explored depth-first via a last-in-first-out stack of nodes; the currently active one is referred to as $LOCAL$ in Algorithm 2.1.

Execution begins with only one subproblem in the $GLOBAL$ queue, which in turn only has the root node $\langle X_o \rangle$ on it stack (lines 1-2). In line 4 the next subproblem is taken from the $GLOBAL$ queue and its stack loaded into the $LOCAL$ stack. Lines 5 through 22 then process the subproblem until either of three conditions listed in Section 2.4.1 is met:

1. The node expansion counter z reaches its limit (the rotation threshold Z given as input to the algorithm);
2. The current subproblem is fully solved and the $LOCAL$ stack becomes empty;
3. The current subproblem decomposes further, captured by the figurative call "childSubprob($LOCAL$)."

Subproblem processing proceeds very similarly to standard AOBB; the top node n from the $LOCAL$ stack is removed (line 7) and caching or pruning is attempted (omitted here, cf. Algorithm 1.1 and [82]). Should these both fail the node is expanded:

- If $n = \langle X_i \rangle$ is an OR node, its AND children are simply pushed onto the $LOCAL$ stack, just like in AOBB (lines 8-11);
- If it is an AND node, $n = \langle X_i, x_j \rangle$, the pseudo tree children of its corresponding variable X_i are retrieved (line 13) and the OR children of n are generated accordingly (line 14). If there is only a single child OR node, i.e., there is just single subproblem and no decomposition, it is pushed onto the $LOCAL$ stack as usual (lines 15-16). If



(a) Pseudo tree. **(b)** AND/OR search graph.

Figure 2.3: Example AND/OR search graph for problem in Figure 1.3.

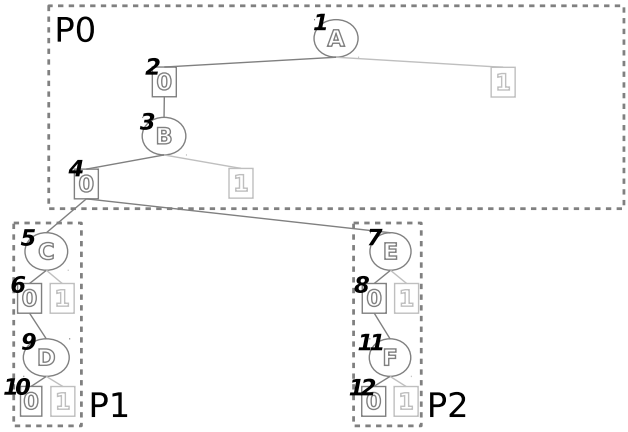
there are two or more subproblems, however, each of them is pushed to the back of the *GLOBAL* queue for subsequent breadth-first processing (lines 17-20).

If no children were generated for n , propagation of cost values is conducted just as in AOBB (lines 21-22).

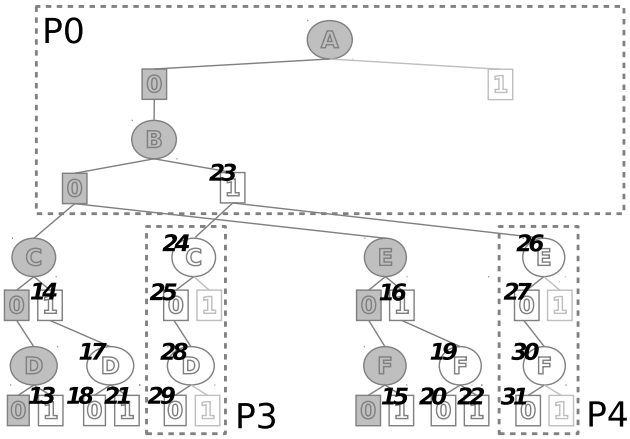
If a subproblem has stopped processing and is not solved yet (i.e., the rotation threshold was reached or further decomposition occurred) it will have a non-empty $LOCAL$ stack that needs to be pushed to the back of the *GLOBAL* queue again (lines 23-34). Finally, when the *GLOBAL* queue is empty, the optimal solution as the value of the root node $\langle X_0 \rangle$ can be returned (line 25).

2.4.3 Example Execution

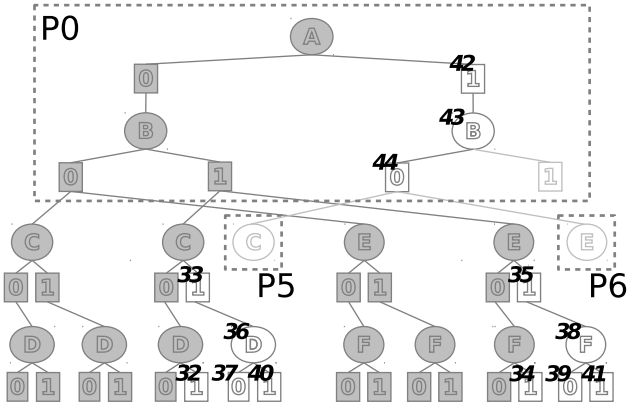
To further illustrate the execution of BRAOBB we revisit the example AND/OR search space from Figure 1.7, reproduced here in Figure 2.3. We demonstrate the application of BRAOBB (with rotation threshold $Z = 2$ and assuming no pruning) in Figure 2.4. Specifically, Figure 2.4a shows the first 12 nodes expanded during the first seven iterations of the outer while loop as follows:



(a) Expansion of nodes 1–12



(b) Expansion of nodes 13–31



(c) Expansion of nodes 32–44

Figure 2.4: BRAOBB exploration ($Z = 2$) at different stages. Nodes are numbered in order of their expansion.

1. Denoting the root subproblem by P0, expand $\langle A \rangle$ and $\langle A, 0 \rangle$ within in before reaching the threshold $Z = 2$. Push P0 to the back of the *GLOBAL* queue.

2. With no decomposition so far rotation returns to P0 (i.e., it is the only subproblem in the *GLOBAL* queue). Expand $\langle B \rangle$ and $\langle B, 0 \rangle$, yielding subproblems P1 and P2 rooted at $\langle C \rangle$ and $\langle E \rangle$, respectively. Since these represent decomposition they are separately added to the back of *GLOBAL* queue (lines 17-20 in Algorithm 2.1) – which is also where P0 is subsequently pushed to.

3. Subproblem P1 is fetched from the front of the *GLOBAL* queue and $\langle C \rangle$ and $\langle C, 0 \rangle$ are expanded within it before the threshold is reached. Since P1 is not solved it is pushed to the *GLOBAL* queue's back.

4. The next subproblem at the front of *GLOBAL* is P2. $\langle E \rangle$ and $\langle E, 0 \rangle$ are expanded within it before reaching the expansion threshold. P2 is pushed to the back of the *GLOBAL* queue since it is not solved yet.

5. Rotate to subproblem P0 from the front of *GLOBAL*; however, it currently has two child subproblems, P1 and P2, so no nodes are expanded within P0 and it is pushed to the back of the *GLOBAL* queue right away.

6. Rotation moves to subproblem P1 from the front of *GLOBAL*. Upon expansion of $\langle D \rangle$ and $\langle D, 0 \rangle$ within P1 a leaf is discovered, which is propagated before P1 gets pushed to the *GLOBAL* queue's back again.

7. Rotate to subproblem P2, expand $\langle F \rangle$ and $\langle F, 0 \rangle$ – which, as a leaf, is propagated. P2 is then pushed to the back of *GLOBAL* again.

Note that at this point a first complete, overall solution can be returned, even though subproblem P1 is not fully solved yet. Contrast this with standard depth-first exploration,

where P1 would have been solved to completion before P2 (and with it a potential overall solution) gets considered at all.

Figures 2.4b and 2.4c illustrate how the search then proceeds to take turns solving subproblems P1 and P2 to completion (nodes 13–22) before "reopening" subproblem P0. Expansion 23 yields two new independent subproblems P3 and P4; their solution is depicted by nodes 24–41. After that subproblem P0 gets reopened, where expanding nodes 42–44 again yields two new subproblems P5 and P6, and so forth.

2.4.4 Analysis of Breadth-Rotating AOBB

In this section we analyze the Breadth-Rotating AOBB algorithm and its properties and contrast it with standard AOBB.

2.4.4.1 Correctness, Completeness, and Complexity

Recall that a heuristic function is said to be admissible if it never underestimates (in a maximization scenario) the cost of the optimal solution to a given subproblem. The mini-bucket heuristic satisfies this requirement [65]. Further recall that n denotes the number of problem variables, k the maximum domain size, h the height of the guiding pseudo tree $\mathcal{T}$ with induced width and w^*.

THEOREM **2.1.** *Breadth-Rotating AOBB is complete and correct assuming an admissible heuristic. Furthermore, when searching an AND/OR search tree (i.e., without caching of redundant subproblems), BRAOBB has time complexity* $O(n \cdot k^h)$ *and space complexity* $O(n)$. *When searching the context-minimal AND/OR search graph (with full caching), time and space complexity are* $O(n \cdot k^{w^*})$.

Proof. Because of the heuristic's admissibility, a subspace is pruned only if it provably cannot yield a better solution than what is already known at this point. Just like standard AOBB the search also remains systematic and all solution trees are considered; the algorithm is guaranteed to eventually terminate and return the optimal solution to the problem.

BRAOBB explores the same underlying AND/OR search space as standard AOBB, hence its asymptotic time complexity remains unchanged, i.e. exponential in h for tree and exponential in w^* for graph search. Space complexity for AND/OR graph search is dominated by the caching and thus also remains unchanged exponential in w^* .

In case of tree search, recall that subproblems with child subproblems are not processed further. Therefore every variable will appear in at most one subproblem at any given time. And since each subproblem is processed depth-first, i.e. in linear space, the space across all subproblems is also linear in n. □

It is worth pointing out that these worst-case bounds are often very loose, because the branch-and-bound scheme is typically very efficient and prunes large parts of the search space. In particular, we observe that in practice the pruning keeps the cache tables from reaching their worst-case exponential size – in none of our experiments (with a 24 hour timeout) did we run into memory issues due to caching. Detailed quantification and analysis pose open questions and are subject to further research.

2.4.4.2 Significance of Z

The rotation threshold Z acts as a safeguard against overly large subproblems, that take a long time to solve optimally (condition (a), Section 2.4.1) or where recursive decomposition does not occur for a long time (condition (b)). Being "stuck" in this way could again impair anytime performance, which is why we limit the number of node expansions before enforcing

a rotation. As we see in Section 2.5, however, practical problems typically exhibit frequent subproblem decomposition along any path in the search space, so a relatively large threshold of $Z = 1000$ or similar is sufficient, if rarely reached.

2.4.4.3 Maximum Queue Size

It is easy to see that the maximum number of entries in the GLOBAL queue is dependent on the number of branchings in the solution tree, corresponding to pseudo tree nodes with more than one successor, since that is where the algorithm generates new child subproblems (lines 17–20, Algorithm 2.1). In particular, decomposition does not occur along the chains in the pseudo tree, i.e., paths where no node (besides the end points) has outdegree greater than 1. The number of queue entries is thus bounded by the number of maximal chains in the pseudo tree. The following is thus fairly straightforward to see:

THEOREM **2.2.** *When exploring an AND/OR search space using a guiding pseudo tree $\mathcal{P}$ with l leaves, the number of subproblems in the GLOBAL queue of BRAOBB is bounded by $2l - 1$.*

Proof. Since $\mathcal{T}$ is a tree with l leaves, there can be at most $l - 1$ branchings (nodes with outdegree greater than 1) in $\mathcal{T}$ to yield these leaves. Each such branching sits at the end of one maximal chain. Together with the leaf chains, we obtain an upper bound of $2l - 1$ maximal chains. □

2.4.4.4 Comparison with Standard AOBB

We expect the anytime performance of BRAOBB to be robust with respect to different subproblem orderings, since the algorithm is not forced to "commit" to a single subproblem

– which we identified as the main reason for the poor anytime behavior of plain AOBB in Section 2.3.1. We will confirm this experimentally in Section 2.5.

The actual number of nodes explored by BRAOBB might differ from plain AOBB (for both graph and tree search), since the pruning behavior of the algorithm can be impacted by the order in which nodes are explored and subproblem solutions produced: On the one hand, solving a subproblem to completion before processing the next (in AOBB) might allow the algorithm to calculate a tighter upper bound using this optimal solution, resulting in better pruning. On the other hand, exploring subproblems concurrently in BRAOBB might lead to a tighter overall lower bound through combining solutions across subproblems as they are discovered (in an anytime fashion). We will revisit this issue in the following experimental section.

2.5 Empirical Evaluation

To validate and compare the performance of the various schemes we recorded their anytime behavior on a variety of problem instances using a common variable ordering and mini-bucket heuristic for each instance (24 hour time limit); unless noted otherwise subproblems were ordered by increasing width (cf. Section 2.3.1). We ran "plain" AOBB, AOBB with the dive extension (cf. Section 2.3.2), and Breadth-Rotating AOBB as presented in Section 2.4; we also included OR branch-and-bound (without problem decomposition) as a baseline. In addition, we ran an advanced stochastic local search (SLS) algorithm [60, 64], both on its own and as a initialization step for our own exhaustive search; in particular we consider the GLS$^+$ implementation from [60] (also cf. Section 1.3.4.2), for which source code is publicly available. Note that as an incomplete search scheme, it does not provide a proof of optimality and always runs for the full 24 hours in our experiments. All algorithms are implemented in C++ and were run on 2.67 GHz Intel Xeon CPUs with 2GB of RAM per core.

2.5.1 Benchmark Instances

Our initial test set (instance name suffix "x1") is comprised of 19 genetic linkage pedigree problems, 50 randomly generated grid networks, 8 mastermind game instances (all part of the UAI 2008 evaluation[1]) as well as 66 protein side-chain prediction problems (taken from [115]). However, several of these instances are relatively simple or have only one complex subproblem, which renders them less interesting for the purpose of this work. Namely, plain AOBB (with subproblems ordered by increasing width) already yields good anytime performance and neither the dive extension nor BRAOBB can provide significant improvements. Hence we also created additional versions of each network with two or three identical copies connected at the root (thus ensuring the presence of more than one complex subproblem), signified by the "x2" and "x3" suffix, respectively. This yields a total of 57 pedigree (each run with three different heuristic strengths), 150 grid, 24 mastermind, and 198 protein prediction instances and resulting in over 90,000 CPU hours worth of experiments.

We show detailed performance results for a representative subset of the problem instances in Section 2.5.2 and in Section 2.5.3 present and discuss summary statistics across all 543 problem instances for OR branch-and-bound, plain AOBB, AOBB with dive extension, and BRAOBB. We briefly touch on BRAOBB's performance for proving optimality in comparison to AOBB in Section 2.5.4. Section 2.5.5 contrasts against the performance of stochastic local search and discusses how our exhaustive search scheme can benefit from it. Section 2.5.6 presents results on two additional very challenging problem classes, protein-protein interaction from the UAI 2010 Challenge and radio link frequency assignments (by the French CELAR agency [2]). Finally, Section 2.5.7 analyzes a number of algorithm parameters empirically.

[1] http://graphmod.ics.uci.edu/uai08/

2.5.2 Detailed Performance Analysis

We begin by showing detailed anytime profiles for a sizable, representative subset of problem instances. Figures 2.5, Figure 2.6, and Figure 2.7 show the results for 10 pedigree linkage, 10 grid network, 10 and protein side-chain prediction instances, respectively, while Figure 2.8 has profiles for four mastermind problems. Each Figure contains results for both the initial problem instances ("x1" suffix) as well as the more complex and thus more interesting ones ("x2" and "x3" suffix). For every problem instance, the plot title specifies number of variables n, maximum domain size k, induced width w along the chosen ordering, and height of the corresponding pseudo tree h. If known, the optimal solution value is indicated by a gray dashed horizontal line. The title of each plot also notes the mini-bucket i-bound; this was typically chosen to fit a 1GB memory limit, except for pedigree instances, where three different heuristic strengths ($i = 7, 10, 15$) were applied for each instance.

2.5.2.1 Linkage Analysis

Figure 2.5 contains results for ten pedigree linkage instances. We first note that OR branch-and-bound does very poorly overall: it only finds an early lower bound in a few of the cases and then provides little improvement over time and never gets close to the optimum. However, since it doesn't exploit subproblem independencies it can in some cases (e.g., pedigree31x2) produce a first solution, albeit a very bad one, earlier than plain AOBB.

As expected, we find that plain AOBB has severely flawed anytime performance in the presence of multiple complex subproblems; it produces a first solution very late or not at all within the time limit (pedigree40x3 and pedigree51x3). The dive extension, largely dependent on the heuristic guidance, is able to alleviate plain AOBB's shortcoming in only three cases (pedigree9x2, pedigree31x2, and pedigree41x3) – on the remaining instances its

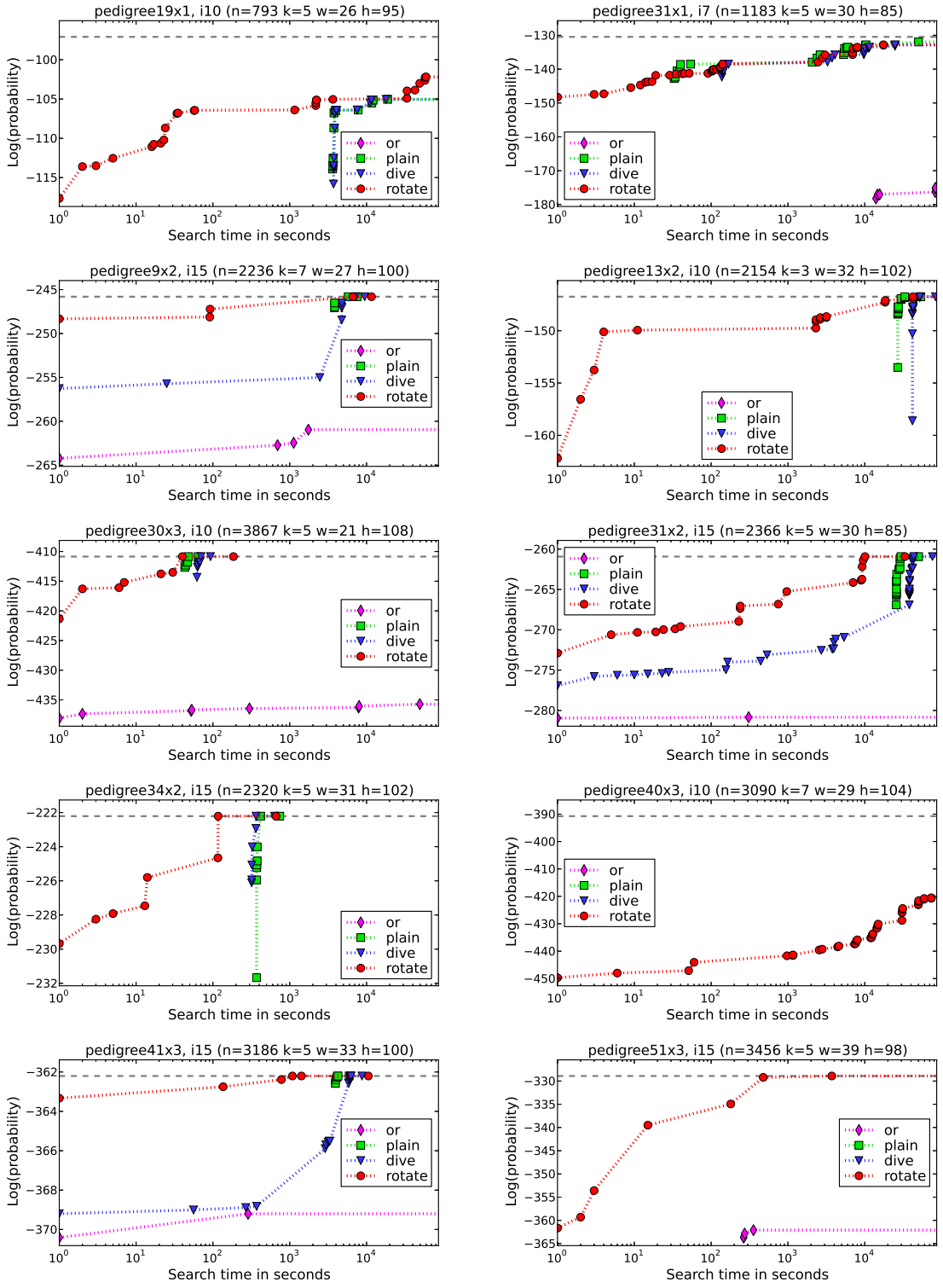


Figure 2.5: Anytime profiles of plain AOBB ("plain"), AOBB with subproblem dive ("dive"), Breadth-Rotating AOBB ("rotate"), and OR branch-and-bound ("or") on selected **pedigree linkage** instances.

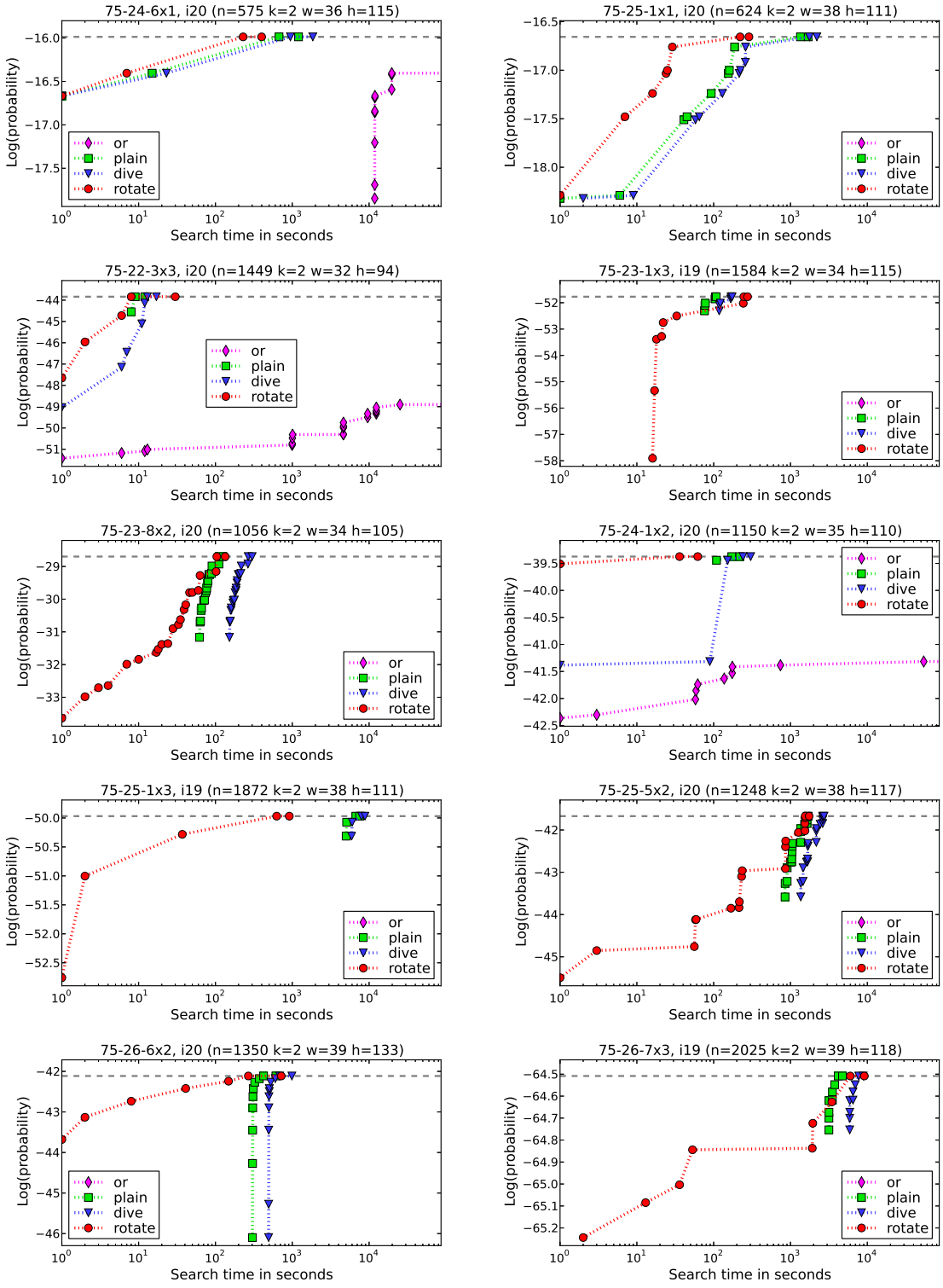


Figure 2.6: Anytime profiles of plain AOBB ("plain"), AOBB with subproblem dive ("dive"), Breadth-Rotating AOBB ("rotate"), and OR branch-and-bound ("or") on selected **grid network** instances.

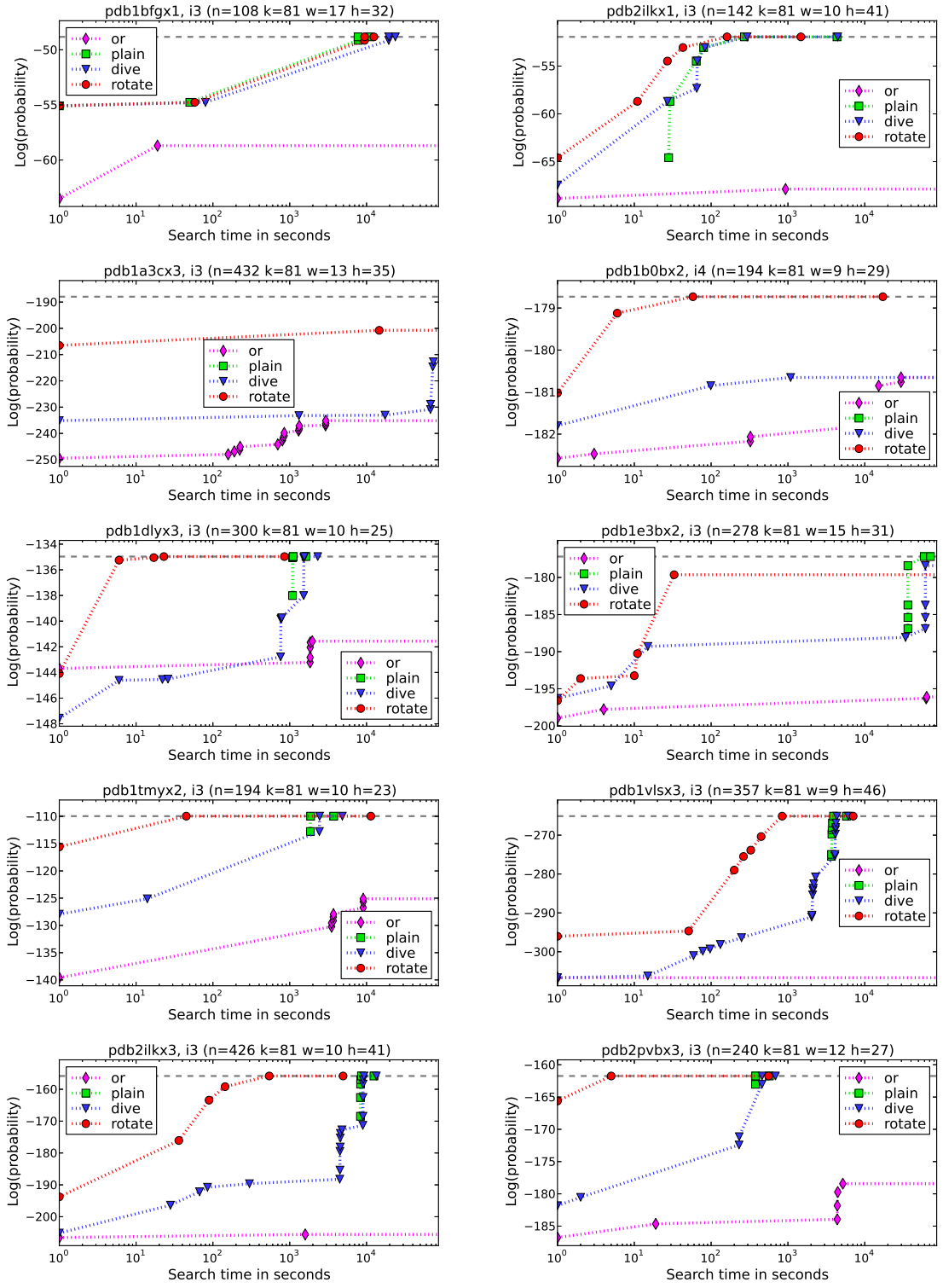


Figure 2.7: Anytime profiles of plain AOBB ("plain"), AOBB with subproblem dive ("dive"), Breadth-Rotating AOBB ("rotate"), and OR branch-and-bound ("or") on selected **pdb side-chain prediction** instances.

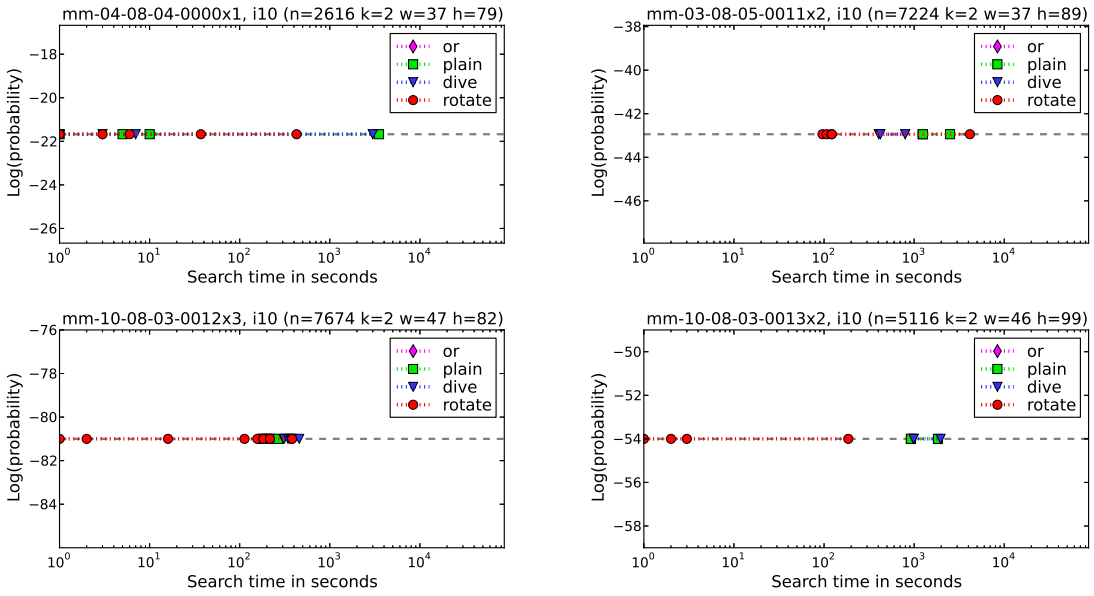


Figure 2.8: Anytime profiles of plain AOBB ("plain"), AOBB with subproblem dive ("dive"), Breadth-Rotating AOBB ("rotate"), and OR branch-and-bound ("or") on selected **mastermind** instances.

performance is identical to plain AOBB (modulo some overhead), demonstrating that this "ad hoc" solution falls short.

BRAOBB, on the other hand, exhibits impressive anytime performance on all instances in Figure 2.5. It always returns a first solution very quickly and continues to improve upon that throughout its execution, even on problems where some or all other schemes fail completely (pedigree40x3 and pedigree51x3).

2.5.2.2 Grid Networks

Anytime profiles for ten grid network instances are plotted in Figure 2.6. Results are very similar to what we found for linkage analysis. OR branch-and-bound fails completely on seven of the instances, with bad performance on the remaining three. Plain actually works well on the two "x1" instances shown (75-24-6x1 and 75-25-1x1) but struggles on the remaining eight "x2" and "x3" ones, due to their multiple complex subproblems. The dive

extension only improves things on two of those cases (75-22-3x3 and 75-24-1x2) but is otherwise equally deficient. Finally, we see again BRAOBB delivering significantly improved anytime performance in all those cases. For instance, on 75-25-1x3 it finds (and proves) the optimal solution in less than half an hour (around 10^3 seconds), while both plain and dive AOBB require over 5000 seconds.

2.5.2.3 Protein Side-chain Prediction

Figure 2.7 shows anytime results on ten protein side-chain interaction instances. These instances are a bit different from pedigree and grid problems in that they exhibit many very small cost values (on the order of 10^{-6}), but have very few actual zeroes in their conditional probability tables. This reduces the likelihood of encountering dead ends in the search space and in turn allows both OR branch-and-bound as well as AOBB with dive extension to reliably return a first solution early, as evident in the plots of Figure 2.7 – note that the solution quality of OR search is generally still lagging.

The performance of plain AOBB, however, is still compromised in the same way as before, by virtue of the multiple complex subproblem, all but one of which will be solved to completion before the first overall solution. Finally, also as before, BRAOBB dominates the anytime performance, often by a large margin both in terms of quality and time of solutions returned.

2.5.2.4 Mastermind

Experimental results for mastermind game instances are shown in Figure 2.8. In contrast to protein side-chain prediction problems, these problems are highly deterministic, with a large number of strictly 0/1-valued conditional probability tables. The resulting anytime profiles are therefore not quite as "interesting" since the first solution is already an optimal one. The sole difference lies in when a given algorithm finds that first solution (and when

optimality is subsequently proven). In all cases, we find that OR branch-and-bound fails to find any solution at all within the time limit; plain AOBB and its dive extension again perform almost identically; finally BRAOBB is always well ahead of the other schemes in finding a solution.

2.5.3 Summary Statistics

Table 2.1 summarizes the entire set of experiments by showing, at different points of time, the number of instances per algorithm for which any solution was found, for which the optimal solution was found, and for which optimality was proven (i.e., when the algorithm terminated). Within each problem class the best value is highlighted in bold for each time stamp. For example, for pedigree networks at 5 seconds (2^{nd} column), OR branch-and-bound, plain AOBB, AOBB with dive, and BRAOBB found solutions for 78, 77, 95, and 161 instances, respectively (first value in each field). Out of those solutions 10, 45, 42, and 50 were optimal ones, respectively (middle value). Finally, optimality was actually proven by the respective algorithm for 8, 41, 36, and 37 instances (last value). The table also contains results for stochastic local search (SLS). At the time stamp of 5 seconds, SLS found a solution for all 171 pedigree instances, but only 21 were actually optimal – and as a local search algorithm, optimality was proven for none. Local search and possible combinations with exact search will be discussed in more detail in Section 2.5.5.

The results in Table 2.1 confirm that BRAOBB yields superior anytime performance: for example, within 1 second it already provides an initial solution on 502 instances overall (out of 543, bottom group), compared to just 232 for plain AOBB, 339 for the dive extension, and 424 for local search; performance remains superior to the other AOBB versions and very competitive with local search for higher time bounds.

	Time bound						
	1 sec	5 sec	10 sec	1 min	5 min	1 hour	24 hours
Pedigree networks (171 total)							
or	77/6/6	78/10/8	82/11/9	84/14/12	87/15/13	91/18/18	94/23/22
plain	65/31/**24**	77/45/**41**	85/**55**/**45**	99/72/**64**	105/80/**75**	113/94/87	136/125/**119**
dive	83/24/17	95/43/36	102/51/43	113/67/61	119/77/72	127/92/87	136/120/113
rotate	**157**/**38**/22	161/**50**/37	162/**55**/44	162/69/59	163/80/71	165/97/87	168/132/112
sls	144/9/0	**171**/21/0	**171**/24/0	**171**/39/0	**171**/66/0	**171**/78/0	**171**/78/0
plain+sls	146/9/0	**171**/13/0	**171**/30/12	**171**/74/62	**171**/84/74	**171**/100/**88**	**171**/129/118
rotate+sls	148/10/0	**171**/13/0	**171**/45/15	**171**/**80**/57	**171**/**93**/72	**171**/**106**/86	**171**/**136**/111
Grid networks (150 total)							
or	45/1/0	47/1/0	51/2/0	55/3/2	62/7/4	67/15/10	78/25/24
plain	47/15/**3**	65/39/**23**	77/52/**40**	94/76/**69**	109/97/89	138/135/**133**	149/**149**/**149**
dive	52/11/1	65/32/13	72/44/27	94/70/64	106/91/84	134/129/123	149/**149**/**149**
rotate	**129**/**24**/1	133/**44**/9	136/**59**/23	140/82/65	143/**107**/**90**	147/**139**/132	149/**149**/**149**
sls	81/0/0	**150**/0/0	**150**/0/0	**150**/2/0	**150**/6/0	**150**/21/0	**150**/21/0
plain+sls	76/0/0	**150**/0/0	**150**/10/1	**150**/79/67	**150**/97/88	**150**/135/132	**150**/**149**/**149**
rotate+sls	83/0/0	**150**/0/0	**150**/14/0	**150**/**83**/64	**150**/106/**90**	**150**/**139**/131	**150**/**149**/**149**
Protein side-chain prediction networks (198 total)							
or	**198**/78/49	**198**/79/52	**198**/80/53	**198**/82/57	**198**/89/61	**198**/90/70	**198**/99/82
plain	114/95/78	120/102/**85**	124/106/**87**	133/117/102	145/132/116	168/163/148	191/188/186
dive	**198**/102/76	**198**/108/84	**198**/110/86	**198**/122/100	**198**/133/112	**198**/161/141	**198**/185/181
rotate	**198**/128/**79**	**198**/133/**85**	**198**/136/86	**198**/151/**104**	**198**/165/**120**	**198**/180/**157**	**198**/190/**190**
sls	**198**/**193**/0	**198**/**198**/0	**198**/**198**/0	**198**/**198**/0	**198**/**198**/0	**198**/**198**/0	**198**/**198**/0
plain+sls	**198**/**193**/0	**198**/**198**/0	**198**/**198**/51	**198**/**198**/81	**198**/**198**/98	**198**/**198**/132	**198**/**198**/169
rotate+sls	**198**/191/0	**198**/**198**/0	**198**/**198**/47	**198**/**198**/83	**198**/**198**/104	**198**/**198**/140	**198**/**198**/172
Mastermind networks (24 total)							
or	0/0/**0**	0/0/**0**	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
plain	6/6/**0**	7/7/**0**	7/7/0	9/9/**3**	12/12/6	20/20/18	**24**/**24**/**24**
dive	6/6/**0**	7/7/**0**	8/8/**1**	10/10/**3**	12/12/6	23/23/**22**	**24**/**24**/**24**
rotate	**18**/**16**/**0**	**18**/**16**/**0**	18/16/0	19/**19**/**3**	**24**/**24**/**10**	**24**/**24**/21	**24**/**24**/**24**
sls	1/1/**0**	**18**/14/**0**	**24**/**18**/0	**24**/18/0	**24**/19/0	**24**/19/0	**24**/19/0
plain+sls	1/1/**0**	**18**/13/**0**	**24**/13/0	**24**/14/**3**	**24**/16/6	**24**/22/18	**24**/**24**/**24**
rotate+sls	2/2/**0**	**18**/13/**0**	**24**/13/0	**24**/17/**3**	**24**/**24**/9	**24**/**24**/21	**24**/**24**/**24**
Overall (543 total)							
or	320/85/55	323/90/60	331/93/62	337/99/71	347/111/78	356/123/98	370/147/128
plain	232/147/**105**	269/193/**149**	293/220/**172**	335/274/**238**	371/321/286	439/412/386	500/486/**478**
dive	339/143/94	365/190/133	380/213/157	415/269/228	435/313/274	482/405/373	507/478/467
rotate	**502**/**206**/102	510/**243**/131	514/266/153	519/321/231	528/376/**291**	534/440/**397**	539/495/475
sls	424/203/0	**537**/233/0	**543**/240/0	**543**/257/0	**543**/289/0	**543**/316/0	**543**/316/0
plain+sls	421/203/0	**537**/224/0	**543**/251/64	**543**/365/213	**543**/395/266	**543**/455/370	**543**/500/460
rotate+sls	431/203/0	**537**/224/0	**543**/**270**/62	**543**/**378**/207	**543**/**421**/275	**543**/**467**/378	**543**/**507**/456

Table 2.1: Summary statistics over 543 instances for OR branch-and-bound, plain AOBB, AOBB with dive extension, breadth-rotating AOBB, stochastic local search, as well as plain and breadth-rotating AOBB with 10 seconds of initial local search. In each case we list the number of cases for which, within the respective time bound, (1) any solution was found, (2) the optimal solution was found, (3) optimality was proven.

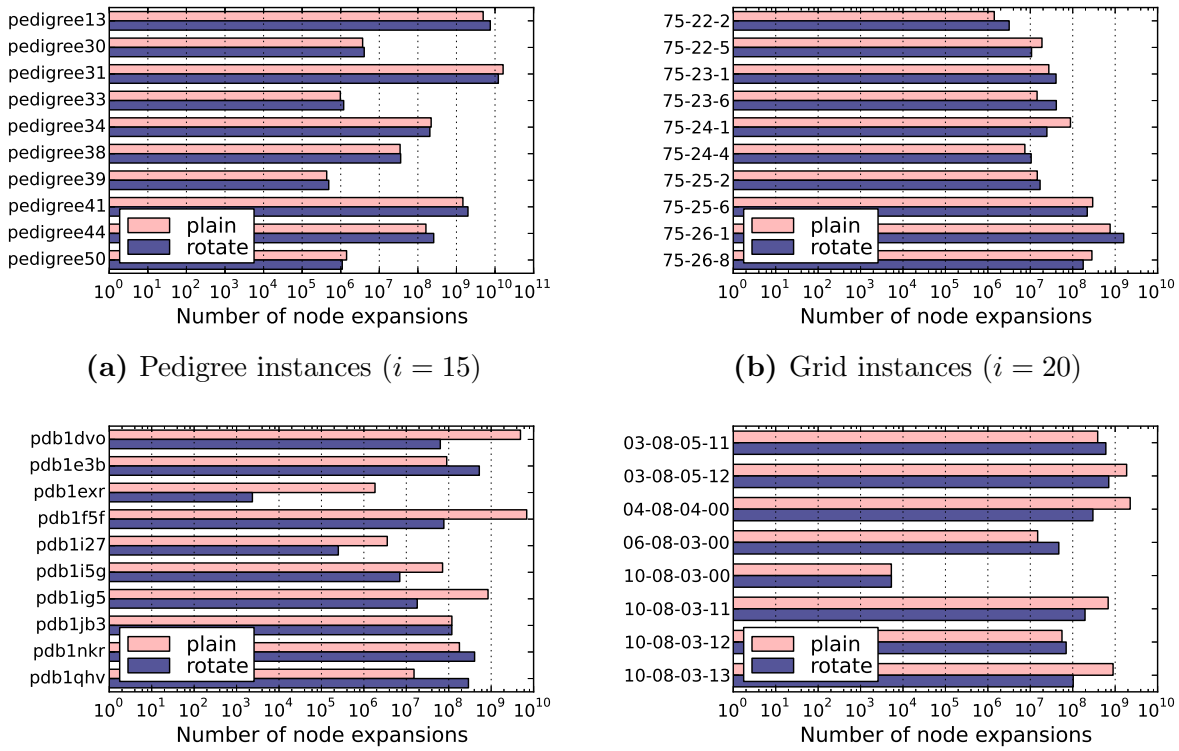
It is important to note that BRAOBB finds the optimal solution quicker than the other schemes (with the exception of local search on side-chain prediction), for example for overall 266 instances after 10 seconds (versus 220 for plain). Similarly, in the full 24 hours, BRAOBB found the optimal solution to 495 instances, versus 486 for plain and just 316 for local search.

We observe that SLS does very well on the side-chain prediction networks – these problems have only a few hundred variables but a large maximum domain size of 81. On the other problem classes, however, with thousands of variables and smaller maximum domains, BRAOBB shows better performance, in particular with respect to finding the optimal solution. The reason for this lies in the heuristic used by AOBB: mini-bucket space complexity is $O(nk^i)$ – large domain size bounds k thus necessitate a significantly lower i-bound ($i = 3$ in case of the side-chain prediction problems), which leads to far less accurate heuristics. Recall that branch-and-bound uses the mini-bucket heuristic both for pruning as well as for value ordering upon variable instantiation (cf. Section 1.3.3) – SLS, on the other hand, does not depend on this kind of heuristic.

2.5.4 Proving Optimality

While not the focus of the present work, we consider for a moment the algorithm's performance in terms of proving optimality, in particular comparison to plain AOBB. Section 2.4.4 stated that different exploration strategies influence the level of pruning the algorithm can apply, since it will impact the availability of subproblem solutions for bounding purposes.

With that in mind, Table 2.1 shows that plain AOBB seems to have a very slight edge in terms of proving optimality. For instance, it proved optimality for 149 instances overall at 5 seconds versus 131 for BRAOBB or 172 versus 153 instances at 10 seconds. AOBB is still a bit ahead after 1 minute, falls behind somewhat at 5 minutes and 1 hour, but ends up



(a) Pedigree instances ($i = 15$)

(b) Grid instances ($i = 20$)

(c) Side-chain prediction instances ($i = 3$)

(d) Mastermind instances ($i = 10$)

Figure 2.9: Comparison of number of node expansions needed by plain AOBB and BRAOBB to prove optimality for a subset of problem instances from each class (i-bound specified per class).

proving optimality for three more instances than BRAOBB at the 24 hour timeout mark. We note again that, as an incomplete solver, local search proves no optimality at all.

To provide a more detailed perspective, Figure 2.9 compares the number of node expansions that plain AOBB and BRAOBB require to prove optimality of a solution for a number of problem instances (note the horizontal log scale). The results confirm our analysis of Table 2.1: AOBB has a very slight edge overall, but we see individual cases going in both AOBB and BRAOBB's favor. Notably, results are fairly close for pedigree, grid, and mastermind instances, while side-chain prediction problems exhibit a difference of up to two orders of magnitude in some cases. We suspect this might be related to the relatively weak heuristic in

the case of side-chain prediction problems ($i = 3$ is maximum possible), but the full analysis is subject to future research.

2.5.5 Combining Local and Exhaustive Search

Looking again at Table 2.1, we notice that SLS can sometimes find solutions more quickly than any kind of exhaustive search – in particular it has found a solution for all 543 problems after 10 seconds (versus 514 instances for BRAOBB and just 293 for plain AOBB). As outlined above, however, SLS is often quickly outperformed by AOBB and BRAOBB in terms of finding the optimal solution, let alone proving optimality (which is impossible with local search). For instance, local search has found only 257 optimal solutions after 1 minute (versus 321 for BRAOBB and 274 for plain AOBB) or 316 at the 24 hour timeout (compared to 495 for BRAOBB, 486 for plain AOBB).

We have thus devised simple, combined schemes that run local search for 10 seconds as a preprocessing step; the resulting solution is then used as an initial lower bound for the exhaustive search. Results for plain AOBB and BRAOBB augmented in this way, denoted "plain+sls" and "rotate+sls," respectively, are included in Table 2.1. Figure 2.10 also shows detailed anytime profiles on ten problem instances, comparing local and exhaustive search, as well as their combinations.

Indeed we see "plain+sls" and "rotate+sls" match local search in terms of initial performance and quickly returning a solution (deviations here are due to randomization). Just as for SLS, however, initial solution quality can be inferior to BRAOBB (cf. instances pedigree41x3 and 75-25-1x3, for instance), but after 10 seconds the combined schemes quickly catch up to plain AOBB and BRAOBB, respectively, as local search preprocessing finishes and exhaustive search takes over. Here "rotate+sls" has the edge over "plain+sls" in terms of getting to

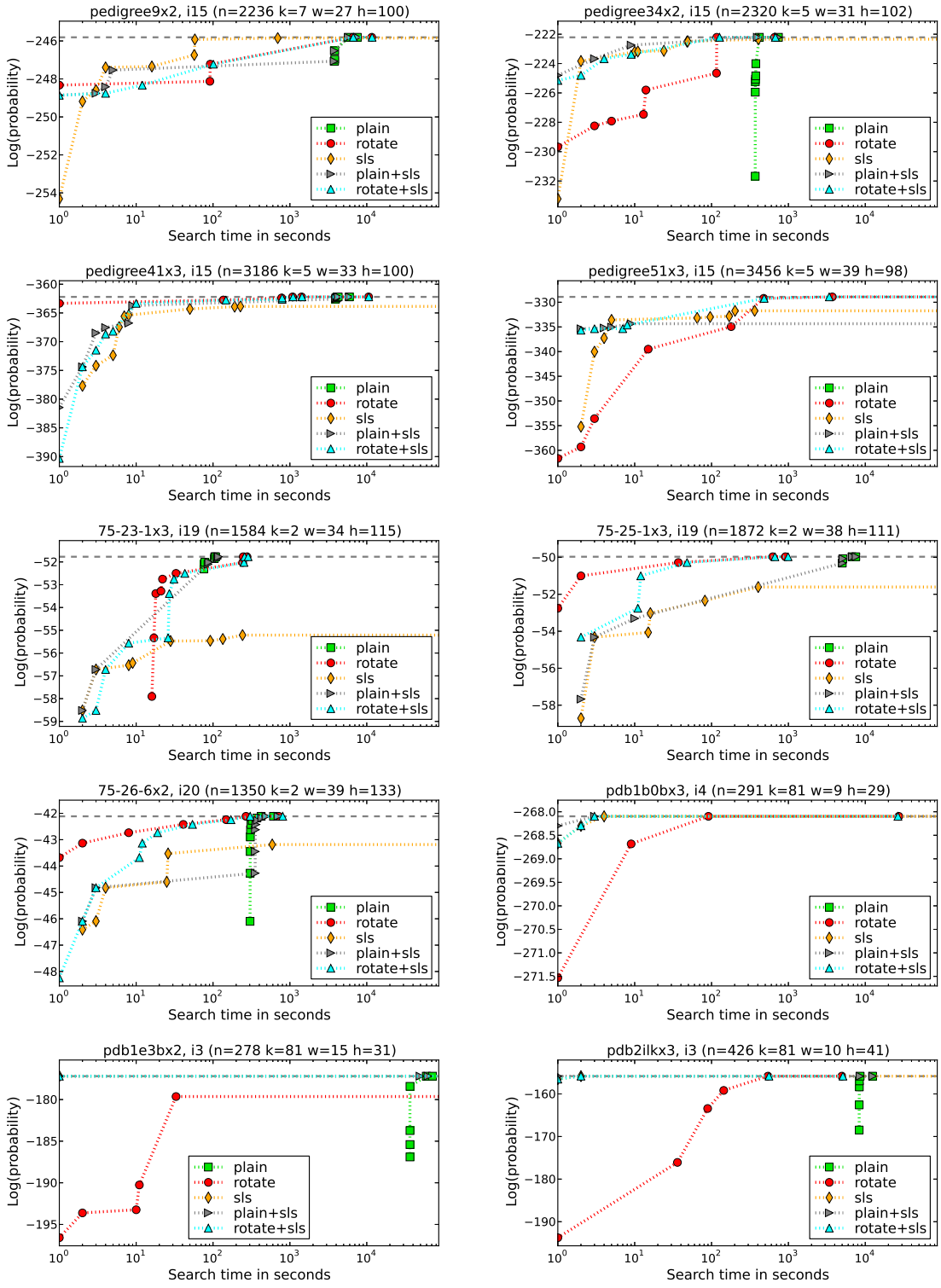


Figure 2.10: Anytime profiles comparing exhaustive AOBB against SLS and combinations of the two on select problem instances.

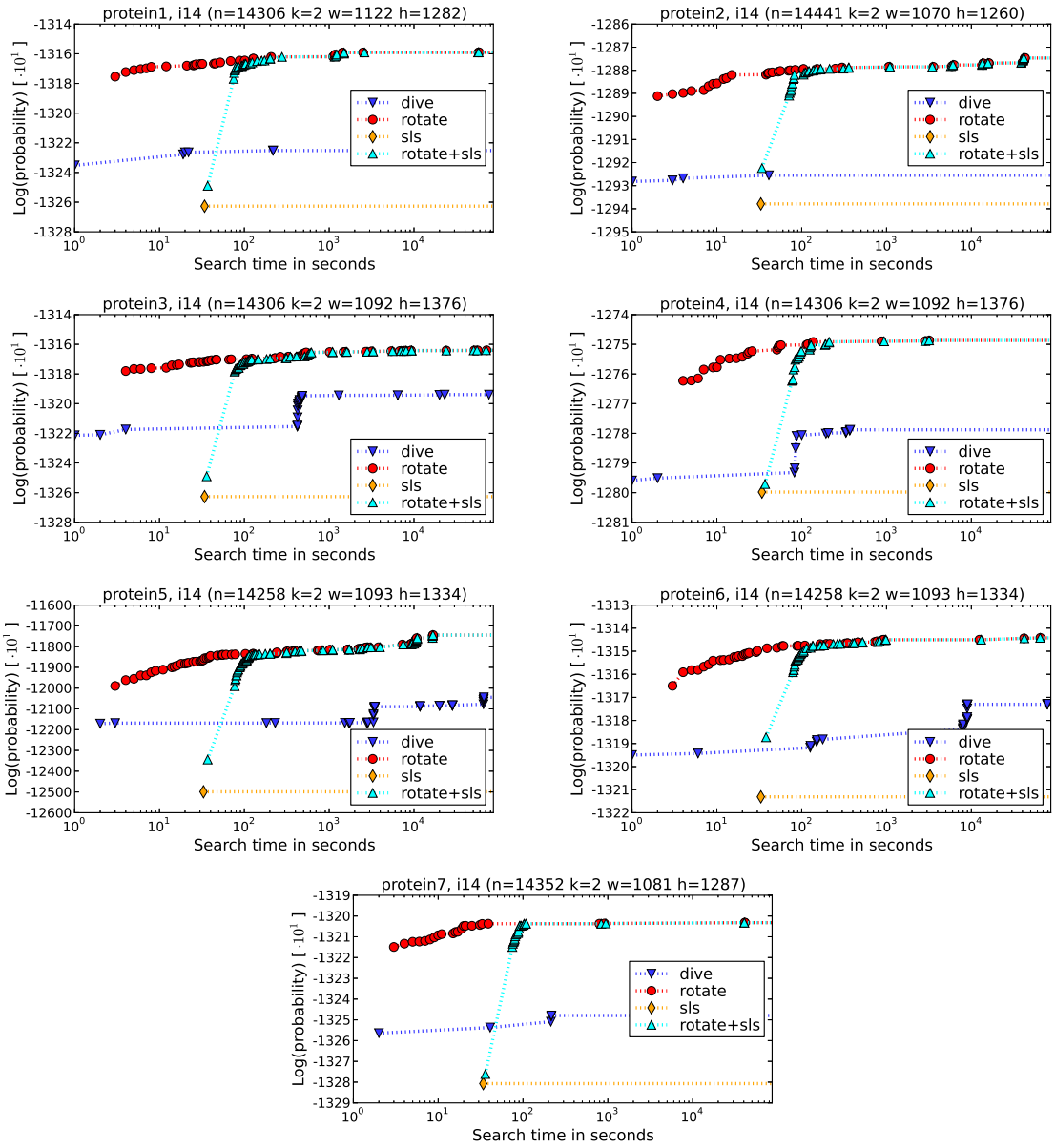


Figure 2.11: Anytime profiles on seven very hard protein-protein interaction instances from the UAI'10 and PASCAL'11 Inference Challenges.

and proving optimality. Overall we therefore believe that "rotate+sls" best combines the benefits of the two search paradigms.

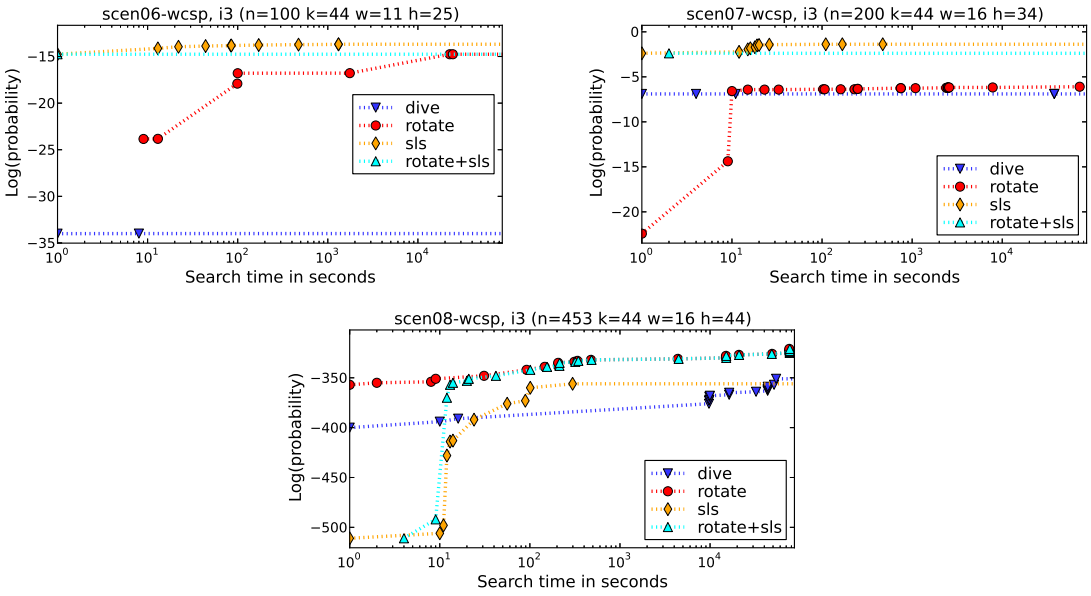


Figure 2.12: Anytime profiles on three very hard WCSP instances of CELAR radio link frequency assignment problems, encoded as MPE.

2.5.6 Additional Problem Classes

In addition to the benchmark set used in the previous sections, we consulted two additional, very challenging classes of problems protein-protein interaction instances (seven problems were made available from the UAI 2010 Challenge [35]) and three CELAR radio link frequency assignment instances converted from weighted constraint satisfaction problems (e.g., [2]), for all of which optimal solutions are unavailable. OR branch-and-bound and plain AOBB failed to produce a solution within the 24 hour time limit on any of these problems and have thus been omitted here.

Figure 2.11 shows anytime profiles for all seven protein-protein interaction networks; in all cases BRAOBB and even the initial subproblem dive restore the anytime performance that plain AOBB is lacking. We note, however, that as before the solution quality of AOBB with initial dive is decidedly inferior. BRAOBB and AOBB with dive also outperform SLS (BRAOBB drastically so); it appears to struggle with the large number of over 14,000 problem variables and never improves upon its initial solution – it is unclear whether this

an issue with the specific implementation we have available (GLS$^+$ from [60]) or whether it is an inherent issue with local search. "rotate+sls" suffers from this as well during its local search preprocessing, but quickly catches up to BRAOBB, as seen in previous sections.

CELAR networks in Figure 2.12, on the other hand, have fewer variables and large domain sizes (maximum $k = 44$). As it was the case for protein side-chain prediction problems, this results in a weaker mini-bucket heuristic (with a low i-bound of 3) for AOBB. This again gives an advantage to SLS, which does not rely on this kind of heuristic and is able to outperform BRAOBB in two of the three cases, scen06-wcsp and scen07-wcsp. However, in both of these two cases we see that SLS improves its solution only little with time. The combined "rotate+sls," which matches the initial performance of SLS but not these later improvements, is thus still a good compromise, since it also yields leading performance on the third instance scen08-wcsp.

2.5.7 BRAOBB Analysis

In the following we investigate several aspects of BRAOBB more closely and compare some of its properties to plain AOBB empirically.

2.5.7.1 Heuristic Accuracy

We've argued above how the performance of BRAOBB can suffer if the heuristic is very inaccurate, i.e., the i-bound of the mini-buckets is low. This became specifically evident when comparing against local search on instances with large domain sizes (protein side-chain prediction and CELAR radio link problems).

Here we compare the different AOBB schemes with respect to their sensitivity for the heuristic's accuracy. Figure 2.13 contrasts plain AOBB, dive, and BRAOBB each with two different

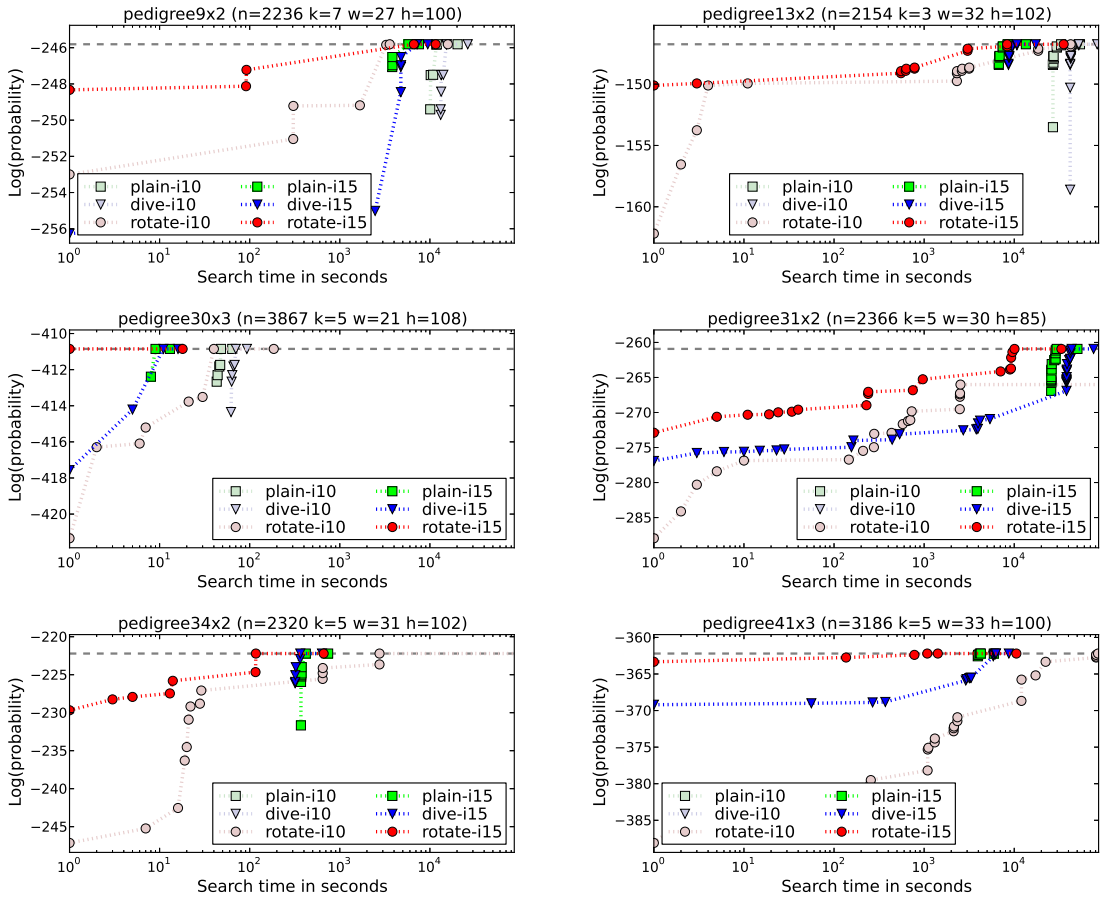


Figure 2.13: Impact of heuristic accuracy on anytime performance: comparing i-bound 10 and 15 on two pedigree instances.

heuristics, parametrized by the mini-bucket i-bound. In all cases plain AOBB fails or does poorly due to problem decomposition (with an advantage, however, for the stronger heuristic). AOBB with dive depends very much on the heuristic and fails or does poorly with the weaker one; on pedigree31x2, for instance, it is unable to produce a solution within 24 hours using $i = 10$, but exhibits acceptable anytime behavior with the stronger $i = 15$.

In contrast, BRAOBB appears to be significantly more robust with regards to the heuristic strength and exhibits acceptable anytime profiles even with $i = 10$. Nevertheless, it still profits from a stronger heuristic and further improves its performance with $i = 15$, in most cases significantly.

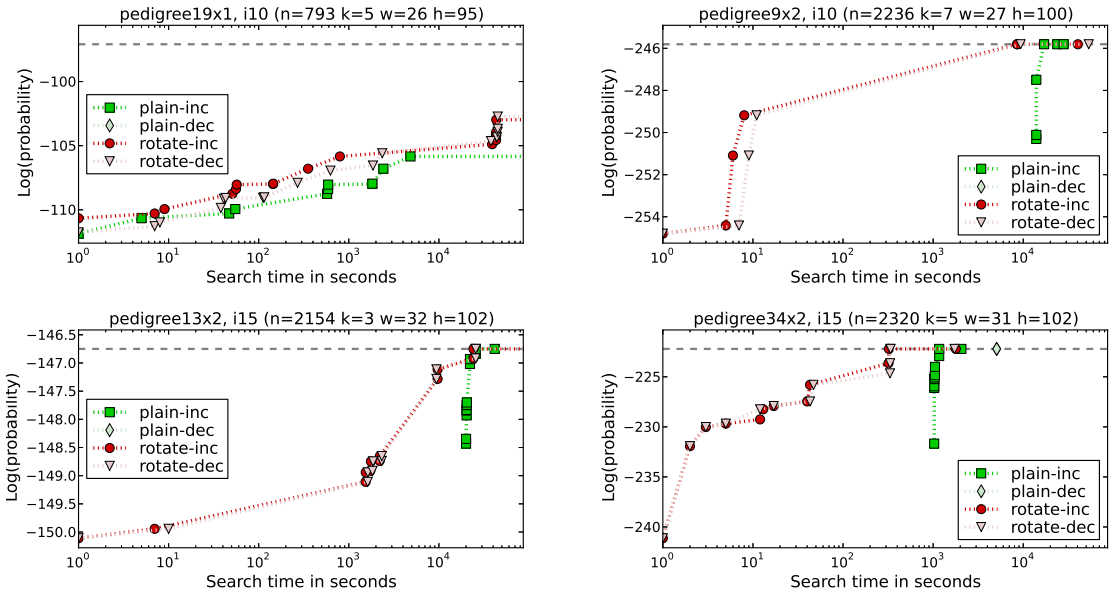


Figure 2.14: Impact of subproblem ordering on anytime performance: subproblems ordered by increasing ("-inc" suffix in plot) and decreasing ("dec") induced width for both plain AOBB and BRAOBB.

2.5.7.2 Subproblem Ordering

Going back to Section 2.3.1, Figure 2.14 compares the performance of BRAOBB with subproblems ordered by increasing and decreasing width. For reference, we also include plain AOBB; as shown earlier, on most instances it fails to produce any solution within the given time limit with subproblems ordered by decreasing width. Ordered by increasing width the only favorable case for plain AOBB is when there is only one complex subproblem (exemplified by pedigree19x1 in Figure 2.14). In contrast our new scheme BRAOBB is very robust and delivers nearly the same performance regardless of subproblem ordering in all cases.

2.5.7.3 Rotation Threshold

Finally, we conducted experiments with different values for the rotation threshold Z in Algorithm 2.1, ranging from 10 to 10,000,000 node expansions. Figure 2.15 shows ex-

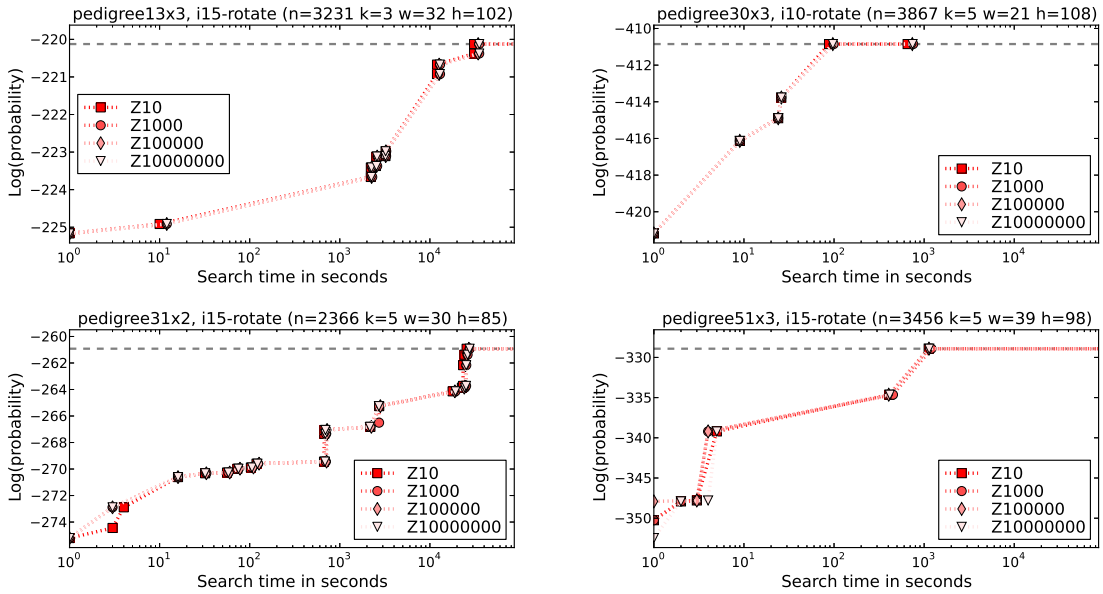


Figure 2.15: Impact of rotation threshold Z: anytime profiles of BRAOBB run with $Z \in \{10, 1000, 100000, 10000000\}$.

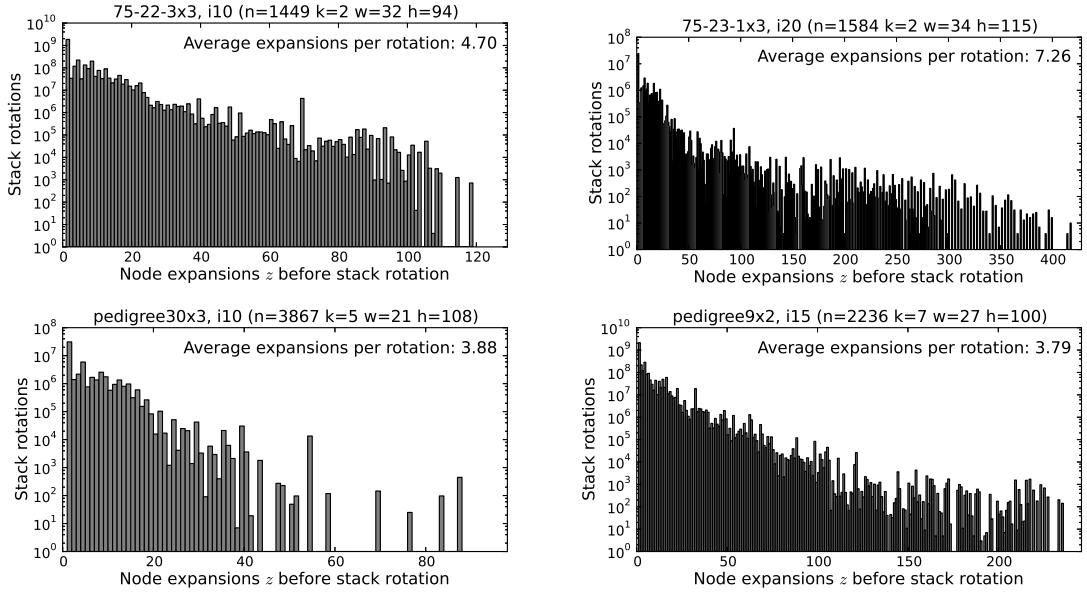


Figure 2.16: Histograms showing number of node expansions between stack rotations for four different runs of BRAOBB (note the vertical log scale). The rotation threshold was set to $Z = 1000$ in each case, but evidently never reached in practice.

emplary results on four pedigree instances, plotting solution quality over time for $Z \in \{10, 1000, 100000, 10000000\}$. We see that performance is virtually identical in each case.

To explain this behavior, we conducted a number of runs of BRAOBB in which we recorded the number of node expansions between stack rotations (i.e., the maximum values of the z counter in Algorithm 2.1). Representative results are shown in Figure 2.16 in the form of histograms: noting the vertical log scale, we observe that the majority of stack rotations happens after only very few node expansions (on the order of at most a few hundred), further confirming the analysis in Section 2.4.4. In particular, we note that in the cases shown in Figure 2.16 the expansion threshold $Z = 1000$, as one of three conditions for subproblem rotation, is never actually met in practice.

2.6 Conclusion to Chapter 2

Exploiting problem decomposition in search methods has been proven to yield significantly better overall complexity in many cases. Yet this chapter has demonstrated how it can be in direct conflict with the depth-first nature of branch-and-bound, thus impairing the important anytime properties of this class of algorithms. Specifically, to obtain an overall result, a partial solution is required from every independent subproblem, which we have shown to be in direct contradiction to the depth-first, consecutive processing of subproblems.

As an "ad hoc" fix, we argued how this effect can be avoided when only one of the decomposed subproblems is relatively hard, in which case the simple ones should be processed first. Using AND/OR Branch-and-Bound, the effectiveness of this approach was shown experimentally, but we validated its obvious limitations as well. We devised a "quick fix" that employs an initial greedy subproblem dive, but whose performance we found to be lacking due to heavy dependence on the underlying heuristic.

The main contribution of this work is the new scheme *Breadth-Rotating AND/OR Branch-and-Bound (BRAOBB)*, which periodically iterates over the different subproblems in a "breadth-first" manner. Yet we have shown that it retains certain desirable properties of the depth-first strategy. In particular, not accounting for caching the number of nodes BRAOBB needs to keep in memory still only grows linearly in the number of variables or the depth of the search space – in contrast to best-first search schemes that are inherently exponential.

We presented an exhaustive set of successful experiments on problems from several different problem classes, including a number of instances that are too hard to solve exactly. The results confirmed the vastly improved anytime performance of BRAOBB, especially in cases where standard depth-first branch-and-bound and its "ad hoc" extensions fail. We also showed BRAOBB to be very competitive with a state-of-the-art stochastic local search algorithm, in many cases even surpassing it (unless the mini-bucket heuristic is very inaccurate). In addition, we demonstrated how the two paradigms can be combined to get the best of both worlds.

2.6.1 Winning the PASCAL 2011 Inference Challenge

The power of our enhanced algorithm was recently further demonstrated when competing in the PASCAL 2011 Inference Challenge [36]. In particular, we submitted an entry based on BRAOBB with initial stochastic local search, as described in Section 2.5.5, combined with the following enhancements:

1. We re-parametrize the problem using a MPLP procedure, a message passing algorithm that iteratively shifts costs between functions or cluster of functions by solving a linear programming relaxation of the localized problem [61]. This is first applied on the original problem graph and subsequently on a higher-order join graph. In each case an

efficient implementation (courtesy of Alexander Ihler) typically allows us to perform several thousand iterations.

2. When computing the mini-bucket heuristic, we apply a single pass of MPLP as described in the previous point.

3. We use a highly efficient implementation (courtesy of Kalev Kask) of the greedy variable ordering schemes *min-fill* and *min-degree*, with optimized data structures, randomization through pooling, and early termination of unpromising iterations [67]. This allows us to run tens of thousands of variable ordering computations in many cases, often yielding orderings with lower induced width and better asymptotic complexity.

4. We perform an initial run of Limited Discrepancy Search with discrepancy limit 2, a systematic but incomplete search scheme described in Section 1.3.4.1. For most problems this step finishes in less than a second.

This combined solver placed first in all three categories (20 second, 20 minute, and 1 hour time limit, respectively) of the MPE track of the PASCAL 2011 Inference Challenge [36]; it was invited for presentation at the UAI conference in August 2012 on Catalina Island, CA.

2.6.2 Open Questions

Possible future research directions include more elaborate rotation schemes, for instance assigning the rotation threshold dynamically based on subproblem-specific heuristic estimates. Given the observations in Section 2.5.7.3, however, it is unclear whether these would have a noticeable impact in practice.

Secondly, while the focus in this work has been on anytime performance and using AND/OR search for approximate inference, the ideas presented might be of value in improving

exact reasoning as well. Specifically, Sections 2.4.4 and 2.5.4 touched upon how varying the schedule of exploration, both in terms of value ordering and subproblem ordering, can impact the pruning of the algorithm. Developing a better understanding of the factors that determine this interdependence would enable us to use it in our favor, with the intention of shortening the time it takes to prove an optimal solution.

In this context, it should also be worthwhile to investigate general complexity issues, such as the dependence on the guiding pseudo tree and the underlying variable ordering. Two key parameters here are the pseudo tree height and induced width, which play a large part in bounding the asymptotic complexity of AND/OR tree and graph search, respectively. The interplay between these two in determining the actual runtime complexity of AOBB in general and its anytime performance in particular constitutes an open research question.

Chapter 3

Complexity Prediction of AND/OR Branch-and-Bound

3.1 Introduction

This chapter investigates the runtime complexity of AND/OR Branch-and-Bound. In particular, we aim to estimate the run time of the algorithm for a given problem instance and set of parameters, most importantly the variable ordering and mini-bucket heuristic i-bound.

The hardness of a graphical model problem is commonly judged by its structural parameters, based on the asymptotic complexity bound of the algorithm in question. In the case of AOBB this is $O(n \cdot k^w)$, i.e., exponential in the problem's induced width w along a given variable ordering (cf. Section 3.2). Due to the asymptotic nature of this bound, however, this intuition does usually not allow us to get a very precise idea of a particular problem's solution time, be it to determine feasibility in light of limited resources, or to choose among a set of variable orderings, possibly with the same induced width, or other algorithm parameters. For this we need a better understanding of problem complexity and algorithm runtime, a

notion sometimes also referred to as the "empirical hardness" of a problem instance (see for instance [78, 103]).

A challenging and interesting problem in itself, estimating the algorithm's performance on a given problem is of particular importance in the context of parallelizing AOBB. Here we aim to find a set of subproblems to be processed in parallel, with runtimes as close to each other as possible; the underlying objective is to ensure load balancing and maximize resource utilization, as will be discussed in-depth in Chapter 4. In this context the scope of the estimation problem widens to arbitrary subproblems within an overall problem search space.

Not limiting ourselves to a problem instance and corresponding search space in its entirety, but instead considering conditioned subproblems and subspaces also gives a much larger set of examples to work with and will therefore form the basis for the experimental evaluation in this chapter.

3.1.1 Contributions

The contributions of this chapter can be summarized as follows:

- We revisit the asymptotic complexity bound of AOBB and derive a finer-grained version, the so-called state space bound. However, we demonstrate that this bound is commonly still very loose in practice, indicating that structural parameters alone are not sufficient to determine problem hardness.
- In contrast to most existing work on estimation of search complexity based on sampling, we propose to take a learning approach and derive a general class of models for the complexity of AOBB. Motivated by the exponential nature of search spaces, we formulate the number of expanded nodes as exponential in a linear combination of

a collection of subproblem features. Under a log transformation this approach corresponds to the well-studied problem of linear regression.

- We lay out a set of 35 subproblem features as the basis for this regression model. Notably, besides structural properties like the induced width and domain size, these features also include more dynamic attributes, e.g., subproblem upper and lower bounds maintained by AOBB and pruning ratios extracted from a very small search sample. All features are generic to AOBB and not specific to a particular problem class.

- We outline four levels of learning and analyze their applicability and relevance in practice: in order of increasing generality, we consider learning complexity models for subproblems of a single problem instance, across several instances from a single class, and across instances from several known classes. In addition, we investigate the possibility of estimating complexities of instances from an unseen problem class, which can be seen as a form of *transfer learning* [93].

- Experimental evaluation is conducted on instances from four different problem classes, according to the levels of generality defined previously, with overall positive results. Namely, all but the most general level of transfer learning yield complexity model instances that produce good to very good prediction performance.

- We analyze the informativeness of the different features via their contribution to the learned model instances. In doing so we are able to demonstrate that indeed dynamic features such as subproblem cost bounds are the most meaningful and crucial to assessing the hardness of a subproblem instance.

3.1.2 Chapter Outline

Section 3.2 provides some necessary background for the work presented in this chapter, including the finer-grained asymptotic bound (Section 3.2.1) and an overview of related work (Section 3.2.2).

In Section 3.3 we present our main contribution, a general approach to complexity estimation that builds on statistical regression learning. In particular, Section 3.3.2 introduces the linear regression model over 35 specific problem features identified in Section 3.3.3. We delineate four different levels of learning for our evaluation and list the specific regression algorithms we considered in Sections 3.3.4 and 3.3.5, respectively.

Section 3.4 describes how we learn our models in practice and presents in-depth empirical evaluation. After describing the set of underlying problem instances and classes in Section 3.4.1 as well as the general experimental setup in Section 3.4.2, Sections 3.4.3 through 3.4.6 give an in-depth report of the experimental results. Sections 3.4.8 and 3.4.7 provide further analysis of the experimental results. Section 3.5 concludes the chapter.

3.2 Background & Related Work

This section lays out the foundations for the contributions in this chapter. It revisits the asymptotic upper bound and develops a more fine-grained version. Experimental results, however, show that this bound is still very loose in practice. We then report on related work and its limitations in our specific context.

To decouple the analysis of runtime complexity from the specific computing power of the CPUs in our experimental environment, we instead measure performance in number of nodes expanded by AOBB, i.e., we consider the size of the *explored search space*. For simplicity,

we will count the number of AND nodes in particular (it's easy to see that this dominates the number of OR nodes).

This metric scales linearly with the run time of the search process (i.e., not accounting for preprocessing times like mini-bucket computation) and is independent of computing power.

3.2.1 A Finer-grained Asymptotic Bound

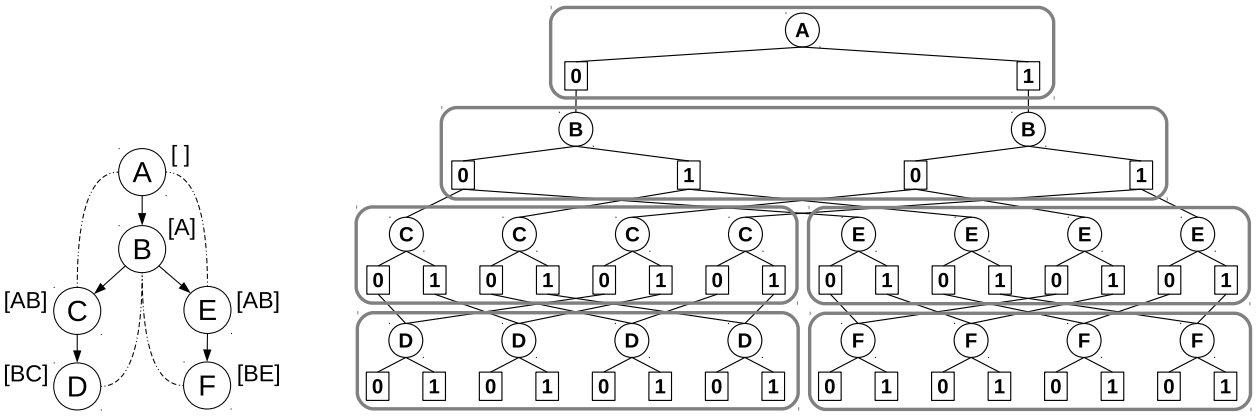
Recall from Section 1.2 and [29] the asymptotic upper bound on the size of the context-minimal AND/OR search graph, $O(n \cdot k^{w+1})$, where n is the number of problem variables, k the maximum domain size across all variables, and w the induced width of the problem along a given variable ordering.

Indeed $n \cdot k^{w+1}$ upper bounds the number of AND nodes, but because of its asymptotic nature the bound is typically very loose. In particular, this is due to the following two implicit simplifying assumptions:

1. Every variable has the same, maximum domain size k,
2. The context size of each variable is the same, maximum w.

A finer-grained upper bound on the size of the context-minimal AND/OR search graph can be obtained by computing the maximum possible *state space* (similar to [69]). To derive this measure, we partition the search space into "clusters," one for each variable's contribution. In other words, we group together search nodes that correspond to the same variable in the graphical model.

Figure 3.1 illustrates this on the example problem from Figure 1.3 by grouping nodes into six clusters, one for each problem variable.



(a) Pseudo tree. **(b)** AND/OR search graph with variable clusters.

Figure 3.1: Example AND/OR search graph from Figure 1.7, for problem in Figure 1.3, with variable clusters marked.

To formalize, denote with $C(X_i)$ the context of variable X_i in the given AND/OR search space and recall that D_i is the variable domain of X_i. The maximum state space size, SS, can then be expressed as follows:

$$SS \;=\; \sum_{i=1}^{n} \; |D_i| \cdot \prod_{X_j \in C(X_i)} |D_j| \tag{3.1}$$

Namely, compute for each problem variable X_i the product of its domain size and the domain sizes of the variables $X_j \in C(X_i)$ in its context. By construction, this bounds the number of AND nodes that X_i can contribute to the context-minimal AND/OR search graph.

For AOBB, however, these state space bounds still tend to be very loose, since they don't account for two important algorithmic aspects:

- If AOBB encounters inconsistencies (such as zero probabilities in a Bayesian network), it can discard the subspace below the current node and backtrack. In fact, this exploitation of determinism in the problem specification is relevant not only in the context of AOBB, but also for general search procedures in a non-optimization setting, such as likelihood computation or solution counting.

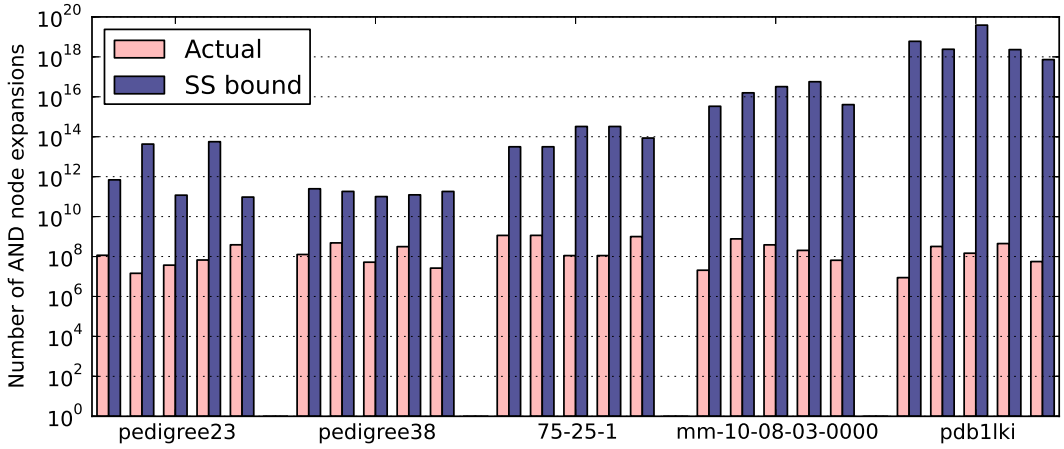


Figure 3.2: Number of actual AND node expansions performed by AOBB vs. state space bound for five problem instances, each run with five different variable orderings (2 pedigree, 1 grid, 1 mastermind, 1 protein side-chain prediction).

- As a branch-and-bound scheme, AOBB can often prune large parts of the search space by comparing previously found solutions against an admissible heuristic estimate of the subproblem below the current search node, as it is provided by the mini-bucket scheme, for instance.

Figure 3.2 illustrates this on five different problem instances from various domains, each run with five different variable orderings. For each ordering we show the actual number of AND node expansions by AOBB and the respective state space bound computed as in Equation 3.1. It is evident that the state space bound doesn't really exhibit any correlation with the actual number of node expansions and that it is typically several orders of magnitude too large (note the vertical log scale).

In the following, we will therefore develop a method to bound or estimate problem complexity ahead of time in a more accurate fashion. We begin by surveying survey some related work in this area, which will serve to motivate our own approach using statistical regression learning.

3.2.2 Related Work

Early efforts in search space estimation go back to Knuth in 1975 with work that is concerned with general backtrack trees (i.e., a traditional OR search space) [70]. Knuth's method performs random probes without backtracking, each time selecting one child at each level i and recording the sequence of observed branching factors b_i. For a given probe of depth d the estimate is simply $\sum_{i=0}^{d} \prod_{j=1}^{i} b_j = 1 + b_1 + b_1 b_2 + \ldots$. The overall estimate is taken to be the average over all probes. This method is asymptotically unbiased, namely the expected value it computes is the correct size of the search tree. However, Knuth emphasized the large variance of the estimator.

Knuth's method was extended in 1992 by Chen, who adopted a stratified sampling approach within each random probe [18]. Namely, search nodes are classified into *types* or *strata*, and instead of one node per depth level as in Knuth's method, one node per type is expanded. Note that when the type is determined only by a node's depth, stratified sampling is equivalent to Knuth's method, and when each node defines its own unique type, we explore the full search space. Chen proposed a type system based on a node's number of children and proved that his scheme has reduced variance compared to Knuth's original method.

However, as Knuth earlier pointed out, his method as well as Chen's extension do not directly apply to the explored search space of branch-and-bound algorithms [70]. In particular, the bounds that are used by branch-and-bound for pruning decisions along a given search path are not known ahead of time and depend on the prior search history. In other words, Knuth and Chen's methods assume that the entire explored search tree is (implicitly) known in advance, namely for any partial path ending in a node n we can determine what its children will be in the final explored tree. This does not generally hold for branch-and-bound, where the shape of the explored tree depends on the order in which the search progresses – node expansion and pruning differs depending on the best solution found so far. As a consequence

the random probes of Knuth and Chen's scheme cannot accurately capture the pruning behavior of branch-and-bound we are interested for this thesis. (It is worth noting, however, that they can still be applied to the limited problem of estimating the work required to prove optimality of a cost bound if the optimal cost is provided as an input.)

Similar methods of runtime estimation have been proposed by Korf et al. for Iterative Deepening A* [72], a linear-space version of the heuristic search algorithm A*. They employ sampling to get an estimate of the asymptotic branching factor of the explored search space and try to characterize the distribution over values of the heuristic function, which they combine for an overall runtime estimation.

Lelis et al. combined this with Chen's method, proposing a stratified sampling approach for predicting the runtime of IDA* [76, 77], using the heuristic function as part of the stratifier. However, since IDA* and A* are best-first search algorithms, neither of these estimation approaches can easily be extended to depth-first branch-and-bound schemes like AOBB – just like Knuth and Chen's methods, they don't account for the bounds that the algorithm updates throughout its progress, which are used for heuristic pruning.

More recently, other authors have proposed online estimation methods that are more suited for the specifics of branch-and-bound-type algorithms. Kilby et al. [68] introduced the *Weighted Backtrack Estimator* (WBE), which can also be seen as an extension of Knuth's method. As an online procedure, it is run as part of branch-and-bound (or another chronological backtracking algorithm) where it keeps track of all "branch lengths" seen so far, i.e., the depth of terminal nodes in the explored search space. At any point, the overall estimate is a weighted average of individual estimates resulting from each such branch, computed just as in Knuth's method. Each element's weight is the probability of reaching the corresponding leaf node through random probing (1 over the product of variable domain sizes along the branch). For instance, if all variable domains are binary and D is the multiset of branch lengths encountered so far, the estimate can be computed as $\frac{\sum_{d \in D} 2^{-d}(2^{d+1}-1)}{\sum_{d \in D} 2^{-d}}$, where $2^{d+1}-1$

is the size of a full binary tree of depth d and 2^{-d} is the probability of reaching a particular leaf at that depth.

Kilby et al. also propose a second method, called the *recursive estimator* [68]. Defined only for binary search trees, it also keeps track of node counts for each solved subproblem. At any point, estimates are computed by "guessing" that an (unexplored) subtree to the right will have the same size as an (explored) subtree to the left, a logic that is applied recursively.

Cornuéjols et al. [21] proposed another estimation approach (which they left unnamed). It is also an online procedure, targeted specifically at branch-and-bound schemes over OR search trees. As the main search algorithm runs as usual, their method keeps track of the shape of the search tree, in particular its maximum depth, the "waist" level (i.e., the depth level with the largest number of explored nodes), and the last complete level. From these parameters they construct a piecewise linear model of the overall branching factor and use that to compute an estimate of the overall search tree size.

Both [21] (effective branching factor prediction) and [68] (WBE and recursive estimator) provide some limited experimental evaluation in the context of optimization problems. Cornuéjols et al. report on a number of mixed integer programming instances (MIP), while Kilby et al. only include summary statistics over a set of Traveling Salesman problem instances (TSP). And while limited in scope, results are mixed in either case. Owing to their online nature, in many cases significant headway into the search process is required to yield fair estimates. Furthermore, neither of these methods is well-defined for search graphs like the context-minimal AND/OR search graph and it is not obvious how AOBB's caching of context-unifiable subproblems could be properly accounted for in the estimation.

A different approach for estimating the size of the explored search space, and thereby problem runtime, was developed by Leyton-Brown, Xu and others in a series of papers that apply machine learning methods, in particular regression techniques.

In one line of work [78], they consider a set of problems from the domain of combinatorial auctions, to be solved by the CPLEX solver. Given a set of previously solved instances and their respective runtimes, their method extracts a number of domain-dependent as well as domain-independent features from each instance. These are given as input to a regression learning algorithm to find a suitable model for runtime prediction according to specific criteria.

Similar concepts were applied to develop SATzilla, a portfolio solver for SAT problems that competed very successfully in a number of SAT competitions [114]. As before, a number of domain-dependent and domain-independent problem features are identified for a number of sample instances. In an offline step, these sample instances are solved by the various SAT solvers in the SATzilla portfolio and their respective solutions times recorded. Subsequently, a separate runtime estimator is learned for each solver. Given a new instance, the different estimators are applied and the SAT solver with the lowest runtime estimate is chosen to process the problem.

In either case results were promising throughout, further evidenced by SATzilla's competitive success. One major advantage of these regression learning based approaches is that most of the estimation complexity is shifted to an offline step, where a sample set is compiled and estimators are trained. Computing estimates for a new instance is comparatively cheap.

This makes this kind of method particularly well-suited for our purposes of parallelizing AOBB, where we often need to consider and estimate the complexity of hundreds, if not thousands of subproblems (cf. Chapter 4). In Section 3.3 we will therefore develop a regression-based estimation scheme for general AND/OR Branch-and-Bound search, which will later form one key component of our parallel implementation.

3.2.3 Earlier Work

For completeness we mention here prior work that we conducted with a focus on the complexity of search for solution counting or likelihood computations [88, 89]. In this work we aimed to exploit determinism in the function tables to obtain tighter upper bounds on the number of expanded nodes. It was motivated by the recently introduced concept of *hypertree decompositions*, which is tree decomposition of the problem's hypergraph. A *hypergraph* of a graphical model $(X, D, F, \otimes)$ is a graph (V, E) that still has the variables as its vertices, $V = X$, but has *hyperedges* that are subsets of variables corresponding to the scopes of the functions F. Alternatively, a hypertree decomposition can be defined as an extension of a tree decomposition as follows:

DEFINITION **3.1** (hypertree decomposition)**.** *[48, 49] A* (generalized) hypertree decomposition *of a graphical model $(X, D, F, \otimes)$ is a tree $T = (V, E)$ with clusters V and edges E, together with labeling functions χ and ψ that associate with each vertex $v \in V$ two sets $\chi(v) \subset X$ and $\psi(v) \subset F$ such that:*

1. *For each $f_i \in F$ there exists $v \in V$ such that $f_i \in \psi(v)$ and $scope(f_i) \subset \chi(v)$, i.e., each function and its scope is contained in at least one cluster.*
2. *For each $X_i \in X$ the set $\{v' \in V \mid X_i \in \chi(v)\}$ forms a connected subtree of T; this is also called the "running intersection" or "connectedness" property.*
3. *For each $v \in V$, we have $\chi(v) \subset \bigcup_{f \in \psi(v)} scope(f)$, i.e., each cluster is fully "covered" by its associated functions' scopes.*

The hypertree width *of a hypertree decomposition is defined as $hw := \max_v |\psi(v)|$. The hypertree width hw^* of a graphical model is the minimum hypertree width over all its hypertree decompositions.*

We note that conditions 1. and 2. above are actually the same as for a tree decomposition (cf. Definition 1.7), hence every hypertree decomposition is also a tree decomposition. However, the third condition allows us to reason about problem complexity in terms of the problem's function table properties. To that end we introduce the notion of *tightness* of a function:

DEFINITION **3.2** (tightness)**.** *The* tightness t_f *of a function* f *is the number of relevant tuples (e.g., nonzero entries in conditional probability tables, allowed tuples in constraints).*

Intuitively, we can store and process function f in a "compressed" form, with only the t_f relevant tuples. Given a hypertree decomposition, one can then modify an inference algorithm to make use of these compact representations. In [48] the complexity of processing a hypertree decomposition for solving a constraint satisfaction problem is shown to be exponential in hw, with a dominant factor of t^{hw} (where t bounds the tightness of the constraints). This was extended in [66] to any graphical model that is absorbing relative to 0. (A graphical model is absorbing relative to a 0 element if its combination operator has the property that $x \otimes 0 = 0$ for all x; for example, multiplication has this property while summation does not.)

In terms of search over AND/OR spaces, we note that the search algorithm backtracks – i.e., the current node doesn't get expanded – if it encounters a dead end, caused by evaluating one or more functions with an inconsistent value. In addition, we recall the close correspondence between the guiding pseudo tree and the resulting context-minimal AND/OR search space on the one hand, and a suitable tree decomposition on the other hand (cf. Section 1.2.3) – this observation also formed the basis for the state space bound derived in Section 3.2.1.

Our work presented in [88, 89] ties these two concepts together by further tightening the state space bound through exploiting function tightness information.

- As in the state space bound, each variable's contribution is considered separately, with the product of domain sizes of the variable and its context as the starting point.

- For each such variable cluster, a greedy algorithm is applied, iteratively trying to cover a subset of variables in the cluster with as tight a function as possible, provided the function's scope is fully instantiated along the path to the root. (Note that function scopes often overlap significantly.)

- The result, per cluster, is an upper bound obtained by multiplying the tightness of the functions in the computed covering with the domain sizes of the uncovered variables. Summing over all clusters yields an overall upper bound.

We point out that function scopes typically overlap significantly and the greedy covering only considers the tightness and in particular doesn't perform any kind of resolution or function combination operations – the result is thus an upper bound. We also note that the above formulation resembles a weighted variant of the well-known, NP-complete set covering problem.

Full details of the scheme as well as empirical evaluation are provided in [88, 89]. In the presence of determinism in a problem instance, the resulting bounds are shown to often be much tighter than the state space bound. In some cases they can reflect, with reasonable accuracy, the number of node expansions performed by the search algorithm. Recall, however, that this work was conducted in the context of likelihood computations and solution counting, i.e., not for combinatorial optimization problems. It therefore doesn't account for the powerful pruning of branch-and-bound-style algorithms such as AOBB and was not immediately considered for the estimation work describe here. However, integrating these two lines of work, for instance by using the upper bound described above as a feature in the regression scheme developed subsequently, carries potential for future research.

3.3 Complexity Prediction as Regression Learning

This section derives and describes in-depth our regression-based approach to runtime complexity prediction of AOBB for a given problem instance or subproblem. Our work is similar to, and partly inspired by, Leyton-Brown et al.'s work in estimating the runtime of the CPLEX on combinatorial auction problems [78] as well as Xu et al.'s contributions to predicting the fastest solver from an algorithm portfolio in the SATzilla system [114].

In contrast to their methods, however, we don't limit ourselves to a particular problem class but instead devise a general scheme applicable to any set of graphical model optimization problems. This has direct implications for the kinds of features we can consider within our learning approach on, namely it rules out many domain-specific aspects that Leyton-Brown et al. and Xu et al. made use of.

Before going into the specifics of our contribution, however, Section 3.3.1 provides a brief summary of some of the central concepts of learning that we will employ.

3.3.1 Learning Background

In the following we survey some of the concepts and terminology of supervised learning that form the basis of our work described subsequently. Our exposition is based on established textbooks on machine learning and learning theory (e.g., [59, 112]). The basic theoretical building blocks of a general learning problem can be defined as follows:

- The *learning domain*, an arbitrary set $\mathcal{X}$ from which samples x are drawn, according to some unknown distribution $\mathcal{D}$. Samples are typically represented by a number of (numerical) features; we will denote the i-th feature by $\phi_i(x)$.

- The set of *target values* $\mathcal{Y}$, which get assigned to each sample by an (unknown) function $f\colon \mathcal{X} \to \mathcal{Y}$.
- A set $\mathcal{S}$ of *training data* $(x_j, y_j) \in \mathcal{X} \times \mathcal{Y}$, i.e. samples from $\mathcal{X}$ with known target $f(x_j) = y_j$.

The learning task is then to use the training data to find a function $\hat{f}\colon \mathcal{X} \to \mathcal{Y}$, called a *model* or *hypothesis*, that can be used to predict for a new sample $x' \in \mathcal{X}$ its target $\hat{f}(x')$, using the sample features $\phi_i(x')$. Typically we limit ourselves to a specific class of hypotheses $\mathcal{F}$ from which $\hat{f}$ can be chosen (e.g., the class of all linear combinations of sample features).

If $\mathcal{Y}$ is a finite set of discrete categories ("labels"), we speak of a *classification* task. If $\mathcal{Y}$ is continuous or quantitative we have a *regression* learning problem. For our work on predicting the runtime complexity of AOBB we will limit ourselves to the latter, in particular we set $\mathcal{Y} = \mathbb{R}$.

3.3.1.1 Evaluating Learning

To assess the quality of a hypothesis $\hat{f}$ we first define the error or *loss function* $l\colon \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ that allows us to determine, for each sample $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the loss, or error, of a prediction via $l(\hat{f}(x), y) = l(\hat{f}(x), f(x))$. The loss or, more commonly, *risk* of the estimator $\hat{f}$ is then defined as the expected error over the learning domain $\mathcal{X}$ with respect to the distribution $\mathcal{D}$:

$$L_{\mathcal{D}}(\hat{f}) \;:=\; E_{x \sim D}[l(\hat{f}(x), f(x))] \tag{3.2}$$

$L_{\mathcal{D}}(\hat{f})$ is also known as the *generalization error*. However, the full set $\mathcal{X}$ as well as the distribution $\mathcal{D}$ is usually unknown, which makes $L_{\mathcal{D}}(\hat{f})$ infeasible to compute and optimize against. Instead, under the assumption that the training data $\mathcal{S}$ are drawn *i.i.d.* (indepen-

dently and identically distributed) from $\mathcal{D}$, we can define the *empirical loss* for given training data $\mathcal{S} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ as:

$$L_{\mathcal{S}}(\hat{f}) \;:=\; \frac{1}{m}\sum_{j=1}^{m} l(\hat{f}(x_j), y_j) \tag{3.3}$$

This quantity is also referred to as the *training error* or *empirical risk*. Hence, a learning algorithm that aims to find $\hat{f}$ that minimizes $L_{\mathcal{S}}(\hat{f})$ as a proxy for optimizing $L_{\mathcal{D}}(\hat{f})$ is said to follow the principle of *empirical risk minimization* (ERM) [112]. Given a model $\hat{f}$ that minimizes $L_{\mathcal{S}}(\hat{f})$ over a training set $\mathcal{S}$, we can try to assess the model's generalization by evaluating it over a *test set* $\mathcal{T}$ of samples drawn i.i.d. from $\mathcal{D}$ (different from the training set $\mathcal{S}$). We compute the *test error* $L_{\mathcal{T}}(\hat{f})$ as an indication of the generalization error.

A simple loss function that is often used is the squared error defined through $l(\hat{f}(x), y)) := (\hat{f}(x) - y)^2$, which results in the very common *mean squared error* (MSE) measure:

$$MSE_{\mathcal{S}}(\hat{f}) \;=\; \frac{1}{m}\sum_{j=1}^{m} (\hat{f}(x_j) - y_j)^2 \tag{3.4}$$

Clearly, lower values are better and a MSE of 0 would imply a perfect fit of the model $\hat{f}$ to the sample set $\mathcal{S}$. Other loss functions are possible, but the MSE is widely used in practical applications, well understood, and implemented in many mature software libraries, which is why we will employ it as well.

3.3.1.2 Overfitting and Regularization

A potential danger of following the ERM principle lies in *overfitting*, where the learned hypothesis $\hat{f}$ performs very well on the training data $\mathcal{S}$ but behaves very poorly on new samples from $\mathcal{D}$, i.e., it achieves low training error but sees high prediction error. Intuitively,

$\hat{f}$ focuses too much on the specifics of the training set $\mathcal{S}$ and therefore generalizes poorly to unseen samples from the learning domain $\mathcal{X}$.

A common way to overcome this issue lies in *regularization* [9] (sometimes called *shrinkage* [59]), which is closely related to the principle of *structural risk minimization* [112]. The underlying concept is to penalize complex models that are likely to overfit the training data just as described, i.e. to "force" the learning of simpler hypotheses. This is achieved by adding a penalty $p(\hat{f})$, the regularization term, to the empirical loss function under consideration:

$$L_{\mathcal{S}}(\hat{f}) \;:=\; \frac{1}{m}\sum_{j=1}^{m} l(\hat{f}(x_j), y_j) \;+\; \alpha\, p(\hat{f}) \tag{3.5}$$

Here $\alpha > 0$ is the regularization parameter that controls the strength of regularization (often determined experimentally in practice). The definition of the penalty function p typically depends on the form of hypotheses considered through $\mathcal{F}$. As Section 3.3.5 will show, an added benefit of regularization is that it can yield sparser learned models, which aid interpretability.

The following Sections will outline our choices for the class of models $\mathcal{F}$ we propose (Section 3.3.2), the set of sample features we define (Section 3.3.3), as well as the different learning domains we investigate in the context of AOBB (Section 3.3.4). In addition, Section 3.3.5 describes the algorithms we used to learn particular model instances for a given training set.

3.3.2 Modeling AOBB Complexity

This section describes the class of hypotheses, or models, we consider for capturing the complexity of subproblems in the explored search space of AND/OR Branch-and-Bound. To begin, we identify each such subproblem with its root node n, in which we implicitly include

the path from n to the overall root of the tree, which defines the subproblem context. As noted above, we measure a subproblem's search complexity via the number of node expansions AOBB requires for its solution, denoted $N(n)$. For simplicity, we let $N(n)$ take values in $\mathbb{R}$, the target set of our learning setup.

To devise a concrete model class, we remind ourselves of the exponential nature of combinatorial search spaces: general search spaces are often derived as exponential in the number of variables (see for instance [87, Chapter 9]) and as Section 1.3.1 showed AND/OR search trees (without caching) are exponential in the height of the guiding pseudo tree while context-minimal AND/OR search graphs have size exponential in the induced width of the variable ordering.

All of the above can be seen as numerical *features* of the problem or subproblem search space rooted at n; as indicated in Section 3.3.1 we will denote these features by $\phi_i(n)$ – the full set of subproblem features we consider will described in Section 3.3.3. We then aim to capture the aforementioned exponential relationship by characterizing $N(n)$ as an exponential function of those features as follows:

$$N(n) \;=\; \exp\Big(\sum_i \lambda_i \phi_i(n)\Big) \tag{3.6}$$

Here $\lambda_i \in \mathbb{R}$ is the weight of feature ϕ_i, and each full set of λ_i's represents a single model or hypothesis from the hypothesis space $\mathcal{F}$ as described in Section 3.3.1.

The choice of the exponent in Equation 3.6 as a weighted sum allows us to consider the log complexity and obtain the following, convenient expression:

$$\log N(n) \;=\; \sum_i \lambda_i \phi_i(n) \tag{3.7}$$

Given a sample set $\mathcal{S} = \{n_1, \ldots, n_m\}$ of size m, representing a set of m AND/OR subproblems, we employ the mean squared error as our loss function (cf. Section 3.3.1) and obtain the following empirical loss, written as a function of λ, the set of λ_i:

$$L(\lambda) \;=\; \frac{1}{m} \sum_{j=1}^{m} \Big(\sum_{i} \lambda_i \phi_i(n_j) - \log N(n_j) \Big)^2 \tag{3.8}$$

In other words, finding parameter values λ_i that minimize this empirical loss can be interpreted as a well-known *linear regression* problem, which has received extensive treatment in the literature (see for instance [9, 33, 59, 102]). A number of different algorithms for this task, including regularization approaches, will be described in Section 3.3.5.

3.3.3 Subproblem Sample Features

This section describes the features we extract for each subproblem, represented by its root node n, as mentioned previously. Having good features as the basis of the regression analysis is clearly crucial – only with an expressive set of parameters will there be a chance of learning an effective and robust model.

The overall process of assembling features for learning can be divided into two conceptually distinctive steps: In *feature engineering* we use our understanding, intuition, and prior knowledge of the problem space to come up with a list of possible problem features. *Feature selection*, on the other hand, involves analysis and experimental evaluation to select only a subset of the available features, to reduce computational complexity of learning and improve interpretability and generalizability of the learned models (e.g., [9, 59]).

Our focus here will be on feature engineering only – while we did experiment with dedicated feature selection schemes like *forward selection*, *backward selection*, and combinations thereof

Subproblem variable statistics (static):

1: Number of variables in subproblem.

2-6: Min, Max, mean, average, and std. dev. of variable domain sizes in subproblem.

Pseudo tree depth/leaf statistics (static):

7: Depth of subproblem root in overall search space.

8-12: Min, max, mean, average, and std. dev. of depth of subproblem pseudo tree leaf nodes, counted from subproblem root.

13: Number of leaf nodes in subproblem pseudo tree.

Pseudo tree width statistics (static):

14-18: Min, max, mean, average, and std. dev. of induced width of variables within subproblem.

19-23: Min, max, mean, average, and std. dev. of induced width of variables within subproblem, *conditioned on subproblem root context.*

State space bound (static):

24: State space size upper bound on subproblem search space size (cf. Section 3.2.1).

Subproblem cost bounds (dynamic):

25: Lower bound L on subproblem solution cost, derived from current best overall solution.

26: Upper bound U on subproblem solution cost, provided by mini bucket heuristics.

27: Difference $U - L$ between upper and lower bound, expressing "constrainedness" of the subproblem.

Pruning ratios (dynamic), based on running AOBB for $5n$ node expansions:

28: Ratio of nodes pruned using the heuristic.

29: Ratio of nodes pruned due of determinism (zero probabilities, e.g.)

30: Ratio of nodes corresponding to pseudo tree leaf.

AOBB sample (dynamic), based on running AOBB for $5n$ node expansions:

31: Average depth of terminal search nodes within probe.

32: Average node depth within probe (denoted $\bar{d}$).

33: Average branching degree, defined as $\sqrt[\bar{d}]{5n}$.

Various (static):

34: Mini bucket i-bound parameter.

35: Max. subproblem variable context size minus mini bucket i-bound.

Table 3.1: Summary of 35 features extracted from each subproblem, forming the basis for regression learning.

[102], we eventually abandoned this in favor of regularized learning algorithms that provide "built-in" feature selection, as will be detailed in Section 3.3.5.

Table 3.1 shows the full list of subproblem features ϕ_i that we consider. In determining this set, we recall our earlier comments regarding the state space size bound. In particular, in Section 3.2.1 we attributed its shortcomings to the fact that it does not capture the powerful pruning of AOBB, which is based on the cost functions of the problem and highly dynamic. Accordingly, we can divide our feature set into two conceptual classes as follows:

- **Static**, which can be precompiled from the problem graph and pseudo tree. These include obvious parameters like the number of variables, domain size information, and induced width (or more generally variable context size statistics) which make up the asymptotic complexity bound of AOBB. We also add the state space size bound on the subproblem search space as derived above. Moreover, we include properties such as number of pseudo tree leaves and their depth (capturing decomposition) as well as subproblem height. Finally, we add the mini-bucket i-bound and its difference to the subproblem induced width, in an effort to capture how "far" the heuristic is from the true solution cost.

- **Dynamic**, which are computed at runtime, as the individual subproblems are considered. One such group of features is based on cost bound information, i.e., it incorporates the problem's cost functions. This includes the subproblem cost upper bound (as returned by the mini-bucket heuristic), the current cost lower bound (as derived from the current best solution known to AOBB), as well as the difference between the two (to measure the "constrainedness" of the subproblem). Secondly, we run full AOBB for a limited (small) number of node expansions and extract various statistics from that sample, such as the ratio of pruned nodes or average depth of leaf nodes.

An important practical consideration in our parallelization context is that none of the dynamic features should be costly to compute, since the scheme will potentially consider thousands of subproblems. The cost bound information is readily available as part of AOBB and does not incur any computational overhead. Running an AOBB sample for each subproblem, however, can take considerable time depending on the sample size. We have experimented with different node expansion limits and found a value of $5n$ to be a good compromise, where n is again the number of problem variables. This number seemed large enough to yield stable ratios (i.e., values don't change significantly for larger expansion limits) while at the same time remaining computationally feasible (our AOBB implementation typically expands some hundred thousand nodes per second).

3.3.4 Subproblem Learning Domains

In order to evaluate the performance of the proposed estimation procedure from a statistical learning perspective, namely to assess its training and prediction error, we need to specify the learning domain $\mathcal{X}$ over which we aim to learn make predictions and from which subproblem samples are assumed to be drawn.

In fact, in the following we consider four levels of learning domains, corresponding to four different designs in the context of parallelizing AOBB. These levels are increasingly more general, reflecting the fact that instances from one problem class often exhibit similar characteristics (for instance, a certain range of variable domain sizes, or a high degree of determinism).

3.3.4.1 Learning per Problem Instance

In *per-instance learning*, the learning domain $\mathcal{X}$ consists of subproblems from a single problem instance. In the context of parallelizing AOBB, this corresponds to learning a separate complexity model for every problem instance. This approach thus has limited relevance in practice, since subproblem samples would have to be drawn and a new model would have to be learned for every new problem instance the parallel scheme encounters. However, it can function as a baseline for assessing predictive performance.

3.3.4.2 Learning per Problem Class

In *per-class learning*, we take the learning domain $\mathcal{X}$ to be subproblems of problems from a specific class. In the parallelization context we learn a separate model for every problem class we consider. Given a new problem instance from that class, however, we redeploy the learned model. For example, we would learn a single model for all pedigree problems and apply it for subsequent input instances from this class, but we'd require a different model for other problem classes like protein side-chain prediction.

3.3.4.3 Learning across Problem Classes

In *cross-class learning*, we take the learning domain $\mathcal{X}$ to be subproblems of problems from several classes and learn a single, unified model across all of them. Subsequent instances from any of the covered problem classes are processed using the same model. In the context of parallelization, this could translate to a parallel scheme that uses a single complexity model for all problem classes under consideration.

3.3.4.4 Learning for Unseen Problem Classes

For the previous three levels of learning, training and test samples in each case are drawn from the same learning domain $\mathcal{X}$ under the same distribution $\mathcal{D}$. *Unseen-class* learning, the last and most general level of learning, deviates from this by taking the learning domain $\mathcal{X}$ to be subproblems of problems from several classes, just as in the previous, third level. In contrast to that, however, test samples will be drawn from a different domain $\mathcal{X}'$, consisting of subproblem instances from a so far unseen problem class. This level of learning thus resembles a form of *transfer learning*, a relatively recent area of research within machine learning [93].

We note that all training and test sample subproblems still originate within the context of AND/OR Branch-and-Bound search spaces. An alternative formulation would thus be one large unified domain $\mathcal{X}$ with two different, disjoint distributions $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ from which training and test samples are drawn, respectively.

Either way, this approach relaxes the assumption that training and test data are sampled i.i.d. from the same learning domain $\mathcal{X}$ and distribution $\mathcal{D}$. In parallelization terms this is clearly the most general level, allowing a single parallel scheme, with a single estimation model, to be applied to instances from any possible problem class, even those that were not the basis for the learned model.

3.3.5 Regression Algorithms

This section describes a number of learning algorithms that we considered for learning the complexity model developed in Section 3.3.1, i.e., given a training set of m sample subproblems $\mathcal{S} = \{n_1, \ldots, n_m\}$ we aim to find a set of weights λ_i for the linear model given in

Equation 3.7. In all cases we used the implementations available in the *scikit-learn* machine learning Python library [97].

An established, simple baseline algorithm is *ordinary least squares* (OLS) [9, 59], which yields parameter values for λ_i that directly minimize the MSE loss function from Equation 3.8. In its geometric interpretation, OLS minimizes the Euclidean norm $||X\Lambda - Y||_2$, where X is the design matrix of subproblem features (one subproblem per row), Λ the parameter vector of λ_i's to be learned, and Y the vector of log problem sizes $\log N(n_i)$. In practice, however, OLS often turns out to have numerical issues (because of singular or near-singular matrix inversion, for instance) and can be prone to overfitting.

To address these issues with OLS, *ridge regression* introduces a regularization term to the loss function, as described in Section 3.3.1. In particular, it uses the L_2 norm of the parameter vector Λ. Ridge regression hence minimizes $||X\Lambda - Y||_2^2 + \alpha||\Lambda||_2^2$ or equivalently,

$$L_{ridge}(\lambda) \;=\; \frac{1}{m}\sum_{j=1}^{m}\Big(\sum_i \lambda_i\phi_i(n_j) - \log N(n_j)\Big)^2 + \alpha\sum_i \lambda_i^2 \tag{3.9}$$

We recall that $\alpha > 0$ is a parameter that controls the amount of "shrinkage," i.e., the strength of regularization. The addition of the L_2 regularization term encourages small parameter values λ_i, thereby helping to combat overfitting. In addition, ridge regression alleviates numerical issues of OLS [59].

A closely related learning method is *lasso regression* [59, 110]. Similar to ridge regression, it introduces a regularization term to the loss function; however, in this case the L_1 norm of the parameter values λ_i is used. More formally, lasso regression minimizes the following loss function:

$$L_{lasso}(\lambda) \;=\; \frac{1}{m}\sum_{j=1}^{m}\Big(\sum_i \lambda_i\phi_i(n_j) - \log N(n_j)\Big)^2 + \alpha\sum_i |\lambda_i| \tag{3.10}$$

As before, $\alpha > 0$ is the parameter that controls regularization strength. Due to the nature of the L_1 norm, larger values of α causes an increasing number of parameters λ_i to become 0 and "drop out" of the resulting model [110]. Lasso regression thus has the advantage of "built-in" feature selection. At the same time, however, the L_1 regularization also means that only one out of several strong but correlated features tends to get selected by lasso regression [59].

In our experiments we have found ridge regression and lasso regression to yield very similar results in terms of prediction accuracy, with no notable, consistent advantage for either method. In the following we will therefore employ lasso regression, since it has the advantage of compact, interpretable models due to its built-in feature selection.

3.3.6 Non-linear Regression

In addition to the purely linear regression analysis proposed above, we also explored a higher-order approach. In particular, we took inspiration from Leyton-Brown et al. [78], who report improved prediction performance using *quadratic feature expansion*, albeit in the context of combinatorial auctions.

Quadratic feature expansion, sometimes also referred to as "quadratic regression," works by adding new features in the form of pairwise products of the original features; namely, for every pair of subproblem features ϕ_i, ϕ_j with $i \leq j$, we create a new feature $\phi_i \cdot \phi_j$. In our case, 35 features are extended with $\binom{35}{2}$ pairwise products, yielding a total of 665 features. We then perform linear regression on the expanded feature set, thereby effectively fitting a polynomial of 2^{nd} degree.

In practice, however, we didn't find any meaningful improvement that would justify the substantial increase in computational complexity of model training and the reduced interpretability of the quadratic model.

3.4 Model Learning & Evaluation

This section describes how we learn the complexity model introduced in Section 3.3 in practice and presents in-depth empirical evaluation and analysis of it; specifically, we will be learning instances of the lasso regression model as defined in Equation 3.10. We begin by describing in Section 3.4.1 the problem classes and instances whose subproblems make up the learning domain $\mathcal{X}$. Section 3.4.2 explains the general experimental setup, including performance measures and the choice of regularization parameter.

Sections 3.4.3 through 3.4.6 report results on the four increasingly general levels of learning, following the exposition of Section 3.3.4 above; Section 3.4.7 provides summary and interpretation of these results. In Section 3.4.8 we analyze the results with respect to the informativeness of the different subproblem features defined in Section 3.3.3.

3.4.1 Benchmark Instances

We first describe the various problem classes and instances that we use to learn and evaluate the complexity prediction model, given by Equation 3.10 in Section 3.3 above. As mentioned, our evaluation will be based on subproblems drawn from a number of problem instances. This approach makes it easier to compile a sizable sample set, but more importantly it will also directly tie into the design and analysis of parallel AOBB in Chapter 4.

After assembling a set of reasonably complex base problems from four different problem classes, we run each problem instance with varying i-bounds for the mini-bucket heuristic; for each problem an initial lower bound on the overall solution cost is generated through 5 seconds of local search. Subproblems are generated from different depths d in the search space, taken from experimental results recorded during the development of parallel AOBB – in short, a central search process applies conditioning up to the specified depth d, thereby implying a *parallelization frontier*. Nodes in this frontier represent independent subproblem with different context instantiations that can be solved separately and in parallel. Full details and analysis of parallel AOBB will be provided in Chapter 4.

The four problem classes under consideration are as follows:

- **Linkage analysis ("pedigree"):** Each of these networks is an instance of a genetic linkage analysis problem on a particular pedigree, i.e., a family ancestry chart over several generations, annotated with phenotype information (observable physical traits, inheritable diseases, etc.) [40, 41]. Originally aimed at $P(e)$ sum-product likelihood computation, these problems have gained popularity as MPE benchmarks due to their complexity and real-world applicability and have been included in recent inference competitions [23, 35].

- **Haplotype inference ("largeFam"):** These networks also encode genetic pedigree instances into a Bayesian network. However, the encoded task is the haplotyping problem, which differs from linkage analysis and necessitates different conversion and data preprocessing steps to generate the graphical model input to our algorithms [1, 39].

- **Protein side-chain prediction ("pdb"):** These networks correspond to side-chain conformation prediction tasks in the protein folding problem [115]. The resulting instances have relatively few nodes, but very large variable domains, forcing a very low mini-bucket i-bound of 3 and generally rendering most instances very complex.

instance	n	m	k	w	h	i	d	p	T_{seq}
pedigree7	1068	1068	4	32	90	7	9	1280	93380
pedigree7	1068	1068	4	32	90	8	8	640	30717
pedigree7	1068	1068	4	32	90	6	9	1280	118383
pedigree9	1118	1118	7	27	100	7	11	1024	58657
pedigree9	1118	1118	7	27	100	8	10	512	41061
pedigree9	1118	1118	7	27	100	6	11	1024	101172
pedigree13	1077	1077	3	32	102	9	10	1024	102385
pedigree13	1077	1077	3	32	102	10	9	512	23949
pedigree13	1077	1077	3	32	102	8	10	1024	252654
pedigree19	793	793	5	25	98	16	6	1440	375110
pedigree19	793	793	5	25	98	15	6	1440	NA
pedigree31	1183	1183	5	30	85	11	10	1024	433029
pedigree31	1183	1183	5	30	85	12	9	512	16238
pedigree31	1183	1183	5	30	85	10	10	1024	NA
pedigree33	798	798	4	28	98	4	8	96	6010
pedigree33	798	798	4	28	98	5	6	24	1482
pedigree34	1160	1160	5	31	102	12	11	948	96122
pedigree34	1160	1160	5	31	102	11	12	1912	350574
pedigree34	1160	1160	5	31	102	10	12	1896	NA
pedigree39	1272	1272	5	21	76	4	6	128	6632
pedigree39	1272	1272	5	21	76	5	5	64	2202
pedigree39	1272	1272	5	21	76	3	6	128	NA
pedigree41	1062	1062	5	33	100	10	10	1408	46819
pedigree41	1062	1062	5	33	100	11	9	704	27583
pedigree41	1062	1062	5	33	100	9	10	1408	25607
pedigree44	811	811	4	25	65	6	9	1120	95830
pedigree44	811	811	4	25	65	7	8	560	16443
pedigree44	811	811	4	25	65	5	9	1120	207136
pedigree51	1152	1152	5	39	98	21	10	1024	164817
pedigree51	1152	1152	5	39	98	20	10	1024	101788

Table 3.2: List of **pedigree linkage** problem instances and their parameters. n is the number of problem variables, m the number of cost functions, k the maximum variable domain size, and w and h the induced width and height of the guiding pseudo tree, respectively. i denotes the mini-bucket heuristic i-bound and d the depth at which subproblems were generated, whose resulting number is given as p. Lastly, T_{seq} is the runtime of sequential AOBB for comparison (1 week timeout denoted "NA").

- **Grid networks ("75-"):** Randomly generated grid networks of size 25x25 and 26x26 with roughly 75% of the probability table entries set to 0. From the original set of problems used in the UAI'08 Evaluation, only a handful proved difficult enough for inclusion here [23].

The particular problem instances and parameter combinations are summarized in Tables 3.2, 3.3, 3.4, and 3.5, respectively. Each table lists a comprehensive set of parameters: the number

instance	n	m	k	w	h	i	d	p	T_{seq}
largeFam3-11-57	2670	2670	3	37	95	17	10	180	18312
largeFam3-11-57	2670	2670	3	37	95	16	10	180	35820
largeFam3-11-59	2711	2711	3	32	73	16	8	200	3023
largeFam3-11-59	2711	2711	3	32	73	15	8	200	35457
largeFam3-13-58	3352	3352	3	31	88	18	8	200	7647
largeFam3-13-58	3352	3352	3	31	88	17	8	200	7379
largeFam3-15-53	3384	3384	3	32	108	18	13	2496	98346
largeFam3-15-53	3384	3384	3	32	108	17	13	2831	345544
largeFam3-15-55	3588	3588	3	35	84	16	10	2688	292713
largeFam3-15-55	3588	3588	3	35	84	15	10	2688	NA
largeFam3-15-59	3730	3730	3	31	84	19	9	936	43307
largeFam3-15-59	3730	3730	3	31	84	18	9	942	28613
largeFam3-16-56	3930	3930	3	38	77	16	12	2629	489614
largeFam3-16-56	3930	3930	3	38	77	15	12	2707	NA
largeFam3-16-56	3930	3930	3	38	77	16	10	900	489614
largeFam4-12-50	2569	2569	4	28	80	14	6	864	33676
largeFam4-12-50	2569	2569	4	28	80	13	6	864	NA
largeFam4-12-55	2926	2926	4	28	78	14	9	768	25905
largeFam4-12-55	2926	2926	4	28	78	13	9	1024	104837
largeFam4-17-51	3837	3837	4	29	85	16	12	704	66103
largeFam4-17-51	3837	3837	4	29	85	16	11	352	66103

Table 3.3: List of **largeFam haplotype** problem instances and their parameters. See Table 3.2 for column legend.

instance	n	m	k	w	h	i	d	p	T_{seq}
pdb1a6m	124	521	81	15	34	3	3	511	198326
pdb1duw	241	743	81	9	32	3	3	784	627106
pdb1e5k	154	587	81	12	43	3	2	1046	112654
pdb1f9i	103	387	81	10	24	3	2	6534	68804
pdb1ft5	172	645	81	14	33	3	3	5281	81118
pdb1hd2	126	448	81	12	27	3	2	3777	101550
pdb1huw	152	587	81	15	43	3	5	1588	545249
pdb1kao	148	568	81	15	41	3	4	3241	716795
pdb1nfp	204	791	81	18	38	3	4	3812	354720
pdb1rss	115	448	81	12	35	3	4	1336	378579
pdb1vhh	133	556	81	14	35	3	2	1842	944633

Table 3.4: List of **pdb side-chain prediction** problem instances and their parameters. See Table 3.2 for column legend.

instance	n	m	k	w	h	i	d	p	T_{seq}
75-25-1	624	626	2	38	111	14	10	96	15402
75-25-1	624	626	2	38	111	12	11	192	77941
75-25-1	624	626	2	38	111	12	8	128	77941
75-25-7	624	626	2	37	120	18	7	72	21694
75-25-7	624	626	2	37	120	16	8	144	297377
75-25-7	624	626	2	37	120	16	9	216	297377
75-26-2	675	677	2	39	120	20	8	144	8053
75-26-2	675	677	2	39	120	16	8	144	25274
75-26-2	675	677	2	39	120	16	9	288	25274
75-26-9	675	677	2	39	124	16	8	120	59609
75-26-9	675	677	2	39	124	16	9	240	59609
75-26-9	675	677	2	39	124	18	9	240	66533
75-26-10	675	677	2	39	124	20	11	416	28413
75-26-10	675	677	2	39	124	16	11	384	46985
75-26-10	675	677	2	39	124	16	10	192	46985

Table 3.5: List of **grid** problem instances and their parameters. See Table 3.2 for column legend.

of problem variables n, the number of functions m, the maximum variable domain size k, as well as the induced w and pseudo tree height h along a given minfill variable ordering. We also include the i-bound of the mini-bucket heuristic as well as the cut-off depth d at which subproblems were generated, whose resulting number is given as p. Finally, for reference T_{seq} denotes the runtime of sequential AOBB on the full problem.

The total number of subproblem samples across all 77 problem instances is over 68,000. However, to account for the large variance in subproblem count per instance and balance the problems' contribution, we randomly select at most 250 subproblems from each instance, leaving us with a final evaluation set of slightly over 17,000 subproblems across all instances and classes. Note that we normalized each feature to have zero mean and unit standard deviation before applying the learning algorithm.

3.4.2 Outline of Experiments

In the following sections we learn and apply our regression model, as established in Section 3.3.2, using the benchmark instances described above. We divide the results along the four conceptual levels of learning laid out in Section 3.3.4.

For each level of learning, we present results on select instances (four from each problem class) through scatter plots, depicting the actual complexity of subproblems against their estimated ones. In each case we also report the training error "TER" of the learned model on the training data, the estimation error "MSE" on the test data, as well as the Pearson Correlation Coefficient "PCC," as defined in Equation 3.11. (A larger set of plots is given in Appendix A, page 286.)

In addition, Tables 3.6 through 3.9, one per problem class, list MSE, PCC, and TER for all of the 77 problem instances, as well as aggregated measures across experiments for each class – where aggregate MSE and TER are weighted averages over the instances' respective error values (weighted by each instance's number of subproblem samples), and aggregate PCC is a simple average over the instance's PCC values. Table 3.10 summarizes these aggregate measures and also lists overall aggregates, across all problem classes.

3.4.2.1 Secondary Performance Measure

As derived in Section 3.3.1, our primary measure of learning performance is the mean squared error, or MSE, which is what our approach of linear regression is aiming to minimize (cf. Equation 3.8). It measures how close, on average, each predicted complexity is to the actual subproblem complexity.

In the context of load balancing for parallelism, however, we can additionally consider a secondary metric, the *Pearson correlation coefficient* (PCC) [101]. Given a set of subproblems

n_j, denote by N and $\hat{N}$ the vector of actual and predicted subproblem complexities $N(n_j)$ and $\hat{N}(n_j)$, respectively. The PCC of N and $\hat{N}$ is then simply the covariance between the two, normalized by the product of each vector's standard deviation. More formally, writing μ_N and $\mu_{\hat{N}}$ for the mean of N and $\hat{N}$, respectively, we have the following:

$$\begin{aligned} PCC(N, \hat{N}) &= \frac{cov(N, \hat{N})}{\sigma_N \sigma_{\hat{N}}} \\ &= \frac{\sum_j \big(N(n_j) - \mu_N\big)\big(\hat{N}(n_j) - \mu_{\hat{N}}\big)}{\sqrt{\sum_j \big(N(n_j) - \mu_N\big)^2}\sqrt{\sum_j \big(\hat{N}(n_k) - \mu_{\hat{N}}\big)^2}} \end{aligned} \tag{3.11}$$

The PCC is bounded by $[-1, 1]$, where 1 implies perfect linear correlation and -1 anticorrelation. In our parallelization context a value close to 1 is hence desirable, as it signifies a model likely to correctly distinguish between hard and easy subproblems or, more generally, provide a reliable ordering of subproblems in terms of their complexity. This will be important for the parallel algorithm's load balancing, where one objective is to identify and avoid bottlenecks in the form of overly complex subproblems, achieved by iteratively breaking the hardest subproblem into smaller pieces. Full details will be presented in Chapter 4.

3.4.2.2 Regularization Parameter α

Before conducting the main part of our evaluation in each of the following sections, we select the regularization parameter α for the lasso regression of Equation 3.10. We choose to do this separately for each of the four levels of learning outlined in Section 3.3.4 – we feel that learning only one global value of α would have been too general since it doesn't accurately reflect the practical deployment within parallel AOBB, where only one of the four levels would be selected.

Like for the final evaluation set mentioned above, we randomly select a subset of each instance's overall sample set (different from the final evaluation set). We then divide it

into four fifths training data and one fifth test data. For a range of α values we apply lasso regression as in Equation 3.10 on the training data, according to the particular level of learning, and evaluate the error of the resulting model on the test data. The range of values we consider is $[10^0, 10^{-4}]$ in intervals of -0.25 for the exponent, i.e., $-\log(\alpha) \in \{1, 0.75, 0.5, \ldots, -3.75, -4\}$. Here $\alpha = 10^0$ implies the strongest regularization, while $\alpha = 10^{-4}$ is the weakest. We choose the value of α that minimizes the test error; results will be reported in the respective subsequent sections.

3.4.3 Learning per Problem Instance

We begin by considering the "per-instance" level of learning, i.e., a separate complexity model is learned for every problem instance. On this most limited level of learning, both training and test samples are drawn from the subproblems of a given problem instance. We employ 5-fold cross validation for performance evaluation, i.e., we partition each instance's sample set into 5 disjoint subsets, then predict the complexity of each subset by learning a model from the remaining four.

As noted above, the practical relevance of this scenario in the parallelization context is limited, since it would entail sampling a sizable set of subproblems and learning a complexity model from these for each problem instance considered. However, it is useful to study the performance of our regression model.

Regularization. To select the regularization parameter α for the lasso regression model (Equation 3.10), Figure 3.3 plots the training and test error, averaged across instances, on a separate validation set as a function of α as described in Section 3.4.2.2. We note that test and training error are very close across the entire range, i.e., we see virtually no overfitting across the range of α's tested (which would manifest itself in increasing test error beyond

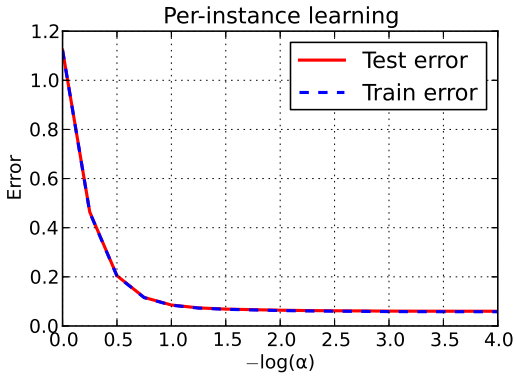


Figure 3.3: Training and test error on separate validation set as a function of regularization parameter α for **per-instance** learning.

a certain point). Based on the plot, we choose $\alpha = 10^{-3} = 0.001$, beyond which the error curves appears to flatten out.

3.4.3.1 Select Per-Instance Learning Results

Using $\alpha = 0.001$ we learn a lasso regression model (Equation 3.10) for each problem instance, using 5-fold cross validation as described above. Estimation results for four select instances per problem class are shown in the form of scatter plots in Figures 3.4 through 3.7 (with more plots in Appendix A.1). Though of no significance for evaluating results, the different colors in each plot mark the distinct cross validation folds. In all cases we observe very good estimation results, as detailed in the following.

Pedigree instances. Scatter plots for four select pedigree linkage instances are given in Figure 3.4 and demonstrate very good performance. For instance, for ped7 (top left) we observe training and test error of just 0.076 and 0.078, respectively. The correlation coefficient of 0.932 is also very good – we can confirm visually that the points of the scatter plot are fairly close to a diagonal (which represents the "perfect" match of estimated and actual complexities). Results for the other pedigree instances in Figure 3.4 are similar, with

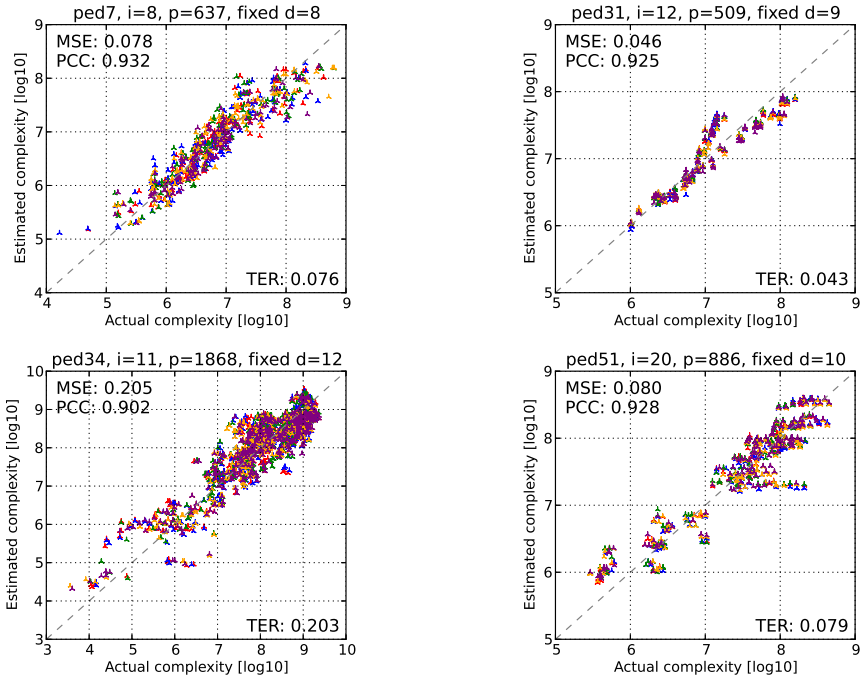


Figure 3.4: Select **per-instance** estimation results on **pedigree linkage** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

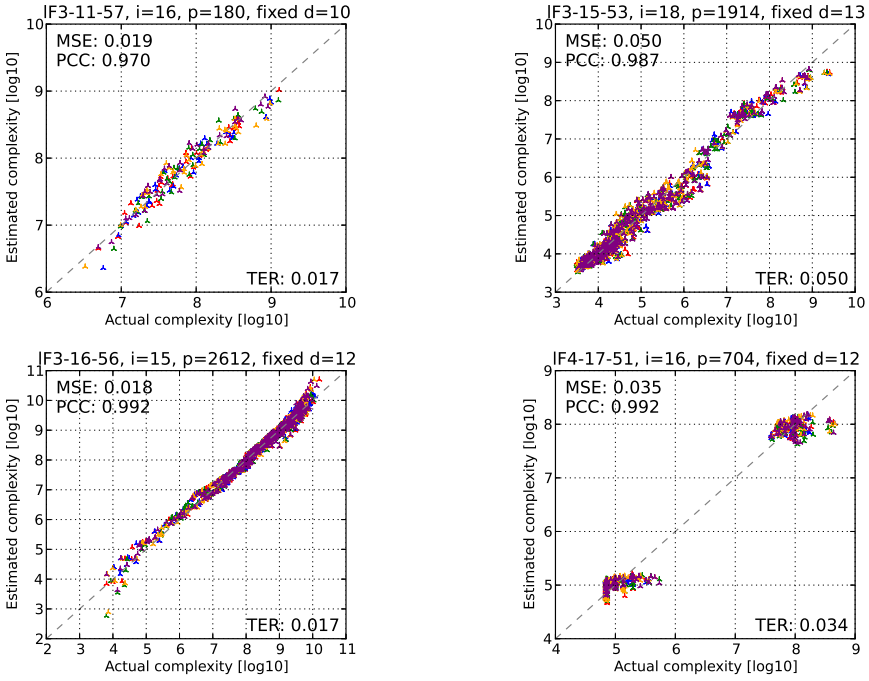


Figure 3.5: Select **per-instance** estimation results on **largeFam haplotyping** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

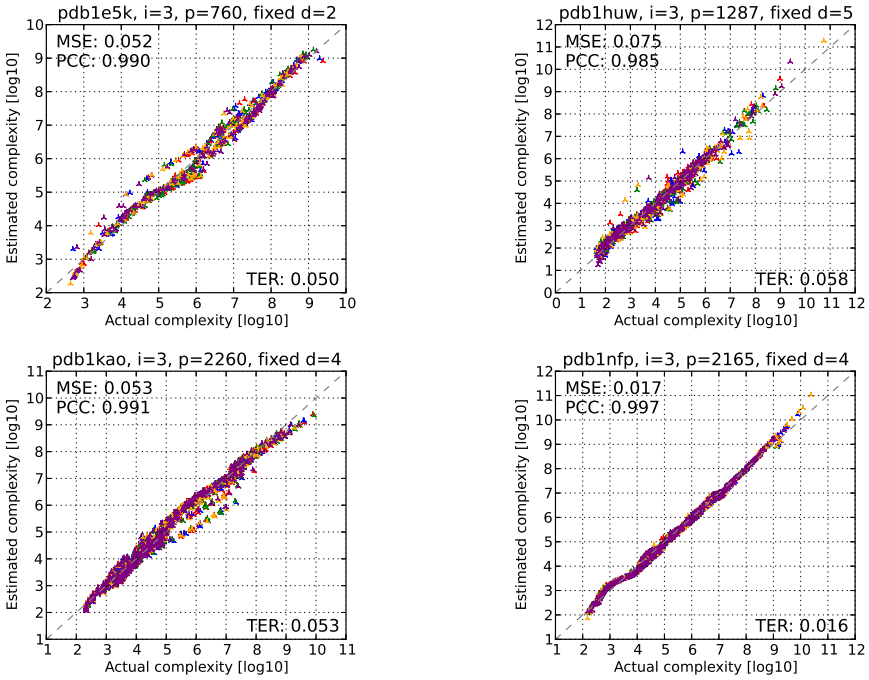


Figure 3.6: Select **per-instance** estimation results on **pdb side-chain prediction** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

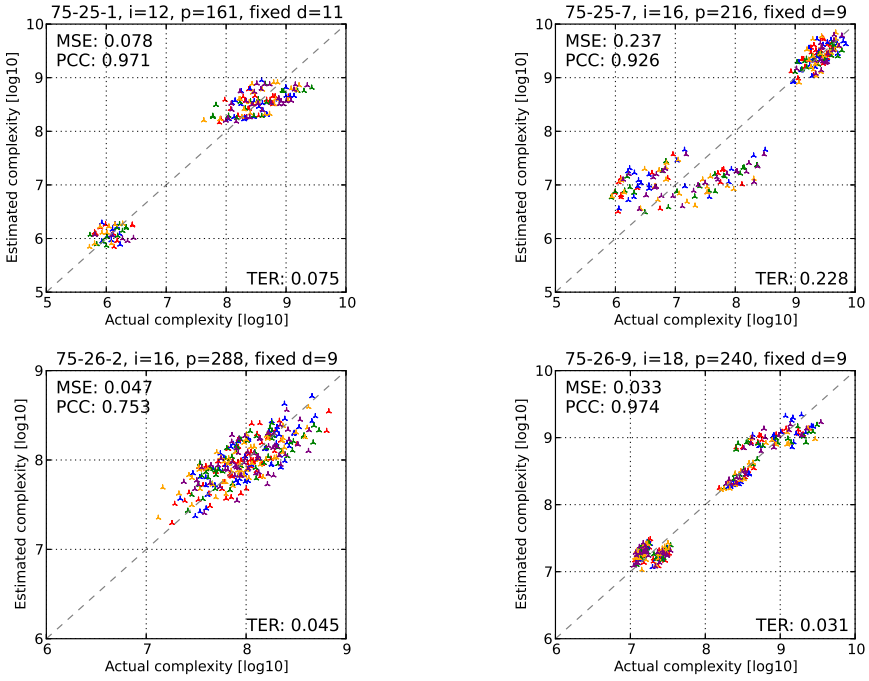


Figure 3.7: Select **per-instance** estimation results on **grid network** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

only ped34 (bottom left) seeing a slightly increased training/test error of 0.203 / 0.205 and slightly worse (but still very good) PCC of 0.902.

LargeFam instances. Figure 3.5 has results of per-instance learning for four largeFam haplotyping instances, with comparable training and test error values, all at 0.050 or below. Correlation coefficients are even better than for the pedigree class and lie between 0.970 (lF3-11-57) and 0.992 (lF3-16-56). We note that instance lF4-17-51 exhibits two distinct groups of subproblems with vastly different complexities (around 10^5 and 10^8 node expansions, respectively), which is very accurately reflected in the predictions as well.

Pdb instances. Pdb side-chain prediction instances, with per-instance estimation results shown in Figure 3.6, are interesting in the sense that the actual complexities of subproblems, even though all originating from the same search depth d, span a tremendous range (e.g., from approx. 10^2 to 10^{10} for pdb1nfp) – a reminder that subproblem structure alone is an insufficient indication for hardness. Nevertheless, the plots in Figure 3.6 indicate that we are able to learn models that capture the same wide range very accurately. In fact, we again observe very good error values of 0.075 (test error of pdb1huw) or lower. PCC results are exceptional as well, with value of 0.985 (pdb1huw) or higher, all the way to an impressive 0.997 for pdb1nfp.

Grid instances. Finally, Figure 3.7 plots exemplary results of per-instance estimation for grid network instances. We observe performance similar to the pedigree instances in Figure 3.4. Training and test error is typically below 0.08, with the exception of instance 75-25-7 (top right plot) where the test error is 0.237. In this case, however, we also observe that our learned model is very capable of picking out the "cluster" of complex subproblems in the top right corner of the plot. This is crucial in the context of parallel AOBB, where these complex subproblems can easily turn into bottlenecks with respect to parallel performance (cf. Chapter 4).

3.4.3.2 Overall Per-Instance Learning Results

We confirm that the above selected results are representative by consulting Tables 3.6 through 3.9 (pages 128–130), which show MSE, PCC, and TER for the complete set of 77 problem instances, as well as aggregated measures per problem class (where aggregate MSE and TER are averages weighted by the instances' subproblem counts, while aggregate PCC is a simple average). In particular, we note that the aggregate MSE for per-instance learning is below 0.1 for all four problem classes (0.081, 0.035, 0.056, and 0.095 for pedigree, largeFam, pdb, and grids, respectively). Secondly, the aggregate correlation coefficients, with values of 0.800, 0.944, 0.989, and 0.824, respectively, represent good to very good results as well.

Overall, we can therefore attest very convincing predictive performance of our learning approach on the level of per-instance learning, with generally excellent, low error and strong correlation coefficient values. Notable is also the apparent absence of overfitting, with virtually identical training and test error values across all instances (cf. also Figure 3.3). As mentioned initially, however, the obvious caveat of these positive results is the lack of direct applicability of instance-specific learning in the context of parallel AOBB.

3.4.4 Learning per Problem Class

This section considers "per-class" learning, where we learn a joint model for a specific problem class, which is more general than the per-instance approach discussed in the previous section. In the parallelization context, this translates to learning a separate model for each problem class we want to run parallel AOBB on, and reusing the learned model for new instances from that class – which renders this approach immediately more applicable in practice than per-instance learning in the previous section.

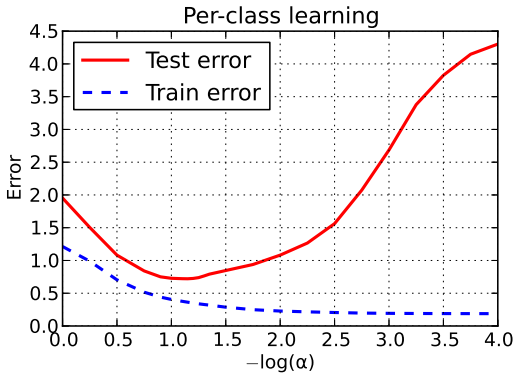


Figure 3.8: Training and test error on separate validation set as a function of regularization parameter α for **per-class** learning.

To evaluate the predictive performance of our regression scheme at this per-class level, we learn and evaluate models as follows: For a given problem instance, we draw subproblem samples from all other instances of the same problem class. We learn a model (Equation 3.10) using this training set and evaluate it on the original problem, whose subproblems thus become the test set. For instance, to perform estimation on a set of subproblems of pedigree13, we draw training samples from all other pedigree problem instances in our overall set – this lets pedigree13 appear as a "new" problem in terms of prediction. As a reminder, to compensate for the large variance in the number of subproblems samples available per instance (column p in Tables 3.2 through 3.5), we choose no more than 250 subproblems from each instance for our training set.

Intuitively, this approach should be more demanding than the previous per-instance level, since we are aiming to learn a model that covers several problem instances instead of just one. However, since we limit ourselves to a single problem class, it is natural to expect that these instances have certain things in common (problem structure or distribution of cost values, for instance), which impact AOBB's performance in similar ways.

Regularization. Following the methodology given in Section 3.4.2.2, the regularization parameter was set to $\alpha = 10^{-1.25} \approx 0.06$, which minimizes the test error in Figure 3.8. In

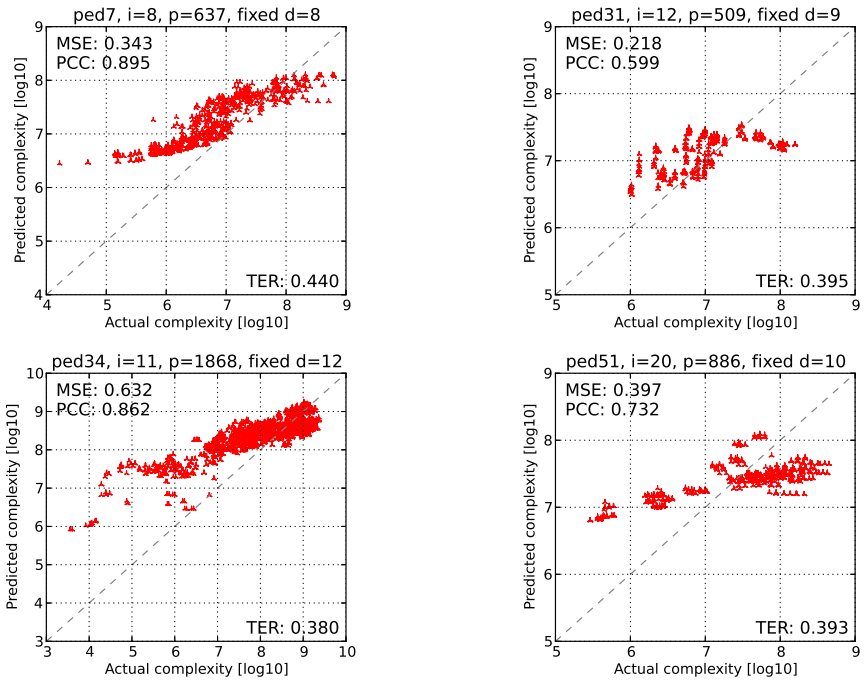


Figure 3.9: Select **per-class** estimation results on **pedigree linkage** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

contrast to Figure 3.3, we observe an increasing test error as regularization is decreased (i.e., as $-\log(\alpha)$ grows beyond 1.5), while the training error decreases further – a clear sign of overfitting of the learned model to the training set, as discussed in Section 3.3.1.2. We also note the distinct gap between training and test error and the generally higher error compared to Figure 3.3, indicating the increased hardness of the per-class learning task.

3.4.4.1 Select Per-Class Learning Results

Detailed results on a subset of problem instances are presented as scatter plots in Figures 3.9 through 3.12, with a more comprehensive set again available in Appendix A.2 (page 286). Results are still good, although somewhat deteriorated from per-instance learning in the previous Section 3.4.3. We elaborate in the following paragraphs.

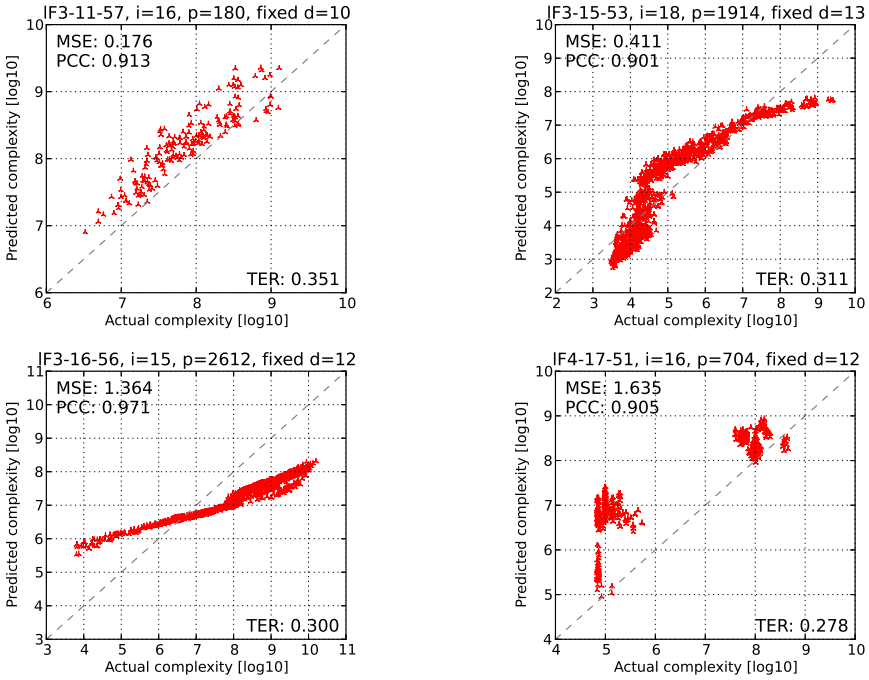


Figure 3.10: Select **per-class** estimation results on **largeFam haplotyping** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

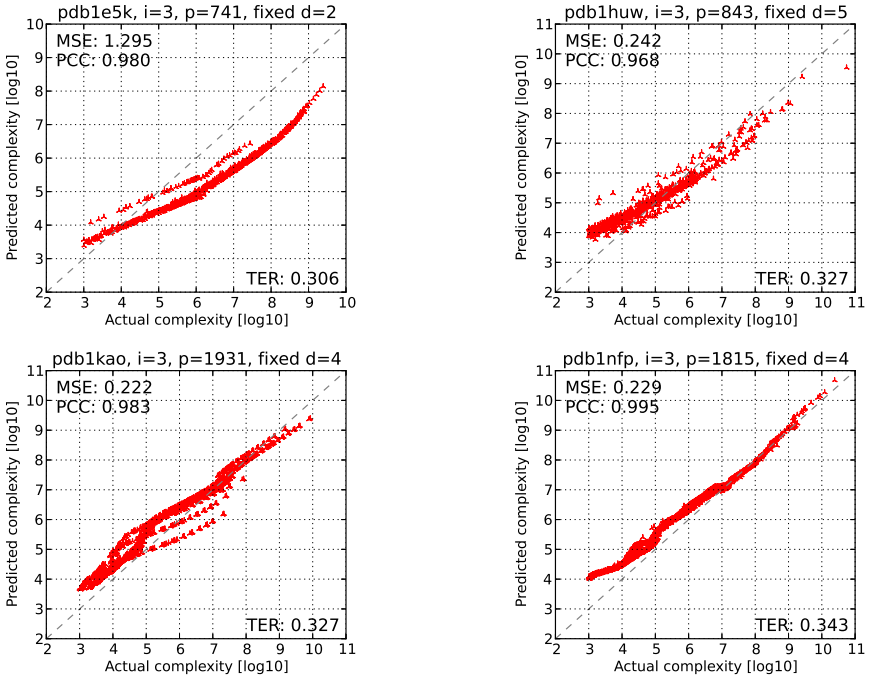


Figure 3.11: Select **per-class** estimation results on **pdb side-chain prediction** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

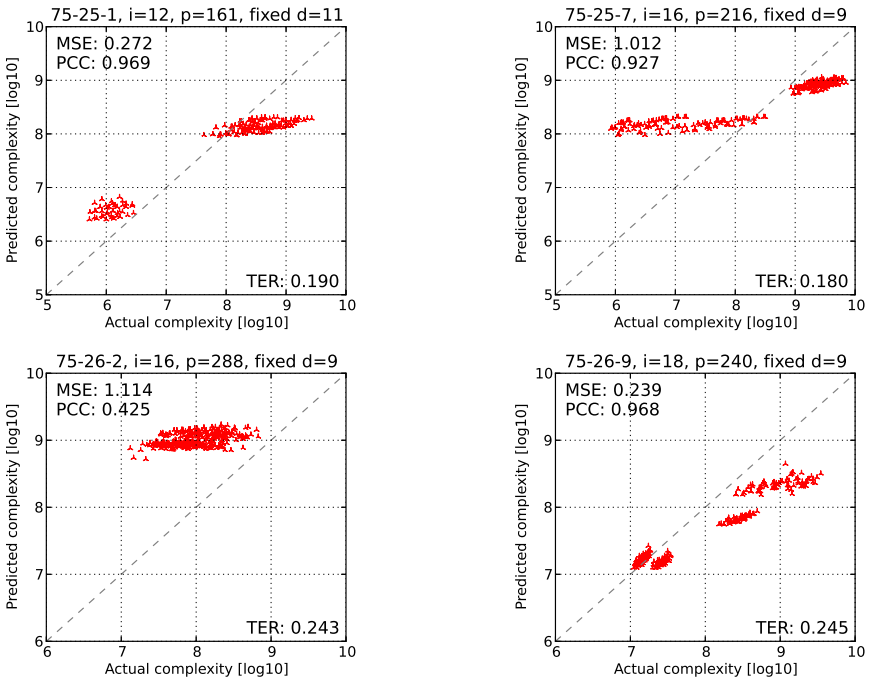


Figure 3.12: Select **per-class** estimation results on **grid network** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

Pedigree instances. Prediction results on four pedigree linkage instances are presented in Figure 3.9. Compared to per-instance learning the test error ("MSE") shows a measurable increase, for instance from 0.078 to 0.343 for instance ped7 or from 0.205 to 0.632 for ped34, which is also the highest error out of the four instances shown. We also note that the correlation coefficient decreases relative to per-class learning: ped7 and ped34 still exhibit good PCC values of 0.895 and 0.862, respectively, but ped31 drops to 0.599.

LargeFam instances. Figure 3.10 has scatter plots for four largeFam haplotyping instances. For lF3-11-57 and lF3-15-53 (top left and top right), performance is decent, with test MSE values of 0.176 and 0.411, respectively. However, for lF3-16-56 and lF4-17-51, the test error is quite a bit higher (1.364 and 1.635, respectively). Nevertheless, in all four cases we observe very good correlation coefficients (PCC) of over 0.9.

Pdb instances. Moving to pdb side-chain prediction instances in Figure 3.11, we similarly see an increased test error from per-instance prediction. For instance, the MSE value on pdb1huw increases to 0.242 (from 0.075, cf. Figure 3.6). pdb1e5k even sees an MSE of 1.295 (up from 0.052). In all cases, however, PCC values remain excellent, at 0.968 (pdb1huw) or higher.

Grid instances. Lastly, Figure 3.12 shows per-class prediction results on four grid instances. Again we find increased test error (MSE) compared to per-instance learning: 75-25-1 and 75-26-9, while increased, are still good with 0.272 and 0.239, respectively, while 75-25-7 and 75-26-2 both see their test MSE climb above 1. On the other hand, with the exception of 75-26-2, PCC values remain very good at above 0.9.

3.4.4.2 Overall Per-Class Learning Results

As before, we confirm our findings by consulting the "per class" columns of Tables 3.6 through 3.9 (pages 128–130). Within each problem class we can identify a handful of outliers with MSE values greater than 1 (though pdb1rss exhibits the only MSE greater than 2 with 2.879) but also numerous very good results with low test errors close to 0.1 or 0.2. Aggregate MSE values are therefore quite satisfactory with 0.615, 0.886, 0.667, and 0.617 for pedigree, largeFam, pdb, and grid instances, respectively. Similarly, aggregate correlation coefficients decreased from per-instance learning, but still acceptable, with values of 0.630, 0.846, 0.977, and 0.683, respectively.

3.4.5 Learning across Problem Classes

We now consider the level of "cross-class" learning, where we learn a single lasso regression model following Equation 3.10 across all problem classes under consideration and reuse this

model for subsequent new problem instances from either of these classes. In the context of parallelization, this puts us in the appealing position of maintaining only a single parallel solver (with a "universal" complexity model) that can be deployed across problem classes, assuming we have knowledge of these problem classes to be considered and can acquire sufficient sample subproblems from each of them.

In order to evaluate performance at this cross-class level, we conduct the following procedure: Given a problem instance to be evaluated, we take the training set to be subproblems from all other instances, regardless of problem class. For instance, to evaluate performance on pedigree13, our training set will be made up of subproblems of all other pedigree instances, as well as all largeFam, pdb, and grid instances. As for per-class learning (cf. Section 3.4.4), proceeding in this way lets pedigree13 appear as a "new" instance to the learned model.

Compared to learning separate models per problem class in the previous section, this approach can be seen as a more demanding learning task since it encompasses a larger variety of problems, with potentially more diverse structure and different runtime behavior. At the same time, however, we may be able to extract characteristics that impact runtime complexity but which were not evident within just a single problem class. In addition, considering several problem classes increases the number of training samples available to learn our model from, which is another potential advantage.

Regularization. We set the regularization parameter for cross-class learning at $\alpha = 10^{-2.25} \approx 0.06$, according to the minimum of the test error in Figure 3.13, obtained as previously outlined in Section 3.4.2.2. Compared to Figure 3.8 (for per-class learning), we note the generally larger training error, which can be taken to confirm a more varied learning domain. However, it also seems that models learned across classes don't overfit as quickly as those learned per class.

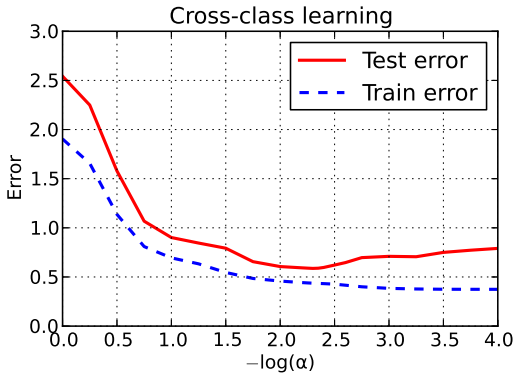


Figure 3.13: Training and test error on separate validation set as a function of regularization parameter α for **cross-class** learning.

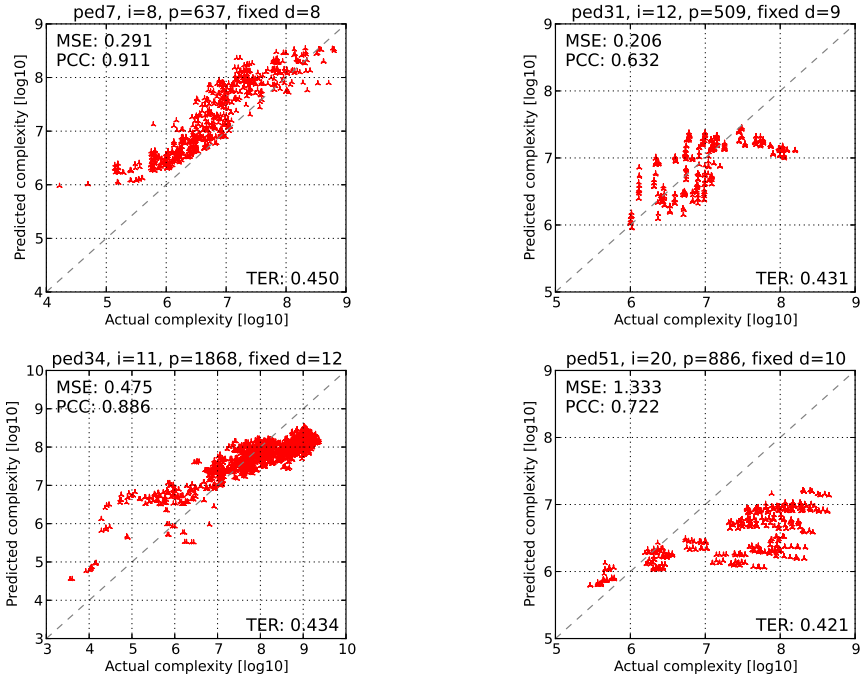


Figure 3.14: Select **cross-class** estimation results on **pedigree linkage** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

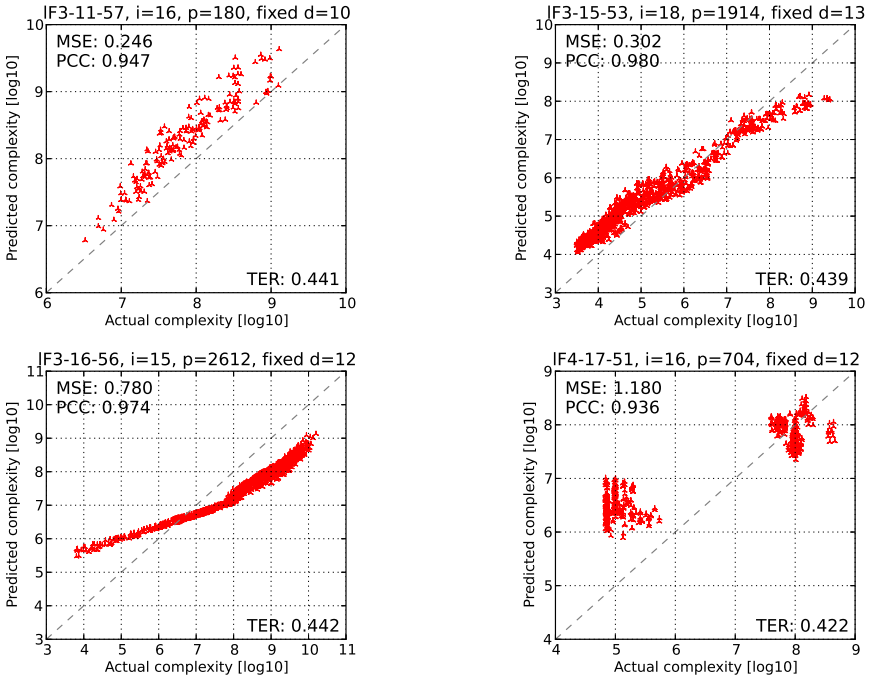


Figure 3.15: Select **cross-class** estimation results on **largeFam haplotyping** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

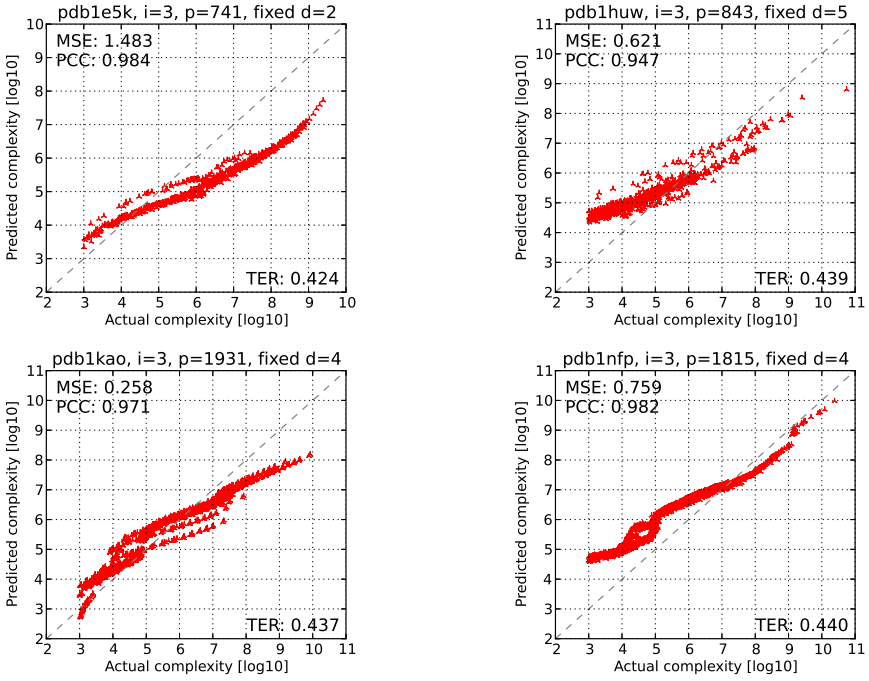


Figure 3.16: Select **cross-class** estimation results on **pdb side-chain prediction** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

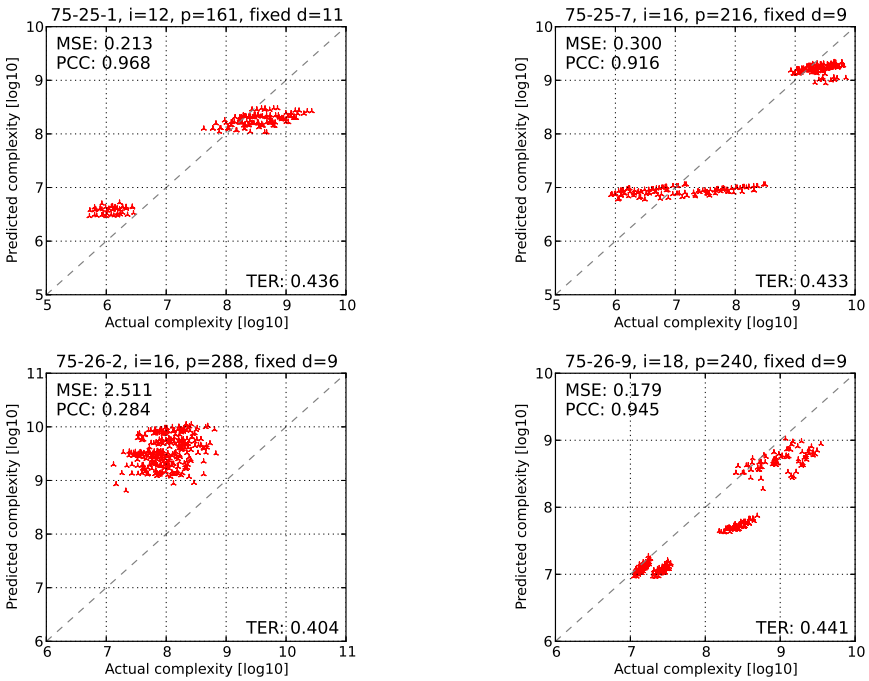


Figure 3.17: Select **cross-class** estimation results on **grid network** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

3.4.5.1 Select Cross-Class Learning Results

As in the previous two sections, we plot detailed estimation results on a subset of problem instances in Figures 3.14 through 3.17, with additional plots in Appendix A.3 (page 287). Compared to per-class learning in the previous section, we observe similar and in some cases even slightly improved results. The following paragraphs provide more detail.

Pedigree instances. Scatter plots of actual and estimated complexity on four pedigree linkage instances are shown in Figure 3.14. On three out of the four instances we actually find that the test error (MSE) improves – from 0.343, 0.218, and 0.632 (per-class learning) to 0.291, 0.206, and 0.475 (cross-class) on ped7, ped31, and ped34, respectively. Only on ped51 do we see an increase in error to 1.333. Similarly, we note better correlation coefficients in the present set of results, although not quite at the level of per-instance learning – for

instance, here we obtain a PCC of 0.911 for ped7, which was 0.932 with per-instance and 0.895 with per-class learning.

LargeFam instances. Select cross-class learning results on largeFam haplotyping instances are plotted in Figure 3.15. Again we see the test MSE improve on three instances relative to per-class learning. Notably, lF3-16-56 improves from 1.364 in Figure 3.10 to 0.780 here. The only decline in MSE is on lF3-11-57, which moves slightly from 0.176 to 0.246. PCC values, on the other hand, go up on all four instances shown here, most notably on lF3-15-53 from 0.901 (per-class learning) to 0.980 (cross-class learning).

Pdb instances. Figure 3.16 has detailed results on four pdb side-chain prediction instances. For this problem class we see a slight increase in MSE, compared per-class learning, on all four instances shown. Test error on pdb1kao moves only from 0.222 to 0.258 for cross-class learning, with the largest increase in case of pdb1nfp (0.229 to 0.759). Correlation coefficients remain largely very similar when moving to the more general cross-class learning, with only minor changes.

Grid instances. Finally, we contrast actual and predicted subproblem complexities on four grid network instances in Figure 3.17. Again we see improved performance relative to per-class learning on three of the instances – for instance from 1.012 MSE per-class to 0.300 here on 75-25-7. 75-26-2, however, takes a sizable hit in prediction performance and declines to a test error of 2.511. The latter example, with a relatively small range of actual complexities, also yields by far the worst PCC of 0.284 (down from 0.425 in per-class learning), while the other three example in Figure 3.17 produce very good correlation coefficients above 0.9.

3.4.5.2 Overall Cross-Class Learning Results

The above observations can be confirmed via the complete set of results in Tables 3.6 through 3.9 (pages 128–130), specifically consulting the cross-class column. We see aggregate MSE

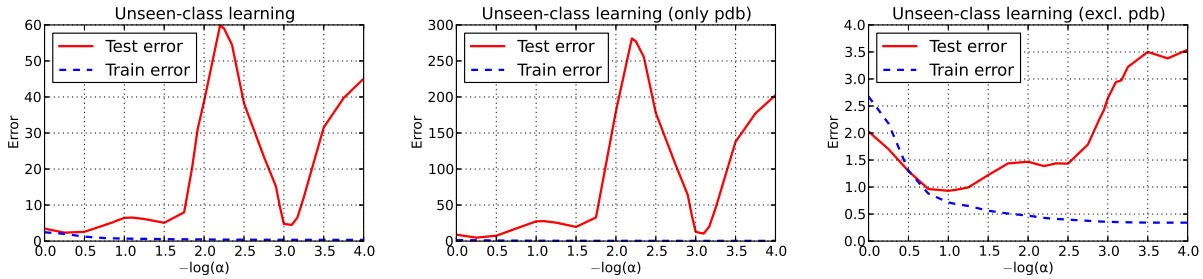
values for the four problem classes of 0.568, 0.499, 0.758, and 0.749 (pedigree, largeFam, pdb, and grids, respectively) – notably all well below 1. In particular, the aggregate test error (MSE) for pedigree and largeFam problems is below the values observed for per-class learning (0.615 and 0.886, respectively), while MSE increased slightly for pdb and grids (from 0.667 and 0.617, respectively). The average PCC follows the same pattern, improving for pedigree and largeFam instances (to 0.651 and 0.894, respectively) and decreasing a bit for pdb and grid problems (to 0.938 and 0.643, respectively). Overall, we can attest satisfactory prediction performance, comparable to per-class learning, with the added benefit of only requiring one complexity model across multiple problem classes.

3.4.6 Learning for Unseen Problem Classes

This section investigates the final, most general and most challenging level of learning defined in Section 3.3.4. At this level we aim to predict complexities for problems from a previously unseen problem class. In a parallelization context having this kind of complexity model would obviously be very convenient, since it would allow us to apply parallel AOBB "blindly" to problem instances from any class, known or unknown – without a doubt a very ambitious goal.

In terms of the terminology introduced in Section 3.3.1, we consider that training and test data are drawn from disjoint sample sets $\mathcal{X}$ and $\mathcal{X}'$, which can be viewed as a form of transfer learning (cf. Section 3.3.4 and [93]). In our practical evaluation we achieve this by systematically withholding samples of a given problem class from the training set. For instance, in order to assess the prediction performance on pedigree13 (the test set), we compile a training set by drawing subproblems from largeFam, pdb, and grid instances.

Regularization. Picking a value for the regularization parameter α is not as straightforward as before, reflecting the ambitious nature of the learning task at hand. In particular, we



(a) All instances. **(b)** Only pdb instances. **(c)** Excluding pdb instances.

Figure 3.18: Training and test error on separate validation set as a function of regularization parameter α for **cross-class** learning.

observe a large intermediate peak in the test error of Figure 3.18a, which shows the average error across all instances. Upon closer inspection, we note that this peak is entirely due to the pdb side-chain prediction instances, for which the average error is shown in Figure 3.18b – the model learned from pedigree, largeFam, and grid instances does not generalize to pdb problems well at all. On non-pdb instances (Figure 3.18c), on the other hand, we find behavior in line with earlier per-class and cross-class learning results (Figures 3.8 and 3.13); namely, both training and test error decrease initially with shrinking α, before overfitting sets in at around $-\log(\alpha) = -1.5$ and the test error deteriorates again. We therefore choose a regularization parameter of $\alpha = 10^{-1} = 0.1$, fully expecting that results on pdb side-chain prediction instances will be disappointing.

3.4.6.1 Select Unseen-Class Learning Results

Figures 3.19 through 3.22 show results for select problem instances, grouped by problem class, with additional plots in Appendix A.4 (page 287). We elaborate on performance details in the following paragraphs.

Pedigree instances. Results for four pedigree linkage problem instances are plotted in Figure 3.19. In comparison with cross-class learning in Section 3.4.5, we actually observe an improved MSE in three cases, most notably ped51 which goes from 1.333 (Fig. 3.14) to

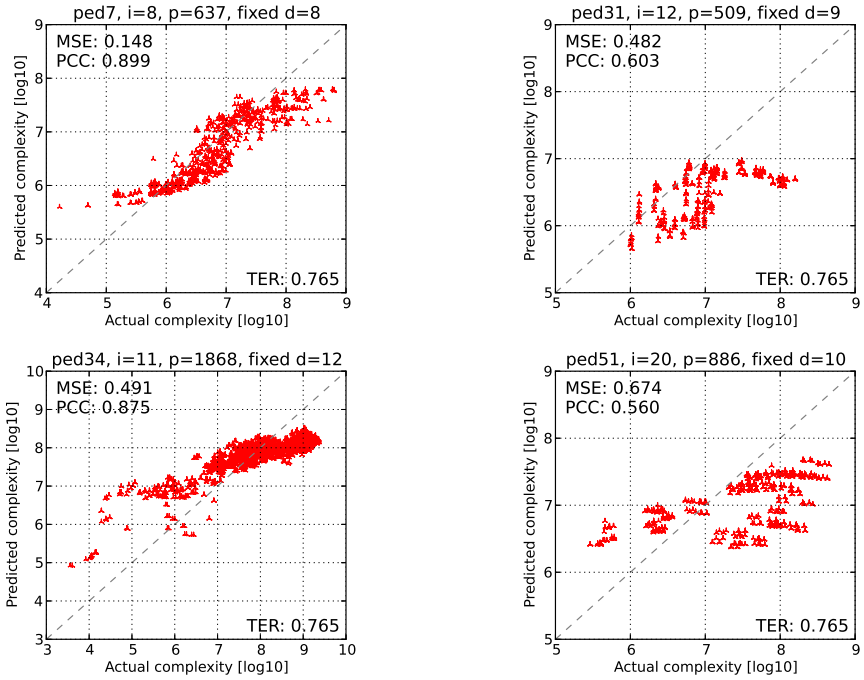


Figure 3.19: Select **unseen-class** estimation results on **pedigree linkage** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

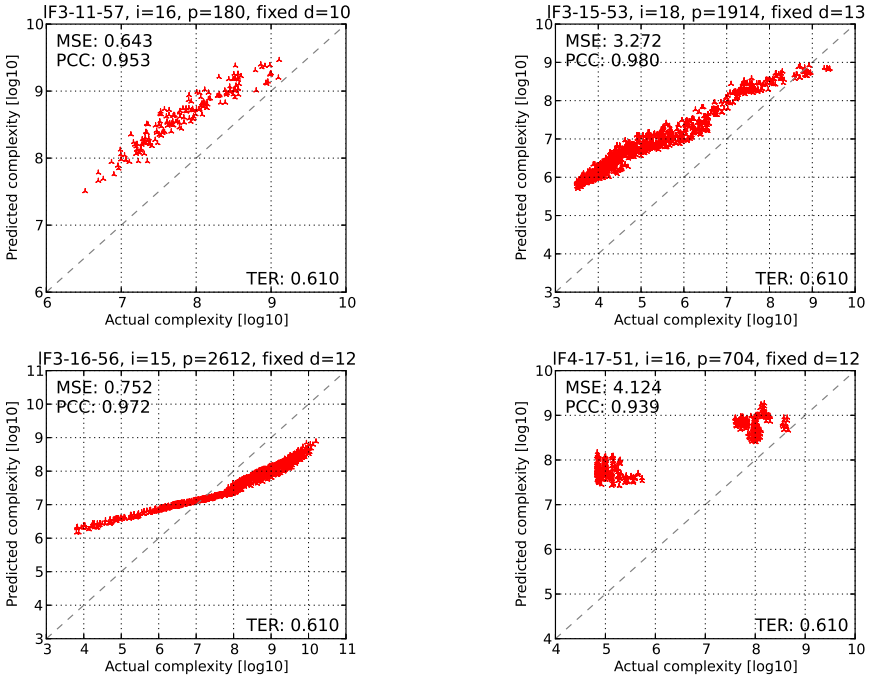


Figure 3.20: Select **unseen-class** estimation results on **largeFam haplotyping** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

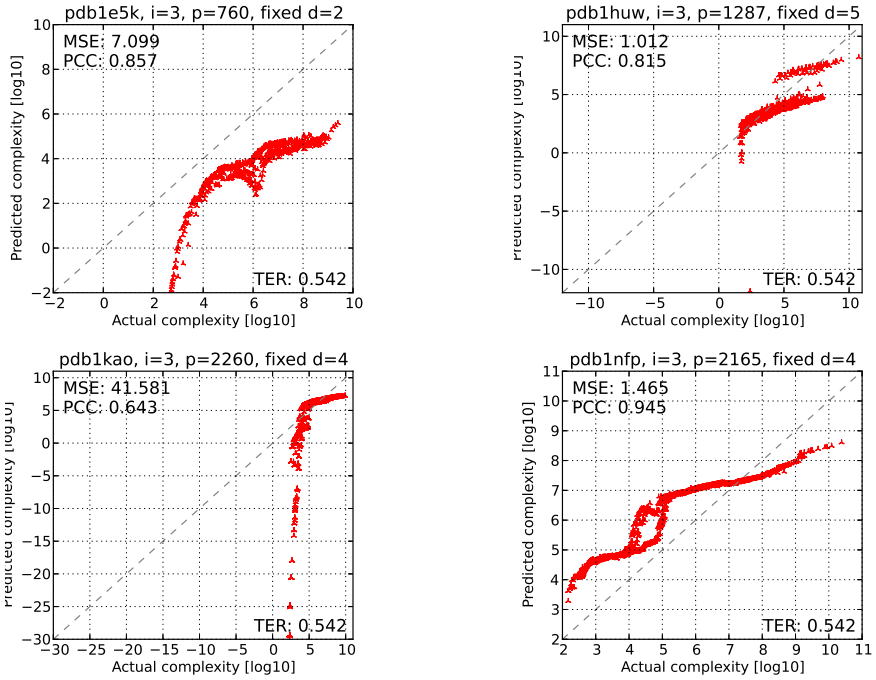


Figure 3.21: Select **unseen-class** estimation results on **side-chain prediction** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

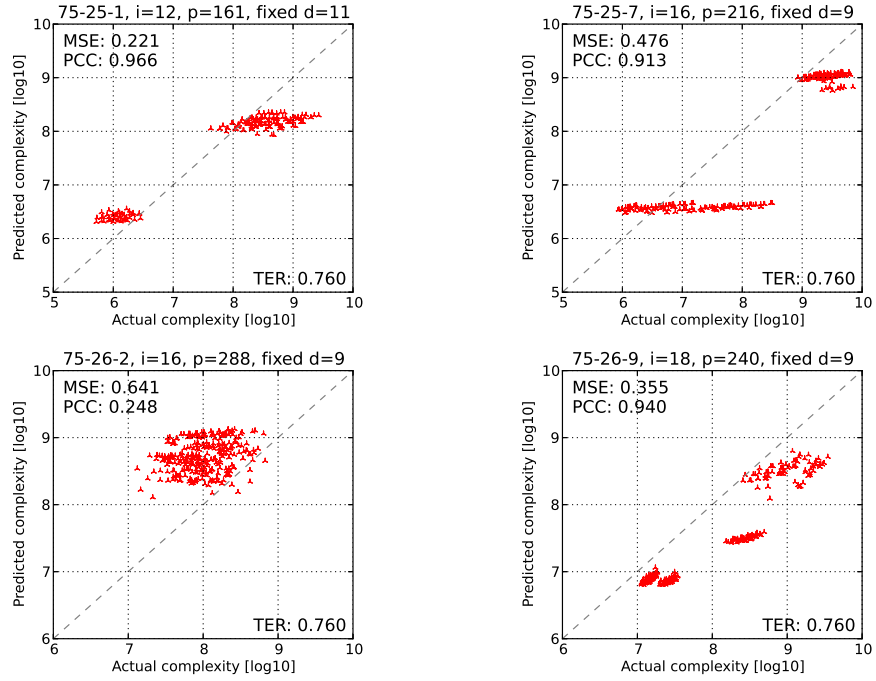


Figure 3.22: Select **unseen-class** estimation results on **grid network** problem instances. Plotted are actual subproblem complexities against predicted ones. Also specified are training error TER, test error MSE, and correlation coefficient PCC.

0.674; only on ped31 does the MSE increase to 0.482 (from 0.206). Correlation coefficients do suffer slightly, with ped51 yielding the weakest results of 0.560, down from 0.722 in Figure 3.14. Ped7 and ped34 retain a relatively high PCC around of almost 0.9.

LargeFam instances.. Figure 3.20 has four scatter plots for largeFam haplotyping instances. Here we find that MSE values deteriorate compared to cross-class learning on three of the instances, most notably on lF3-15-53 and lF4-17-51 from 0.302 and 1.180 (cf. Fig. 3.16) to 3.272 and 4.124, respectively. In both cases we find that the predictions systematically overestimate the actual complexities, but are consistent (relative to each other) within each problem instance. This is confirmed by the excellent PCC results well above 0.9, similar to what we observed for cross-class learning, if not minimally improved (e.g., lF3-11-57 at 0.953, up from 0.947).

Pdb instances. A set of scatter plots for predictions on four pdb side-chain prediction instances is shown in Figure 3.21. Recalling our observations from the regularization parameter selection above, we do indeed find some very negative results in terms of MSE test error. pdb1kao stands out with an MSE of 41.581 (up from 0.258 in cross-class learning, Figure 3.16), but pdb1e5k is also quite bad with a test MSE of 7.099 (up from 1.483). Notably, in both cases we see a number of negative predicted values for the simplest subproblems (in terms of actual complexity), which is not very meaningful as a log complexity (i.e., 10^{-30} node expansions in case of pdb1kao) – we could lower bound predictions at 0, but that would not address the inherent inaccuracy of the learned model. The measured correlation coefficients also suffer in comparison to cross-class prediction, particularly because of severe underestimates like in the case of ped1kao (PCC of 0.643, down from 0.971).

Grid instances. The last problem class we consider for this most general level of learning are grid networks, with four scatter plots shown in Figure 3.22. After the disappointing pdb results, here we obtain good results again. The MSE test error improves significantly for 75-26-2, from 2.511 (cf. Fig. 3.17) to 0.641. It increases slightly for the other three instances,

but stays below 0.5 in each case. Finally, correlation coefficients (PCC) are very good at above 0.9 and very close to cross-class learning, with the exception of 75-26-2 (PCC 0.248, from 0.284).

3.4.6.2 Overall Unseen-Class Learning Results

To put the described results into a wider context, we again consult Tables 3.6 through 3.9, which contain MSE and PCC measures for all instances, as well as aggregates per problem class. We find this most challenging level of transfer learning for unseen problem classes crucially reflected in the pdb results (Table 3.8), with a number of very high individual MSE test errors and an aggregate of 27.373. The PCC for pdb instance also drops to 0.804 (which is, however, still very respectable).

On the remaining three problem classes, results are fairly good and in some cases quite close to cross-class learning examined in Section 3.4.5. The aggregate MSE for pedigree instances, for instances, increases only slightly, from 0.568 for cross-class learning to 0.703 here (PCC remains similar with 0.643). For largeFam haplotyping instances in Table 3.7, the aggregate MSE increases quite a bit compared to cross-class learning, from 0.499 to 1.246, but not at the expense of average PCC (0.908, from 0.894). Finally, grid networks in Table 3.9 actually see their aggregate MSE improve from cross-class learning with a value of 0.532 (from 0.749), with again a similar PCC of 0.633.

3.4.7 Summary of Results

We have trained and evaluated our proposed regression model (Equation 3.10) on the four levels of learning laid out in Section 3.3.4, trading off between the wider applicability of the learned models and the challenges of capturing increasingly general sample sets. Results

			per instance			per class			across classes			unseen class		
instance	i	d	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER
ped7	7	9	0.050	0.956	0.049	0.220	0.920	0.440	0.096	0.939	0.450	0.304	0.930	0.765
ped7	8	8	0.078	0.932	0.076	0.343	0.895	0.440	0.291	0.911	0.450	0.148	0.899	0.765
ped7	6	9	0.054	0.856	0.053	0.162	0.807	0.440	0.125	0.830	0.450	0.801	0.820	0.765
ped9	7	11	0.019	0.612	0.018	0.274	0.477	0.439	0.202	0.462	0.449	0.063	0.478	0.765
ped9	8	10	0.016	0.598	0.014	0.174	0.375	0.439	0.181	0.375	0.449	0.059	0.347	0.765
ped9	6	11	0.015	0.730	0.015	0.078	0.602	0.439	0.271	0.631	0.449	0.609	0.637	0.765
ped13	9	10	0.046	0.773	0.045	0.262	0.479	0.420	0.137	0.518	0.445	0.586	0.494	0.765
ped13	10	9	0.008	0.955	0.008	0.118	0.571	0.420	0.759	0.602	0.445	0.238	0.577	0.765
ped13	8	10	0.037	0.804	0.036	0.458	0.496	0.420	0.093	0.547	0.445	0.804	0.510	0.765
ped19	16	6	0.112	0.964	0.109	1.711	0.893	0.359	0.790	0.904	0.429	0.721	0.896	0.765
ped19	15	6	0.154	0.896	0.152	1.094	0.584	0.359	1.084	0.606	0.429	1.446	0.568	0.765
ped31	11	10	0.050	0.841	0.049	0.768	0.460	0.395	0.835	0.576	0.431	1.656	0.581	0.765
ped31	12	9	0.046	0.925	0.043	0.218	0.599	0.395	0.206	0.632	0.431	0.482	0.603	0.765
ped31	10	10	0.067	0.735	0.066	1.463	0.645	0.395	1.362	0.695	0.431	2.711	0.678	0.765
ped33	4	8	0.008	0.884	0.006	0.687	0.874	0.416	1.103	0.885	0.438	0.363	0.873	0.765
ped33	5	6	0.004	0.900	0.002	0.313	0.898	0.416	0.532	0.854	0.438	0.073	0.908	0.765
ped34	12	11	0.161	0.935	0.156	1.012	0.882	0.380	0.400	0.897	0.434	0.467	0.883	0.765
ped34	11	12	0.205	0.902	0.203	0.632	0.862	0.380	0.475	0.886	0.434	0.491	0.875	0.765
ped34	10	12	0.223	0.903	0.222	0.758	0.832	0.380	0.696	0.877	0.434	0.711	0.869	0.765
ped39	4	6	0.118	0.855	0.091	0.384	0.455	0.391	0.444	0.378	0.438	0.791	0.427	0.765
ped39	5	5	0.147	0.673	0.111	0.571	0.344	0.391	0.421	0.250	0.438	0.676	0.183	0.765
ped39	3	6	0.305	0.640	0.270	1.893	0.094	0.391	0.533	0.106	0.438	2.178	0.089	0.765
ped41	10	10	0.043	0.902	0.043	0.446	0.824	0.430	0.187	0.854	0.451	0.107	0.829	0.765
ped41	11	9	0.030	0.944	0.029	0.499	0.838	0.430	0.371	0.872	0.451	0.122	0.857	0.765
ped41	9	10	0.032	0.912	0.031	0.414	0.846	0.430	0.179	0.878	0.451	0.071	0.878	0.765
ped44	6	9	0.010	0.664	0.010	0.372	0.583	0.392	0.040	0.615	0.450	0.419	0.605	0.765
ped44	7	8	0.043	0.946	0.042	1.523	0.945	0.392	0.360	0.947	0.450	0.122	0.946	0.765
ped44	5	9	0.012	0.690	0.012	0.206	0.646	0.392	0.202	0.662	0.450	0.842	0.654	0.765
ped51	21	10	0.102	0.816	0.100	0.418	0.686	0.393	1.561	0.616	0.421	0.619	0.611	0.765
ped51	20	10	0.080	0.928	0.079	0.397	0.732	0.393	1.333	0.722	0.421	0.674	0.560	0.765
ped51	19	10	0.066	0.849	0.063	1.157	0.816	0.393	2.380	0.806	0.421	1.326	0.800	0.765
Aggregate			*0.081*	*0.800*	*0.079*	*0.615*	*0.630*	*0.406*	*0.568*	*0.651*	*0.439*	*0.703*	*0.643*	*0.765*

Table 3.6: Full complexity estimation results for **pedigree linkage** instances at different levels of learning. Listed are test error (MSE), correlation coefficient (PCC), and training error (TER) per instance as well as aggregated across instances.

			per instance			per class			across classes			unseen class		
instance	i	d	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER
lF3-11-57	17	10	0.023	0.961	0.021	0.062	0.922	0.351	0.087	0.952	0.441	0.539	0.949	0.610
lF3-11-57	16	10	0.019	0.970	0.017	0.176	0.913	0.351	0.246	0.947	0.441	0.643	0.953	0.610
lF3-11-59	16	8	0.010	0.980	0.009	0.368	0.952	0.358	0.492	0.957	0.434	0.568	0.961	0.610
lF3-11-59	15	8	0.009	0.990	0.008	0.218	0.978	0.358	0.078	0.969	0.434	0.165	0.974	0.610
lF3-13-58	18	8	0.033	0.969	0.029	0.344	0.883	0.356	0.373	0.927	0.432	0.243	0.941	0.610
lF3-13-58	17	8	0.043	0.961	0.039	0.233	0.894	0.356	0.300	0.932	0.432	0.279	0.943	0.610
lF3-15-53	18	13	0.050	0.987	0.050	0.411	0.901	0.311	0.302	0.980	0.439	3.272	0.980	0.610
lF3-15-53	17	13	0.051	0.988	0.051	0.682	0.862	0.311	0.415	0.957	0.439	3.234	0.959	0.610
lF3-15-55	16	10	0.017	0.920	0.017	1.823	0.258	0.289	0.362	0.540	0.443	0.073	0.704	0.610
lF3-15-55	15	10	0.013	0.846	0.013	0.438	0.307	0.289	0.112	0.702	0.443	0.022	0.806	0.610
lF3-15-59	19	9	0.069	0.970	0.067	0.214	0.961	0.353	0.321	0.967	0.447	0.731	0.968	0.610
lF3-15-59	18	9	0.045	0.977	0.044	0.136	0.969	0.353	0.295	0.971	0.447	0.647	0.974	0.610
lF3-16-56	16	12	0.030	0.990	0.030	0.739	0.964	0.300	0.310	0.977	0.442	0.541	0.978	0.610
lF3-16-56	15	12	0.018	0.992	0.017	1.364	0.971	0.300	0.780	0.974	0.442	0.752	0.972	0.610
lF3-16-56	16	10	0.014	0.994	0.013	0.460	0.959	0.300	0.217	0.980	0.442	0.391	0.981	0.610
lF4-12-50	14	6	0.063	0.854	0.056	1.716	0.833	0.303	0.482	0.843	0.445	0.126	0.837	0.610
lF4-12-50	13	6	0.061	0.804	0.060	1.958	0.728	0.303	0.352	0.740	0.445	0.114	0.746	0.610
lF4-12-55	14	9	0.095	0.683	0.093	0.121	0.664	0.334	0.191	0.670	0.416	0.588	0.672	0.610
lF4-12-55	13	9	0.037	0.994	0.037	0.667	0.987	0.334	2.668	0.926	0.416	3.906	0.907	0.610
lF4-17-51	16	12	0.035	0.992	0.034	1.635	0.905	0.278	1.180	0.936	0.422	4.124	0.939	0.610
lF4-17-51	16	11	0.033	0.994	0.030	1.892	0.950	0.278	1.412	0.933	0.422	3.978	0.934	0.610
Aggregate			*0.035*	*0.944*	*0.035*	*0.886*	*0.846*	*0.322*	*0.499*	*0.894*	*0.436*	*1.246*	*0.908*	*0.610*

Table 3.7: Full complexity estimation results for **largeFam haplotyping** instances at different levels of learning. Listed are test error (MSE), correlation coefficient (PCC), and training error (TER) per instance as well as aggregated across instances.

			per instance			per class			across classes			unseen class		
instance	i	d	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER
pdb1a6m	3	3	0.041	0.990	0.039	0.201	0.988	0.334	0.282	0.967	0.443	48.594	0.871	0.542
pdb1duw	3	3	0.015	0.997	0.015	0.200	0.991	0.329	0.414	0.974	0.441	2.307	0.843	0.542
pdb1e5k	3	2	0.052	0.990	0.050	1.295	0.980	0.306	1.483	0.984	0.424	7.099	0.857	0.542
pdb1f9i	3	2	0.070	0.978	0.069	1.598	0.964	0.296	1.295	0.903	0.434	24.906	0.651	0.542
pdb1ft5	3	3	0.017	0.995	0.016	0.173	0.985	0.331	0.670	0.976	0.442	8.409	0.950	0.542
pdb1hd2	3	2	0.018	0.996	0.018	0.067	0.989	0.326	0.421	0.939	0.443	127.322	0.871	0.542
pdb1huw	3	5	0.075	0.985	0.058	0.242	0.968	0.327	0.621	0.947	0.439	1.012	0.815	0.542
pdb1kao	3	4	0.053	0.991	0.053	0.222	0.983	0.327	0.258	0.971	0.437	41.581	0.643	0.542
pdb1nfp	3	4	0.017	0.997	0.016	0.229	0.995	0.343	0.759	0.982	0.440	1.465	0.945	0.542
pdb1rss	3	4	0.178	0.980	0.171	2.879	0.929	0.201	1.475	0.714	0.430	7.456	0.697	0.542
pdb1vhh	3	2	0.100	0.980	0.098	0.148	0.976	0.337	0.377	0.963	0.442	3.607	0.698	0.542
Aggregate			*0.056*	*0.989*	*0.054*	*0.667*	*0.977*	*0.314*	*0.758*	*0.938*	*0.438*	*27.373*	*0.804*	*0.542*

Table 3.8: Full complexity estimation results for **pdb side-chain prediction** instances at different levels of learning. Listed are test error (MSE), correlation coefficient (PCC), and training error (TER) per instance as well as aggregated across instances.

			per instance			per class			across classes			unseen class		
instance	i	d	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER
75-25-1	14	10	0.066	0.963	0.058	0.215	0.915	0.190	0.248	0.862	0.436	0.261	0.866	0.760
75-25-1	12	11	0.078	0.971	0.075	0.272	0.969	0.190	0.213	0.968	0.436	0.221	0.966	0.760
75-25-1	12	8	0.086	0.653	0.072	1.039	0.475	0.190	0.895	0.454	0.436	1.692	0.446	0.760
75-25-7	18	7	0.016	0.995	0.015	1.681	0.959	0.180	0.457	0.971	0.433	0.136	0.967	0.760
75-25-7	16	8	0.253	0.921	0.228	1.111	0.926	0.180	0.305	0.909	0.433	0.572	0.907	0.760
75-25-7	16	9	0.237	0.926	0.228	1.012	0.927	0.180	0.300	0.916	0.433	0.476	0.913	0.760
75-26-2	20	8	0.041	0.753	0.017	0.671	0.212	0.243	1.678	0.143	0.404	0.227	0.095	0.760
75-26-2	16	8	0.041	0.751	0.038	0.746	0.655	0.243	2.318	0.545	0.404	0.331	0.517	0.760
75-26-2	16	9	0.047	0.753	0.045	1.114	0.425	0.243	2.511	0.284	0.404	0.641	0.248	0.760
75-26-9	16	8	0.033	0.967	0.029	0.580	0.877	0.245	0.624	0.800	0.441	1.201	0.814	0.760
75-26-9	16	9	0.052	0.949	0.047	0.350	0.904	0.245	0.501	0.816	0.441	0.825	0.824	0.760
75-26-9	18	9	0.033	0.974	0.031	0.239	0.968	0.245	0.179	0.945	0.441	0.355	0.940	0.760
75-26-10	20	11	0.094	0.584	0.087	0.564	0.308	0.203	0.204	0.262	0.446	0.184	0.254	0.760
75-26-10	16	11	0.130	0.677	0.125	0.380	0.294	0.203	0.481	0.320	0.446	0.505	0.312	0.760
75-26-10	16	10	0.131	0.521	0.126	0.162	0.439	0.203	0.605	0.451	0.446	0.841	0.424	0.760
Aggregate			*0.095*	*0.824*	*0.088*	*0.617*	*0.683*	*0.211*	*0.749*	*0.643*	*0.432*	*0.532*	*0.633*	*0.760*

Table 3.9: Full complexity estimation results for **grid network** instances at different levels of learning. Listed are test error (MSE), correlation coefficient (PCC), and training error (TER) per instance as well as aggregated across instances.

	per instance			per class			across classes			unseen class		
class	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER	MSE	PCC	TER
Pedigree	0.081	0.800	0.079	0.615	0.630	0.406	0.568	0.651	0.439	0.703	0.643	0.765
LargeFam	0.035	0.944	0.035	0.886	0.846	0.322	0.499	0.894	0.436	1.246	0.908	0.610
Pdb	0.056	0.989	0.054	0.667	0.977	0.314	0.758	0.938	0.438	27.373	0.804	0.542
Grids	0.095	0.824	0.088	0.617	0.683	0.211	0.749	0.643	0.432	0.532	0.633	0.760
Overall	*0.061*	*0.866*	*0.059*	*0.721*	*0.742*	*0.363*	*0.587*	*0.751*	*0.437*	*6.453*	*0.731*	*0.688*

Table 3.10: Summary of complexity estimation results at different levels of learning. Listed are test error (MSE), correlation coefficient (PCC), and training error (TER), aggregated per problem class and across all classes.

have been evaluated both in terms of their prediction error (MSE), as well as through the correlation coefficient PCC of actual and estimated complexities (a useful metric in the context of parallel AOBB, to be developed in Chapter 4).

Learning per problem instance, examined in Section 3.4.3, provided a good baseline with excellent results throughout, in the form of very low prediction errors (0.061 aggregated, cf. Table 3.10) and generally excellent correlation coefficients (0.866 on average). However, we noted that it has limited relevance in practice, since each new problem instance would require extensive sampling of subproblems to train on.

Learning per problem class in Section 3.4.4 is more meaningful from a practical point of view, as the learned model can be reused for new problems within the given problem class. Our experiments here showed good performance; predictions are somewhat less accurate than for per-instance learning, but the prediction error still yields an acceptable 0.721 on average (Table 3.10, "per class" column) – we also see the average PCC value decrease to 0.742. Nevertheless the results suggest that runtime performance of AOBB behaves similarly for instances within the same problem class, and that our prediction approach can capture this behavior fairly well.

A more general approach is presented by learning across problem classes, which was the subject of Section 3.4.4. Here the learning domain encompassed subproblems of instances from four different problem classes. It is likely to be more diverse and thus potentially more challenging to generalize for our learning scheme. On the other hand, certain dependencies might become evident in this setting that are not visible within a given problem class. In our experiments we observed performance very similar to, if not sometimes slightly better than, per-class learning, with an aggregate MSE of 0.587 and overall average PCC 0.751 (Table 3.10). This suggests that our approach is suitable to capture sufficient runtime complexity dependencies at this level as well.

The last learning approach we considered is learning for unseen problem classes, the most general and, at the same time, most challenging level of learning. As before model learning is performed on subproblems from several classes, yet testing happens on an entirely different, new problem class. In our experiments we systematically leave out subproblems of that problem class from the training set. For three of the four problem classes studied here this approach works reasonably well and yields respectable error values on almost all instances, similar to the values observed at lower learning levels.

Only for pdb side-chain prediction, using a model learned from pedigree, largeFam, and grid instances, do the results deteriorate notably, in some cases dramatically, as evident by

the aggregate MSE of 27.373 for pdb instances. Evidently, the pdb side-chain prediction problem class is too different from the other classes considered here. In particular, its higher maximum variable domain size (up to $k = 81$, cf. Table 3.4) and resulting inaccurate mini-bucket heuristic with low i-bound are very much unlike the other three classes. Overall, while expected to some degree, this result highlights the limitations of our learning approach.

However, we note that the correlation coefficient stayed fairly high throughout most of our experiments – even in the aforementioned case of unseen-class learning for pdb instances. This indicates that the learned models should have a marked ability to discriminate between subproblems of different complexity relative to each other. This will be helpful in the context of parallelizing AOBB in Chapter 4, where we aim to select the hardest subproblems for further splitting.

3.4.8 Feature Informativeness

As mentioned in Section 3.3.5, linear regression has the advantage that the resulting models can be straightforward to interpret. Namely, to assess the informativeness of feature ϕ_i we simply consider the absolute value of its coefficient λ_i in the learned regression model. Assuming normalized training data (i.e., each subproblem feature is transformed to have zero mean and unit variance), features with larger absolute values $|\lambda_i|$ contribute more to the predictions and are thus intuitively more informative.

Furthermore, recall from Section 3.3.5 that the L_1-regularization in lasso regression implicitly performs feature selection by assigning $\lambda_i = 0$ for some of the ϕ_i. In our case, we learned a model using the entire set of subproblems (over 17,000, cf. Section 3.4.1) and obtained a model with non-zero λ_i for nine features. These features and the absolute values of their coefficients are shown in Table 3.11. In addition, each feature's *cost of omission* ("coo") as defined in [78] is given. The cost of omission is the normalized difference between the

Feature ϕ_i	$\lvert\lambda_i\rvert$	coo
Average branching degree in probe	0.57	100
Average leaf node depth in probe	0.39	87
Subproblem upper bound minus lower bound	0.22	17
Ratio of nodes pruned by heuristic in probe	0.20	27
Max. context size minus mini bucket i-bound	0.19	16
Ratio of leaf nodes in probe	0.18	10
Subproblem upper bound	0.11	7
Std. dev. of subproblem pseudo tree leaf depth	0.06	2
Depth of subproblem root node in overall space	0.05	2

Table 3.11: Features ϕ_i present in the linear model trained by lasso regression (Eq. 3.10) on instances from all problem classes, with their model coefficients λ_i and their normalized cost of omission ("coo").

test error of the model with all nine features and the test error of a model trained with the respective feature omitted (using 5-fold cross-validation in all cases).

The particular set of features can be somewhat misleading, however, since lasso regression tends to pick only one of several highly correlated features [110]. Yet it is useful to gain a conceptual understanding of which kind of features are more informative than others. In particular, we observe that the four highest-weight features are dynamic, extracted from a limited AOBB probe or based on the initial subproblem bounds. Only the fifth feature, maximum subproblem context size minus mini-bucket i-bound, is static, with a normalized cost of omission of 16. This ties in to Section 3.2.1, where we observed that the asymptotic complexity bound of AOBB as well as the state space bound, both based on static, structural parameters, yield little information in this context.

3.5 Conclusion to Chapter 3

This chapter considered the problem of estimating the runtime complexity, in number of node expansions, of AND/OR Branch-and-Bound (AOBB). Chapter 4 illustrates the significance

of this task in the context of parallelizing AOBB, but it is interesting and challenging on its own.

Asymptotic complexity analysis, exponential in the induced width, and even finer-grained state-space upper bounds are generally very loose, since they don't account for determinism or, more significantly, the pruning power of AOBB and the accompanying mini-bucket heuristic, which we demonstrated empirically.

Most related work is either (a) based on sampling portions of the search space to be estimated, often relying on rather large samples that take a long time to compute, (b) not applicable to branch-and-bound search techniques and their powerful pruning, or (c) tailored to a specific class of problems, such as combinatorial auctions or proving/disproving Boolean satisfiability problems (SAT, typically not actually optimization queries). In contrast, we proposed a general scheme based on statistical regression analysis, where a complexity model is learned from runtime results of earlier experiments in an offline step. This keeps the time needed for complexity prediction of a new problem of subproblem to a minimum (which is crucial in the context of parallel AOBB).

As the basis for regression learning, we define a set of 35 features that encompass static, structural properties like the induced width as well as more dynamic attributes such as upper and lower cost bounds, derived from the problem's function tables. We then define a general class of complexity models that formulate the size of the explored search space as exponential in a linear combination of the defined features, motivated by the inherently exponential nature of search spaces. Predicting the log number of nodes can thus be formulated as a well-studied linear regression problem, to which we apply lasso regularization, which helps avoid overfitting and functions as a feature selection method.

Based on this approach, we learn and evaluate complexity models on four different problem classes and four different levels of learning, with varying practical applicability: in order

of increasing generality, we learn complexity models for each problem instance separately, across several instances from a single class, and across instances from several classes. Lastly, we investigate the possibility of estimating complexities for problem from a previously unseen problem class (a form of transfer learning).

Experimental performance is overall positive, with very good results for per-instance learning and decreased, but still good performance metrics for per-class and cross-class learning. On the most ambitious level of learning, predicting instances of an unseen problem class, our approach performs reasonably well on three out of four classes. However, the scheme reaches its limits and yields poor results on the fourth problem class, where instances have some distinctly different characteristics. In this context in particular it would be worthwhile to widen the evaluation to a more varied set of problem classes.

Finally, we note that, across all levels of learning, the majority of prediction results exhibited a high degree of correlation between actual and estimated complexities. This suggests that the learned models can reliably predict the order of subproblems in terms of their runtime. We hope to exploit this property for the load balancing of parallel AND/OR Branch-and-Bound, to be described in Chapter 4.

3.5.1 Open Questions & Future Work

We can identify a number of interesting directions for future work. First, we would want to consider additional problem classes, with more varied characteristics. Besides the question of predictive performance in general, this should be especially interesting in the context of transfer learning – Section 3.4.6 has yielded somewhat sobering results in this regard, and further investigation could be very illuminating. A central challenge in that regard is finding a sizable set of suitable problem instances, i.e., instances that are not too complex to be feasible, but also not too easy.

Secondly, our work thus far has been limited to linear regression models (quadratic feature expansion can be seen as a linear model with quadratic terms), which have yielded good results in most experiments. However, one could also consider truly nonlinear regression approaches, for instance employing gradient descent methods [9] with a loss function other than the mean squared error.

Thirdly, we have focused on predicting the runtime of subproblems of a larger AND/OR search space. This was done in the wider context of parallelizing AND/OR Branch-and-Bound search, but also to facilitate assembly of sufficiently larger sets of subproblem samples to apply learning on. Results should carry over to estimation of "full" search spaces (i.e., not subproblems), but meaningful empirical validation is made difficult by the challenge of compiling large enough sets of suitable problem instance samples.

Chapter 4

Parallelizing AND/OR Branch-and-Bound

4.1 Introduction

Chapter 2 considered AND/OR Branch-and-Bound running in sequential execution on a single CPU and proposed Breadth-Rotating AOBB for better anytime performance. This improves the algorithm's applicability as a scheme for approximate inference, but it does not address the case where a proof of optimality is required. This chapter, in contrast, will then focus on pushing the boundaries of AOBB for exact inference.

Solving MPE problems exactly is known to be NP-hard in general [104]. In practice, the limiting factor tends to be the induced width or tree width of a given problem instance (cf. Chapter 1), with many relevant problems rendered infeasible, and even harder ones introduced continually. Given todays availability and pervasiveness of inexpensive, yet powerful computers, connected through local networks or the Internet, it is only natural to "split" these complex problems and exploit a multitude of computing resources in parallel, which is

at the core of the field of distributed and parallel computing (see, for instance, the textbook by Grama et al. [51]).

Here we put optimization problems over graphical models in this parallelization context, by describing a parallelized version of AND/OR Branch-and-Bound. In particular, we adapt and extend the established concept of parallel tree search [51, 53, 74], where the search tree is explored centrally up to a certain depth, and the remaining subtrees are solved in parallel. In the graphical model context we explore the search space of partial instantiations up to a certain point and solve the resulting conditioned subproblems in parallel.

Our distributed framework is built with a general grid computing environment in mind, i.e., a set of autonomous, loosely connected systems – notably, we don't assume any kind of shared memory or dynamic load balancing which many parallel or distributed search implementations build upon (see Section 4.2.2).

The primary challenge in our work will therefore be to determine a priori a set of subproblems with balanced complexity, so that the overall parallel runtime will not be dominated by just a few of them. As shown in Chapter 3, however, in the context of optimization and AOBB, it is very hard to reliably predict and balance subproblem complexity ahead of time because of the algorithm's pruning power. This is where we will apply the complexity models developed in Chapter 3, in the hope that it will enable us to detect and circumvent bottlenecks in subproblem runtime.

4.1.1 Contributions

We first give an overview of the landscape of parallel and distributed computing in Section 4.2 and put our approach in context. In Section 4.3 we describe our parallel setup in more detail and present the parallel AND/OR Branch-and-Bound in two variants: one that bases its

parallelization decision on a fixed cutoff depth, and one that uses the complexity prediction proposed in Chapter 3 in an attempt to balance subproblem complexity. To our knowledge, it is the first implementation of its kind, i.e. an exact optimization algorithm for graphical models, running on a computational grid.

Section 4.4 provides in-depth algorithm analysis, including a number of examples. In particular, we illustrate different sources of overhead incurred as a consequence of the distributed execution environment. We also discuss characteristics of the parallelization frontier and investigate the question of optimality.

Related to that, in Section 4.5 we conduct a detailed investigation of redundancies in the overall search space explored by the parallel search. We identify two sources for this, both of which have their origin in the lack of communication across these parallel subproblem solution processes: unavailability of subproblem solutions as bounding information for pruning, as well as lack of caching for unifiable (sub-)subproblems across parallel CPUs.

This is followed by an extensive experimental evaluation in Section 4.6. We examine and analyze overall performance (i.e. runtime) and corresponding relative parallel speedup on a variety of instances from four different problem classes, using varying degrees of parallelism and different numbers of parallel CPUs. Further consideration is given to parallel resource utilization as well as the extent of the parallel redundancies in practice, which is shown to be far less pronounced than the theory in Section 4.5 suggests.

Experimental results are mostly positive. For relatively low and medium number of CPUs (20 and 100, respectively), we are able to show good parallel performance on many problem instances – the variable-depth scheme is often superior, provided that the complexity estimates don't exhibit any significantly underestimated outliers. At the same time results with 500 CPUs hint at the limitations of the current implementation, at which point the parallel

search space redundancies, while still far from their theoretical worst, become significant enough to meaningfully hinder performance.

Section 4.7 summarizes our contributions and outlines how parallel AOBB has been integrated into Superlink-Online SNP, a real-world inference platform used by geneticists and medical researchers worldwide. Finally, we briefly suggest potential future research directions to extend the algorithms and address some of the issues we identified.

4.1.2 Chapter Outline

Section 4.2 gives an overview of distributed computing in general and parallel search implementations in particular. Section 4.3 then proposes our two specific implementations of parallel AND/OR Branch-and-Bound search. In Section 4.4 we proceed to analyze some of the algorithms' properties, before Section 4.5 specifically investigates and illustrates the issue of redundancies in the parallel search space. Section 4.6 begins by describing our experimental setup and benchmark problem instances in Sections 4.6.1 and 4.6.2. Sections 4.6.3 through 4.6.7 then perform extensive empirical evaluation and analysis, which is summarized in Section 4.6.8. Section 4.7 concludes.

4.2 Background & Related Work

In this section we summarize relevant concepts and terminologies from parallel and distributed computing and put our contribution in context. We will also survey related work and delineate our approach against it.

4.2.1 Parallel & Distributed Computing

The notions of *parallel computing* and *distributed computing* have considerable overlap and there is no clear line to be drawn between them; sometimes the terms are even used interchangeably. One distinction that is commonly made, however, is to consider how tightly coupled the concurrent processes are. In particular, distributed systems are typically more loosely coupled than parallel ones, with each process having its own, private memory. The latter is often also referred to as the *distributed-memory* model, in contrast to *shared-memory* [47, 51, 81].

Historically, "distributed systems" were often just that, namely geographically distributed, but this connotation has weakened over the years to include, for instance, locally networked computers. Further distinctions can be made regarding *cluster computing* or *grid computing*, where a computational grid is often regarded as a larger-scale, more heterogeneous incarnation of a cluster of (more uniform) computers [43].

4.2.1.1 Data Parallelism

Well-known applications of large-scale grid computing are, for instance, SETI@home [4] and Folding@home [8], which employ thousands of processors worldwide, often volunteered home computers, for analysis of cosmic radio signals and protein folding simulation, respectively. Both of these are based on the concept of *data parallelism*, where concurrency is achieved by independently processing parts of a huge set of input data in parallel.

It is worth pointing out that this assumption of independence regarding the computation of each data item renders these applications *embarrassingly parallel* – a common classification in parallel computing given to problems for which a straightforward subproblem separation takes little to no effort and yields very good results.

A general framework in a similar context that has gained much popularity in recent years, in particular for industrial applications on massive cluster systems, is MapReduce [25]. It is designed to independently apply a "Map" operation and subsequent "Reduce" combinator to each entry in terabytes of data in a distributed fashion, with a variety of implementations available (e.g., the Hadoop library [113]).

4.2.1.2 Task Parallelism

Our contribution of parallelizing AND/OR Branch-and-Bound is similar in its use of the cluster or grid computing paradigm, but its approach to parallelization is not inherently data based – in fact, in terms of raw size our input problem specifications often measure just a few hundred kilobytes. Rather, the primary objective is to distribute an exponential amount of computations, a notion also referred to as *task parallelism*.

A fitting example of this challenge is the Superlink-Online system, which similarly uses vast numbers of computers around the world to perform genetic linkage analysis on general pedigrees [105, 106]. To be specific, it converts the pedigree data into a Bayesian network and computes the likelihood of a number of specific sets of evidence using a sum-product algorithm. This corresponds to computing the LOD scores that are used to signify genetic linkage in the input [40].

In fact, from the outset one of the motivations for and goals of our research has been to eventually adapt our parallel AOBB implementation for integration into the Superlink-Online system, where the task of computing maximum likelihood haplotypes for a given pedigree can be expressed as an MPE query.

This objective also determines the particular parallel environment we consider, which we describe in more detail in Section 4.3.1.

4.2.2 Parallel Tree Search

A general way of distributing the depth-first exploration of a search tree across multiple processors is presented by the *parallel tree search* paradigm [51, 74].

At the core of this approach the search space is partitioned into disjoint parts, at least as many as there are processors, which are then assigned to the different processors to handle. Since depth-first algorithms are often implemented using a stack data structure, this approach is also referred to as *stack splitting* in the literature [51]. Namely, the stack of the sequential algorithm is split into distinct parts for the concurrent processes.

Over the years the parallel tree search concept has been developed and applied in a variety of incarnations across many domains, from classic Vertex-Cover [80] and Traveling Salesman problems [111] to planning tasks in robotics [15]. Adaptations have been proposed for parallelizing alpha-beta pruning and general game tree search [38], an area that has gained renewed prominence through IBM's massively parallel Deep Blue chess-playing system [13] or more recent, very successful advances of parallel Monte-Carlo tree search in the game of Go [14, 16].

In the 1990s research was also conducted on parallel search for specific parallel architectures. In the context of SIMD systems (single instruction, multiple data – in contrast to multiple instruction, multiple data common today) in particular there were efforts to parallelize heuristic search algorithms like IDA* (leading to SIMD-IDA* or SIDA* [98]) and A* (resulting in PRA*, parallel retracting A* [37]). As in other shared memory search implementations, load balancing was conducted dynamically at runtime (in intervals, to accommodate the SIMD architecture), with a hashing function used to assign newly generated nodes to processors. Another central challenge at the time was presented by the limited system memory, which PRA* addressed by selectively "retracting" expanded nodes [37].

Finally, we note that the SAT (Boolean satisfiability) community has shown great interest in parallel search as well, since most state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland procedure (DPLL [24]), a depth-first backtrack search algorithm. Consequently, several SAT solvers based on parallel tree search have been proposed (e.g., [20, 63]). However, the focus in recent years has shifted to parallelized portfolio solvers [54].

4.2.2.1 Parallel Branch-and-Bound

Since branch-and-bound is inherently a depth-first search algorithm, many of the results summarized above are directly applicable in its parallelization. In fact, alpha-beta pruning for game trees can be seen as a form of branch-and-bound [38].

The most crucial addition of branch-and-bound over standard depth-first search lies in keeping track of the current lower bound on the solution cost (assuming a maximization problem), which the algorithm compares against heuristic estimates to prune subtrees. In a shared-memory parallel setup, this global bound can be synchronized across processors, for which various schemes have been proposed in the literature [46, 52, 53].

Faced with a lack of shared memory in grid and cluster systems as well as limited or no inter-process communication, this exchanging and updating of bound information is no longer possible – each processor is limited to its locally known bound, which can lead to additional node expansions. We will explore this issue and possible (partial) remedies more closely in the context of AOBB in Section 4.4.

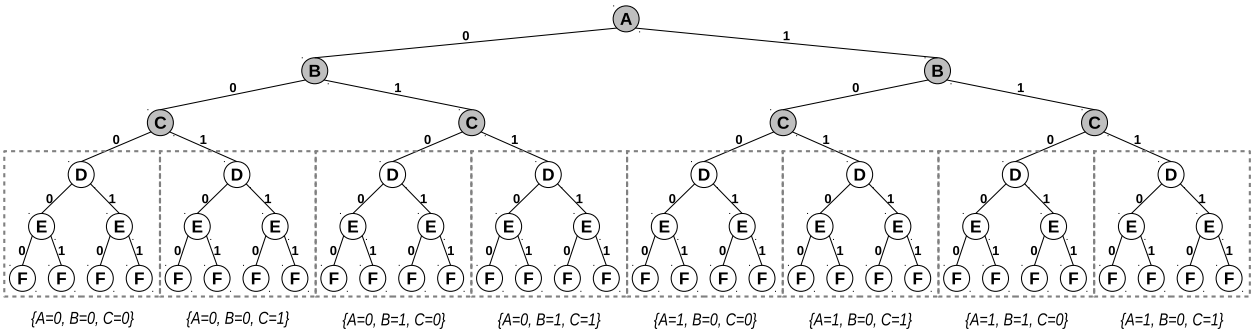
4.2.2.2 Load Balancing

One of the crucial issues in parallel tree search is clearly the choice of partitioning. In particular, the goal is to make sure each processor gets an equal share of the overall workload, to minimize the amount of idle time across CPUs and, equivalently, optimize the overall runtime. This issue is commonly referred to as *load balancing*.

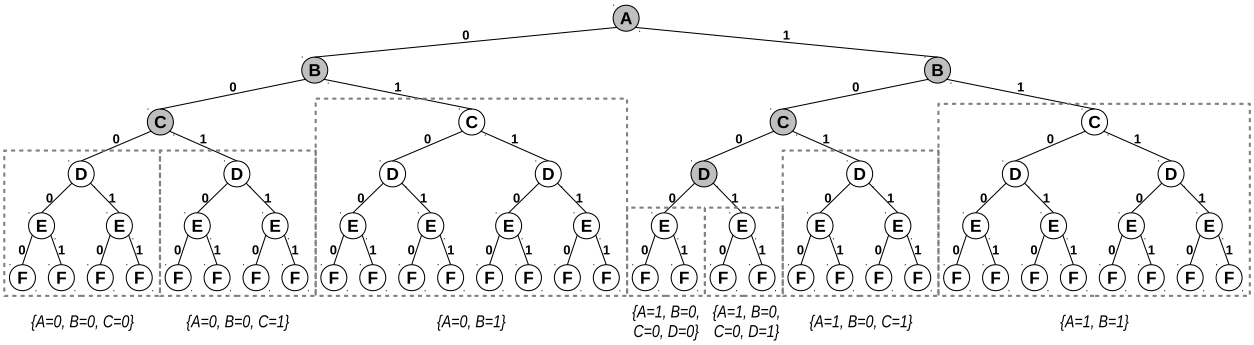
To illustrate, imagine a scenario where all but one processor completes their assigned task almost immediately, while the remaining CPU continues to work for a long time, thus delaying the overall solution. Ideally, at the opposite end of the load balancing spectrum, all processors would finish at the same time, so that no idle time occurs. Assuming a fixed overall workload of T seconds and p parallel processors, the overall parallel runtime in the latter, balanced case would be T/p. In the former, more extreme case, however, the overall parallel runtime would still be close to T – clearly not an efficient use of parallel resources (performance metrics will be discussed in more detail in Section 4.2.4).

In shared-memory approaches to parallel computing, this problem is often tackled through *dynamic load balancing* [51, 73], where an initial partitioning of the search space is dynamically adapted over time. Namely, if one processor runs out of work it can be assigned (or request) part of some other processor's partition of the search space to restore load balancing, where the question of when and how to perform these reassignments is one of the central research issues. Dependent on the implementation, this approach is sometimes also referred to as *work stealing* [19]. Using message passing schemes, dynamic load balancing can also be implemented for distributed-memory architectures [47, 81].

In distributed systems where inter-process communication is prohibitively expensive, or even altogether infeasible because of technical restrictions, dynamic load balancing is not an option – this applies, for instance, to many grid approaches discussed earlier and the Superlink-Online framework in particular. In this case, a suitable partitioning must be found ahead of



(a) Parallelization frontier at fixed depth $d = 3$, yielding 8 subproblems.



(b) Variable-depth parallelization frontier, yielding 7 subproblems.

Figure 4.1: OR search parallelization applied to the OR tree from Figure 1.5. Conditioning nodes are shaded gray, the respective conditioning set is given below each subproblem .

time to facilitate efficient *static load balancing*. Namely, since transfer of workload among processors is no longer possible, the initial partitioning of the search space should be as balanced as possible.

As Chapter 3 has demonstrated, however, the exact size of a search space is not always easy to assess in advance. Static load balancing is therefore very challenging, in particular for a branch-and-bound scheme like AOBB, and in fact its analysis and implementation for AOBB constitutes one of the main contributions of our work.

4.2.3 Parallel OR Search Example

To illustrate the concept of parallel tree search, we give an example in the context of regular OR tree search. Namely, independent subproblems are not decomposed and unifiable sub-

problems are not cached. We build upon the example search space over six binary variables from Figure 1.5.

The underlying principle consists of exploring a small part of the search space centrally, at each step breaking the overall problem into smaller pieces through conditioning. Consequentially, we refer to this part of the search space as the *conditioning space*. Its boundary with the conditioned subproblems is called the *parallelization frontier* or *parallel cutoff*; it is made up of the root nodes of the resulting concurrent subproblems.

Example **4.1.** *A first example is given in Figure 4.1a. Here the parallelization frontier is placed at constant depth $d = 3$ (note that we define the overall problem root node to have depth 0). This yields eight subproblems, each conditioned on a different instantiation of the variables A, B, and C. In other words, $\{A, B, C\}$ is a conditioning set for the parallel subproblems.*

We note that a fixed-depth parallel cutoff as in Example 4.1 is fairly natural choice, as it renders the eight subproblems in Figure 4.1a to be of equal size, with seven nodes each, i.e., they appear to be balanced perfectly. In practice, however, a branch-and-bound algorithm might only explore a small part of each subproblem search space due to its pruning power and determinism in the problem specification. This discrepancy, which was also at the core of Chapter 3 (cf. Section 3.2.1, in particular), can lead to significant imbalance of subproblem runtimes in practice, since the effects of pruning can play out very differently in each subproblem.

It is therefore very convenient that we are not limited to a uniform conditioning set for parallelization. Rather, the search framework gives us considerable flexibility with regards to the parallelization frontier. In particular, we can vary the cutoff depth for different parts of the conditioning space.

EXAMPLE **4.2.** *An example of variable-depth parallel cutoff is shown in Figure 4.1b. In this case, we define one subproblem for the conditioning* $\{A = 0, B = 1\}$*, while the subspace for* $\{A = 0, B = 0\}$ *is broken up further by conditioning on* C*. The overall number of parallel subproblems is seven.*

Ideally, a variable-depth parallel cutoff will be able to compensate for the varying degree of exploration by branch-and-bound in different subproblems. And in fact, this will be one of the central issues we investigate in our empirical evaluation in Section 4.6.

4.2.4 Assessing Parallel Performance

Parallel and distributed algorithms in general, and parallel search implementations in particular, can be evaluated from a variety of standpoints, accounting for the many objectives that are involved in their design [51]. Specifically, given a parallel search algorithm and its base sequential version, we can collect and report the following metrics:

- **Sequential runtime** T_{seq}**.** The wall-clock runtime of the sequential algorithm.
- **Sequential node expansions** N_{seq}**.** The number of node expansions by sequential AOBB.
- **Parallel runtime** T_{par}**.** The elapsed wall-clock time from when the parallel scheme is started to when all concurrent processes have finished and the overall solution has been returned.
- **Parallel node expansions** N_{par}**.** The number of node expansions counted across all parallel processes.
- **Parallel speedup** $S_{par} := T_{seq}/T_{par}$**.** The relative speedup of the parallel scheme over the sequential algorithm.

- **Parallel overhead** $O_{par} := N_{par}/N_{seq}$**.** The relative amount of additional work (in terms of node expansions) induced by parallelization.

- **Parallel resource utilization** U_{par}**.** If T^i_{par} is the runtime of parallel processor i, $1 \leq i \leq C$, we denote $T_{max} := \max_j T^j_{par}$ and define $U_{par} := \frac{1}{C} \sum_{i=1}^{C} T^i_{par}/T_{max}$ as the average processor utilization, relative to the longest-running processor.

The definition and interpretation of T_{seq} and T_{par} as well as N_{seq} and N_{par} is straightforward. Regarding the parallel speedup S_{par} we note that, in the ideal case, it will be close to the number of concurrent processors. In practice, however, issues like communication overhead, network delays, and inherent redundancies make this hard to achieve; specifics will be discussed in Section 4.4.

The parallel overhead O_{par} is ideally 1, i.e., the number of nodes expanded overall by the parallel scheme is the same as for sequential AOBB. As Section 4.4 will detail, however, inherent search space redundancies in the parallel scheme again make this hard to achieve, just as for the optimal speedup.

Lastly, the parallel resource utilization U_{par} with $0 < U_{par} \leq 1$ measures the efficiency of load balancing. A value close to 1 indicates very balanced load distribution, with all concurrent processes finishing at about the same time. Values closer to 0 signify substantial load imbalance, with most processors finishing long before the last one.

4.2.5 Amdahl's Law

In regard to parallel performance and parallel speedup in particular it is worth mentioning *Amdahl's law*, named after its author Gene Amdahl [3]. It comprises the simple observation that the possible speedup of a parallel program is limited by its strictly sequential portion. Namely, if only a fraction p of a given workload can be parallelized, even with unlimited

parallel resources the speedup can never exceed $1/(1-p)$. For instance, if $p = 0.9$, i.e., 90% of a computation can be parallelized, the maximum achievable parallel speedup is $1/(1-0.9) = 10$. More generally:

THEOREM **4.1** (Amdahl's Law)**.** *[3] If a fraction p of a computation can sped up by a factor of s through parallelization, the overall speedup cannot exceed $1/(1-p+p/s)$.*

For instance, if $p = 0.9$ and $s = 10$, the overall speedup will be approx. $1/(1-0.9+0.09) \approx 5.26$. We will put our results in this context when analyzing our parallel scheme in Section 4.4 and when conducting experimental evaluation in Section 4.6.

4.2.6 Other Related Work

We point out the work of Allouche et al. [2], which is similar in that it proposes a method to solved weighted CSPs in parallel (which could be generalized to general max-product problems like MPE). However, their approach is based on inference through variable elimination. At its core, it exactly solves (in parallel) the clusters of a tree decomposition, conditioned on the separator instantiations. They also describe a method to obtain suitable tree decompositions, bounding the space of separator instantiations through iteratively merging decomposition clusters. According to the authors, load imbalance was not an issue with their approach in their (limited) set of practical experiments.

A fairly young field where parallel search is an active area of research is *distributed constraint reasoning* [117], which is concerned with solving *distributed constraint satisfaction problems* (DCSPs) and *distributed constraint optimization problems* (DCOPs). Over the last decade or so, several search-based parallel algorithms have been proposed in distributed constraint reasoning. Notable examples for solving DCSPs include ABT (Asynchronous Backtracking) by Yokoo et al. [118] and extensions by Zivan and Meisels [120]. For DCOPs there are, for

instance, ADOPT (Asynchronous Distributed Optimization) by Modi et al. [86], which is based on parallel best-first search and BnB-ADOPT, an adaptation of ADOPT to depth-first search principles by Yeoh et al. [116].

While there are some shared concepts with AOBB and parallel tree search as outlined above (e.g., ADOPT and BnB-ADOPT exploit a pseudo tree structure), the underlying principles of distributed constraint reasoning are very different. In particular, the term "distributed" is used to indicate a multi-agent setting where each agent only has partial knowledge of the problem, with its state represented by a subset of the problem variables. The key differences between the various schemes cited above are how communication between agents is organized, i.e., what kind of messages are sent and to which agent(s).

Agents are also typically assumed to be low-powered devices with limited computational power, fairly expensive inter-agent communication (e.g., in terms of electrical power required for radio transmission), and sometimes limitations on what kind of information may be shared between agents. These assumptions then determine the performance metrics that are typically applied to DCSP and DCOP algorithms. Namely, evaluation is performed with regard to the number of messages sent between agents or the number of constraint checks each agent performs to solve a problem – computation time or parallel speedup, on the other hand, are only secondary and sometimes not considered at all. A direct comparison to our contribution in this chapter is therefore not easily attainable.

4.3 Parallel AND/OR Branch-and-Bound

In the following we will introduce our implementation of parallel AND/OR Branch-and-Bound, based on the parallel tree search concept outlined in Section 4.2.2. To begin, Section

4.3.1 lays out the parallel environment we build upon, in line with the exposition of Section 4.2.1.

We then propose two variants of parallel AOBB that differ in how they determine the parallelization frontier. The first, in Section 4.3.2, chooses the subproblem root nodes at a fixed depth, in line with parallel tree search described in Section 4.2.2. In contrast, the second approach introduced in Section 4.3.3 uses estimates of subproblem runtime to determine a variable-depth frontier.

4.3.1 Parallel Setup

As indicated in Section 4.2.1, our approach to parallelizing AND/OR Branch-and-Bound is built on a grid computing framework. Namely, we assume a set of independent computer systems, each with its own processor and memory, that are connected over some network (e.g., a local network or the Internet).

We will further assume a *master-worker* organization (also known as *master-slave*), where one designated *master host* directs the remaining *worker hosts*. In particular, the master determines the parallel subproblems and assigns them to the workers as *jobs*; it collects the results and compiles the overall solution. Communication among workers is assumed infeasible – in fact, because of firewalls or other network restrictions, workers might not even be aware of each other.

Clearly, this grid approach entails a crucial limitation in terms of algorithm design by explicitly forbidding synchronization between workers, thereby forcing subproblems to be processed fully independently. On the other hand, it also brings with it a number of advantages, which make it particularly suitable for large-scale parallelism. Subproblems that are currently executing can easily be preempted (or the executing system may fail) and restarted elsewhere,

since no other running job depends on it. Parallel resources can readily be added to or removed from the grid on the fly. In general, the lack of synchronization also inherently facilitates scaling, with the only possible bottleneck located in the management of parallel resources by the central master host. For these reasons, this kind of grid paradigm is sometimes also referred to as *opportunistic computing*.

As mentioned earlier, this setup matches that of Superlink-Online [106], a high-performance online system for genetic linkage analysis. It enables researchers and medical practitioners to use tens of thousands of CPUs across multiple locations, including volunteered home computers, for large-scale genetic studies, to great success [105].

Internally, Superlink-Online is built upon a specific grid software package, also called middleware, the *Condor* distributed workload management system [107, 108], which we will also employ for our setup. Condor provides an abstraction layer on top of the bare parallel resources which, among other things, transparently handles the following:

- Centralized tracking and managing of parallel resources.
- Assigning jobs to available worker hosts.
- Distributing the necessary input files to workers and transmitting their output back to the master host.
- Gracefully handling resource failures (e.g., by automatically rescheduling jobs).

Condor also exposes a powerful mechanism for prioritizing jobs, but neither our system nor Superlink-Online makes use of it; parallel jobs are simply assigned to available resources on a first-come, first-served basis.

Finally, we point out that real-world grid systems are almost always shared-access resources, with many users submitting jobs of varying complexity at different points of time. Together

Algorithm 4.1 Master process for fixed-depth parallelization.

Given: Pseudo tree $\mathcal{T}$ with root X_0, cutoff depth d_{cut}.
Output: Ordered list of subproblem root nodes for grid submission.

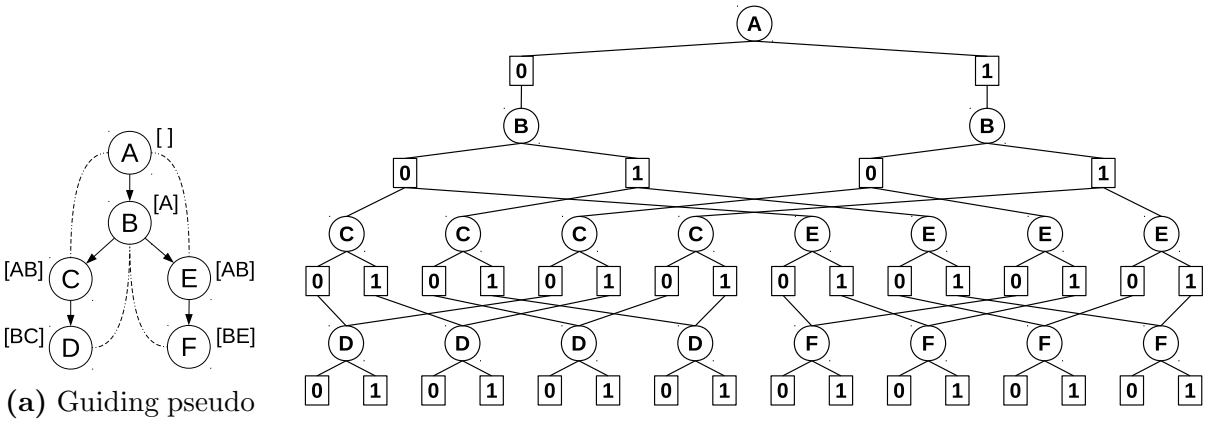
```
1: Stack ← ∅  // last-in-first-out stack data structure
2: Stack.push(⟨X_0⟩) // root node at depth 0
3: while |Stack| > 0 :
4:    n ← Stack.pop()
5:    if depth(n) == d_cut :
6:       grid_submit(n)
7:    else
8:       for n' in children(n) :
9:          Stack.push(n')
```

with the opportunistic nature discussed above (i.e., the set of available parallel resources fluctuates over time), this can make controlled experiments, to measure overall parallel runtime and resulting speedup, notoriously tricky in practice. And at the same time, results of carefully executed experiments do not always carry over directly into real-world systems.

In our experiments (cf. Section 4.6) we will mostly rely on an "idealized" grid environment (i.e., with a stable number of processors and little to no interference from other users), but also some carefully designed simulations.

4.3.2 Fixed-depth Parallelization

This section introduces our "baseline" parallel AOBB with a fixed-depth parallelization frontier. It explores the conditioning space centrally, on the master host, up to a certain depth. It thereby applies the natural choice of parallel cutoff, leading to subproblems that are structurally the same, just as it was done for OR search parallelization in Example 4.1 (cf. Section 4.2.3). For convenience, in the AND/OR context we consider a depth level to consist of an OR node and its AND children – this implies that the roots of the parallel subproblems will always be OR nodes.



(a) Guiding pseudo tree with contexts.

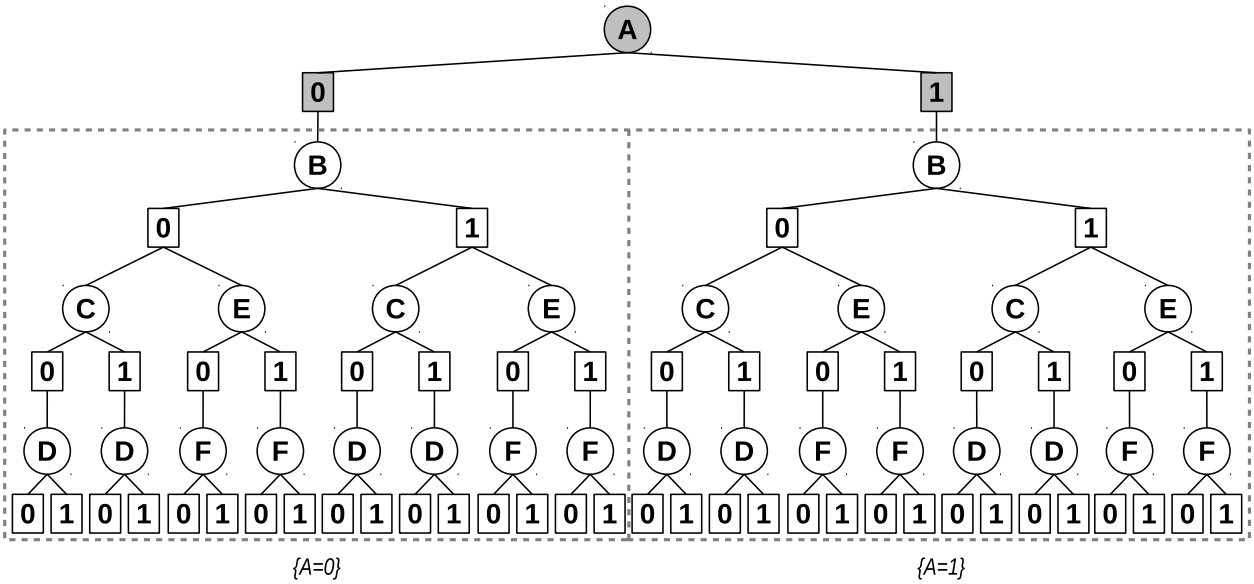
(b) AND/OR search graph with 19 OR and 38 AND nodes.

Figure 4.2: Example AND/OR search graph for problem in Figure 1.3.

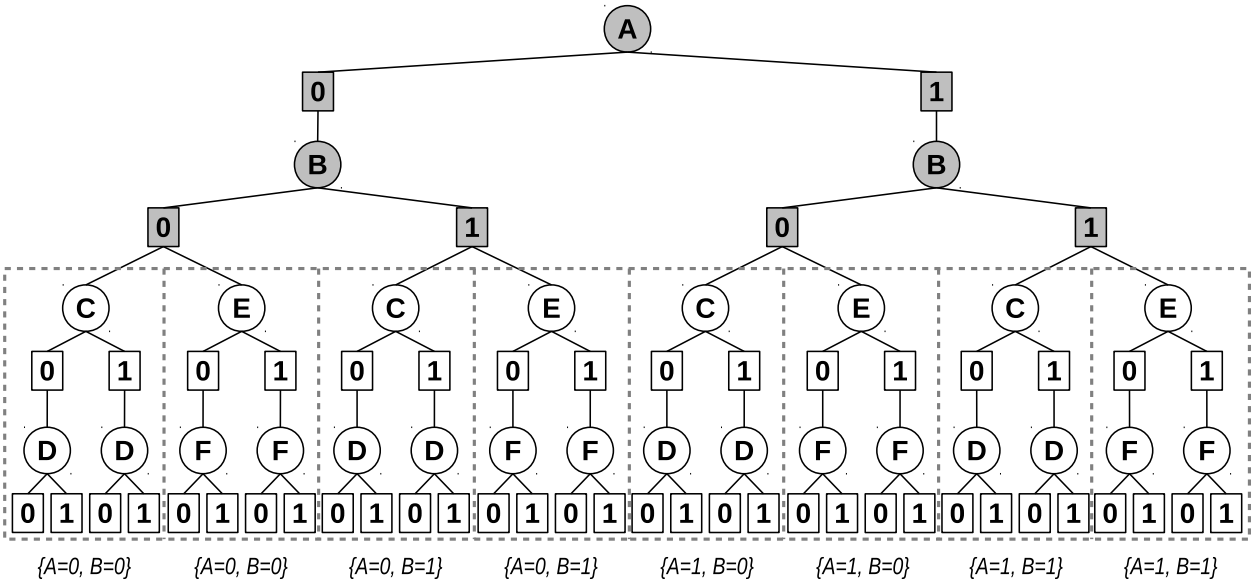
Pseudo code for this simple scheme is shown in Algorithm 4.1. It expands all nodes up to a given depth d_{cut} in a depth-first fashion. The subproblems represented by the nodes at depth d_{cut} are marked for submission to the grid, for parallel solving. We note that the call to "children(n)" in line 8 of Algorithm 4.1 can be implemented to apply branch-and-bound-style pruning, using a separately provided initial lower or upper bound on the problem's solution cost (obtained, for instance, through incomplete, local search).

Example **4.3.** *To illustrate, we apply Algorithm 4.1 to the problem from Example 1.8, for which the AND/OR search graph is shown again in Figure 4.2. The result of setting the cutoff depth at* $d = 1$ *is depicted in Figure 4.3a , the conditioning set* $\{A\}$ *yields two subproblems. Figure 4.3b, on the other hand, shows the outcome of setting* $d = 2$ *, i.e., a static conditioning set of* $\{A, B\}$*, which gives eight subproblems – notably, subproblem decomposition below* B *presents an additional source of parallelism, with independent subproblems processable in parallel.*

Already at this point, we note that the conditioning process can impact the caching of unifiable subproblems in AND/OR Branch-and-Bound graph search (cf. Section 1.3.1). In particular, the subproblems rooted at D and F are unified in Figure 4.2, yet in Figures 4.3a and 4.3b they are spread across different parallel subproblems rooted at C and E,



(a) Parallelization frontier at fixed depth $d = 1$, yielding 2 subproblems.



(b) Parallelization frontier at fixed depth $d = 2$, yielding 8 subproblems.

Figure 4.3: AND/OR search parallelization at fixed depth, applied to the example problem and search space from Figures 1.3 and 1.7, respectively. Conditioning nodes are shaded gray, the respective conditioning set is specified below each subproblem.

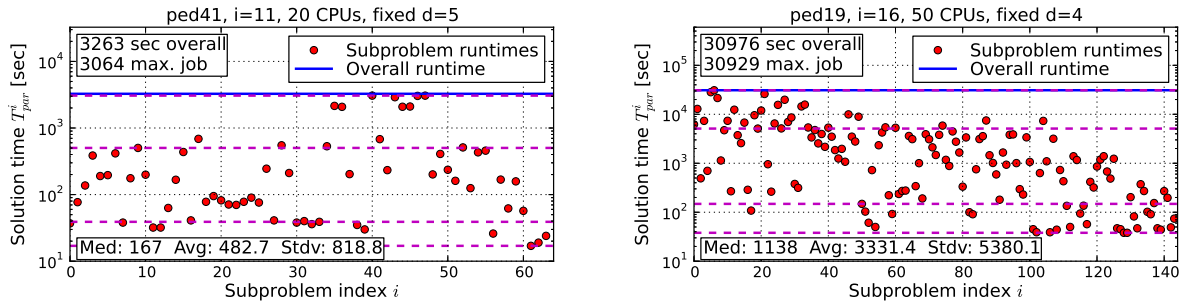


Figure 4.4: Subproblem statistics for two runs of fixed-depth parallel AOBB. Each dot represents a single subproblem, plotted in the order in which they were generated. Dashed horizontal lines mark the 0^{th}, 20^{th}, 80^{th}, and 100^{th} percentile, the solid horizontal line is the overall parallel runtime using the number of CPUs specified in the plot title.

respectively, and unification is no longer possible (since we assume no sharing of information between workers). We will analyze this issue in-depth in Section 4.4.

Finally, we point out that the cutoff depth d_{cut} is assumed to be given as an input parameter. It could, however, also be derived from other objectives, such as the minimum desired number of subproblems p. In case of a problem instance with binary variables, for instance, we could easily compute $d = \lceil \log_2 p \rceil$; generalization for non-binary domains is straightforward.

4.3.2.1 Underlying vs. Explored Search Space

We note that Example 4.3 and Figures 4.2 and 4.3 can be somewhat misleading since they depict the full, underlying context-minimal AND/OR search graph. As we indicated earlier, however, in practice large parts of the underlying search space are ignored by AOBB, because of determinism or pruning based on the mini-bucket heuristic. This leads to a much smaller *explored search space* – a discrepancy that was also at the core of Chapter 3, which developed a learning approach to better estimate the runtime complexity of a given subproblem ahead of time.

To illustrate the practical impact in the context of parallel AOBB, Figure 4.4 gives some runtime statistics for two different runs of the fixed-depth parallel scheme. Shown are, for two

different problems from the domain of pedigree linkage analysis, the runtimes of subproblems generated at a fixed depth d, as listed in the plot title. We also indicate, by a solid horizontal line, the overall runtime of parallel AOBB using this particular parallelization frontier.

In the context of this section, we point out two things in particular. First, all subproblems originated at the same depth, thus the size of their underlying search space is in fact the same. As expected, however, we observe significant variance in the size of the explored search space. This is captured by the subproblem runtimes plotted in Figure 4.4, which range over about two and three orders of magnitude for ped41 and ped19, respectively. Second, the overall runtime is heavily dominated by only a handful of subproblems, which is very detrimental to parallel performance and thus a scenario we aim to avoid. In the following we therefore propose a more flexible variant of parallel AOBB.

4.3.3 Variable-depth Parallelization

This section describes a second variant of parallel AND/OR Branch-and-Bound, which employs the flexibility provided by AND/OR search to place the parallelization frontier at a variable cutoff depth. Namely, subproblems within one parallel run can be chosen at different depths, in line with the example in Section 4.2.3.

The question of how to determine the actual cutoff, however, was left open in Section 4.2.3. Recall that the central motivations for a variable-depth cutoff were to better balance subproblem runtimes and, related to that, avoid performance bottlenecks through long-running subproblems. In the following, we thus propose an iterative, greedy scheme that employs complexity estimates of subproblems to decide the parallelization frontier; notably, we can apply the estimation models developed in Chapter 3.

Algorithm 4.2 Master process for variable-depth parallelization.

Given: Pseudo tree $\mathcal{T}$ with root X_0, subproblem count p, complexity estimator $\hat{N}$.
Output: Ordered list of subproblem root nodes for grid submission.

```
1: Frontier ← {⟨X_0⟩}
2: while |Frontier| < p :
3:     n' ← arg max_{n∈Frontier} N̂(n)
4:     Frontier ← Frontier \ {n'}
5:     F ← Frontier ∪ children(n')
6: while |Frontier| > 0 :
7:     n' ← arg max_{n∈Frontier} N̂(n)
8:     Frontier ← Frontier \ {n'}
9:     grid_submit(n')
```

Algorithm 4.2 gives pseudo code for this approach. Starting with just the root node, the algorithm gradually grows the conditioning space, at each point maintaining the frontier of potential parallel subproblem root nodes. In each iteration, the node with the largest complexity estimate is removed from the frontier (lines 3-4) and its children added instead (line 5). This is repeated until the frontier encompasses a desired number of parallel subproblems p. At that point these subproblems are marked for grid submission, in descending order of their complexity estimates – it makes sense to process the larger subproblems first so that, in case the number of subproblems exceeds the number of parallel CPUs, smaller subproblems towards the end can be assigned to workers that finish early.

We point out that this policy of assigning parallel jobs in a hard-to-easy fashion corresponds to the LPT algorithm ("longest processing time") for the *multiprocessor scheduling* optimization problem [34], which in turn is a special case of the *job-shop scheduling* optimization problem , which is known to be NP-complete for 3 or more parallel resources [45]. Introduced already in the 1960s, LPT is often used for its simplicity; it has been proven to be optimal within a factor of $\frac{4}{3} - \frac{1}{3c}$ from the best possible overall runtime, where c is the number of parallel resources considered [50]. Note, however, that this bound assumes that the exact job runtimes are fully known ahead of time, which is not the case in our setting.

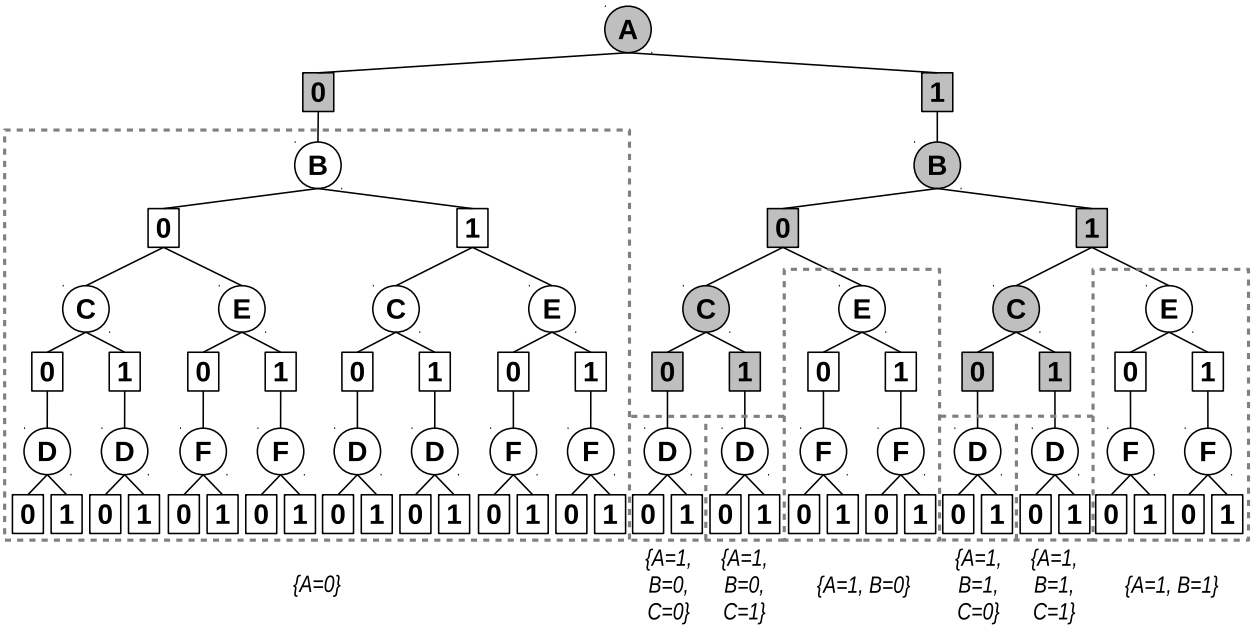


Figure 4.5: AND/OR search parallelization at variable depth, applied to the example search space from Figure 4.2, respectively, yielding seven subproblems. Conditioning nodes are shaded gray, the respective conditioning set is specified below each subproblem.

EXAMPLE **4.4.** *Figure 4.5 shows an example of a variable-depth parallel cutoff applied to the same problem as in Example 4.3. With seven parallel subproblems overall, the subproblem with conditioning set* $\{A = 0\}$ *is not broken up further, while* $\{A = 1\}$ *is split (more than once, in fact) into a total of six subproblems. As before we point out the impact of parallelization on caching for nodes of variables* D *and* F*, which we will analyze in the next section. Note also that Figure 4.5 again only depicts the underlying search space – the explored search space for each subproblem might only comprise a small sub space when processed by AOBB.*

Finally, we point out that instead of providing the desired number of subproblems p as input, one could equally use other parameters. For instance, a straightforward alternative would be to set an upper bound on subproblem complexity, where subproblems are broken into pieces through conditioning while their estimated complexity exceeds the provided bound. However, our focus here will be on targeting a specific number of subproblems, which facilitates a direct comparison with the fixed-depth parallel cutoff.

We perform in-depth analysis of the above algorithms in the following section.

4.4 Algorithm Analysis

In this section we provide analysis of the parallel algorithms' properties, also taking into account the performance measures introduced in Section 4.2.4. Section 4.4.1 describes sources of parallel overhead that are common to distributed computing, and how they manifest in our context. Section 4.4.2 considers some of the challenges in choosing a parallel cutoff. Description and analysis of redundancies in the parallel search space are provided in their own Section 4.5.

4.4.1 Distributed System Overhead

When compared to standard, sequential AOBB, parallel AOBB as described above does inevitably incur overhead in a variety of forms, by virtue of its distributed execution and operating environment. The following paragraphs distinguish the different sources of overhead and, if possible, quantify their practical effects based on our experiments.

4.4.1.1 Parallelization Decision

One of the first tasks of the parallel scheme lies in determining the parallel cutoff, algorithms for which were described in Sections 4.3.2 and 4.3.3. Performed by a master host on the grid, this computation involves a number of steps:

- General preprocessing, like problem parsing and evidence elimination, variable order computation, and mini-bucket heuristic compilation, needs to be performed as part of the the initial master process.

  Runtime for these steps can vary greatly depending on the problem instance and, most centrally, the chosen i-bound of the mini-bucket heuristic, but can exceed one minute in some cases. Note, however, that these steps and their respective runtimes are actually the same as for sequential AOBB.

- The master process gradually expands the conditioning set, until either a fixed depth has been reached (Algorithm 4.1) or a predetermined number of parallel subproblems has been generated (Algorithm 4.2). We can show the following:

  THEOREM **4.2.** *Assuming a branching degree of at least 2, the number of node expansions required in the conditioning space to obtain p parallel subproblems is $O(p)$, i.e., linear in p.*

  Proof. Consider a conditioning search space with p leaf nodes representing subproblems. There can be at most $\frac{p}{2}$ parent internal nodes to the p leaves, since a branching degree of at least 2 is assumed. These nodes in turn have at most $\frac{p}{2^2}$ parents, and so on, all the way to the root node. Thus, $\frac{p}{2}+\frac{p}{2^2}+\frac{p}{2^3}+\ldots+1 = p(\frac{1}{2}+\frac{1}{2^2}+\ldots)+1 \leq p+1$ bounds the number of expanded, internal nodes. □

  Even in the case of several thousand subproblems, the number of such conditioning operations is thus fairly small (relatively to the full search space). In practice, the longest runtimes we observed for the conditioning step were up to 2 minutes for just under 10000 subproblems on a 2.8 GHz Intel Xeon CPU. This includes application of our complexity estimation scheme (cf. Chapter 3) each time a subproblem is split and conditioned further.

- Finally, the master process needs to write the parallel subproblem information, including the variable ordering, upper and lower bounds, and the individual conditioning sets, to a set of files, which will be passed as input to sequential AOBB on the worker hosts. This process can usually be completed in a few seconds.

In the context of Amdahl's Law (cf. Section 4.2.4), the above steps can be seen as the non-parallelizable part of the computation. As a consequence, parallel AOBB will likely not work that well, i.e., yield suboptimal parallel performance, if the problem instance at hand is too easy (say, for instance, less than 20-30 minutes with sequential processing).

4.4.1.2 Communication and Scheduling Delays

Once the parallel cutoff is determined and the respective subproblem specification files have been generated, the information is submitted to the worker hosts. This is achieved by invoking the grid middleware's job submission service, in our case provided by the Condor software (cf. Section 4.3.1).

Parsing and processing the parallel job descriptions can take Condor some time in practice, around 4-5 seconds for each 1000 subproblems on our system – for extreme cases, with tens of thousands of subproblem submissions, we observed delays of several minutes.

Subsequently the grid management software will match jobs to available parallel resources, i.e., worker hosts, transmit input files as necessary, and start remote execution of sequential AOBB. After a job, corresponding to a single parallel subproblem, finishes, the grid software retrieves its output, transfers it back to the master host, and makes the now idle CPU available for the next job.

In our experience, this transfer of input/output and general job setup adds, on average, about 2-3 seconds on top of the actual runtime of sequential AOBB on the worker host.

These numbers are based on experiments using a grid spanning 324 CPUs on a relatively fast local network with a powerful host in charge of the Condor scheduling; we expect this overhead to increase for more (geographically) distributed grids or a less powerful scheduling machine.

4.4.1.3 Repeated Preprocessing

As mentioned, once a job has been submitted to a worker host on the grid, sequential AOBB is invoked on the conditioned subproblem. This relies on some of the same data structures that have been compiled during preprocessing in the master conditioning process, most crucially the pseudo tree to guide the search and the mini-bucket heuristic for AOBB's pruning decision.

Since the variable ordering is transmitted as part of the subproblem specification, recomputing the pseudo tree (or the sub pseudo tree relevant to the subproblem) is an easy task and usually takes a second or two.

The issue is less straightforward regarding the mini-bucket heuristic. Its compilation complexity is exponential in the selected i-bound, which can translate to significant amounts of time – up to several minutes on some instances. In principle, this computation has be repeated on a worker host for each parallel subproblem – a potential source of overhead, in particular for subproblems that turn out to require little actual search. On the other hand, the subproblem's context instantiation can be taken into account (i.e., plugged in) when recomputing the mini-bucket heuristic. This has the potential to reduce the size of the mini-bucket tables and yield a stronger heuristic, which can induce more pruning within the subproblem.

An alternative would be to write the mini-bucket heuristic that has been computed in the master process to a file, transmit that with the subproblem description, and load it into

AOBB on the worker host. However, the tables of a mini-bucket heuristic can often occupy several hundred megabytes, if not 1–2 gigabytes in memory. The cost of transmitting this kind of information across the network, in particular in the context of hundreds of worker hosts, is thus clearly prohibitive in practice. In our implementation, the mini-bucket heuristic is therefore recomputed on the worker hosts for each parallel subproblem as described above.

4.4.2 Parallel Cutoff Characteristics

We now consider the question of optimality regarding the parallel cutoff found by our parallelization scheme. We focus on two particular aspects, namely the "balancedness" of the parallel subproblems (which directly impacts parallel resource utilization) as well as the size of the largest subproblem (which often determines the overall parallel runtime). Namely, we ask whether there is another parallelization frontier of the same size that is more balanced or has a smaller largest subproblem.

As mentioned before, the fixed-depth scheme yields subproblems that have the same structure, yet by design it is oblivious to the actual number of explored nodes (which can vary vastly), even if that number was known. Thus no claim of optimality can be made regarding either the balancedness of parallel subproblems nor the minimality of the largest one – in fact, given the results from Sections 3.2.1 and 4.3.2.1 in particular, it is very likely that the obtained parallel cutoff will be suboptimal in both regards. In the following we therefore focus on the variable-depth parallelization frontier.

4.4.2.1 Largest Subproblem

The size of the largest parallel subproblem is important since it will dominate the overall parallel runtime in the case where the number of subproblems is equal or very close to the number of parallel CPUs.

THEOREM **4.3.** *If the subproblem complexity estimator $\hat{N}$ is exact, i.e., $\hat{N}(n) = N(n)$ for all nodes n, then Algorithm 4.2 will return the parallelization frontier of size p with the smallest possible largest subproblem, i.e., no other parallelization frontier of size p can have a strictly smaller largest subproblem.*

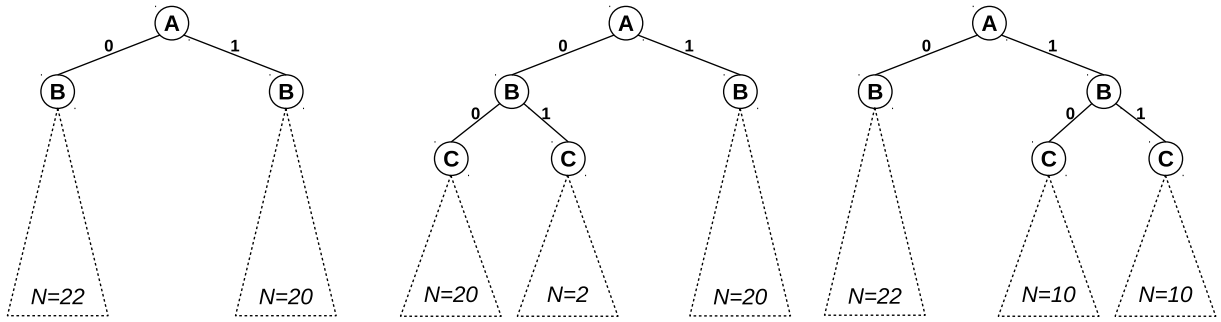
Proof. Induction over p. Base case $p = 1 \to 2$: Trivial. Inductive step $p = k \to k + 1$: Consider a parallelization frontier F_k of size k and denote by n^* the node corresponding to the largest subproblem, i.e., $n^* = \text{argmax}_{n \in F_k} N(n)$. Because $\hat{N}(n) = N(n)$ for all n, Algorithm 4.2 will expand n^* and obtain parallelization frontier F_{k+1} of size $k+1$. Choosing any other $n \in F_k$, $n \neq n^*$ would result in a parallelization frontier F'_{k+1} that still has n^* in it and thus the same $\max_{n \in F'_{k+1}} N(n) = N(n^*)$, which cannot possibly be better than F_{k+1}.

□

This property even holds if we don't assume an exact estimator $\hat{N}$, as long as we can correctly identify the current largest subproblem at each iteration. This is guaranteed, for instance, if the correlation coefficient of estimated and actual complexities (denoted PCC in Chapter 3) is equal to 1, regardless of the estimator's mean squared error (MSE).

4.4.2.2 Subproblem Balancedness

We furthermore aim to characterize the balancedness of the subproblems in the parallelization frontier. To that end, the measure we consider is the variance over subproblem runtimes.



(a) State with 2 subproblems. **(b)** Greedy split of $\{A = 0\}$. **(c)** Alternate split of $\{A = 1\}$.

Figure 4.6: Example of subproblem splitting decision by the greedy, variable-depth parallelization scheme. Applying Algorithm 4.2 in state (a) will lead to (b), while (c) would be more balanced (N denotes each subproblem's complexity).

However, it turns out the we cannot make any claims regarding optimality of the parallel cutoff, even if an exact estimator $\hat{N}$ is available. The following counter example illustrates.

EXAMPLE **4.5.** *Assume we run Algorithm 4.2, the greedy variable-depth parallelization scheme, with a desired subproblem count of* $p = 3$. *Furthermore assume a conditioning space after the first iteration (which splits the root note) as depicted in Figure 4.6a, with two parallel subproblems of size 22 and 20, respectively.*

The greedy scheme will pick the left node $\{A = 0\}$ *for splitting, with two resulting new subproblems of size 20 and 2, respectively, as shown in Figure 4.6b. The average subproblem size is then* $(20 + 2 + 20)/3 = 14$ *with variance* $(6^2 + 12^2 + 6^2)/3 = 72$.

Instead splitting the subproblem $\{A = 1\}$ *on the right, however, would yield two new subproblems, both of size 10, as depicted in Figure 4.6c. The average subproblem size is still* $(22 + 10 + 10)/3 = 14$, *but the variance is lower with* $(8^2 + 4^2 + 4^2)/3 = 32$.

Example 4.5 demonstrates that even with an subproblem complexity estimator, optimality cannot be guaranteed with respect to subproblem balancedness. It also outlines some of the underlying intricacies of the problem we're trying to solve. In particular, by the nature of branch-and-bound search a given subproblem that is split further through conditioning can

yield parts of vastly varying complexity (cf. Figure 4.6b), which is in direct contradiction to our objective of balancing the parallel workload.

4.5 Parallel Redundancies

Section 4.4.1 illustrated the overhead introduced in the parallel AOBB implementation by virtue of the grid paradigm; examples include additional processing time to determine the parallel subproblems as well as delays inherent to the distributed system and grid approach in general. In contrast, this section will investigate in-depth the overhead stemming from redundancies in the actual search process, namely the expansion of search nodes that would not have been explored in pure sequential execution. Consequentially, it is clear that the problem of parallelizing AND/OR search is far from embarrassingly parallel (cf. Section 4.2.1).

We distinguish two principled sources of search space redundancies as follows:

- **Impacted pruning** due to unavailability of bounding information across workers.
- **Impacted caching** of unifiable subproblems across workers.

In the following we investigate these aspects in detail. In particular, we will explain how both issues are caused by the lack of communication among worker nodes. Secondly, we will analyze and bound the magnitude of redundancies from impacted caching using the problem instance's structure.

4.5.1 Impacted Pruning through Limited Bounds Propagation

One of the strengths of AND/OR Branch-and-Bound in particular (and any branch-and-bound scheme in general) lies in exploiting a heuristic for pruning of unpromising subproblems. Namely, the heuristic overestimates the optimal solution cost below a given node, i.e., it provides an upper bound (in a maximization setup). AOBB compares this estimate against the best solution found so far, i.e., a lower bound. If this lower bound exceeds the upper bound of a node n, the subproblem below n can be disregarded, or pruned, since it can't possibly yield an improved solution. Chapter 3 and in particular Section 3.2.1 have demonstrated the significant effect that pruning can have on the size of the explored search space.

One key realization is that the pruning mechanism of branch-and-bound relies inherently on the algorithm's depth-first exploration. Namely, subproblems are solved to completion before the next sibling subproblem is considered, where the best solution found previously is used as a point of reference for pruning. (This depth-first property is also at the core of Chapter 2, where its impact is considered in the context of anytime performance.)

In the context of parallelizing AOBB on a computational grid, however, this property is compromised. Parallel subproblems, as determined by the parallel cutoff, are processed independently and on different worker hosts. And because these hosts typically can't communicate, or aren't even aware of each other, lower bounds (i.e., conditionally optimal subproblem solutions) from an earlier subproblem are not available for pruning in later ones. Here "earlier" and "later" refers to the order in which the subproblems would have been considered by sequential AOBB. The following example illustrates:

Example **4.6.** *Figure 4.7 shows the top part of the search space from Figure 4.3a,. augmented with some cost-related information (assuming a max-product setting): assigning variable A to 0 and 1 incurs cost* 0.9 *and* 0.7*, respectively; the heuristic estimates for the two*

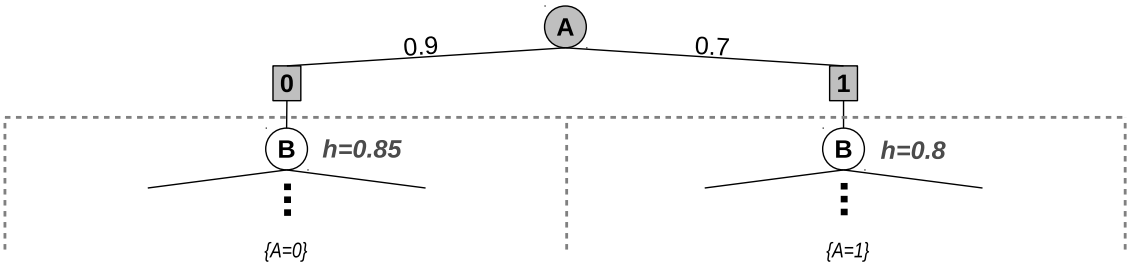


Figure 4.7: Example of impacted pruning across subproblems. Depending on the optimal solution cost to subproblem $\{A = 0\}$, subproblem $\{A = 1\}$ could be pruned. *Max-product* setting.

subproblems below variable B *are* 0.85 *and* 0.8 *, respectively. Assume that the optimal solution to the subproblem below variable* B *for* $\{A = 0\}$ *is* 0.7 *.*

In fully sequential depth-first AOBB, the current best overall solution after exploring $\{A = 0\}$ *is thus* $0.9 \cdot 0.7 = 0.63$ *. When AOBB next considers the right subproblem at node* B *with* $\{A = 1\}$ *, the pruning check compares that overall solution against the heuristic estimate for the subproblem below* B *(including the parent edge labels). And since* $0.63 > 0.7 \cdot 0.8 = 0.56$ *the subproblem below node* B *for* $\{A = 1\}$ *cannot possibly yield an improved overall solution (recall that the heuristic overestimates) and is pruned at this point.*

Now assume a parallel cutoff at fixed depth $d = 1$*, yielding two parallel subproblems rooted at nodes* B *with* $\{A = 0\}$ *and* $\{A = 1\}$ *respectively, as indicated in Figure 4.7. In this case the optimal solution to the left subproblem* $\{A = 0\}$ *would not be available to the worker host processing the subproblem with* $\{A = 1\}$ *on the right. That means that node* B *on the right will be expanded and its children processed depth-first by AOBB in the worker host, resulting in redundant computations.*

The specific effect of the (un)availability of lower bounds and its impact on pruning is hard to analyze. From a practical point of view, however, we aim to address this issue in our implementation as follows: As part of preprocessing in the master host, before the actual exploration of the conditioning space, we run a few seconds of incomplete search (e.g.,

stochastic local search [60] or limited discrepancy search [58, 99]). This yields an overall solution that is possibly not optimal, i.e., it is a lower bound. It is subsequently transmitted with each parallel subproblem, to aid in the pruning decisions of sequential AOBB running on the worker hosts.

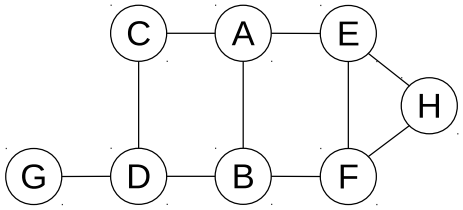
4.5.2 Impacted Caching across Parallel Subproblems

A second aspect that is at least partially compromised by the grid implementation of parallel AOBB lies in the caching of unifiable subproblems. Observed already in Examples 4.3 and 4.4, this section will provide in-depth analysis of this issue – in the context of the graphical model paradigm, we also refer to it as *structural redundancy*.

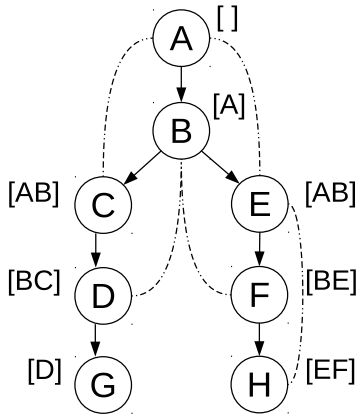
Recall that in AND/OR graph search, certain subproblems can be unified based on their context, i.e., the partial instantiation that separates a subproblem from the rest of the network (cf. Section 1.3.1). The following example expands upon the problem instance from Example 1.8 to reiterate and illustrate:

EXAMPLE **4.7.** *Figure 4.8a shows an example primal graph with eight problem variables A through H. A possible pseudo tree with induced width 2 is shown in Figure 4.8b, annotated with each variable's context. The corresponding context-minimal AND/OR search graph is shown in Figure 4.8c, having a total of 50 AND nodes. We note the following:*

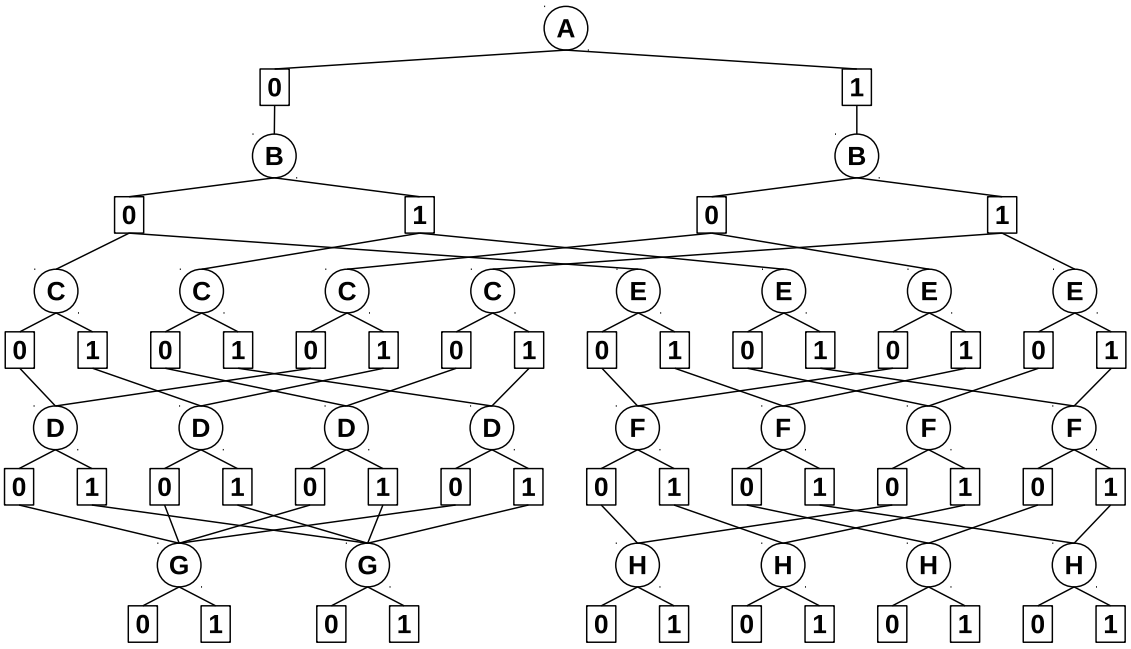
- *Like in Example 1.8, subproblem decomposition is exhibited below variable B, with two children C and E in the pseudo tree.*
- *As before, the context of variables D and F is $\{B, C\}$ and $\{B, E\}$, respectively. There are thus only four possible context instantiations (variable A not being in either context), which allows for caching of the subproblems rooted at D and F, with two incoming edges each from the level above.*



(a) Primal graph with eight variables.



(b) Pseudo tree, induced width 2.



(c) Context-minimal AND/OR search graph.

Figure 4.8: Example problem with eight binary variables, pseudo tree along ordering A, B, C, D, E, F, G, H (induced width 2), and corresponding context-minimal AND/OR search graph.

- *The context of variable G is $\{D\}$. Hence there are only two possible context instantiations, leading to four incoming edges per subproblem rooted at G.*
- *The context of variable H is $\{E, F\}$. With four possible context instantiations, we observe two incoming edges per subproblem rooted at H from the level above.*

As we have noted, some of the caching is compromised by the conditioning process, since the respective cache tables are not shared across worker hosts and parallel subproblems. The following sections quantify this effect. We focus on the fixed-depth parallelization scheme, but the application to a variable-depth parallel cutoff is straightforward. Similar to the state space bound derived for sequential AOBB in Section 3.2.1, our analysis will provide an upper bound on the overall *parallel search space*, i.e., the conditioning space and all parallel subproblem search spaces.

4.5.2.1 Parallel Search Space Bound

We assume a pseudo tree $\mathcal{T}$ with n nodes for a graphical model $(X, D, F, \otimes)$ with variables $X = \{X_1, \ldots, X_n\}$ (cf. Definition 1.1). For simplicity, we assume $|D_i| = k$ for all i, i.e., a fixed domain size k. Let X_i be an arbitrary variable in X, then $h(X_i)$ is the depth of X_i in $\mathcal{T}$, where the root of $\mathcal{T}$ has depth 0 by definition; $h := \max_i h(X_i)$ is the height of $\mathcal{T}$. $L_j := \{X_i \in X \mid h(X_i) = j\}$ is the set of variables at depth j in $\mathcal{T}$, also called a *level*. For every variable $X_i \in X$, we denote by $\Pi(X_i)$ the set of ancestors of X_i along the path from the root node to X_i in $\mathcal{T}$, and for $j < h(X_i)$ we define $\pi_j(X_i)$ as the ancestor of X_i at depth j along the path from the root in $\mathcal{T}$.

As before, by $\text{context}(X_i)$ we denote the set of variables in the context of X_i with respect to $\mathcal{T}$ (cf. Definition 1.8), and $w(X_i) := |\,\text{context}(X_i)|$ is the width of X_i.

DEFINITION **4.1** (conditioned context, conditioned width)**.** *Given a node* $X_i \in X$ *and* $j < h(X_i)$*,* $\text{context}_j(X_i)$ *denotes the* conditioned context of X_i *when placing the parallelization frontier at depth* j *, namely,*

$$\text{context}_j(X_i) \;:=\; \{X' \in \text{context}(X_i) \,|\, h(X') \geq j\} \;=\; \text{context}(X_i) \setminus \Pi(\pi_j(X_i))\,. \quad (4.1)$$

Correspondingly, the conditioned width *of variable* X_i *is defined as* $w_j(X_i) := |\,\text{context}_j(X_i)|\,.$

The following results extend the state space bound derived for sequential AOBB in Chapter 3. Section 3.2.1 in particular showed how, by virtue of context-based subproblem caching, each variable X_i cannot contribute more AND nodes to the context-minimal AND/OR search space than the number of different assignments to its context (times its own domain size), which in our present analysis amounts to $k^{w(X_i)+1}$. Summing over all variables gives the overall state space bound SS, which we will refer to here as SS_{seq} for clarity. We can rewrite it as a summation over the levels of the search space using the notation introduced above:

$$SS_{seq} \;=\; \sum_{i=1}^{n} k^{w(X_i)+1} \;=\; \sum_{j=0}^{h} \sum_{X' \in L_j} k^{w(X')+1}\,. \quad (4.2)$$

Starting from Equation 4.2, we now assume the introduction of a parallelization frontier at fixed depth d. Up to and including level d, caching is not impacted and the contribution to the state space remains the same. Below level d however, caching is no longer possible across the parallel subproblems, while it is not impacted within.

THEOREM **4.4** (Parallel search space)**.** *With the parallelization frontier at depth* d*, the overall number of AND nodes across conditioning search space and all parallel subproblems*

is bounded by:

$$SS_{par}(d) \;=\; \sum_{j=0}^{d} \sum_{X' \in L_j} k^{w(X')+1} \;+\; \sum_{j=d+1}^{h} \sum_{X' \in L_j} k^{w(\pi_d(X'))+w_d(X')+1} \tag{4.3}$$

Proof. The first part of the sum, over levels L_0 through L_d, remains unchanged from Equation 4.2, since the conditioning search space is still subject to full caching. We then note that the variables in level L_d are those rooting the parallel subproblems that are solved by the worker hosts. For a given subproblem root variable $\hat{X} \in L_d$ we can compute the number of possible context instantiations as $k^{w(\hat{X})}$, expressing how many different parallel subproblems rooted at $\hat{X}$ may be generated. For a variable X' that is a descendant of $\hat{X}$ in $\mathcal{T}$ (i.e., $\pi_d(X') = \hat{X}$), the contribution to the search space within a single subproblem is $k^{w_d(X')+1}$, based on its conditioned width $w_d(X')$. The overall contribution of X', across all parallel subproblems, is therefore $k^{w(\hat{X})} \cdot k^{w_d(X')+1} = k^{w(\pi_d(X'))+w_d(X')+1}$; summing this over all variables at depth greater than d yields the second half of the sum in Equation 4.3. □

Observe that $SS_{par}(0) = SS_{par}(h) = SS_{seq}$. For $d = 0$ the entire problem is a single "parallel" subproblem, executed at one worker host. In the case $d = h$ the conditioning space ends up covering the entire search space, solved centrally by the master host.

4.5.2.2 Parallel Search Space Example

This section presents an example to better illustrate Theorem 4.4. Figures 4.9, 4.10, and 4.11 show examples of parallel search spaces resulting from introducing a fixed-depth parallelization frontier to the context-minimal AND/OR search graph from Example 4.7 / Figure 4.8c, with a state space bound of $SS_{seq} = 50$ AND nodes. We explain each figure in detail in the following:

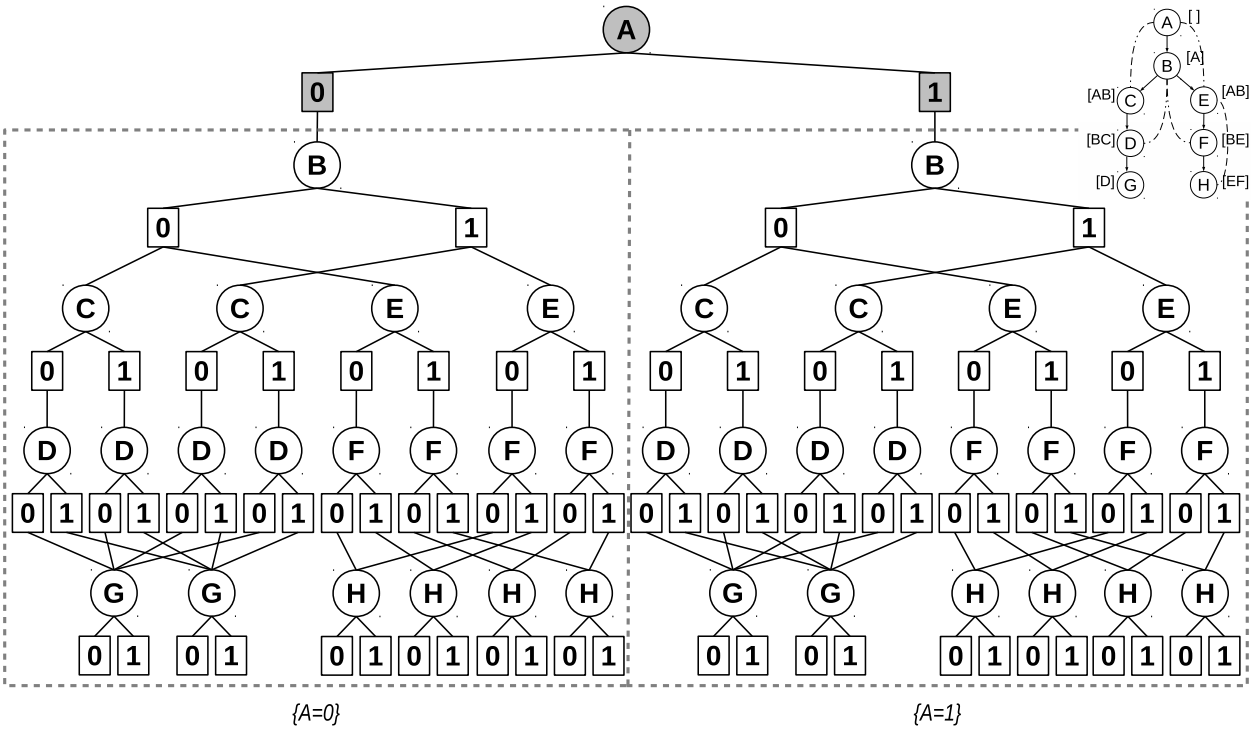


Figure 4.9: Illustration of parallelization impact on caching for AND/OR search graph from Figure 4.8c (page 172), with parallel cutoff at fixed depth $d = 1$. Guiding pseudo tree from Figure 4.8b included for reference.

EXAMPLE **4.8.** *Figure 4.9 is based on a parallel cutoff at depth $d = 1$, i.e., the conditioning set is only $\{A\}$. We note that A is in the context of B which roots the parallel subproblems. It is not in the contexts of D, F, G, or H, though – therefore each of these variables' contribution to the overall search space increases by a factor of two, the domain size of A. Note also that not all caching is voided, as evident by the nodes for G and H, with four and two edges incoming from the level above, respectively. The overall number of AND nodes in the parallel search space of Figure 4.9 is $SS_{par}(1) = 78$; the conditioning space has 2 and each parallel subproblem has 38 AND nodes.*

EXAMPLE **4.9.** *In Figure 4.10 we demonstrate the effect of placing the parallel cutoff at depth $d = 2$, implying a conditioning set of $\{A, B\}$. The root nodes of the parallel subproblems are thus C and E, both of which have context $\{A, B\}$. Compared to a cutoff with $d = 1$, no additional redundancy is introduced with respect to nodes for D and F, since those variables*

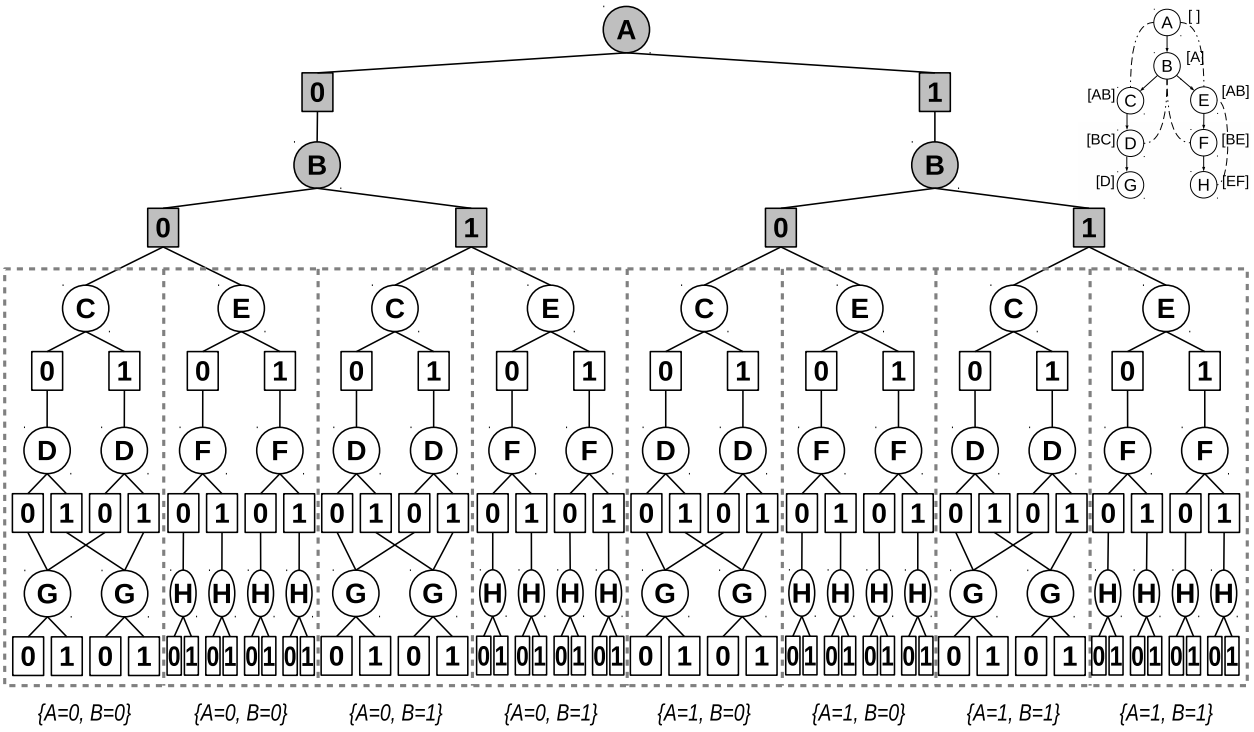


Figure 4.10: Illustration of parallelization impact on caching for AND/OR search graph from Figure 4.8c (page 172), with parallel cutoff at fixed depth $d = 2$. Guiding pseudo tree from Figure 4.8b included for reference.

have B in their context. Variables G and H, however, don't have B in their context, which is why some of caching is lost relative to $d = 1$ and the number of nodes corresponding to G and H increases twofold across all parallel subproblems. Again, note that some caching is preserved for G, with two incoming edges each. The overall number of AND nodes in the parallel search space of Figure 4.10 is $SS_{par}(2) = 102$; the conditioning space has 6, and the parallel subproblems have 10 (root C) or 14 (root E) AND nodes, respectively.

Example **4.10.** *The parallel search space in Figure 4.11 is based on a parallel cutoff at depth $d = 3$, with a conditioning set $\{A, B, C, E\}$. That makes D and F the root nodes of the parallel subproblems, which have context $\{B, C\}$ and $\{B, E\}$, respectively. In particular, the parallel subproblems don't depend on A and can hence be cached at the leaves of the conditioning space in the master host, as indicated by two incoming edges each in Figure 4.11. In contrast to Figure 4.10 with cutoff $d = 2$, however, caching for nodes G and H is*

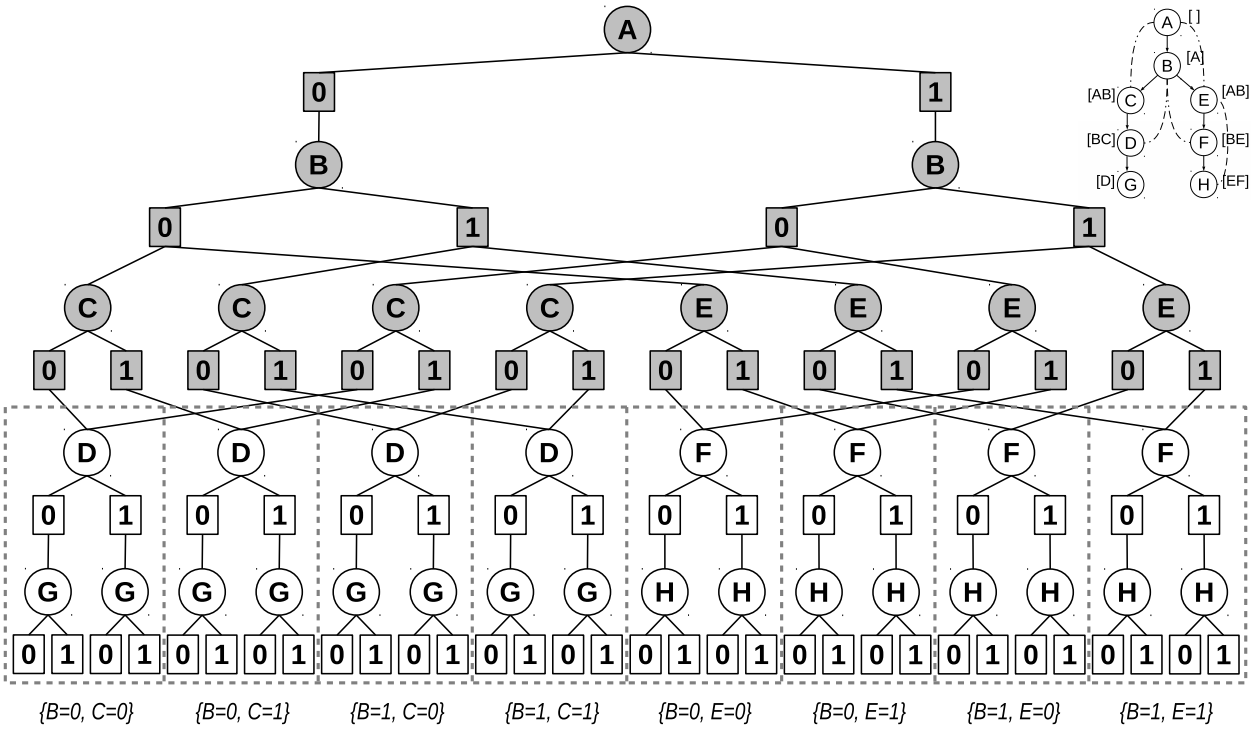


Figure 4.11: Illustration of parallelization impact on caching for AND/OR search graph from Figure 4.8c (page 172), with parallel cutoff at fixed depth $d = 3$. Guiding pseudo tree from Figure 4.8b included for reference.

no longer applicable, since the respective unifiable subproblems are now spread across different parallel jobs. Overall, the number of AND nodes in the parallel search space of Figure 4.11 is $SS_{par}(3) = 70$*; the conditioning space and each parallel subproblem have 22 and 6 AND nodes, respectively.*

4.5.2.3 Analysis of Structural Redundancy

Table 4.1 summarizes the properties of the parallel search space of the example problem and pseudo tree given in Figure 4.8 for the full range of parallel cutoff depths $0 \leq d \leq 5$. We list the overall parallel state space size $SS_{par}(d)$ as well as the size of the conditioning space, the number of parallel subproblems, and the size of the largest one.

	d					
	0	1	2	3	4	5
parallel space $SS_{par}(d)$	50	78	102	70	50	50
conditioning space	0	2	6	22	38	50
no. of subproblems	1	2	8	8	6	0
max. parallel subproblem	50	38	14	6	2	–
cond. space + max. subprob	50	40	20	28	40	50

Table 4.1: Parallel search space statistics for example problem from Figure 4.8 with varying parallel cutoff depth d.

As expected, we note that $SS_{par}(0)$ and $SS_{par}(5)$ match the sequential state space bound $SS_{seq} = 50$. In the case of $d = 0$ we have no conditioning and one large parallel subproblem, for $d = 5$ the entire problem space is covered by the conditioning space in the master host. We also see that $SS_{par}(d)$ increases monotonically until it peaks at $d = 2$, from which point on it decreases monotonically. This convexity, however, is owed to the simplicity of the example problem – in general there could be several local maxima, depending on the structure of the problem and the chosen pseudo tree. However, it's easy to see that $SS_{par}(d) \geq SS_{seq}$ for $0 \leq d \leq h$, i.e., conditioning can only increase the size of the parallel search space.

Clearly, the parallel search space measure $SS_{par}(d)$ does not account for parallelism – in other words, while the overall parallel search space bound might go up with increased cutoff depth d, we hope that the additional parallelism will compensate for it, yielding a net gain in terms of parallel run time.

We can therefore instead consider a metric based on the assumption that the parallel subproblems are solved in parallel. In particular, if the number of CPUs exceeds the number of subproblems, we only need to consider the size of the largest subproblem an add it to the size of the conditioning space. The resulting measure can be seen as indication of parallel performance, it is given in the last row of Table 4.1.

We observe that the minimum is actually achieved at $d = 2$, where $SS_{par}(d)$ takes its maximum. This observation is a manifestation of *Amdahl's Law* as discussed in Section

4.2.4. Recall its statement that parallel performance and speedup is limited by the non-parallelizable portion of a computation. In our context, a deeper cutoff depth d increases the size of the conditioning space, which is limited to sequential, non-parallelizable exploration in the master host.

We conclude by pointing out that the above discussion is helpful in understanding some of the trade-offs regarding the choice of parallelization frontier in parallel AOBB. The practical implications, however, are limited. Like the sequential state space bound, denoted SS_{seq} above, the parallel search space metrics considered above are upper bounds on the number of nodes expanded by parallel AOBB. Sections 3.2.1 and 4.3.2.1 demonstrated that these bounds are typically very loose in practice since they don't account for determinism in the problem specification and the algorithm's pruning power. The following section will provide additional evidence of this, with a more comprehensive empirical confirmation and analysis to follow in Section 4.6.

4.5.2.4 More Practical Examples

The above example problem helped in visualizing the issue of structural redundancies, but it was also quite simple. To provide a better understanding of the practical implications on real-world problems, here we anticipate the more substantial evaluation in Section 4.6 and present empirical results for two example instances, pedigree51, a linkage analysis instance, and largeFam3-13-59, a haplotyping problem. Both problems are very hard to solve, with 1152 and 2711 variables, maximum domain size of 5 and 3, and induced width of 39 and 31, respectively, taking 28 and 5 $^1/_2$ hours with sequential AOBB.

For each of these two problems Figure 4.12 displays the following as a function of the cutoff depth d:

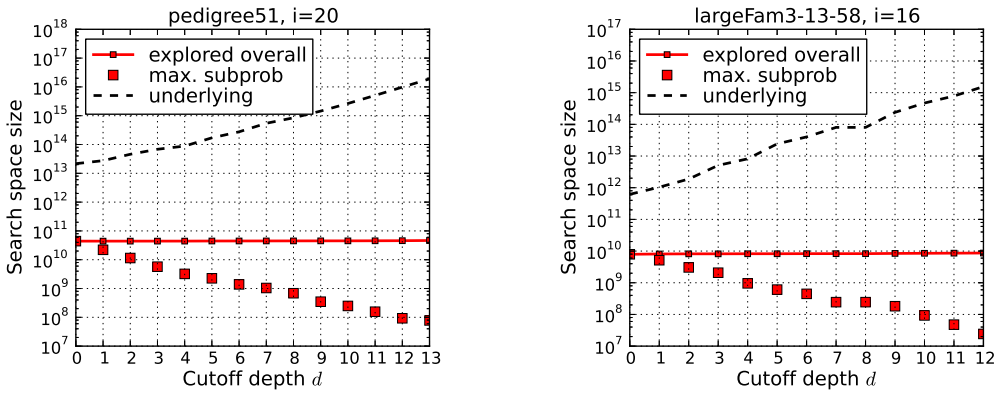


Figure 4.12: Comparison of underlying parallel search space size vs. size of explored search space (summed across subproblems) and largest subproblem, as a function of the cutoff depth d.

- "underlying": the size of *underlying* parallel search space $SS_{par}(d)$ (Equation 4.3).
- "explored overall": the overall *explored* search space, i.e., the number of nodes expanded across the master conditioning space and all subproblems at the given level,
- "max. subprob": the explored size of the largest subproblem at level d.

We can make the following observations:

- The underlying parallel search space, plotted with a dashed line, indeed grows exponentially with d (note the vertical log scale) as a consequence of structural redundancies introduced by parallelization.
- However, the overall explored search space grows very little, if at all, as d is increased – far from the rapidly growing underlying search space bound. Closer analysis in Section 4.6.6 will show, in fact, that the overall explored search space grows linearly in with increasing d, in many cases with a small slope.
- In line with previous results, the upper bound presented by the size of the underlying parallel search space is very loose (by orders of magnitude), even for low cutoff depths d.

- Finally, we see that subproblem complexity, expressed in Figure 4.12 via the explored size of the largest one, does indeed decrease as the cutoff depth d is deepened.

Overall these results tentatively confirm that the redundancies derived above, while substantial in theory, are far less pronounced in practice. We will revisit this issue in-depth in Section 4.6.6.

4.6 Empirical Evaluation

In this section we will evaluate and analyze the performance of parallel AOBB, as proposed in the previous Section 4.3. We first outline the experimental setup and grid environment in Section 4.6.1. Section 4.6.2 describes the problem instances on which we evaluate performance. In Section 4.6.3 we outline our evaluation methodology and illustrate a number of central performance characteristics through three in-depth case studies. Building on that, detailed evaluation is then performed for the different performance measures defined in Section 4.2.4:

- Section 4.6.4 surveys the overall parallel performance results in terms of parallel runtime T_{par} and corresponding speedup S_{par} .
- In Section 4.6.5 we discuss the results in the context of parallel resource utilization U_{par} , a secondary performance metric introduced earlier.
- Similarly, the subject of parallel redundancies (detailed in Section 4.5) and the resulting overhead O_{par} is investigated in Section 4.6.6.
- Related to the issue of speedup, Section 4.6.7 will specifically focus on the question of scaling of the parallel performance with increasing number of CPUs.

Finally, Section 4.6.8 will summarize the empirical results as well as our analysis of them.

The experiments that form the basis of our evaluation encompass over 1400 parallel runs across 75 benchmark cases (i.e., combinations of problem instance and mini-bucket i-bound), amounting to the equivalent of approx. 91 thousand CPU hours, i.e., over 10 years of computation time. A comprehensive subset of results is presented and discussed in this section, with complete tables included in Appendix B for reference.

4.6.1 Experimental Setup

Experimental evaluation was conducted on an in-house cluster of 27 computer systems, each with dual 2.67 GHz Intel Xeon 6-core CPUs and 24 GB of RAM, running a recent 64-bit Linux operating system. This makes for a total of 324 CPU cores with 2GB of available RAM per core. Note that each core is treated separately by the Condor grid management system. In the following we will hence use the terms "CPU," "core," and "processor" interchangeably, always referring to a single worker in the Condor grid system.

The master host of our parallel scheme as well as the Condor grid scheduler was a separate, dedicated 2.8 GHZ Intel Xeon quad-core system with 16 GB of RAM. This system was chosen because of its additional redundancy through RAID hard drive mirroring – in principle any of the other, "regular" machines could have been designated the master host without significantly altering the results. All systems are connected over a local gigabit network.

The parallel schemes described above were implemented in C++. The full source code is available under an open-source GPL license at `http://github.com/lotten/daoopt`.

Algorithm 4.3 Pseudo code for simulation of parallel run with p CPUs.

Given: Master host preprocessing time T_{pre}, list of subproblem runtimes $T_{par}^1, \ldots, T_{par}^C$, target number of CPUs p.

Output: Overall parallel runtime, i.e., termination of last worker.

```
1: workers ← array(p, 0)  // Array of size p with all entries set to 0.
2: for i ← 1 to p :
3:    workers[i] ← T_pre  // Initialize each worker time to T_pre
4: for j ← 1 to C :   // over subproblems
5:    i* ← arg min_i workers[i]
6:    workers[i*] ← workers[i*] + T_par^j  // Assignment of subproblem j to worker i*
7: return  max_i workers[i]
```

4.6.1.1 Mapping Subproblems to CPUs

We point out that the number of subproblems does not have to match the number of CPUs. Clearly, if the CPU count exceeds the number of subproblems, some CPUs will sit idle. In the case where the number of subproblems is greater than the available number of CPUs, however, the Condor system holds submitted parallel jobs in a first-in-first-out (FIFO) queue; each time a processor completes a subproblem and reports its solution, it is immediately assigned the next subproblem in the queue.

4.6.1.2 Simulating Large Number of CPUs

With 324 CPUs at our disposal, as noted above, we are able to directly run any parallel experiment targeting that number of CPUs (or fewer) and simply observe and record its performance at runtime. To consider experiments with a more than 324 CPUs, we can exploit the independent nature of the parallel subproblems to efficiently simulate the respective results.

In particular, we can simply run the parallel scheme with fewer CPUs, as available, and record each parallel subproblem's runtime, including network transmission, job setup time, and other distributed overhead (see Section 4.4.1). Afterwards, simulating execution with p

CPUs is straightforward, with pseudo code shown in Algorithm 4.3. Given are the master host preprocessing time T_{pre} and an ordered list of recorded subproblem runtimes T_{par}^{i}. We maintain an array of worker runtimes, with all p entries initially set to the master preprocessing time T_{pre}, since the worker hosts are just idle during that period (lines 1–3). We then iterate over the parallel subproblems in order, at each point adding the runtime of subproblem j to the worker node with the lowest current runtime (lines 4–6). Finally, the longest worker runtime is returned as the overall parallel runtime in line 7.

Instead of returning only the overall parallel runtime, Algorithm 4.3 can easily be extended to return more diverse metrics, as outlined in Section 4.2.4.

We confirmed the relative accuracy of this simulation by comparing its output on runs with fewer than 324 CPUs against the actual results. For very large number of CPUs (tens of thousands) there are likely scaling effects like network saturation that are not captured by the simulation, but for CPU counts on the same order of magnitude as our experimental setup we are confident that the simulated results have high accuracy.

4.6.2 Benchmark Problem Instances

This section will introduce and describe the set of benchmarks that form the basis of our experimental evaluation. We consider instances from the same four problem classes as in Chapter 3. The description of each problem class from Section 3.4.1 in particular is repeated below:

- **Linkage analysis ("pedigree"):** Each of these networks is an instance of a genetic linkage analysis problem on a particular pedigree, i.e., a family ancestry chart over several generations, annotated with phenotype information (observable physical traits, inheritable diseases, etc.) [40, 41]. Originally aimed at $P(e)$ sum-product likelihood

computation, these problems have gained popularity as MPE benchmarks due to their complexity and real-world applicability and have been included in recent inference competitions [23, 35].

- **Haplotype inference ("largeFam"):** These networks also encode genetic pedigree instances into a Bayesian network. However, the encoded task is the haplotyping problem, which differs from linkage analysis and necessitates different conversion and data preprocessing steps to generate the graphical model input to our algorithms [1, 39].

- **Protein side-chain prediction ("pdb"):** These networks correspond to side-chain conformation prediction tasks in the protein folding problem [115]. The resulting instances have relatively few nodes, but very large variable domains, forcing a very low mini-bucket i-bound of 3 and generally rendering most instances very complex.

- **Grid networks ("75-"):** Randomly generated grid networks of size 25x25 and 26x26 with roughly 75% of the probability table entries set to 0. From the original set of problems used in the UAI'08 Evaluation, only a handful proved difficult enough for inclusion here [23].

Statistics for the pedigree instances are shown in Table 4.2, largeFam problems are covered in Table 4.3, pdb instances are listed in Table 4.4, while Table 4.5 contains details for the grid benchmarks. For each problem we list the number of problem variables n , the number of cost functions m , the maximum domain size k , and the induced width w and pseudo tree height h along a given minfill variable ordering. We also include the runtime, in seconds, of sequential AOBB for one or more i-bounds as listed (which determines the heuristic accuracy).

								Subproblems for fixed depth d												
instance	n	m	k	w	h	i	T_{seq}	1	2	3	4	5	6	7	8	9	10	11	12	13
ped13	1077	1077	3	32	102	8	252654	2	4	8	16	32	64	128	256	512	1024	2048	4096	6144
						9	102385	2	4	8	16	32	64	128	256	512	1024	2048	4096	6144
ped19	793	793	5	25	98	16	375110	4	12	48	144	288	1440	2880	5752	7672	11254	14968		
ped20	437	437	5	22	60	3	5136	2	6	12	32	96	160	480	800	3200	6400			
						4	2185	2	6	12	32	96	160	480	800	3200	6400			
ped31	1183	1183	5	30	85	10	1258519	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
						11	433029	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
						12	16238	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
ped33	798	798	4	28	98	4	6010	2	3	6	6	12	24	48	96	192	384	768	1536	1536
ped34	1160	1160	5	31	102	10	962006	3	5	10	20	30	60	90	180	360	716	952	1896	3752
						11	350574	3	5	10	20	30	60	90	180	360	720	956	1912	3808
						12	96122	3	5	10	20	30	60	90	180	360	716	948	1896	3728
ped39	1272	1272	5	21	76	4	6632	2	4	8	16	64	128	384	768	1152	2304	4608		
						5	2202	2	4	8	16	64	128	384	768	1152	2304	4608		
ped41	1062	1062	5	33	100	9	25607	3	8	16	32	64	128	176	352	704	1408	2176	4352	8556
						10	46819	3	8	16	32	64	128	176	352	704	1408	2176	4352	8576
						11	27583	3	8	16	32	64	128	176	352	704	1408	2176	4352	8460
ped44	811	811	4	25	65	5	207136	2	4	8	16	64	112	336	560	1120	2240	4480	8960	17920
						6	95830	2	4	8	16	64	112	336	560	1120	2240	4480	8960	17920
ped50	514	514	6	17	47	3	4135	2	4	24	144	720	2160	5760	14401					
						4	1780	2	4	24	144	720	2160	5760	14400					
ped51	1152	1152	5	39	98	20	101788	2	4	8	16	32	64	128	256	512	1024	2048	4064	7968
						21	164817	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
ped7	1068	1068	4	32	90	6	118383	2	4	12	32	96	160	480	640	1280	1280	2560	3840	7680
						7	93380	2	4	12	32	96	160	480	640	1280	1280	2560	3840	7680
						8	30717	2	4	12	32	96	160	480	640	1276	1276	2552	3816	7588
ped9	1118	1118	7	27	100	6	101172	2	4	8	16	32	32	64	128	256	512	1024	2048	4096
						7	58657	2	4	8	16	32	32	64	128	256	512	1024	2048	4096
						8	41061	2	4	8	16	32	32	64	128	256	512	1024	2048	4096

Table 4.2: Statistics for **pedigree linkage** instances with number of subproblems generated by fixed-depth parallel AOBB at various depths d.

4.6.2.1 Cutoff Depth and Subproblem Count

The remaining columns of Tables 4.2 through 4.5 list the number of parallel subproblems generated when running parallel AOBB with a parallelization frontier at various fixed depths d as indicated; i.e., in some cases several subproblems might have been pruned by the master process (recall that the conditioning space is also processed by AOBB). These subproblem counts will serve as a reference for subsequent variable-depth experiments in the empirical evaluation of Sections 4.6.3 through 4.6.6.

								Subproblems for fixed depth d											
instance	n	m	k	w	h	i	T_{seq}	1	2	3	4	5	6	7	8	9	10	11	12	13
lF3-11-57	2670	2670	3	37	95	15	121311	2	4	6	18	30	30	60	60	120	180	360	1080	1440
						16	35820	2	4	6	18	30	30	60	60	120	180	360	1080	1440
						17	18312	2	4	6	18	30	30	60	60	120	180	360	1080	1440
lF3-11-59	2711	2711	3	32	73	14	35457	3	5	10	10	30	50	150	200	600	1000	2000	2000	4000
						15	8523	3	5	10	10	30	50	150	200	596	992	1962	1962	3886
						16	3023	3	5	10	10	30	50	150	200	600	1000	1999	1999	3992
lF3-13-58	3352	3352	3	31	88	14	46464	2	4	12	20	60	100	200	200	600	1200	2000	4000	6400
						16	20270	2	4	12	20	60	100	200	200	600	1200	1998	3990	6390
						18	7647	2	4	12	20	60	100	200	200	591	1181	1958	3858	
lF3-15-53	3384	3384	3	32	108	17	345544	2	4	12	16	34	46	78	201	358	632	1093	1927	2831
						18	98346	2	4	12	16	32	44	68	165	284	526	912	1572	2496
lF3-15-59	3730	3730	3	31	84	18	28613	2	4	8	20	40	80	240	476	942	1855	3633	7098	13781
						19	43307	2	4	8	20	40	80	240	476	936	1830	3571	6964	13482
lF3-16-56	3930	3930	3	38	77	15	1891710		3	9	15	43	71	205	470	934	934	1827	2707	7582
						16	489614	2	3	9	15	42	70	201	455	900	900	1766	2629	7122
lF4-12-50	2569	2569	4	28	80	13	57842	3	12	24	72	288	864	3456	5760					
						14	33676	3	12	24	72	288	864	3456	5760					
lF4-12-55	2926	2926	4	28	78	13	104837	2	4	8	16	64	128	256	512	1024	1024	1792	1792	3072
						14	25905	2	4	8	16	48	96	192	384	768	768	1536	1536	3072
lF4-17-51	3837	3837	4	29	85	15	10607	2	4	4	8	16	32	40	56	128	152	176	352	400
						16	66103	2	4	8	16	32	64	80	112	256	304	352	704	800

Table 4.3: Statistics for **largeFam haplotyping** instances with number of subproblems generated by fixed-depth parallel AOBB at various depths d.

								Subproblems for fixed depth d					
instance	n	m	k	w	h	i	T_{seq}	1	2	3	4	5	6
pdb1a6m	124	521	81	15	34	3	198326	9	81	511	15318		
pdb1duw	241	743	81	9	32	3	627106	9	54	784	15081		
pdb1e5k	154	587	81	12	43	3	112654	66	1046	11321			
pdb1f9i	103	387	81	10	24	3	68804	81	6534				
pdb1ft5	172	645	81	14	33	3	81118	27	118	5281			
pdb1hd2	126	448	81	12	27	3	101550	79	3777				
pdb1huw	152	587	81	15	43	3	545249	9	42	293	654	1588	2597
pdb1kao	148	568	81	15	41	3	716795	27	215	752	3241		
pdb1nfp	204	791	81	18	38	3	354720	6	48	336	3812		
pdb1rss	115	448	81	12	35	3	378579	8	109	908	1336		
pdb1vhh	133	556	81	14	35	3	944633	27	1842	67760			

Table 4.4: Statistics for **pdb side-chain prediction** instances with number of subproblems generated by fixed-depth parallel AOBB at various depths d.

								Subproblems for fixed depth d													
instance	n	m	k	w	h	i	T_{seq}	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
75-25-1	624	626	2	38	111	12	77941	2	4	8	16	16	32	64	128	192	192	192	384	768	1152	2112
						14	15402	2	4	8	8	8	16	32	64	96	96	192	288	576	864	1584
75-25-3	624	626	2	37	115	12	104037	2	4	4	6	6	12	24	48	48	72	144	288	576	960	1536
						15	33656	2	4	4	6	6	12	24	48	48	72	144	288	576	960	1536
75-25-7	624	626	2	37	120	16	297377	2	3	6	12	24	36	72	144	216	288	504	1008	2016	2688	3360
						18	21694	2	3	6	12	24	36	72	144	216	288	504	1008	2014	2661	3325
75-26-10	675	677	2	39	124	16	46985	2	4	8	8	16	16	32	64	128	192	384	384	768	1280	1280
						18	26855	2	4	8	8	16	24	48	80	160	240	480	480	960	1216	1216
75-26-2	675	677	2	39	120	16	25274	2	4	8	12	24	48	96	144	288	384	640	1280	1280	2560	3840
						20	8053	2	4	8	12	24	48	96	144	288	384	640	1280	1280	2560	3840
75-26-6	675	677	2	39	133	10	199460	2	4	8	16	32	64	128	128	128	256	384	576	1152	2304	4608
						12	64758	2	4	8	16	32	64	128	128	128	256	384	576	1152	2304	4608
75-26-9	675	677	2	39	124	16	59609	2	4	8	16	24	48	96	120	240	480	960	1920	3840	3840	7680
						18	66533	2	4	8	16	24	48	96	120	240	480	960	1920	3840	3840	7680
						20	5708	2	4	8	16	24	48	96	120	240	320	640	1280	2560	2560	5120

Table 4.5: Statistics for **grid network** instances with number of subproblems generated by fixed-depth parallel AOBB at various depths d.

Note that not all cutoff depths were run for every problem instance. In particular, a relatively deep parallel cutoff makes little sense for easy problems (with low T_{seq}) for a number of reasons:

- Conceptually, we quickly approach the limits of Amdahl's Law, as described in Section 4.2.5. Namely, the preprocessing and central conditioning step make up an exceedingly large proportion of the computation, quickly limiting the attainable speedup.
- A deep cutoff on an easy problem instance will predominantly yield near-trivial subproblems. This leads to "thrashing," where the distributed overhead from communication and job setup time far exceeds the actual computation time.
- Finally, our current implementation hits the limitations of the operating system (in particular, the underlying file system) when working with tens of thousands of subproblems, due to its reliance on temporary files for subproblem solutions and statistics. In practice, this causes a variety of error messages and generally unpredictable behavior and is extremely challenging to troubleshoot. Bypassing these limits is not impossi-

ble, but would require re-engineering and re-implementing major parts of the parallel system.

Looking at the progression of subproblem counts for each problem instance, we notice that in most cases this number increases by an integer factor – the domain size of the variable that was added to the conditioning set. For instance, for ped13 (Table 4.2) the subproblem count increases by a constant factor of 2 with each depth level. In other cases this factor varies between different integer values from level to level – for for lF4-12-50 in Table 4.3, for example, this sequence of factors is 3, 2, 2, 3, 4, 3, 4 from $d = 1$ through $d = 7$.

Secondly, we observe instances where the relative subproblem count increase is not always integer, or in fact doesn't increase at all from one depth level to the next. Examples include ped9 (Table 4.2), where the subproblem count remains the same going from $d = 5$ to $d = 6$, or lF3-16-56 (Table 4.3), when going from $d = 9$ to $d = 10$. Also in case of lF3-16-56, the move from $d = 4$ to $d = 5$ or $d = 10$ to $d = 11$, for instance, sees the number of subproblems grow from by a factor of slightly less than 3 and 2, respectively. These can be attributed to either determinism, where the conditioning instantiates a cost function with value 0 (assuming a max-product setting), or to pruning based on the mini-bucket heuristic, where AOBB can discard some of the parallel subproblems based on the heuristic estimate of their solution cost.

4.6.2.2 Choice of Mini-bucket i-bound

The i-bound values shown in Tables 4.2 through 4.5 were chosen according to two criteria:

- Feasibility: Larger i-bounds imply a larger mini-bucket structure (size exponential in i), so the limit of 2 GB memory per core implies a natural boundary. For instance,

the large domain sizes of up to $k = 81$ render $i = 3$ the highest possible value for pdb instances (Table 4.4).

- Complexity: In several cases we lowered the i-bound from its theoretical maximum (given the 2 GB memory limit) to make the problem instance harder to solve and more interesting for applying parallelism. Ped7 and ped9, for example, can be solved sequentially in minutes with higher i-bounds.

In many cases we choose to examine a problem instance in combination with more than one i-bound. This will also allow us to investigate how the heuristic strength impacts the parallel performance.

4.6.2.3 A Note on Problem Instance Selection

We point out that finding suitable problem instances is a significant, time-consuming challenge in itself, specifically when targeting parallelism on the scale of hundreds of CPUs. In particular, many problem instances that are solved by AOBB in seconds or minutes are of little interest in the context of parallelization – Amdahl's Law and the inherent distributed overhead severely limit the potential gain in these cases. For instance, if parallel AOBB spends 1 minute of non-parallelizable preprocessing in the master process on a problem that only takes sequential AOBB 20 minutes to solve, the best theoretically attainable speedup is 20. And in practice this is only exacerbated by the overhead from communication and job setup, which increasingly dominates the solution time for very small subproblems. On the other hand there are several hard problems that remain infeasible within realistic time frames even for parallel AOBB on several hundred CPUs – detecting this infeasibility is a costly endeavor, often simply achieved through trial and error.

4.6.3 Methodology and Three In-depth Case Studies

This section provides an introduction to the methodology of our experiments. We put some earlier remarks into context and then provide hands-on illustration by studying in detail the parallel performance of three example instances. The results presented in this section set the stage for the more comprehensive evaluation in subsequent sections.

Given a specific problem instance, we proceed as follows: we first run parallel AOBB with fixed-depth cutoff (Algorithm 4.1) on all benchmark instances for varying cutoff depths d, as stated previously. Besides monitoring the actual parallel performance through the runtime T_{par}, we also record the number of parallel subproblems generated as a function of the cutoff depth d, as indicated Tables 4.2 through 4.5 in Section 4.6.2. We then run parallel AOBB with variable-depth cutoff using these recorded subproblem counts as input (called p in Algorithm 4.2) and record the parallel performance. This allows us to directly compare the two schemes.

Somewhat orthogonal to the number of subproblems is the number of worker hosts (parallel processors) that participate in the parallel computation – as mentioned previously, if the number of subproblems exceeds the number of CPUs, subproblems get assigned in a first-come, first-served manner. Of particular interest in this context is the issue of *scaling*, where one considers how the parallel speedup $S_{par} = \frac{T_{seq}}{T_{par}}$ changes with the number of parallel resources. Ideally, speedup scales linearly with the number of processors. However, since parallelizing AND/OR Branch-and-Bound is far from embarrassingly parallel, as detailed in Section 4.5, this is unrealistic in our case.

With this in mind we choose to conduct all experiments with 20, 100, and 500 CPUs, to capture and assess small-, medium-, and relatively large-scale parallelism. Note that results on 500 CPUs are simulated based on runs with just over 300 cores, as described in Section

instance	i	T_{seq}	#cpu	Cutoff depth d = 2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
				(p=4)		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
lF3-15-59 n=3730 k=3 w=31 h=84	19	43307	20	15858	15694	5909	5470	3649	2845	2744	2501	3482	3505	7222	7238
			100	15858	15694	5909	5470	3434	2247	1494	723	928	741	1540	1536
			500	15858	15694	5909	5470	3434	2247	1414	573	692	260	415	399
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
ped44 n=811 k=4 w=25 h=65	6	95830	20	26776	26836	9716	9481	6741	6811	7959	7947	10103	9763	12418	12472
			100	26776	26836	9716	9481	2344	3586	1799	1700	2126	2276	2545	2543
			500	26776	26836	9716	9481	1659	3586	583	886	536	905	569	824
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
ped7 n=1068 k=4 w=32 h=90	6	118383	20	35387	58872	12338	58121	9031	8515	9654	7319	8705	7582	8236	7693
			100	35387	58872	11956	58121	5122	7690	4860	2306	3929	1814	2644	1649
			500	35387	58872	11956	58121	4984	7690	4359	2086	3294	1301	1764	943

(a) Results for parallel runtime, in seconds.

instance	i	T_{seq}	#cpu	Cutoff depth d = 2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
				(p=4)		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
lF3-15-59 n=3730 k=3 w=31 h=84	19	43307	20	2.73	2.76	7.33	7.92	11.87	15.22	15.78	17.32	12.44	12.36	6.00	5.98
			100	2.73	2.76	7.33	7.92	12.61	19.27	28.99	59.90	46.67	58.44	28.12	28.19
			500	2.73	2.76	7.33	7.92	12.61	19.27	30.63	75.58	62.58	166.57	104.35	108.54
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
ped44 n=811 k=4 w=25 h=65	6	95830	20	3.58	3.57	9.86	10.11	14.22	14.07	12.04	12.06	9.49	9.82	7.72	7.68
			100	3.58	3.57	9.86	10.11	40.88	26.72	53.27	56.37	45.08	42.10	37.65	37.68
			500	3.58	3.57	9.86	10.11	57.76	26.72	164.37	108.16	178.79	105.89	168.42	116.30
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
ped7 n=1068 k=4 w=32 h=90	6	118383	20	3.35	2.01	9.59	2.04	13.11	13.90	12.26	16.17	13.60	15.61	14.37	15.39
			100	3.35	2.01	9.90	2.04	23.11	15.39	24.36	51.34	30.13	65.26	44.77	71.79
			500	3.35	2.01	9.90	2.04	23.75	15.39	27.16	56.75	35.94	90.99	67.11	125.54

(b) Corresponding parallel speedup results, relative to T_{seq}.

Table 4.6: Subset of parallel results on select problem instances. Each entry lists the number of parallel subproblems p as well as, from top to bottom, the performance with 20, 100, and (simulated) 500 parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). Each row's best value is highlighted in gray.

4.6.1; the 500 CPU count is close enough to our actual parallel setup that we feel very confident in the accuracy of the simulated runtimes.

Two different parallel schemes as well as varying number of subproblems and parallel processors lead to a staggering amount of experimental data, not all of which can be presented here. We thus only present a limited yet comprehensive subset, with full result tables given in Appendix B.

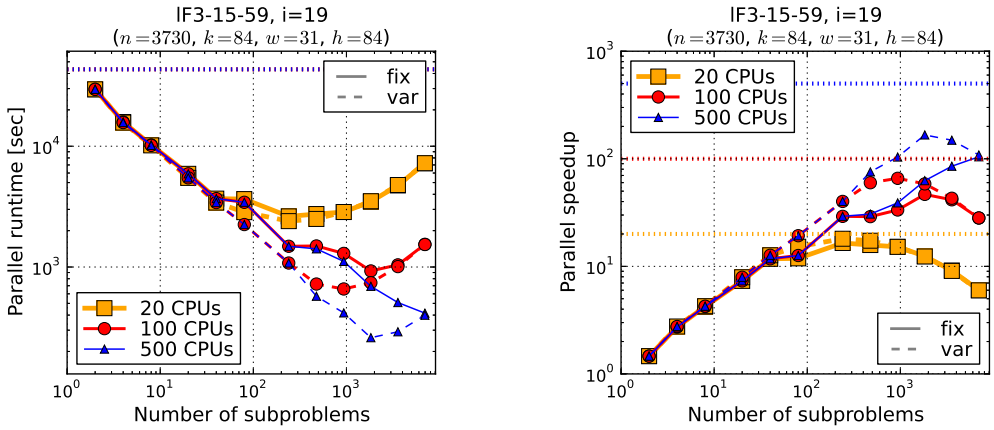
4.6.3.1 Overview of Three Case Studies

We begin by going over detailed results for a number of problem instances, to better explain the experimental methodology and highlight relevant performance characteristics. Table 4.6 shows a subset of parallel results on three particular problem instances, largeFam3-15-59 with i-bound 19, pedigree44 with $i = 6$, as well as pedigree7 with $i = 6$. Sequential runtimes using AOBB are 12 hours, over 26 hours, and almost 33 hours, respectively. Section 4.6.2 gave the instances' induced width as 31, 25, and 32, respectively; for the full set of instance properties refer to Tables 4.2 and 4.3. As discussed above, we use the subproblem count yielded by fixed-depth parallelization (Algorithm 4.1) as input to the variable-depth scheme (Algorithm 4.2), so that we obtain a corresponding parallelization frontier for each fixed-depth cutoff.

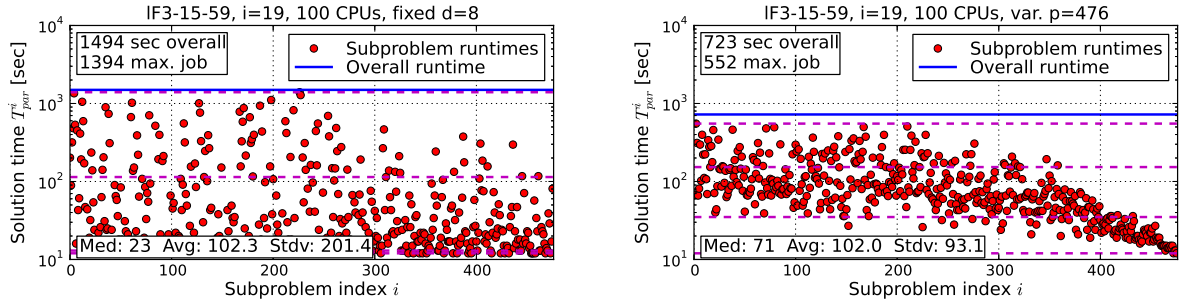
Table 4.6 presents overall parallel results for a subset of cutoff depths $d \in \{2, 4, 6, 8, 10, 12\}$. In particular, Table 4.6a shows the overall parallel runtimes (denoted T_{par} earlier), i.e., the time from the start of preprocessing in the master host to the termination of the last worker. The corresponding parallel speedup values, i.e., the ratio of sequential over parallel runtime, are given in Table 4.6b.

Each field in the two tables contains several values: at the top, the subproblem count p is specified as obtained at the given cutoff depth d for this particular instance; then on the left, with column title "fix," the result (runtime or speedup) of the fixed-depth parallel scheme with, from top to bottom, 20, 100, and 500 CPUs is listed; similarly, on the right (column "var"), we show the result of variable-depth parallel AOBB with 20, 100, and 500 CPUs, run with the same number of subproblems as the fixed-depth scheme. For each CPU count (i.e., each row), the best runtime/speedup is highlighted in gray.

As a general observation, we note that low values of d produce 20 or fewer subproblems, such that parallel performance is identical for 20, 100, or 500 CPUs – there are simply not enough subproblems to make use of the additional parallel resources. For instance, in case



(a) *Left:* parallel runtimes using 20, 100, and 500 CPUs for varying number of subproblems, with sequential AOBB runtime indicated by horizontal dashed line. *Right:* corresponding parallel speedups, with "optimal" values of 20, 100, and 500 marked by dashed horizontal lines.



(b) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 8$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 476$.

Figure 4.13: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on haplotyping instance largeFam3-15-59, $i = 19$.

of largeFam3-15-59 and pedigree44, all cutoffs below $d = 4$ entail at most 20 subproblems and identical performance across all CPU counts. For largeFam3-15-59 setting $d = 6$ creates 80 subproblems, so 20 CPUs are indeed a bit slower overall, while 100 and 500 CPUs still give the same performance. For the two pedigree instances, however, the subproblem count for $d = 6$ is already greater than 100, giving an advantage to 500 available CPUs over just 100.

4.6.3.2 Case Study 1: largeFam3-15-59

To investigate parallel performance in more detail, consider first the largeFam haplotyping problem largeFam3-15-59 with 3730 variables and induced width 31 (cf. Table 4.3), run with i-bound $i = 19$. In parallel execution we obtain a maximum of 6964 subproblems at depth $d = 12$. For better illustration, the parallel results listed in Table 4.6 are plotted in Figure 4.13a, contrasting performance with 20, 100, and 500 CPUs. The left plot captures parallel runtimes for the fixed-depth (solid lines) and variable-depth (dashed lines) scheme as a function of the number of subproblems; the right plot does the same for the corresponding speedup values.

Subproblem Count. We note that, as expected, performance between the three CPU counts only begins to differ as the number of subproblems grows. Notably, we also observe that performance for each CPU count continues to improve further as the number of subproblems is increased beyond the respective CPU count. We see it deteriorate again eventually. This is because initially the overall performance is still dominated by a few long running subproblems, i.e., the parallel load is fairly unbalanced. Further increasing the subproblem count and thereby parallel granularity ideally splits these hard subproblems, allowing the resulting parts to be spread across separate CPUs. In addition, once the number of subproblems exceeds the CPU count, the first-come, first-served assignment of jobs to workers allows those CPUs that finish early to continue working on another subproblem. Eventually, however, overall performance begins to suffer as all parallel CPUs approach saturation and smaller and smaller subproblems induce increasingly more overhead, as outlined in Section 4.4. More analysis of this will be provided in Sections 4.6.5 and 4.6.6.

For the variable-depth scheme in Figure 4.13a, for instance, the turning point of parallel performance for 20, 100, and 500 CPUs lies at 240, 936, and 1830 subproblems, respectively ($d = 7$, $d = 9$, and $d = 10$, not all included in Table 4.6, cf. Table B.8 in Appendix

B). It makes sense intuitively that this "sweet spot" of parallel granularity increases with the number of CPUs. In fact, for largeFam3-11-59 we observe that the "ideal" number of subproblems is roughly one order magnitude larger than the number of parallel CPUs. We will find this confirmed in subsequent parallel results and note that this relation actually matches a fairly common "rule of thumb" in distributed computing, reported for instance by the team of Superlink Online [105].

Fixed-depth vs. Variable-depth. It is evident that in the present case a variable-depth parallelization frontier, using regression-based complexity estimations, has a decided edge over the fixed-depth scheme and yields faster overall runtimes. For instance, with 500 CPUs, the best speedup obtained by variable-depth AOBB in Table 4.6b is 167x ($d = 10$ equivalent), compared to 104x for fixed-depth at $d = 12$. Using 100 CPUs, variable-depth parallelization achieves a speedup of 60x ($d = 8$), while fixed-depth peaks at 47x ($d = 10$).

To illustrate the advantage of variable-depth parallelization, the two plots in Figure 4.13b show the runtimes of the individual subproblems for both the fixed-depth parallel cutoff at depth $d = 8$ (on the left) as well as the corresponding variable-depth parallel cutoff with $p = 476$ subproblems (on the right). Subproblems are indexed in the order of their position in the Condor job queue, which, in case of the variable-depth scheme, is sorted by decreasing complexity estimates. Each plot also includes the overall parallel runtime as a solid horizontal line, as well as the 0^{th}, 20^{th}, 80^{th}, and 100^{th} percentile of subproblem runtimes marked with dashed horizontal lines.

We observe that in spite of the relatively large number of subproblems (476 jobs vs. 100 CPUs), the parallel runtime of the fixed-depth scheme is dominated by a handful of long-running subproblems – the largest subproblem takes 1394 seconds, while the overall runtime is 1494 seconds. Using variable-depth parallelization, on the other hand, the longest-running subproblem takes only 552 seconds and the overall time is 723 seconds – less than half that of the fixed-depth scheme. We furthermore note that the standard deviation in subproblem

runtime, as noted in the plots of Figure 4.13b, is twice as high for fixed-depth parallelization (201 vs. 93 seconds).

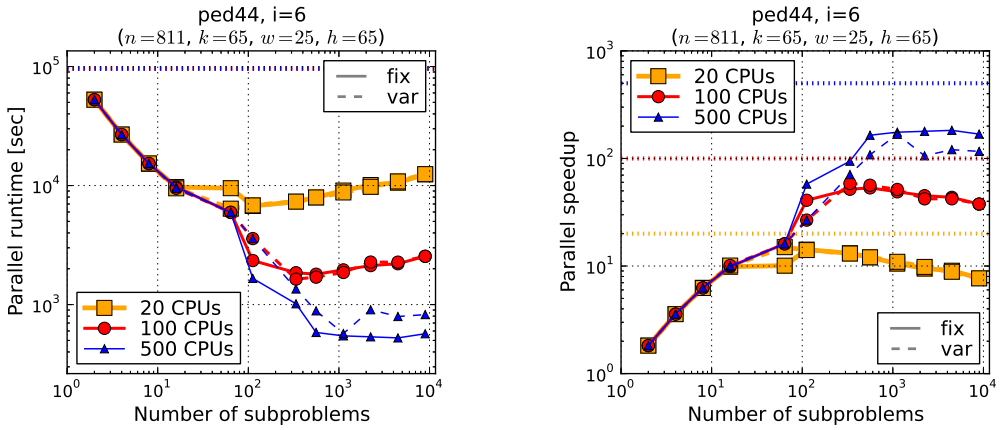
Finally, thanks to the estimation scheme, variable-depth parallelization is fairly successful in pushing the easiest subproblems to the end of the parallel job queue (corresponding to higher subproblem indexes in the plots). This is desirable since these subproblems will be assigned to workers towards the end of the parallel execution and starting a long-running subproblem at that stage would be potentially disastrous to the overall runtime.

Parallel Resource Utilization. Resource utilization will be discussed in more detail later, but we point out already that in this example the average resource utilization for fixed-depth parallelization is only 34% versus 70% for variable-depth (cf. Appendix B Table B.20, page 326). That means that the 100 CPUs used for parallel computation are on average busy 34% and 70%, respectively, relative to the longest-running CPU. In other words, variable-depth parallelization is more than twice as efficient in terms of using parallel resources in this example.

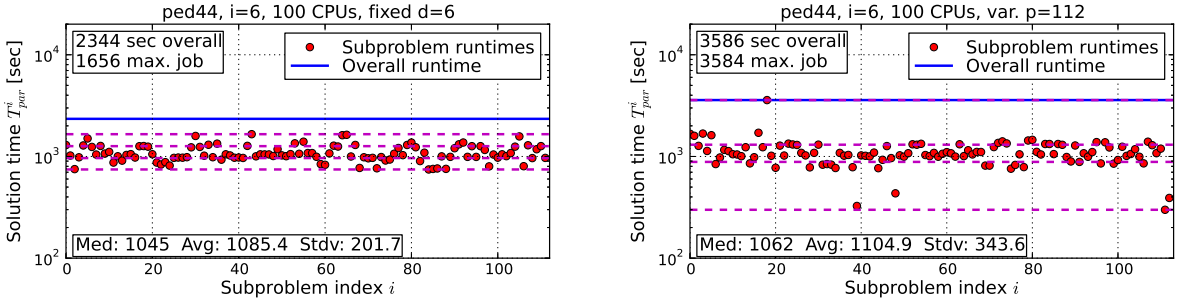
4.6.3.3 Case Study 2: pedigree44

Secondly, we consider pedigree44 with $i = 6$ with 811 variables and induced width 25 (cf. Table 4.2). Sequential runtime is $T_{seq} = 95830$ seconds or 26 hours and 37 minutes. Overall parallel runtime and corresponding speedup are listed in Tables 4.6a and 4.6b, respectively. As before, various aspects of parallel performance are plotted in Figure 4.14 for illustration.

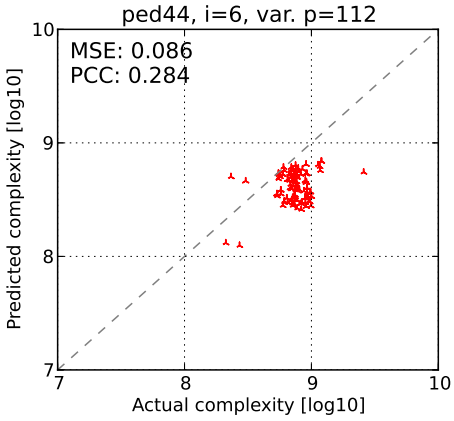
Runtime and Speedup. Figure 4.14a shows plots of runtime (left) and speedup (right) results as a function of subproblem count. Overall results are comparable to the previous case study, with best obtained speedups of approx. 14, 56, and 179 for 20, 100, and 500 CPUs, respectively – corresponding to runtimes of 112 minutes, 28 minutes, and 9 minutes. However, in contrast to the example in the previous section, the fixed-depth scheme appears



(a) *Left:* parallel runtimes using 20, 100, and 500 CPUs for varying number of subproblems, with sequential AOBB runtime indicated by horizontal dashed line. *Right:* corresponding parallel speedups, with "optimal" values of 20, 100, and 500 marked by dashed horizontal lines.



(b) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 6$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 112$.



(c) Scatter plot of actual vs. predicted subproblem complexity (in node expansions) for variable-depth parallelization with $p = 112$ subproblems.

Figure 4.14: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on linkage instance pedigree44, $i = 6$.

to have a slight edge in terms of overall performance, with better results than the variable-depth parallel cutoff for many subproblem counts. This is most notable with 500 CPUs, even though we note that variable-depth parallelization at one point ($d = 9$, not shown in Table 4.6) matches the 9 minutes runtime of the fixed-depth scheme.

Performance Analysis through Subproblem Statistics. To analyze the inferior performance of the variable-depth scheme as observed above, Figure 4.14b illustrates the individual subproblem complexities for fixed-depth (left, with cutoff $d = 6$) and variable-depth (right, with corresponding $p = 112$ subproblems) parallelization: we see that the subproblems produced by fixed-depth parallelization are actually remarkably balanced already, yielding an overall runtime of 2344 seconds. Variable-depth parallelization, on the other hand, gives a single outlier with 3584 seconds that far dominates the overall runtime of 3586 seconds (we also note a handful of outliers that have significantly shorter runtimes but don't affect overall performance).

Recall that the subproblems in the case of variable-depth parallelization are ordered by decreasing complexity estimates. The position of the outlier in Figure 4.14b thus suggests that the complexity estimate of the particular subproblem was subject to considerable inaccuracy. This hypothesis is confirmed by plotting each subproblems' actual vs. predicted complexity in Figure 4.14c – the outlier in question is clearly visible to the right of the main cluster of plot entries. This case study thus serves as an example where the variable-depth scheme, dependent on subproblem complexity estimation, falls short of its intention to avoid long-running subproblems that constitute bottlenecks for the overall performance.

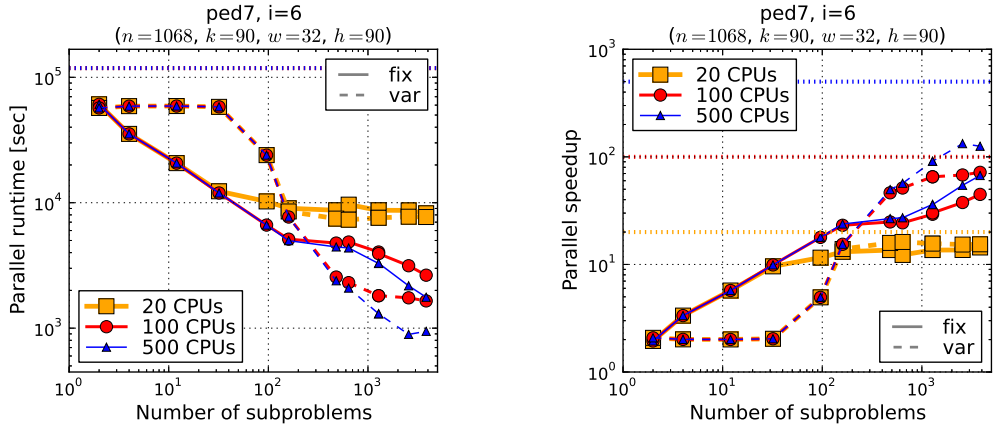
Subproblem Count. The behavior we observe regarding the choice of subproblem count is similar to the previous example instance largeFam3-15-59. In particular, for all three CPU counts parallel performance is identical for small number of subproblems and peaks when the number of subproblems is several times the number of CPUs – for 20, 100, and 500 CPUs at 112, 560, and 2240 subproblems, respectively.

4.6.3.4 Case Study 3: pedigree7

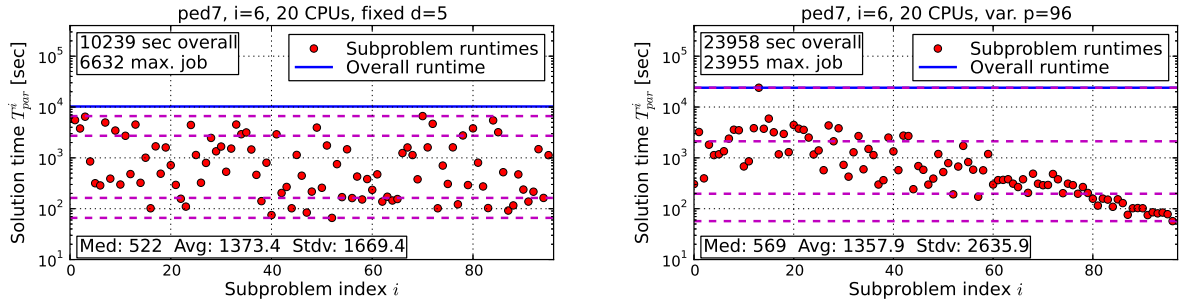
Lastly, we consider pedigree7 with i-bound $i = 6$ which has a sequential runtime of $T_{seq} = 118383$ seconds, or a little under 33 hours. The problem has 1068 variables and induced width 32 (cf. Table 4.2). A subset of parallel runtimes using 20, 100, and 500 CPUs are shown in Table 4.6a, with corresponding speedups in Table 4.6b. As for the previous two examples we plot the progression of runtimes and speedups as the number of subproblem increases in Figure 4.15a.

Runtime and Speedup. In the best case, we observe parallel runtimes of about 2 hours, 27 minutes, and 15 minutes using 20, 100, and 500 CPUs, respectively – this corresponds to speedups of 16, 72, and 126. We note that the parallel runtimes of variable-depth parallelization show poor, almost constant parallel performance up to cutoff depth $p = 32$ subproblems (corresponding to $d = 4$). Just like in the previous example, this suggests that one complex subproblem (or a small number of them) is underestimated significantly by the complexity prediction, thus dominating parallel performance. However, the performance of variable-depth parallelization improves drastically as we increase the number of subproblems beyond $p = 96$ to (corresponding to cutoff depth $d = 6$ and higher.

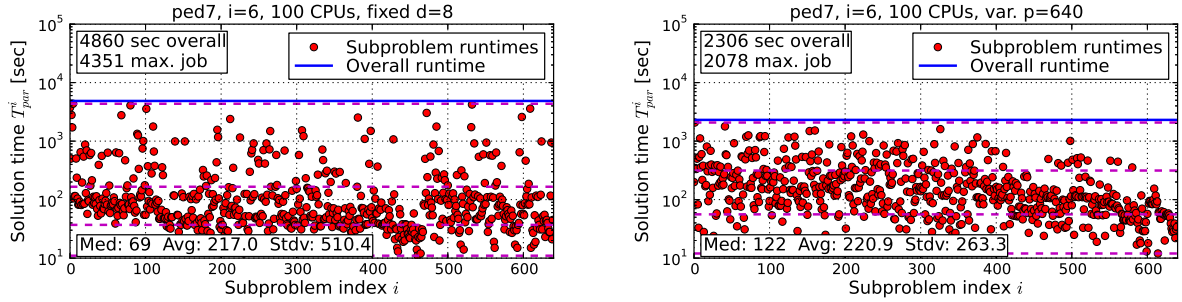
Subproblem Statistics. To investigate the performance of the variable-depth scheme, we again consider the runtime of the individual subproblems. We find our hypothesis of an outlier subproblem confirmed when plotting, for instance, comparing fixed-depth cutoff at $d = 5$ with the corresponding variable-depth run using $p = 96$ (Figure 4.15b). In the latter case the maximum subproblem runtime is 23955 seconds, which directly determines the overall runtime of 23958 seconds – parallel resource utilization using 20 CPUs is accordingly at a low 28% (cf. Section 4.6.5). The fixed-depth run, however, while arguably not very balanced in terms of subproblem complexity, finished in 10239 seconds overall, with the largest subproblem at 6632 seconds (parallel resource utilization comes out to 0.64%). Figure



(a) *Left:* parallel runtimes using 20, 100, and 500 CPUs for varying number of subproblems, with sequential AOBB runtime indicated by horizontal dashed line. *Right:* corresponding parallel speedups, with "optimal" values of 20, 100, and 500 marked by dashed horizontal lines.



(b) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 5$ using 20 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 96$.



(c) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 8$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 640$.

Figure 4.15: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on linkage instance pedigree7, $i = 6$.

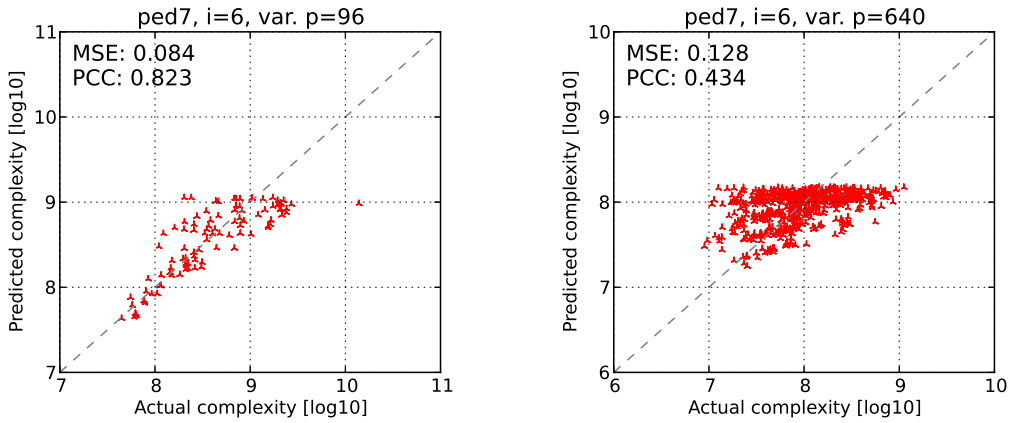


Figure 4.16: Scatter plot of actual vs. predicted subproblem complexity for variable-depth parallel runs with $p = 96$ (left) and $p = 640$ (right) subproblems on pedigree7, $i = 6$.

4.16 (left) further illustrates the issue, plotting actual vs. predicted subproblem complexity for the variable-depth parallel run in question, and clearly showing an outlier to the right of the main group of plot points.

We also noted that performance of the variable-depth scheme sees drastic improvements with more than $p = 96$ subproblems. In this context, Figure 4.15c shows the runtime of individual subproblems for fixed-depth and variable-depth parallelization with 100 CPUs, using cutoff depth $d = 8$ and the corresponding $p = 640$ subproblems, respectively. In addition, Figure 4.16 (right) contrasts the actual and predicted complexities of the variable-depth run. We see that an outlier subproblem with drastically larger complexity is no longer present (even though the prediction mean squared error increased). This and the generally better load balancing (subproblem runtime standard deviation of 263 vs. 510 seconds of fixed-depth scheme) allows the variable-depth parallelization to outperform the fixed-depth variant by a factor of two, 2306 seconds to 4860. Parallel resource utilization is also a lot better, with 62% to 29% (cf. Section 4.6.5).

4.6.4 Overall Parallel Performance

After highlighting and analyzing a variety of performance characteristics through three particular case studies in Section 4.6.3, this section will provide a more general overview and analysis on the parallel runtime and speedup results of our parallel scheme.

To that end we provide a comprehensive subset of results for each problem class, with runtime and speedup tables across all instances. In addition we show additional performance plots of select problem instances, together with more detailed subproblem analysis in a number of cases.

4.6.4.1 Overall Analysis of Linkage Problems

Tables 4.7 and 4.8 show a parallel runtime results for pedigree linkage analysis instances on a subset of cutoff depths while Tables 4.9 and 4.10 list the corresponding speedup values. Full result tables are included in Appendix B.

As for Table 4.6a, each field gives the number of subproblems p as well as parallel runtime/speedup of fixed-depth (left) and variable-depth parallelization (right) with 20, 100, and 500 CPUs, from top to bottom. The last row, in italic font, lists the "best possible" parallel runtime or speedup that could be obtained if we had an unlimited number of CPUs at our disposal – in that case each subproblem would be solved by a separate worker host and the parallel runtime is the sum of preprocessing time and the runtime of the largest subproblem.

As before, the best entry of each row is highlighted by a gray background. In addition, for each pair of fixed-/variable-depth results, if one runtime is faster than the other by more than 10%, it is marked bold. The bottom two rows of each table provide a summary of how

instance	i	T_{seq}	#cpu	Cutoff depth d = 2 fix	2 var	4 fix	4 var	6 fix	6 var	8 fix	8 var	10 fix	10 var	12 fix	12 var
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
ped13 n=1077 k=3 w=32 h=102	8	252654	20	**69461**	136558	**19125**	70897	**16290**	24717	15228	**13822**	14074	13883	14734	14486
			100	**69461**	136558	**19125**	70897	**5945**	18836	**4867**	11027	**3603**	3979	3713	3620
			500	**69461**	136558	**19125**	70897	**5945**	18836	**3339**	11027	**1741**	2816	1516	1608
			∞	***69461***	*136558*	***19125***	*70897*	***5945***	*18836*	***3339***	*11027*	***1658***	*2816*	***1139***	*1399*
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
	9	102385	20	30925	31659	**10539**	11993	**5806**	9706	**5111**	5691	5223	5573	5656	5574
			100	30925	31659	**10539**	11993	**3481**	9706	**1941**	3237	**1337**	2082	1463	1546
			500	30925	31659	**10539**	11993	**3481**	9706	**1682**	3202	**835**	1764	824	841
			∞	*30925*	*31659*	***10539***	*11993*	***3481***	*9706*	***1682***	*3202*	***807***	*1764*	*752*	*765*
				(p=12)		*(p=144)*		*(p=1440)*		*(p=5752)*		*(p=11254)*			
ped19 n=793 k=5 w=25 h=98	16	375110	20	119407	**73960**	30976	**28027**	27290	28435	35187	37384	47882	48784		
			100	119407	**73960**	30976	**20593**	12125	**7294**	7492	7947	9705	9887		
			500	119407	**73960**	30976	**20593**	11930	**5158**	5222	**3301**	2912	**2381**		
			∞	*119407*	***73960***	*30976*	***20593***	*11930*	***5158***	*5079*	***2639***	*2709*	***1934***		
				(p=6)		*(p=32)*		*(p=160)*		*(p=800)*		*(p=6400)*			
ped20 n=437 k=5 w=22 h=60	3	5136	20	1361	1392	454	423	395	433	514	505	1164	1167		
			100	1361	1392	383	381	**111**	238	127	133	249	252		
			500	1361	1392	383	381	**95**	238	**52**	133	67	72		
			∞	*1361*	*1392*	*383*	*381*	***95***	*238*	***41***	*133*	***32***	*43*		
				(p=6)		*(p=32)*		*(p=160)*		*(p=800)*		*(p=6400)*			
	4	2185	20	229	227	**81**	90	79	75	149	149	733	728		
			100	229	227	**51**	90	**21**	24	34	32	161	160		
			500	229	227	**51**	90	**15**	24	12	11	46	47		
			∞	*229*	*227*	***51***	*90*	***15***	*24*	***10***	*11*	*20*	*20*		
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
ped31 n=1183 k=5 w=30 h=85	10	1258519	20	360625	358220	104694	103211	103639	**89404**	85986	81762	81513	80922	82833	81644
			100	360625	358220	104694	103211	48472	49703	30712	**26508**	19653	**17786**	17082	16784
			500	360625	358220	104694	103211	48472	49703	23569	**17423**	8172	**5668**	4607	**3929**
			∞	*360625*	*358220*	*104694*	*103211*	*48472*	*49703*	*23569*	***17423***	*7100*	***5118***	*2033*	***1584***
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
	11	433029	20	107463	108489	30376	29491	31225	**26680**	22809	24087	24048	24254	25193	25315
			100	107463	108489	30376	29491	15695	15780	7333	7668	5748	**5069**	5367	5144
			500	107463	108489	30376	29491	15695	15780	7333	7668	2384	**1820**	1473	1360
			∞	*107463*	*108489*	*30376*	*29491*	*15695*	*15780*	*7333*	*7668*	*2141*	***1820***	*1083*	***818***
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
	12	16238	20	3758	4046	1298	1224	1463	**1126**	1086	**929**	1055	1037	1444	1447
			100	3758	4046	1298	1224	1176	**811**	437	**393**	287	**233**	325	319
			500	3758	4046	1298	1224	1176	**811**	406	393	139	131	115	**95**
			∞	*3758*	*4046*	*1298*	*1224*	*1176*	***811***	*406*	*393*	*127*	*131*	*70*	***63***
				(p=3)		*(p=6)*		*(p=24)*		*(p=96)*		*(p=384)*		*(p=1536)*	
ped33 n=798 k=4 w=28 h=98	4	6010	20	3071	3238	1645	1521	649	**486**	478	467	467	434	594	597
			100	3071	3238	1645	1521	519	**367**	252	**173**	173	**112**	159	**139**
			500	3071	3238	1645	1521	519	**367**	252	**173**	124	**91**	87	**59**
			∞	*3071*	*3238*	*1645*	*1521*	*519*	***367***	*252*	***173***	*124*	***91***	*77*	***49***
				(p=5)		*(p=20)*		*(p=60)*		*(p=180)*		*(p=716)*		*(p=1896)*	
ped34 n=1160 k=5 w=31 h=102	10	962006	20	424691	424270	175178	**144147**	145741	**122690**	109138	**93405**	98912	97309	97118	96468
			100	424691	424270	175178	**144147**	115446	**75577**	42110	39354	27212	**21134**	21438	21187
			500	424691	424270	175178	**144147**	115446	**75577**	41663	39354	13890	**11203**	6670	6136
			∞	*424691*	*424270*	*175178*	***144147***	*115446*	***75577***	*41663*	*39354*	*13680*	***11203***	*5773*	*5926*
				(p=5)		*(p=20)*		*(p=60)*		*(p=180)*		*(p=720)*		*(p=1912)*	
	11	350574	20	216931	217171	**103935**	155893	95723	95475	80649	74447	74526	76593	76817	76397
			100	216931	217171	**103935**	155893	79842	79099	30073	29976	19172	18735	16714	16145
			500	216931	217171	**103935**	155893	79842	79099	30042	29976	10423	**9238**	5282	5692
			∞	*216931*	*217171*	***103935***	*155893*	*79842*	*79099*	*30042*	*29976*	*10423*	***9238***	*4666*	***4194***
				(p=5)		*(p=20)*		*(p=60)*		*(p=180)*		*(p=716)*		*(p=1896)*	
	12	96122	20	26241	26328	20456	**15283**	15829	15743	13996	13785	12819	13081	12979	13421
			100	26241	26328	20456	**15283**	13298	**9028**	5402	5648	3489	**2971**	2887	2972
			500	26241	26328	20456	**15283**	13298	**9028**	5390	5648	2100	**1298**	1014	988
			∞	*26241*	*26328*	*20456*	***15283***	*13298*	***9028***	*5390*	*5648*	*2096*	***1298***	*916*	***625***
				(p=4)		*(p=16)*		*(p=128)*		*(p=768)*		*(p=2304)*			
ped39 n=1272 k=5 w=21 h=76	4	6632	20	2690	2731	1494	1503	697	728	709	**566**	672	677		
			100	2690	2731	1494	1503	596	571	516	**318**	252	**190**		
			500	2690	2731	1494	1503	596	571	491	**304**	181	**129**		
			∞	*2690*	*2731*	*1494*	*1503*	*596*	*571*	*491*	***304***	*169*	***121***		
				(p=4)		*(p=16)*		*(p=128)*		*(p=768)*		*(p=2304)*			
	5	2202	20	793	867	409	**359**	421	**292**	422	**306**	443	452		
			100	793	867	409	**359**	317	292	278	**141**	156	**122**		
			500	793	867	409	**359**	312	292	253	**111**	103	**70**		
			∞	*793*	*867*	*409*	***359***	*312*	*292*	*252*	***109***	*94*	***70***		
Better by 10%				4x	4x	16x	16x	14x	21x	10x	20x	7x	26x	1x	10x
Better by 50%				4x	4x	7x	3x	13x	6x	8x	8x	5x	3x	0x	1x

Table 4.7: Subset of parallel **runtime** results on **linkage instances, part 1 of 2**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 2 fix	2 var	4 fix	4 var	6 fix	6 var	8 fix	8 var	10 fix	10 var	12 fix	12 var
				(p=8)		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
ped41	9	25607	20	16193	**14319**	4737	**2882**	2193	2186	2445	**2118**	2427	2457	2972	2982
n=1062 k=5 w=33 h=100			100	16193	**14319**	4565	**2436**	1307	**793**	827	**576**	545	520	805	**666**
			500	16193	**14319**	4565	**2436**	1272	**793**	650	**340**	251	**211**	446	**225**
			∞	*16193*	***14319***	*4565*	***2436***	*1272*	***793***	*650*	***340***	*222*	***172***	*401*	***148***
				(p=8)		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
	10	46819	20	33309	**17287**	9679	**4851**	3613	3331	3307	3199	3059	3237	3507	3538
			100	33309	**17287**	9573	**4851**	2762	**1711**	1483	**1175**	**904**	1032	**751**	866
			500	33309	**17287**	9573	**4851**	2728	**1711**	1274	1175	**705**	999	519	504
			∞	*33309*	***17287***	*9573*	***4851***	*2728*	***1711***	*1274*	*1175*	***694***	*999*	*514*	***435***
				(p=8)		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
	11	27583	20	20222	**10169**	6101	**3877**	2189	**1866**	2051	**1784**	1988	1964	2268	2266
			100	20222	**10169**	6039	**3877**	1639	**1099**	1010	**553**	**548**	794	548	**491**
			500	20222	**10169**	6039	**3877**	1615	**1099**	900	**497**	**292**	607	259	**179**
			∞	*20222*	***10169***	*6039*	***3877***	*1615*	***1099***	*900*	***497***	***270***	*581*	*212*	***117***
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
ped44	5	207136	20	65724	65934	**20396**	31287	12719	13425	17214	16269	20630	20412	29114	**25629**
n=811 k=4 w=25 h=65			100	65724	65934	**20396**	31287	**3415**	10659	4040	**3600**	4281	4551	5919	5481
			500	65724	65934	**20396**	31287	**3415**	10659	**1554**	1868	**1000**	1405	**1270**	1584
			∞	*65724*	*65934*	***20396***	*31287*	***3415***	*10659*	***1371***	*1868*	***366***	*1002*	***737***	*921*
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
	6	95830	20	26776	26836	9716	9481	6741	6811	7959	7947	10103	9763	12418	12472
			100	26776	26836	9716	9481	**2344**	3586	1799	1700	2126	2276	2545	2543
			500	26776	26836	9716	9481	**1659**	3586	**583**	886	**536**	905	**569**	824
			∞	*26776*	*26836*	*9716*	*9481*	***1659***	*3586*	***459***	*886*	***433***	*774*	***103***	*696*
				(p=4)		*(p=144)*		*(p=2160)*		*(p=14401)*					
ped50	3	4135	20	1485	1477	423	**345**	465	451	1734	1730				
n=514 k=6 w=17 h=47			100	1485	1477	345	345	198	**106**	381	378				
			500	1485	1477	345	345	159	**88**	127	**111**				
			∞	*1485*	*1477*	*345*	*345*	*153*	***88***	*89*	***53***				
				(p=4)		*(p=144)*		*(p=2160)*		*(p=14400)*					
	4	1780	20	272	255	75	**67**	273	277	1551	1549				
			100	272	255	42	41	62	61	342	340				
			500	272	255	42	41	27	**19**	101	99				
			∞	*272*	*255*	*42*	*41*	*22*	***17***	*46*	*42*				
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4064)*	
ped51	20	101788	20	27299	27269	8261	7697	7051	**6225**	6404	5885	6573	6570	9899	9866
n=1152 k=5 w=39 h=98			100	27299	27269	8261	7697	3457	**2658**	2340	**1687**	**1578**	2208	2186	2025
			500	27299	27269	8261	7697	3457	**2658**	1772	**1163**	**704**	1525	852	**748**
			∞	*27299*	*27269*	*8261*	*7697*	*3457*	***2658***	*1772*	***1163***	***681***	*1440*	*689*	*675*
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
	21	164817	20	43197	42435	11542	11279	**9030**	10008	8727	9508	10221	10488	16982	17114
			100	43197	42435	11542	11279	4867	4868	**2349**	2950	2132	2253	3537	3500
			500	43197	42435	11542	11279	4867	4868	2349	**1860**	**908**	1571	**1050**	1165
			∞	*43197*	*42435*	*11542*	*11279*	*4867*	*4868*	*2349*	***1860***	***908***	*1571*	*845*	*778*
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
ped7	6	118383	20	**35387**	58872	**12338**	58121	9031	8515	9654	**7319**	8705	**7582**	8236	7693
n=1068 k=4 w=32 h=90			100	**35387**	58872	**11956**	58121	**5122**	7690	4860	**2306**	3929	**1814**	2644	**1649**
			500	**35387**	58872	**11956**	58121	**4984**	7690	4359	**2086**	3294	**1301**	1764	**943**
			∞	***35387***	*58872*	***11956***	*58121*	***4984***	*7690*	*4359*	***2086***	*3256*	***1301***	*1740*	***876***
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
	7	93380	20	**25119**	47316	**7989**	51318	**5015**	26061	5909	**5366**	6061	**5461**	5924	5706
			100	**25119**	47316	**7989**	51318	**2947**	25320	3002	**1997**	2819	**1505**	2156	**1268**
			500	**25119**	47316	**7989**	51318	**2947**	25320	2615	**1368**	2381	**1180**	1433	**513**
			∞	***25119***	*47316*	***7989***	*51318*	***2947***	*25320*	*2615*	***1368***	*2381*	***1180***	*1421*	***394***
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1276)*		*(p=3816)*	
	8	30717	20	**8913**	18311	**2976**	18357	**2344**	10390	2204	**1938**	2196	**1786**	2109	2069
			100	**8913**	18311	**2976**	18357	**1276**	9856	1146	**1004**	1075	**944**	916	**601**
			500	**8913**	18311	**2976**	18357	**1256**	9856	1090	**854**	1008	**833**	682	**345**
			∞	***8913***	*18311*	***2976***	*18357*	***1256***	*9856*	*1090*	***854***	*1008*	***833***	*648*	***294***
				(p=4)		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
ped9	6	101172	20	**27626**	52049	**10362**	17818	7283	7356	6232	6557	5945	6098	6648	6718
n=1118 k=7 w=27 h=100			100	**27626**	52049	**10362**	17818	**5438**	7016	**1559**	2678	1273	1294	1390	1405
			500	**27626**	52049	**10362**	17818	**5438**	7016	**1406**	2678	**395**	931	**345**	384
			∞	***27626***	*52049*	***10362***	*17818*	***5438***	*7016*	***1406***	*2678*	***395***	*931*	***170***	*321*
				(p=4)		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
	7	58657	20	15391	15640	**5383**	6071	5148	4791	3644	3746	3516	3561	3827	3933
			100	15391	15640	**5383**	6071	2957	2923	1051	1128	806	763	816	834
			500	15391	15640	**5383**	6071	2957	2923	**753**	1128	**234**	356	**221**	413
			∞	*15391*	*15640*	***5383***	*6071*	*2957*	*2923*	***753***	*1128*	***234***	*356*	***99***	*413*
				(p=4)		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
	8	41061	20	10995	10923	4827	4411	3634	**3213**	**2560**	3277	2439	2455	2736	2749
			100	10995	10923	4827	4411	**2259**	3157	**746**	1826	541	583	578	574
			500	10995	10923	4827	4411	**2259**	3157	**604**	1826	**255**	465	155	151
			∞	*10995*	*10923*	*4827*	*4411*	***2259***	*3157*	***604***	*1826*	***255***	*465*	***78***	*108*
Better by 10%				16x	12x	24x	14x	24x	20x	14x	29x	21x	14x	11x	18x
Better by 50%				16x	8x	20x	12x	17x	9x	8x	14x	15x	6x	4x	12x

Table 4.8: Subset of parallel **runtime** results on **linkage instances, part 2 of 2**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d											
				2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
ped13 n=1077 k=3 w=32 h=102	8	252654	20	*(p=4)* **3.64**	1.85	*(p=16)* **13.21**	3.56	*(p=64)* **15.51**	10.22	*(p=256)* 16.59	**18.28**	*(p=1024)* 17.95	18.20	*(p=4096)* 17.15	17.44
			100	**3.64**	1.85	**13.21**	3.56	**42.50**	13.41	**51.91**	22.91	**70.12**	63.50	68.05	69.79
			500	**3.64**	1.85	**13.21**	3.56	**42.50**	13.41	**75.67**	22.91	**145.12**	89.72	166.66	157.12
			∞	***3.64***	*1.85*	***13.21***	*3.56*	***42.50***	*13.41*	***75.67***	*22.91*	***152.38***	*89.72*	***221.82***	*180.60*
	9	102385	20	*(p=4)* 3.31	3.23	*(p=16)* **9.71**	8.54	*(p=64)* **17.63**	10.55	*(p=256)* **20.03**	17.99	*(p=1024)* 19.60	18.37	*(p=4096)* 18.10	18.37
			100	3.31	3.23	**9.71**	8.54	**29.41**	10.55	**52.75**	31.63	**76.58**	49.18	69.98	66.23
			500	3.31	3.23	**9.71**	8.54	**29.41**	10.55	**60.87**	31.98	**122.62**	58.04	124.25	121.74
			∞	*3.31*	*3.23*	***9.71***	*8.54*	***29.41***	*10.55*	***60.87***	*31.98*	***126.87***	*58.04*	*136.15*	*133.84*
ped19 n=793 k=5 w=25 h=98	16	375110	20	*(p=12)* 3.14	**5.07**	*(p=144)* 12.11	**13.38**	*(p=1440)* 13.75	13.19	*(p=5752)* 10.66	10.03	*(p=11254)* 7.83	7.69		
			100	3.14	**5.07**	12.11	**18.22**	30.94	**51.43**	50.07	47.20	38.65	37.94		
			500	3.14	**5.07**	12.11	**18.22**	31.44	**72.72**	71.83	**113.64**	128.82	**157.54**		
			∞	*3.14*	***5.07***	*12.11*	***18.22***	*31.44*	***72.72***	*73.86*	***142.14***	*138.47*	***193.96***		
ped20 n=437 k=5 w=22 h=60	3	5136	20	*(p=6)* 3.77	3.69	*(p=32)* 11.31	12.14	*(p=160)* 13.00	11.86	*(p=800)* 9.99	10.17	*(p=6400)* 4.41	4.40		
			100	3.77	3.69	13.41	13.48	**46.27**	21.58	40.44	38.62	20.63	20.38		
			500	3.77	3.69	13.41	13.48	**54.06**	21.58	**98.77**	38.62	76.66	71.33		
			∞	*3.77*	*3.69*	*13.41*	*13.48*	***54.06***	*21.58*	***125.27***	*38.62*	***160.50***	*119.44*		
	4	2185	20	*(p=6)* 9.54	9.63	*(p=32)* **26.98**	24.28	*(p=160)* 27.66	29.13	*(p=800)* 14.66	14.66	*(p=6400)* 2.98	3.00		
			100	9.54	9.63	**42.84**	24.28	**104.05**	91.04	64.26	68.28	13.57	13.66		
			500	9.54	9.63	**42.84**	24.28	**145.67**	91.04	182.08	198.64	47.50	46.49		
			∞	*9.54*	*9.63*	***42.84***	*24.28*	***145.67***	*91.04*	***218.50***	*198.64*	*109.25*	*109.25*		
ped31 n=1183 k=5 w=30 h=85	10	1258519	20	*(p=4)* 3.49	3.51	*(p=16)* 12.02	12.19	*(p=64)* 12.14	**14.08**	*(p=256)* 14.64	15.39	*(p=1024)* 15.44	15.55	*(p=4096)* 15.19	15.41
			100	3.49	3.51	12.02	12.19	25.96	25.32	40.98	**47.48**	64.04	**70.76**	73.68	74.98
			500	3.49	3.51	12.02	12.19	25.96	25.32	53.40	**72.23**	154.00	**222.04**	273.18	**320.32**
			∞	*3.49*	*3.51*	*12.02*	*12.19*	*25.96*	*25.32*	*53.40*	***72.23***	*177.26*	***245.90***	*619.05*	***794.52***
	11	433029	20	*(p=4)* 4.03	3.99	*(p=16)* 14.26	14.68	*(p=64)* 13.87	**16.23**	*(p=256)* 18.99	17.98	*(p=1024)* 18.01	17.85	*(p=4096)* 17.19	17.11
			100	4.03	3.99	14.26	14.68	27.59	27.44	59.05	56.47	75.34	**85.43**	80.68	84.18
			500	4.03	3.99	14.26	14.68	27.59	27.44	59.05	56.47	181.64	**237.93**	293.98	318.40
			∞	*4.03*	*3.99*	*14.26*	*14.68*	*27.59*	*27.44*	*59.05*	*56.47*	*202.26*	***237.93***	*399.84*	***529.38***
	12	16238	20	*(p=4)* 4.32	4.01	*(p=16)* 12.51	13.27	*(p=64)* 11.10	**14.42**	*(p=256)* 14.95	**17.48**	*(p=1024)* 15.39	15.66	*(p=4096)* 11.25	11.22
			100	4.32	4.01	12.51	13.27	13.81	**20.02**	37.16	**41.32**	56.58	**69.69**	49.96	50.90
			500	4.32	4.01	12.51	13.27	13.81	**20.02**	40.00	41.32	116.82	123.95	141.20	**170.93**
			∞	*4.32*	*4.01*	*12.51*	*13.27*	*13.81*	***20.02***	*40.00*	*41.32*	*127.86*	*123.95*	*231.97*	***257.75***
ped33 n=798 k=4 w=28 h=98	4	6010	20	*(p=3)* 1.96	1.86	*(p=6)* 3.65	3.95	*(p=24)* 9.26	**12.37**	*(p=96)* 12.57	12.87	*(p=384)* 12.87	13.85	*(p=1536)* 10.12	10.07
			100	1.96	1.86	3.65	3.95	11.58	**16.38**	23.85	**34.74**	34.74	**53.66**	37.80	**43.24**
			500	1.96	1.86	3.65	3.95	11.58	**16.38**	23.85	**34.74**	48.47	**66.04**	69.08	**101.86**
			∞	*1.96*	*1.86*	*3.65*	*3.95*	*11.58*	***16.38***	*23.85*	***34.74***	*48.47*	***66.04***	*78.05*	***122.65***
ped34 n=1160 k=5 w=31 h=102	10	962006	20	*(p=5)* 2.27	2.27	*(p=20)* 5.49	**6.67**	*(p=60)* 6.60	**7.84**	*(p=180)* 8.81	**10.30**	*(p=716)* 9.73	9.89	*(p=1896)* 9.91	9.97
			100	2.27	2.27	5.49	**6.67**	8.33	**12.73**	22.85	24.44	35.35	**45.52**	44.87	45.41
			500	2.27	2.27	5.49	**6.67**	8.33	**12.73**	23.09	24.44	69.26	**85.87**	144.23	156.78
			∞	*2.27*	*2.27*	*5.49*	***6.67***	*8.33*	***12.73***	*23.09*	*24.44*	*70.32*	***85.87***	*166.64*	*162.34*
	11	350574	20	*(p=5)* 1.62	1.61	*(p=20)* **3.37**	2.25	*(p=60)* 3.66	3.67	*(p=180)* 4.35	4.71	*(p=720)* 4.70	4.58	*(p=1912)* 4.56	4.59
			100	1.62	1.61	**3.37**	2.25	4.39	4.43	11.66	11.70	18.29	18.71	20.97	21.71
			500	1.62	1.61	**3.37**	2.25	4.39	4.43	11.67	11.70	33.63	**37.95**	66.37	61.59
			∞	*1.62*	*1.61*	***3.37***	*2.25*	*4.39*	*4.43*	*11.67*	*11.70*	*33.63*	***37.95***	*75.13*	***83.59***
	12	96122	20	*(p=5)* 3.66	3.65	*(p=20)* 4.70	**6.29**	*(p=60)* 6.07	6.11	*(p=180)* 6.87	6.97	*(p=716)* 7.50	7.35	*(p=1896)* 7.41	7.16
			100	3.66	3.65	4.70	**6.29**	7.23	**10.65**	17.79	17.02	27.55	**32.35**	33.29	32.34
			500	3.66	3.65	4.70	**6.29**	7.23	**10.65**	17.83	17.02	45.77	**74.05**	94.79	97.29
			∞	*3.66*	*3.65*	*4.70*	***6.29***	*7.23*	***10.65***	*17.83*	*17.02*	*45.86*	***74.05***	*104.94*	***153.80***
ped39 n=1272 k=5 w=21 h=76	4	6632	20	*(p=4)* 2.47	2.43	*(p=16)* 4.44	4.41	*(p=128)* 9.52	9.11	*(p=768)* 9.35	**11.72**	*(p=2304)* 9.87	9.80		
			100	2.47	2.43	4.44	4.41	11.13	11.61	12.85	**20.86**	26.32	**34.91**		
			500	2.47	2.43	4.44	4.41	11.13	11.61	13.51	**21.82**	36.64	**51.41**		
			∞	*2.47*	*2.43*	*4.44*	*4.41*	*11.13*	*11.61*	*13.51*	***21.82***	*39.24*	***54.81***		
	5	2202	20	*(p=4)* 2.78	2.54	*(p=16)* 5.38	**6.13**	*(p=128)* 5.23	**7.54**	*(p=768)* 5.22	**7.20**	*(p=2304)* 4.97	4.87		
			100	2.78	2.54	5.38	**6.13**	6.95	7.54	7.92	**15.62**	14.12	**18.05**		
			500	2.78	2.54	5.38	**6.13**	7.06	7.54	8.70	**19.84**	21.38	**31.46**		
			∞	*2.78*	*2.54*	*5.38*	***6.13***	*7.06*	*7.54*	*8.74*	***20.20***	*23.43*	***31.46***		
Better by 10%				4x	4x	16x	16x	14x	21x	10x	20x	7x	26x	1x	10x
Better by 50%				4x	4x	7x	3x	13x	6x	8x	8x	5x	3x	0x	1x

Table 4.9: Subset of parallel **speedup** results on **linkage instances, part 1 of 2**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 2 fix	2 var	4 fix	4 var	6 fix	6 var	8 fix	8 var	10 fix	10 var	12 fix	12 var
				(p=8)		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
ped41	9	25607	20	1.58	**1.79**	5.41	**8.89**	11.68	11.71	10.47	**12.09**	10.55	10.42	8.62	8.59
n=1062			100	1.58	**1.79**	5.61	**10.51**	19.59	**32.29**	30.96	**44.46**	46.99	49.24	31.81	**38.45**
k=5			500	1.58	**1.79**	5.61	**10.51**	20.13	**32.29**	39.40	**75.31**	102.02	**121.36**	57.41	**113.81**
w=33			∞	*1.58*	***1.79***	*5.61*	***10.51***	*20.13*	***32.29***	*39.40*	***75.31***	*115.35*	***148.88***	*63.86*	***173.02***
h=100				*(p=8)*		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
	10	46819	20	1.41	**2.71**	4.84	**9.65**	12.96	14.06	14.16	14.64	15.31	14.46	13.35	13.23
			100	1.41	**2.71**	4.89	**9.65**	16.95	**27.36**	31.57	**39.85**	**51.79**	45.37	**62.34**	54.06
			500	1.41	**2.71**	4.89	**9.65**	17.16	**27.36**	36.75	39.85	**66.41**	46.87	90.21	92.89
			∞	*1.41*	***2.71***	*4.89*	***9.65***	*17.16*	***27.36***	*36.75*	*39.85*	***67.46***	*46.87*	*91.09*	***107.63***
				(p=8)		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
	11	27583	20	1.36	**2.71**	4.52	**7.11**	12.60	**14.78**	13.45	**15.46**	13.87	14.04	12.16	12.17
			100	1.36	**2.71**	4.57	**7.11**	16.83	**25.10**	27.31	**49.88**	**50.33**	34.74	50.33	**56.18**
			500	1.36	**2.71**	4.57	**7.11**	17.08	**25.10**	30.65	**55.50**	**94.46**	45.44	106.50	**154.09**
			∞	*1.36*	***2.71***	*4.57*	***7.11***	*17.08*	***25.10***	*30.65*	***55.50***	***102.16***	*47.48*	*130.11*	***235.75***
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
ped44	5	207136	20	3.15	3.14	**10.16**	6.62	16.29	15.43	12.03	12.73	10.04	10.15	7.11	**8.08**
n=811			100	3.15	3.14	**10.16**	6.62	**60.65**	19.43	51.27	**57.54**	48.38	45.51	35.00	37.79
k=4			500	3.15	3.14	**10.16**	6.62	**60.65**	19.43	**133.29**	110.89	**207.14**	147.43	**163.10**	130.77
w=25			∞	*3.15*	*3.14*	***10.16***	*6.62*	***60.65***	*19.43*	***151.08***	*110.89*	***565.95***	*206.72*	***281.05***	*224.90*
h=65				*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
	6	95830	20	3.58	3.57	9.86	10.11	14.22	14.07	12.04	12.06	9.49	9.82	7.72	7.68
			100	3.58	3.57	9.86	10.11	**40.88**	26.72	53.27	56.37	45.08	42.10	37.65	37.68
			500	3.58	3.57	9.86	10.11	**57.76**	26.72	**164.37**	108.16	**178.79**	105.89	**168.42**	116.30
			∞	*3.58*	*3.57*	*9.86*	*10.11*	***57.76***	*26.72*	***208.78***	*108.16*	***221.32***	*123.81*	***930.39***	*137.69*
				(p=4)		*(p=144)*		*(p=2160)*		*(p=14401)*					
ped50	3	4135	20	2.78	2.80	9.78	**11.99**	8.89	9.17	2.38	2.39				
n=514			100	2.78	2.80	11.99	11.99	20.88	**39.01**	10.85	10.94				
k=6			500	2.78	2.80	11.99	11.99	26.01	**46.99**	32.56	**37.25**				
w=17			∞	*2.78*	*2.80*	*11.99*	*11.99*	*27.03*	***46.99***	*46.46*	***78.02***				
h=47				*(p=4)*		*(p=144)*		*(p=2160)*		*(p=14400)*					
	4	1780	20	6.54	6.98	23.73	**26.57**	6.52	6.43	1.15	1.15				
			100	6.54	6.98	42.38	43.41	28.71	29.18	5.20	5.24				
			500	6.54	6.98	42.38	43.41	65.93	**93.68**	17.62	17.98				
			∞	*6.54*	*6.98*	*42.38*	*43.41*	*80.91*	***104.71***	*38.70*	*42.38*				
				(p=4)		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4064)*	
ped51	20	101788	20	3.73	3.73	12.32	13.22	14.44	**16.35**	15.89	17.30	15.49	15.49	10.28	10.32
n=1152			100	3.73	3.73	12.32	13.22	29.44	**38.29**	43.50	**60.34**	**64.50**	46.10	46.56	50.27
k=5			500	3.73	3.73	12.32	13.22	29.44	**38.29**	57.44	**87.52**	**144.59**	66.75	119.47	**136.08**
w=39			∞	*3.73*	*3.73*	*12.32*	*13.22*	*29.44*	***38.29***	*57.44*	***87.52***	***149.47***	*70.69*	*147.73*	*150.80*
h=98				*(p=4)*		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
	21	164817	20	3.82	3.88	14.28	14.61	**18.25**	16.47	18.89	17.33	16.13	15.71	9.71	9.63
			100	3.82	3.88	14.28	14.61	33.86	33.86	**70.16**	55.87	77.31	73.15	46.60	47.09
			500	3.82	3.88	14.28	14.61	33.86	33.86	70.16	**88.61**	**181.52**	104.91	**156.97**	141.47
			∞	*3.82*	*3.88*	*14.28*	*14.61*	*33.86*	*33.86*	*70.16*	***88.61***	***181.52***	*104.91*	*195.05*	*211.85*
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
ped7	6	118383	20	**3.35**	2.01	**9.59**	2.04	13.11	13.90	12.26	**16.17**	13.60	**15.61**	14.37	15.39
n=1068			100	**3.35**	2.01	**9.90**	2.04	**23.11**	15.39	24.36	**51.34**	30.13	**65.26**	44.77	**71.79**
k=4			500	**3.35**	2.01	**9.90**	2.04	**23.75**	15.39	27.16	**56.75**	35.94	**90.99**	67.11	**125.54**
w=32			∞	***3.35***	*2.01*	***9.90***	*2.04*	***23.75***	*15.39*	*27.16*	***56.75***	*36.36*	***90.99***	*68.04*	***135.14***
h=90				*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
	7	93380	20	**3.72**	1.97	**11.69**	1.82	**18.62**	3.58	15.80	**17.40**	15.41	**17.10**	15.76	16.37
			100	**3.72**	1.97	**11.69**	1.82	**31.69**	3.69	31.11	**46.76**	33.13	**62.05**	43.31	**73.64**
			500	**3.72**	1.97	**11.69**	1.82	**31.69**	3.69	35.71	**68.26**	39.22	**79.14**	65.16	**182.03**
			∞	***3.72***	*1.97*	***11.69***	*1.82*	***31.69***	*3.69*	*35.71*	***68.26***	*39.22*	***79.14***	*65.71*	***237.01***
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1276)*		*(p=3816)*	
	8	30717	20	**3.45**	1.68	**10.32**	1.67	**13.10**	2.96	13.94	**15.85**	13.99	**17.20**	14.56	14.85
			100	**3.45**	1.68	**10.32**	1.67	**24.07**	3.12	26.80	**30.59**	28.57	**32.54**	33.53	**51.11**
			500	**3.45**	1.68	**10.32**	1.67	**24.46**	3.12	28.18	**35.97**	30.47	**36.88**	45.04	**89.03**
			∞	***3.45***	*1.68*	***10.32***	*1.67*	***24.46***	*3.12*	*28.18*	***35.97***	*30.47*	***36.88***	*47.40*	***104.48***
				(p=4)		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
ped9	6	101172	20	**3.66**	1.94	**9.76**	5.68	13.89	13.75	16.23	15.43	17.02	16.59	15.22	15.06
n=1118			100	**3.66**	1.94	**9.76**	5.68	**18.60**	14.42	**64.90**	37.78	79.48	78.19	72.79	72.01
k=7			500	**3.66**	1.94	**9.76**	5.68	**18.60**	14.42	**71.96**	37.78	**256.13**	108.67	**293.25**	263.47
w=27			∞	***3.66***	*1.94*	***9.76***	*5.68*	***18.60***	*14.42*	***71.96***	*37.78*	***256.13***	*108.67*	***595.13***	*315.18*
h=100				*(p=4)*		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
	7	58657	20	3.81	3.75	**10.90**	9.66	11.39	12.24	16.10	15.66	16.68	16.47	15.33	14.91
			100	3.81	3.75	**10.90**	9.66	19.84	20.07	55.81	52.00	72.78	76.88	71.88	70.33
			500	3.81	3.75	**10.90**	9.66	19.84	20.07	**77.90**	52.00	**250.67**	164.77	**265.42**	142.03
			∞	*3.81*	*3.75*	***10.90***	*9.66*	*19.84*	*20.07*	***77.90***	*52.00*	***250.67***	*164.77*	***592.49***	*142.03*
				(p=4)		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
	8	41061	20	3.73	3.76	8.51	9.31	11.30	**12.78**	**16.04**	12.53	16.84	16.73	15.01	14.94
			100	3.73	3.76	8.51	9.31	**18.18**	13.01	**55.04**	22.49	75.90	70.43	71.04	71.53
			500	3.73	3.76	8.51	9.31	**18.18**	13.01	**67.98**	22.49	**161.02**	88.30	264.91	271.93
			∞	*3.73*	*3.76*	*8.51*	*9.31*	***18.18***	*13.01*	***67.98***	*22.49*	***161.02***	*88.30*	***526.42***	*380.19*
Better by 10%				16x	12x	24x	14x	24x	20x	14x	29x	21x	14x	11x	18x
Better by 50%				16x	8x	20x	12x	17x	9x	8x	14x	15x	6x	4x	12x

Table 4.10: Subset of parallel **speedup** results on **linkage instances, part 2 of 2**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

many times (for the given cutoff depth d) one scheme was better than the other by 10% (as marked bold in the table above) and 50%, respectively.

We observe that if the fourth, "best possible" value is close to the parallel runtimes above it, it indicates that the overall performance is dominated by a single subproblem, as it was the case for some of the examples discussed earlier. Note that this will always be the case when the number of subproblems is smaller than the number of CPUs.

In addition, Figures 4.17 and 4.18 show parallel runtime and speedup plots, respectively, of six different linkage instances from Tables 4.7 through 4.10, contrasting as before fixed-depth and variable-depth parallelization using 20, 100, and 500 CPUs.

General Performance. Results are in line with the examples discussed in-depth previously. On the one hand, we notice a number of excellent outcomes with significant speedup values. For instance, pedigree31 ($i = 10$ and $i = 11$) reaches speedups of around 320 with 500 CPUs, and pedigree9 sees speedups of over 260 for all i-bounds. For 100 CPUs a number of instances see speedups close to, or above 70 – for instance pedigree7 ($i = 6, 7$) and pedigree13 (all i-bounds). Pedigree13 is in fact also one of the best-performing instances with 20 CPUs, with the speedup reaching 20 for $i = 9$. On the other hand, we also observe a number of weaker results, for example on instances with relatively short sequential solution times like pedigree39 with highest speedup of 31.

As before, we recognize a number of cases where the performance of the variable-depth scheme seems to suffer from an imprecise subproblem complexity estimate. In the following we describe two of these, pedigree13 and pedigree9, in more detail.

Pedigree13. One notable example where the fixed-depths scheme maintains a marked edge over the variable-depth cutoff for most depths is pedigree13, both for $i = 8$, shown at the top left of Figures 4.17 and 4.18, and $i = 9$. Across all CPU counts, fixed-depth performance ceases earlier to be bound by the longest-running subproblem; for $i = 8$ and

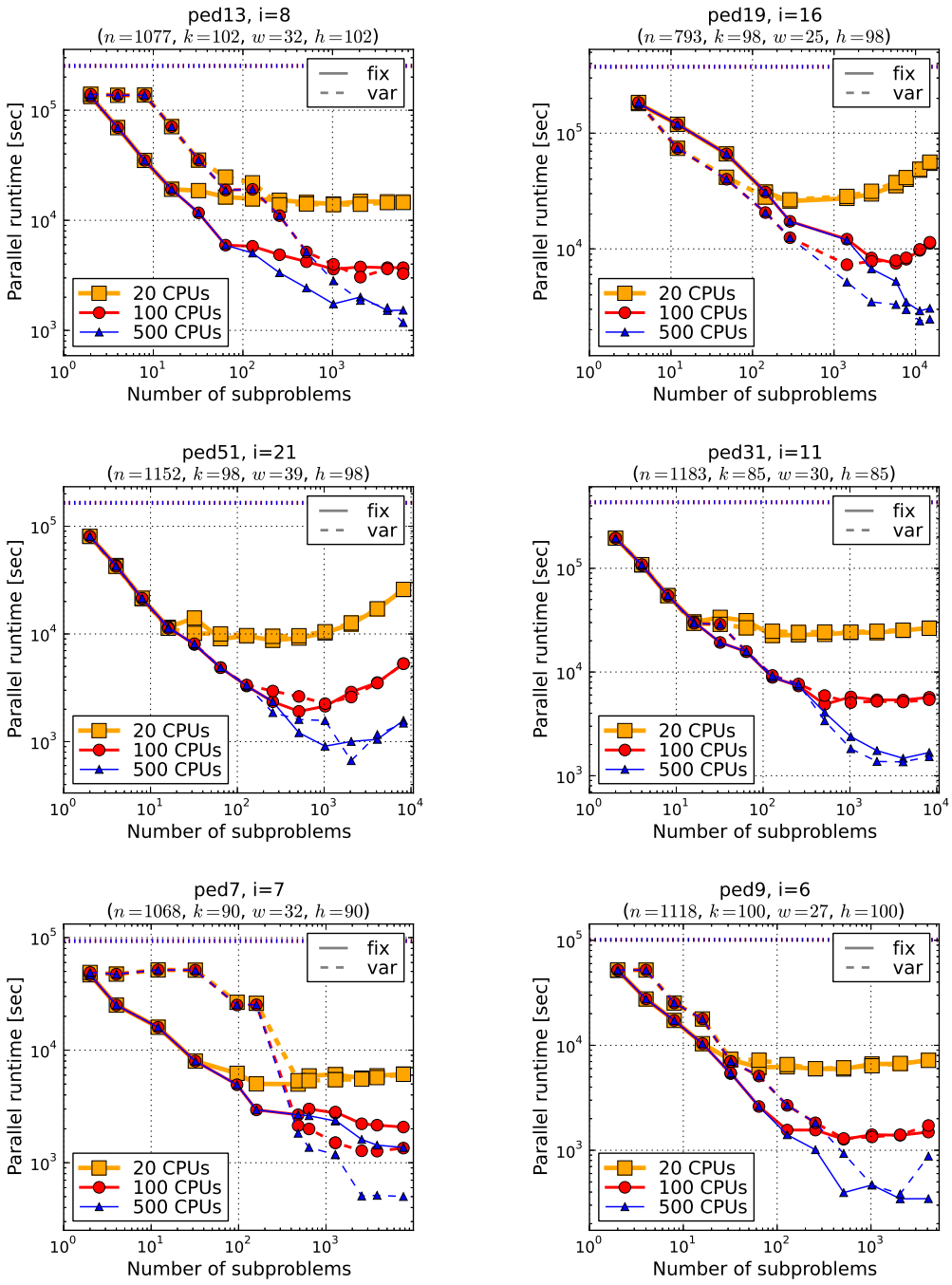


Figure 4.17: Parallel **runtime** plots for select **linkage** problems. Shown is the runtime using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). The instance's sequential solution time T_{seq} is indicated by the dashed horizontal line.

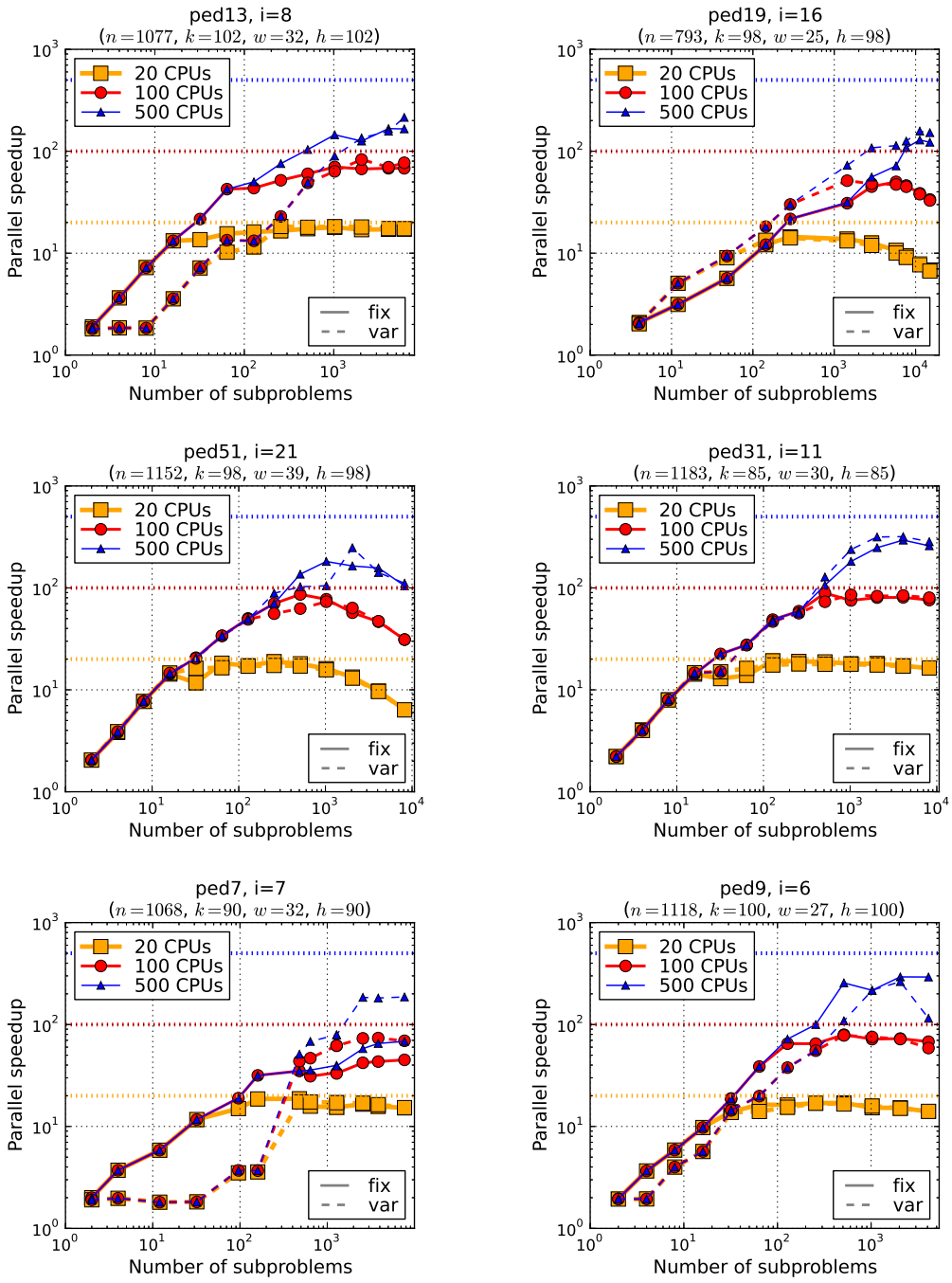


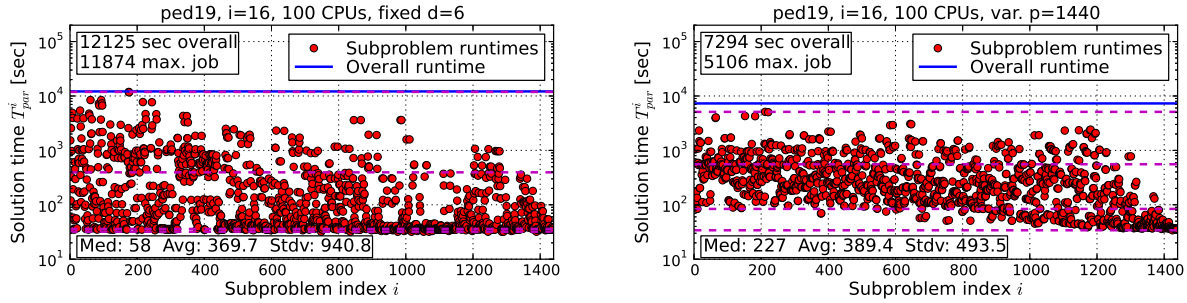
Figure 4.18: Parallel **speedup** plots for select **linkage** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.

100 CPUs at depth $d = 8$, for instance, variable-depth cutoff runtime of 11027 seconds is still determined by the longest running subproblem (as implied by the last row in Table 4.7, in italic), while the fixed-depth scheme takes only 4867 seconds (longest running subproblem 3339 seconds in last row).

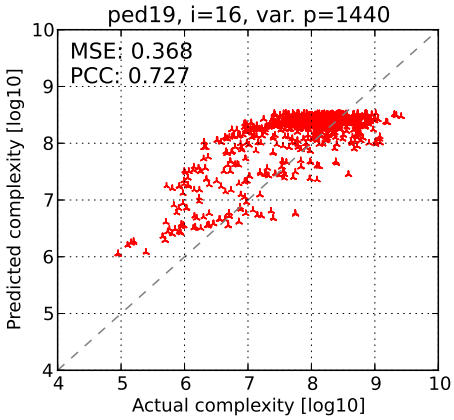
Pedigree9. Another example where fixed-depth parallelization appears ahead of variable-depth throughout most of the range of d values is pedigree9. For both schemes, performance is mostly dominated by the largest subproblem, as indicated by the last of Table 4.8 (especially for the higher CPU counts). Here variable-depth parallelization is often doing worse thanks to a very few underestimated subproblem complexities, as seen on previous experiments.

Pedigree19. Pedigree19, on the other hand, produces one of several "well-behaved" experiments with respect to fixed-depth and variable-depth performance (top right in Figures 4.17 and 4.18). It profits nicely from variable-depth parallelization, outperforming fixed-depth through most of the range of cutoff depths/sizes for all CPU counts and peaking at 2381 seconds (speedup 157) vs. 2912 seconds (speedup 129), with $p = 11254$ subproblems or depth $d = 10$, respectively. Figure 4.19a illustrates this by plotting subproblem runtimes of fixed-depth and variable-depth parallelization at depth $d = 8$ and with $p = 1440$ subproblems, respectively – here the latter is balanced enough that the overall runtime is not dominated by the largest subproblem. Looking at the scatter plot (Figure 4.19b) of actual vs. predicted subproblem complexities for the variable-depth run, we do observe a few minor outliers, but in this case these are not substantial enough to impact the overall performance.

Overall, however, these results are evidence of the Achilles' heel of the variable-depth parallel scheme: its performance relies to some extent on the accuracy of the subproblem complexity estimates. Namely, we have seen that it in some cases it takes just a single subproblem with vastly underestimated complexity to dominate the overall runtime and negatively impact parallel performance (recall also pedigree 7 in Section 4.6.3.4). On the other hand, in the



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 6$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 1440$.



(b) Scatter plot of actual vs. predicted subproblem complexity for variable-depth parallel run with $p = 1440$ subproblems.

Figure 4.19: Performance details of fixed-depth and variable-depth parallel scheme on pedigree19 instance ($i = 16$) with $d = 6$ and corresponding $p = 1440$ subproblems, respectively.

absence of such estimation outliers, we have shown our variable-depth scheme to be effective in improving parallel performance.

Subproblem Count for Fixed/variable-depth. We find that the variable-depth scheme seems to work better (relative to fixed-depth) for larger depths and the corresponding higher subproblem count. At depth $d = 4$, for instance, across both Tables 4.7 and 4.8 fixed-depth has the advantage in 40 cases and 27 cases for the 10% and 50% margin, respectively. Variable-depth is superior by 10% and 50% in 30 and 15 cases, respectively. At depth $d = 8$, however, this changes and variable-depth is superior by 10% and 50% in 49 and 22 cases, respectively, versus 24 and 10 for fixed-depth. Finally, for depth $d = 12$ (which we didn't

run for all instances), variable-depth has a 10% and 50% advantage in 28 and 13 cases, respectively, compared to 12 and 4 for the fixed-depth scheme.

Summary. We have observed a number of promising parallel runtime and speedup results on linkage instances, but also saw some mixed performance. In particular, we identified several of instances with very good speedups: close to 20 for 20 CPUS, 70-80 for 100 CPUs, and in the 200s and even 300s for 500 CPUs. In some other cases, however, we've described results that fall somewhat short, particularly for easier problem instances where the effects of overhead from the grid environment and even Amdahl's Law are more noticeable. We have also reconfirmed that in some cases the variable-depth scheme can be impeded by very inaccurate complexity estimates of a handful of subproblems (or even a single one). This, however, becomes less likely as the number of subproblems grows, which is when we've seen variable-depth parallelization dominate the fixed-depth variant.

4.6.4.2 Overall Analysis of Haplotyping Problems

Tables 4.11 and 4.12 present parallel runtime results on largeFam haplotyping instances and Tables 4.13 and 4.14 show the corresponding speedup values. The format is the same as explained for pedigree instances, i.e. we show parallel runtimes/speedups for fixed-depth and variable-depth parallelization, left and right in each field, respectively, on 20, 100, 500, and "unlimited" CPUs. In addition, Figures 4.20 and 4.21 show plots of parallel runtime and speedup for six of the problems.

General Performance. As for pedigrees in Section 4.6.4.1 we can identify several good results. A number of instances yield speedups of 17 or 18 using 20 CPUs, e.g., 18.86 and 17.6 on largeFam3-13-58 for $i = 14, 16$, respectively, with $d = 7$ – or even above 19 (largeFam4-12-50 for $i = 14$, $d = 5$). Similarly, for 100 CPUs, the highest speedups we see are in the 60s (e.g., 63.86 for largeFam3-15-53 with $i = 17$ at $d = 13$) or 70s (largeFam3-13-58 with $i = 14$,

instance	i	T_{seq}	#cpu	Cutoff depth d = 3 fix	3 var	5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
			20	**34103**	49291	14789	14313	14431	**10438**	12991	**10664**	11669	10866	11038	11574
	15	121311	100	**34103**	49291	14789	14313	13039	**8015**	12868	**5947**	6518	**3148**	4038	**2611**
			500	**34103**	49291	14789	14313	13039	**8015**	12868	**5947**	6501	**2419**	3150	**891**
			∞	***34103***	*49291*	*14789*	*14313*	*13039*	***8015***	*12868*	***5947***	*6501*	***2419***	*3099*	***877***
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
lF3-11-57			20	**9703**	13630	4689	4728	**3535**	3902	3453	**2984**	3002	2943	3508	3542
n=2670 k=3 w=37 h=95	16	35820	100	**9703**	13630	**4214**	4728	3383	3415	3446	**2272**	1733	**1018**	1156	**802**
			500	**9703**	13630	**4214**	4728	3383	3415	3446	**2272**	1733	**970**	935	**392**
			∞	***9703***	*13630*	***4214***	*4728*	*3383*	*3415*	*3446*	***2272***	*1733*	***970***	*935*	***384***
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
			20	**5413**	7022	2285	2436	1933	1848	2047	**1527**	1978	**1772**	2815	2760
	17	18312	100	**5413**	7022	2285	2436	1933	1848	1858	**970**	954	**531**	788	**585**
			500	**5413**	7022	2285	2436	1933	1848	1858	**970**	925	**531**	565	**202**
			∞	***5413***	*7022*	*2285*	*2436*	*1933*	*1848*	*1858*	***970***	*925*	***531***	*549*	***176***
				(p=10)		*(p=30)*		*(p=150)*		*(p=600)*		*(p=2000)*		*(p=4000)*	
			20	6309	5960	4790	**3789**	3214	**2866**	3539	3404	4308	4157	4838	4856
	14	35457	100	6309	5960	4790	**3789**	2147	**1220**	1161	**722**	1309	**871**	1315	**1019**
			500	6309	5960	4790	**3789**	2147	**1220**	1128	**341**	736	**304**	709	**259**
			∞	*6309*	*5960*	*4790*	***3789***	*2147*	***1220***	*1128*	***341***	*638*	***242***	*576*	***129***
				(p=10)		*(p=30)*		*(p=150)*		*(p=596)*		*(p=1962)*		*(p=3886)*	
lF3-11-59			20	1597	1590	1117	1127	959	**786**	1202	1130	1704	1658	2284	2259
n=2711 k=3 w=32 h=73	15	8523	100	1597	1590	1117	1127	530	**242**	469	**245**	404	**356**	508	495
			500	1597	1590	1117	1127	530	**242**	362	**101**	162	**103**	176	**142**
			∞	*1597*	*1590*	*1117*	*1127*	*530*	***242***	*362*	***101***	*154*	***67***	*122*	***78***
				(p=10)		*(p=30)*		*(p=150)*		*(p=600)*		*(p=1999)*		*(p=3992)*	
			20	739	**494**	442	**368**	446	412	788	835	1819	1783	3061	3017
	16	3023	100	739	**494**	372	368	166	157	209	**187**	395	389	664	654
			500	739	**494**	372	368	163	157	143	**71**	129	**112**	190	183
			∞	*739*	***494***	*372*	*368*	*163*	*157*	*143*	***71***	*79*	***65***	*90*	***80***
				(p=12)		*(p=60)*		*(p=200)*		*(p=600)*		*(p=2000)*		*(p=6400)*	
			20	**9384**	11947	3220	3007	2754	**2464**	2695	2562	2795	2727	3459	3441
	14	46464	100	**9384**	11947	**2350**	3007	1337	**1089**	1134	1044	708	**584**	760	751
			500	**9384**	11947	**2350**	3007	1222	**1089**	**884**	1030	380	**340**	224	225
			∞	***9384***	*11947*	***2350***	*3007*	*1222*	***1089***	***884***	*1030*	*380*	***340***	***130***	*180*
				(p=12)		*(p=60)*		*(p=200)*		*(p=600)*		*(p=1998)*		*(p=6390)*	
lF3-13-58			20	5073	5097	1648	**1361**	1244	1152	1366	1389	1575	1567	2829	2825
n=3352 k=3 w=31 h=88	16	20270	100	5073	5097	1478	1361	824	**508**	807	**606**	425	**340**	636	630
			500	5073	5097	1478	1361	800	**508**	742	**490**	302	**179**	207	190
			∞	*5073*	*5097*	*1478*	*1361*	*800*	***508***	*742*	***490***	*282*	***179***	*116*	*111*
				(p=12)		*(p=60)*		*(p=200)*		*(p=591)*		*(p=1958)*		*(p=6121)*	
			20	1705	1597	785	**488**	707	**605**	1049	1024	2502	2483	6933	6918
	18	7647	100	1705	1597	588	**488**	294	**210**	346	**235**	566	538	1472	1464
			500	1705	1597	588	**488**	262	**210**	319	**115**	177	**148**	389	375
			∞	*1705*	*1597*	*588*	***488***	*262*	***210***	*319*	***115***	*103*	*100*	*129*	*120*
				(p=12)		*(p=34)*		*(p=78)*		*(p=358)*		*(p=1093)*		*(p=2831)*	
			20	99673	100130	94280	**59566**	79765	**32938**	34001	**24934**	28754	**24071**	26657	**21190**
	17	345544	100	99673	100130	94280	**59566**	79765	**32938**	32364	**12653**	16594	**8995**	12624	**5411**
			500	99673	100130	94280	**59566**	79765	**32938**	32364	**12271**	16594	**7324**	12230	**4727**
lF3-15-53			∞	*99673*	*100130*	*94280*	***59566***	*79765*	***32938***	*32364*	***12271***	*16594*	***7324***	*12230*	***4727***
n=3384 k=3 w=32 h=108				*(p=12)*		*(p=32)*		*(p=68)*		*(p=284)*		*(p=912)*		*(p=2496)*	
			20	28557	29872	26907	**24321**	23792	**9719**	10796	**7007**	10158	**7481**	9946	9127
	18	98346	100	28557	29872	26907	**24321**	23792	**9719**	9803	**4136**	5403	**2788**	4702	**2592**
			500	28557	29872	26907	**24321**	23792	**9719**	9803	**4136**	5347	**2788**	4020	**1894**
			∞	*28557*	*29872*	*26907*	***24321***	*23792*	***9719***	*9803*	***4136***	*5347*	***2788***	*4020*	***1856***
Better by 10%				16x	4x	6x	18x	1x	31x	2x	35x	0x	33x	1x	22x
Better by 50%				0x	0x	0x	5x	0x	20x	0x	28x	0x	22x	0x	16x

Table 4.11: Subset of parallel **runtime** results on **haplotyping instances, part 1 of 2**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 3 fix	3 var	5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var
				(p=8)		*(p=40)*		*(p=240)*		*(p=942)*		*(p=3633)*		*(p=13781)*	
			20	**5293**	6734	2840	**2068**	1791	**1610**	2045	1959	3227	3302	8120	8209
	18	28613	100	**5293**	6734	2055	2068	893	**585**	824	**462**	730	707	1773	1792
			500	**5293**	6734	2055	2068	842	**535**	636	**462**	319	**211**	518	508
lF3-15-59			∞	***5293***	*6734*	*2055*	*2068*	*842*	***535***	*636*	***462***	*250*	***127***	*253*	***229***
n=3730 k=3 w=31 h=84				*(p=8)*		*(p=40)*		*(p=240)*		*(p=936)*		*(p=3571)*		*(p=13482)*	
			20	10234	10164	3684	3417	2626	2398	2852	2854	4734	4775	11914	11913
	19	43307	100	10234	10164	3684	3417	1485	**1079**	1296	**658**	1042	1008	2534	2541
			500	10234	10164	3684	3417	1485	**1079**	1113	**417**	508	**291**	660	667
			∞	*10234*	*10164*	*3684*	*3417*	*1485*	***1079***	*1113*	***417***	*442*	***177***	*317*	***251***
				(p=9)		*(p=43)*		*(p=205)*		*(p=934)*		*(p=1827)*		*(p=7582)*	
			20	643626	639982	325905	**200608**	164502	**149029**	**160338**	177599	**180036**	198701	**217768**	251809
	15	1891710	100	643626	639982	316651	**186789**	119413	**42519**	46136	**38277**	47363	**40579**	48752	50451
			500	643626	639982	316651	**186789**	119016	**42519**	35911	**15754**	28697	**11309**	17870	**11166**
lF3-16-56			∞	*643626*	*639982*	*316651*	***186789***	*119016*	***42519***	*35893*	***15754***	*26721*	***10225***	*15239*	***4262***
n=3930 k=3 w=38 h=77				*(p=9)*		*(p=42)*		*(p=201)*		*(p=900)*		*(p=1766)*		*(p=7122)*	
			20	182770	**125290**	89282	**56806**	53173	**47562**	56246	58857	**60079**	67647	**78744**	90965
	16	489614	100	182770	**125290**	81351	**56806**	33623	**26361**	19126	**12956**	15158	**13653**	19498	18292
			500	182770	**125290**	81351	**56806**	33425	**26361**	12942	**6514**	10045	**5321**	8158	**4118**
			∞	*182770*	***125290***	*81351*	***56806***	*33425*	***26361***	*12595*	***6514***	*9849*	***5321***	*6573*	***1345***
				(p=24)		*(p=288)*		*(p=3456)*							
			20	4810	4863	3413	3408	4245	4149						
	13	57842	100	4810	4863	1190	**1052**	908	861						
			500	4810	4863	1103	**899**	555	**281**						
lF4-12-50			∞	*4810*	*4863*	*1103*	***899***	*536*	***236***						
n=2569 k=4 w=28 h=80				*(p=24)*		*(p=288)*		*(p=3456)*							
			20	2946	**2423**	1750	1897	2542	2551						
	14	33676	100	2720	**2238**	**637**	1425	575	535						
			500	2720	**2238**	**564**	1425	204	**140**						
			∞	*2720*	***2238***	***564***	*1425*	*151*	***101***						
				(p=8)		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=1792)*		*(p=3072)*	
			20	**16287**	27758	**8247**	15781	**7623**	15799	**7279**	9773	7746	7590	8110	7913
	13	104837	100	**16287**	27758	**3732**	13666	**2278**	13877	**1651**	5958	1953	**1764**	2004	**1721**
			500	**16287**	27758	**3732**	13666	**1689**	13628	**672**	5540	813	**712**	887	863
lF4-12-55			∞	***16287***	*27758*	***3732***	*13666*	***1689***	*13628*	***672***	*5540*	***606***	*691*	*730*	*760*
n=2926 k=4 w=28 h=78				*(p=8)*		*(p=48)*		*(p=192)*		*(p=768)*		*(p=1536)*		*(p=3072)*	
			20	**3595**	6882	**2103**	3599	1968	**1788**	2006	1931	2386	2296	2986	2991
	14	25905	100	**3595**	6882	**1181**	3599	699	759	574	**474**	589	**498**	717	**648**
			500	**3595**	6882	**1181**	3599	**566**	759	281	**216**	**230**	341	375	**233**
			∞	***3595***	*6882*	***1181***	*3599*	***566***	*759*	*255*	***216***	***164***	*341*	*329*	***191***
				(p=4)		*(p=16)*		*(p=40)*		*(p=128)*		*(p=176)*		*(p=400)*	
			20	2819	2785	1322	1287	1223	1165	1053	**933**	1039	994	1312	1285
	15	10607	100	2819	2785	1322	1287	766	773	392	**336**	413	**332**	**310**	487
			500	2819	2785	1322	1287	766	773	333	336	317	332	**136**	391
lF4-17-51			∞	*2819*	*2785*	*1322*	*1287*	*766*	*773*	*333*	*336*	*317*	*332*	***136***	*391*
n=3837 k=4 w=29 h=85				*(p=8)*		*(p=32)*		*(p=80)*		*(p=256)*		*(p=352)*		*(p=800)*	
			20	**15508**	29154	**7281**	28848	**4805**	15050	4224	3934	3922	4213	4049	4086
	16	66103	100	**15508**	29154	**7281**	28848	**3719**	15050	1733	1860	1848	1992	1125	1065
			500	**15508**	29154	**7281**	28848	**3719**	15050	1668	1620	1641	1633	**606**	951
			∞	***15508***	*29154*	***7281***	*28848*	***3719***	*15050*	*1668*	*1620*	*1641*	*1633*	***606***	*951*
Better by 10%				16x	8x	15x	12x	10x	20x	5x	17x	5x	14x	7x	10x
Better by 50%				12x	0x	15x	5x	8x	8x	3x	8x	1x	8x	5x	6x

Table 4.12: Subset of parallel **runtime** results on **haplotyping instances, part 2 of 2**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 3 fix	3 var	5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
			20	**3.56**	2.46	8.20	8.48	8.41	**11.62**	9.34	**11.38**	10.40	11.16	10.99	10.48
	15	121311	100	**3.56**	2.46	8.20	8.48	9.30	**15.14**	9.43	**20.40**	18.61	**38.54**	30.04	**46.46**
			500	**3.56**	2.46	8.20	8.48	9.30	**15.14**	9.43	**20.40**	18.66	**50.15**	38.51	**136.15**
			∞	***3.56***	*2.46*	*8.20*	*8.48*	*9.30*	***15.14***	*9.43*	***20.40***	*18.66*	***50.15***	*39.15*	***138.32***
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
lF3-11-57			20	**3.69**	2.63	7.64	7.58	**10.13**	9.18	10.37	**12.00**	11.93	12.17	10.21	10.11
n=2670 k=3 w=37 h=95	16	35820	100	**3.69**	2.63	**8.50**	7.58	10.59	10.49	10.39	**15.77**	20.67	**35.19**	30.99	**44.66**
			500	**3.69**	2.63	**8.50**	7.58	10.59	10.49	10.39	**15.77**	20.67	**36.93**	38.31	**91.38**
			∞	***3.69***	*2.63*	***8.50***	*7.58*	*10.59*	*10.49*	*10.39*	***15.77***	*20.67*	***36.93***	*38.31*	***93.28***
				(p=6)		*(p=30)*		*(p=60)*		*(p=120)*		*(p=360)*		*(p=1440)*	
			20	**3.38**	2.61	8.01	7.52	9.47	9.91	8.95	**11.99**	9.26	**10.33**	6.51	6.63
	17	18312	100	**3.38**	2.61	8.01	7.52	9.47	9.91	9.86	**18.88**	19.19	**34.49**	23.24	**31.30**
			500	**3.38**	2.61	8.01	7.52	9.47	9.91	9.86	**18.88**	19.80	**34.49**	32.41	**90.65**
			∞	***3.38***	*2.61*	*8.01*	*7.52*	*9.47*	*9.91*	*9.86*	***18.88***	*19.80*	***34.49***	*33.36*	***104.05***
				(p=10)		*(p=30)*		*(p=150)*		*(p=600)*		*(p=2000)*		*(p=4000)*	
			20	5.62	5.95	7.40	**9.36**	11.03	**12.37**	10.02	10.42	8.23	8.53	7.33	7.30
	14	35457	100	5.62	5.95	7.40	**9.36**	16.51	**29.06**	30.54	**49.11**	27.09	**40.71**	26.96	**34.80**
			500	5.62	5.95	7.40	**9.36**	16.51	**29.06**	31.43	**103.98**	48.18	**116.63**	50.01	**136.90**
			∞	*5.62*	*5.95*	*7.40*	***9.36***	*16.51*	***29.06***	*31.43*	***103.98***	*55.58*	***146.52***	*61.56*	***274.86***
				(p=10)		*(p=30)*		*(p=150)*		*(p=596)*		*(p=1962)*		*(p=3886)*	
lF3-11-59			20	5.34	5.36	7.63	7.56	8.89	**10.84**	7.09	7.54	5.00	5.14	3.73	3.77
n=2711 k=3 w=32 h=73	15	8523	100	5.34	5.36	7.63	7.56	16.08	**35.22**	18.17	**34.79**	21.10	**23.94**	16.78	17.22
			500	5.34	5.36	7.63	7.56	16.08	**35.22**	23.54	**84.39**	52.61	**82.75**	48.43	**60.02**
			∞	*5.34*	*5.36*	*7.63*	*7.56*	*16.08*	***35.22***	*23.54*	***84.39***	*55.34*	***127.21***	*69.86*	***109.27***
				(p=10)		*(p=30)*		*(p=150)*		*(p=600)*		*(p=1999)*		*(p=3992)*	
			20	4.09	**6.12**	6.84	**8.21**	6.78	7.34	3.84	3.62	1.66	1.70	0.99	1.00
	16	3023	100	4.09	**6.12**	8.13	8.21	18.21	19.25	14.46	**16.17**	7.65	7.77	4.55	4.62
			500	4.09	**6.12**	8.13	8.21	18.55	19.25	21.14	**42.58**	23.43	**26.99**	15.91	16.52
			∞	*4.09*	***6.12***	*8.13*	*8.21*	*18.55*	*19.25*	*21.14*	***42.58***	*38.27*	***46.51***	*33.59*	***37.79***
				(p=12)		*(p=60)*		*(p=200)*		*(p=600)*		*(p=2000)*		*(p=6400)*	
			20	**4.95**	3.89	14.43	15.45	16.87	**18.86**	17.24	18.14	16.62	17.04	13.43	13.50
	14	46464	100	**4.95**	3.89	**19.77**	15.45	34.75	**42.67**	40.97	44.51	65.63	**79.56**	61.14	61.87
			500	**4.95**	3.89	**19.77**	15.45	38.02	**42.67**	**52.56**	45.11	122.27	**136.66**	207.43	206.51
			∞	***4.95***	*3.89*	***19.77***	*15.45*	*38.02*	***42.67***	***52.56***	*45.11*	*122.27*	***136.66***	***357.42***	*258.13*
				(p=12)		*(p=60)*		*(p=200)*		*(p=600)*		*(p=1998)*		*(p=6390)*	
lF3-13-58			20	4.00	3.98	12.30	**14.89**	16.29	17.60	14.84	14.59	12.87	12.94	7.17	7.18
n=3352 k=3 w=31 h=88	16	20270	100	4.00	3.98	13.71	14.89	24.60	**39.90**	25.12	**33.45**	47.69	**59.62**	31.87	32.17
			500	4.00	3.98	13.71	14.89	25.34	**39.90**	27.32	**41.37**	67.12	**113.24**	97.92	106.68
			∞	*4.00*	*3.98*	*13.71*	*14.89*	*25.34*	***39.90***	*27.32*	***41.37***	*71.88*	***113.24***	*174.74*	*182.61*
				(p=12)		*(p=60)*		*(p=200)*		*(p=591)*		*(p=1958)*		*(p=6121)*	
			20	4.49	4.79	9.74	**15.67**	10.82	**12.64**	7.29	7.47	3.06	3.08	1.10	1.11
	18	7647	100	4.49	4.79	13.01	**15.67**	26.01	**36.41**	22.10	**32.54**	13.51	14.21	5.19	5.22
			500	4.49	4.79	13.01	**15.67**	29.19	**36.41**	23.97	**66.50**	43.20	**51.67**	19.66	20.39
			∞	*4.49*	*4.79*	*13.01*	***15.67***	*29.19*	***36.41***	*23.97*	***66.50***	*74.24*	*76.47*	*59.28*	*63.73*
				(p=12)		*(p=34)*		*(p=78)*		*(p=358)*		*(p=1093)*		*(p=2831)*	
			20	3.47	3.45	3.67	**5.80**	4.33	**10.49**	10.16	**13.86**	12.02	**14.36**	12.96	**16.31**
	17	345544	100	3.47	3.45	3.67	**5.80**	4.33	**10.49**	10.68	**27.31**	20.82	**38.42**	27.37	**63.86**
			500	3.47	3.45	3.67	**5.80**	4.33	**10.49**	10.68	**28.16**	20.82	**47.18**	28.25	**73.10**
lF3-15-53			∞	*3.47*	*3.45*	*3.67*	***5.80***	*4.33*	***10.49***	*10.68*	***28.16***	*20.82*	***47.18***	*28.25*	***73.10***
n=3384 k=3 w=32 h=108				*(p=12)*		*(p=32)*		*(p=68)*		*(p=284)*		*(p=912)*		*(p=2496)*	
			20	3.44	3.29	3.66	**4.04**	4.13	**10.12**	9.11	**14.04**	9.68	**13.15**	9.89	10.78
	18	98346	100	3.44	3.29	3.66	**4.04**	4.13	**10.12**	10.03	**23.78**	18.20	**35.27**	20.92	**37.94**
			500	3.44	3.29	3.66	**4.04**	4.13	**10.12**	10.03	**23.78**	18.39	**35.27**	24.46	**51.93**
			∞	*3.44*	*3.29*	*3.66*	***4.04***	*4.13*	***10.12***	*10.03*	***23.78***	*18.39*	***35.27***	*24.46*	***52.99***
Better by 10%				16x	4x	6x	18x	1x	31x	2x	35x	0x	33x	1x	22x
Better by 50%				0x	0x	0x	5x	0x	20x	0x	28x	0x	22x	0x	16x

Table 4.13: Subset of parallel **speedup** results on **haplotyping instances, part 1 of 2**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 3 fix	3 var	5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var
				(p=8)		*(p=40)*		*(p=240)*		*(p=942)*		*(p=3633)*		*(p=13781)*	
			20	**5.41**	4.25	10.07	**13.84**	15.98	**17.77**	13.99	14.61	8.87	8.67	3.52	3.49
	18	28613	100	**5.41**	4.25	13.92	13.84	32.04	**48.91**	34.72	**61.93**	39.20	40.47	16.14	15.97
			500	**5.41**	4.25	13.92	13.84	33.98	**53.48**	44.99	**61.93**	89.70	**135.61**	55.24	56.32
lF3-15-59			∞	***5.41***	*4.25*	*13.92*	*13.84*	*33.98*	***53.48***	*44.99*	***61.93***	*114.45*	***225.30***	*113.09*	***124.95***
n=3730 k=3 w=31 h=84				*(p=8)*		*(p=40)*		*(p=240)*		*(p=936)*		*(p=3571)*		*(p=13482)*	
			20	4.23	4.26	11.76	12.67	16.49	18.06	15.18	15.17	9.15	9.07	3.63	3.64
	19	43307	100	4.23	4.26	11.76	12.67	29.16	**40.14**	33.42	**65.82**	41.56	42.96	17.09	17.04
			500	4.23	4.26	11.76	12.67	29.16	**40.14**	38.91	**103.85**	85.25	**148.82**	65.62	64.93
			∞	*4.23*	*4.26*	*11.76*	*12.67*	*29.16*	***40.14***	*38.91*	***103.85***	*97.98*	***244.67***	*136.62*	***172.54***
				(p=9)		*(p=43)*		*(p=205)*		*(p=934)*		*(p=1827)*		*(p=7582)*	
			20	2.94	2.96	5.80	**9.43**	11.50	**12.69**	**11.80**	10.65	**10.51**	9.52	**8.69**	7.51
	15	1891710	100	2.94	2.96	5.97	**10.13**	15.84	**44.49**	41.00	**49.42**	39.94	**46.62**	38.80	37.50
			500	2.94	2.96	5.97	**10.13**	15.89	**44.49**	52.68	**120.08**	65.92	**167.27**	105.86	**169.42**
lF3-16-56			∞	*2.94*	*2.96*	*5.97*	***10.13***	*15.89*	***44.49***	*52.70*	***120.08***	*70.79*	***185.01***	*124.14*	***443.85***
n=3930 k=3 w=38 h=77				*(p=9)*		*(p=42)*		*(p=201)*		*(p=900)*		*(p=1766)*		*(p=7122)*	
			20	2.68	**3.91**	5.48	**8.62**	9.21	**10.29**	8.70	8.32	**8.15**	7.24	**6.22**	5.38
	16	489614	100	2.68	**3.91**	6.02	**8.62**	14.56	**18.57**	25.60	**37.79**	32.30	**35.86**	25.11	26.77
			500	2.68	**3.91**	6.02	**8.62**	14.65	**18.57**	37.83	**75.16**	48.74	**92.02**	60.02	**118.90**
			∞	*2.68*	***3.91***	*6.02*	***8.62***	*14.65*	***18.57***	*38.87*	***75.16***	*49.71*	***92.02***	*74.49*	***364.03***
				(p=24)		*(p=288)*		*(p=3456)*							
			20	12.03	11.89	16.95	16.97	13.63	13.94						
	13	57842	100	12.03	11.89	48.61	**54.98**	63.70	67.18						
			500	12.03	11.89	52.44	**64.34**	104.22	**205.84**						
lF4-12-50			∞	*12.03*	*11.89*	*52.44*	***64.34***	*107.91*	***245.09***						
n=2569 k=4 w=28 h=80				*(p=24)*		*(p=288)*		*(p=3456)*							
			20	11.43	**13.90**	19.24	17.75	13.25	13.20						
	14	33676	100	12.38	**15.05**	**52.87**	23.63	58.57	62.95						
			500	12.38	**15.05**	**59.71**	23.63	165.08	**240.54**						
			∞	*12.38*	***15.05***	***59.71***	*23.63*	*223.02*	***333.43***						
				(p=8)		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=1792)*		*(p=3072)*	
			20	**6.44**	3.78	**12.71**	6.64	**13.75**	6.64	**14.40**	10.73	13.53	13.81	12.93	13.25
	13	104837	100	**6.44**	3.78	**28.09**	7.67	**46.02**	7.55	**63.50**	17.60	53.68	**59.43**	52.31	**60.92**
			500	**6.44**	3.78	**28.09**	7.67	**62.07**	7.69	**156.01**	18.92	128.95	**147.24**	118.19	121.48
lF4-12-55			∞	***6.44***	*3.78*	***28.09***	*7.67*	***62.07***	*7.69*	***156.01***	*18.92*	***173.00***	*151.72*	*143.61*	*137.94*
n=2926 k=4 w=28 h=78				*(p=8)*		*(p=48)*		*(p=192)*		*(p=768)*		*(p=1536)*		*(p=3072)*	
			20	**7.21**	3.76	**12.32**	7.20	13.16	**14.49**	12.91	13.42	10.86	11.28	8.68	8.66
	14	25905	100	**7.21**	3.76	**21.93**	7.20	37.06	34.13	45.13	**54.65**	43.98	**52.02**	36.13	**39.98**
			500	**7.21**	3.76	**21.93**	7.20	**45.77**	34.13	92.19	**119.93**	**112.63**	75.97	69.08	**111.18**
			∞	***7.21***	*3.76*	***21.93***	*7.20*	***45.77***	*34.13*	*101.59*	***119.93***	***157.96***	*75.97*	*78.74*	***135.63***
				(p=4)		*(p=16)*		*(p=40)*		*(p=128)*		*(p=176)*		*(p=400)*	
			20	3.76	3.81	8.02	8.24	8.67	9.10	10.07	**11.37**	10.21	10.67	8.08	8.25
	15	10607	100	3.76	3.81	8.02	8.24	13.85	13.72	27.06	**31.57**	25.68	**31.95**	**34.22**	21.78
			500	3.76	3.81	8.02	8.24	13.85	13.72	31.85	31.57	33.46	31.95	**77.99**	27.13
lF4-17-51			∞	*3.76*	*3.81*	*8.02*	*8.24*	*13.85*	*13.72*	*31.85*	*31.57*	*33.46*	*31.95*	***77.99***	*27.13*
n=3837 k=4 w=29 h=85				*(p=8)*		*(p=32)*		*(p=80)*		*(p=256)*		*(p=352)*		*(p=800)*	
			20	**4.26**	2.27	**9.08**	2.29	**13.76**	4.39	15.65	16.80	16.85	15.69	16.33	16.18
	16	66103	100	**4.26**	2.27	**9.08**	2.29	**17.77**	4.39	38.14	35.54	35.77	33.18	58.76	62.07
			500	**4.26**	2.27	**9.08**	2.29	**17.77**	4.39	39.63	40.80	40.28	40.48	**109.08**	69.51
			∞	***4.26***	*2.27*	***9.08***	*2.29*	***17.77***	*4.39*	*39.63*	*40.80*	*40.28*	*40.48*	***109.08***	*69.51*
Better by 10%				16x	8x	15x	12x	10x	20x	5x	17x	5x	14x	7x	10x
Better by 50%				12x	0x	15x	5x	8x	8x	3x	8x	1x	8x	5x	6x

Table 4.14: Subset of parallel **speedup** results on **haplotyping instances, part 2 of 2**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

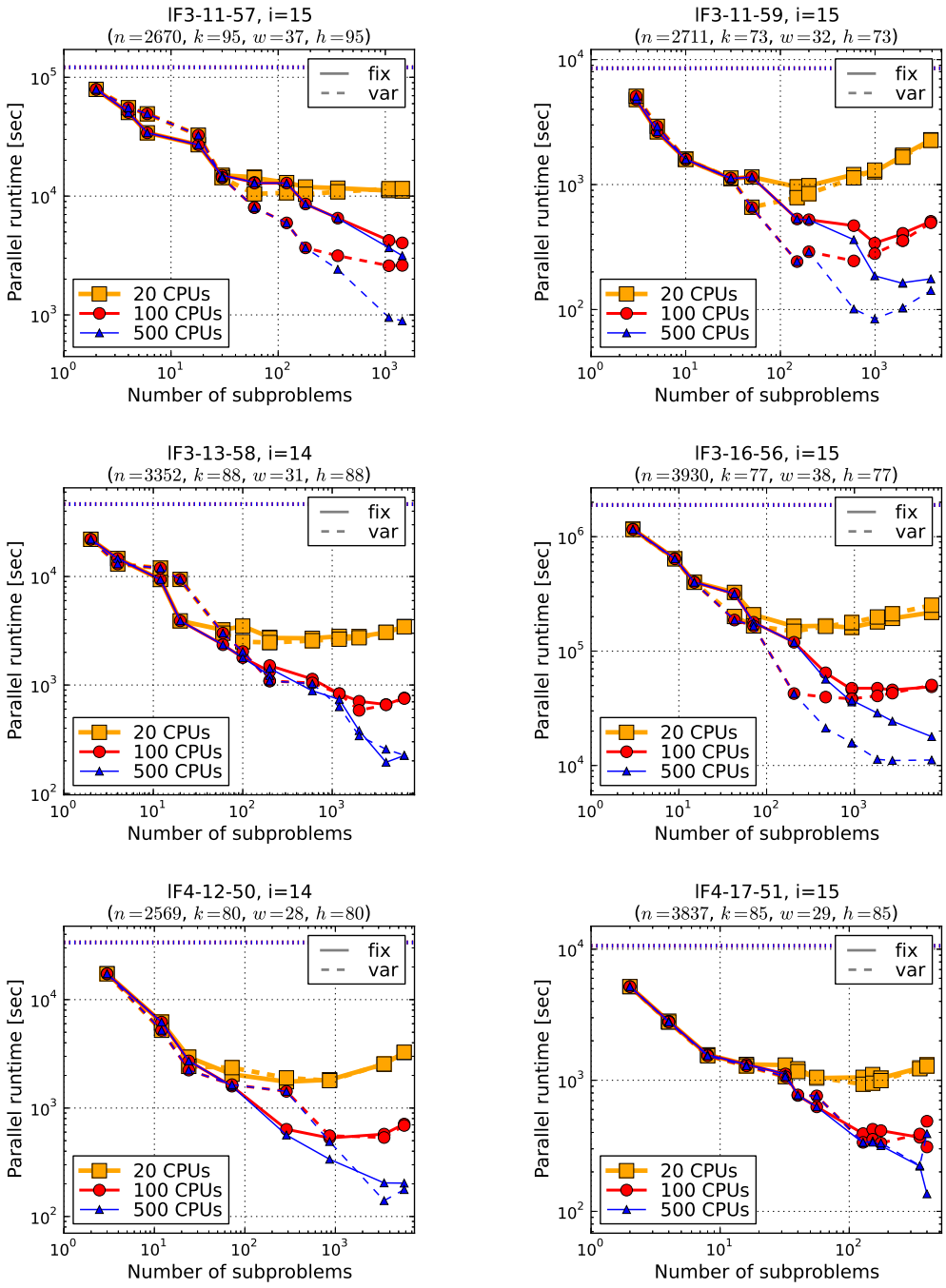


Figure 4.20: Parallel **runtime** plots for select **haplotyping** problems. Shown is the runtime using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). The instance's sequential solution time T_{seq} is indicated by the dashed horizontal line.

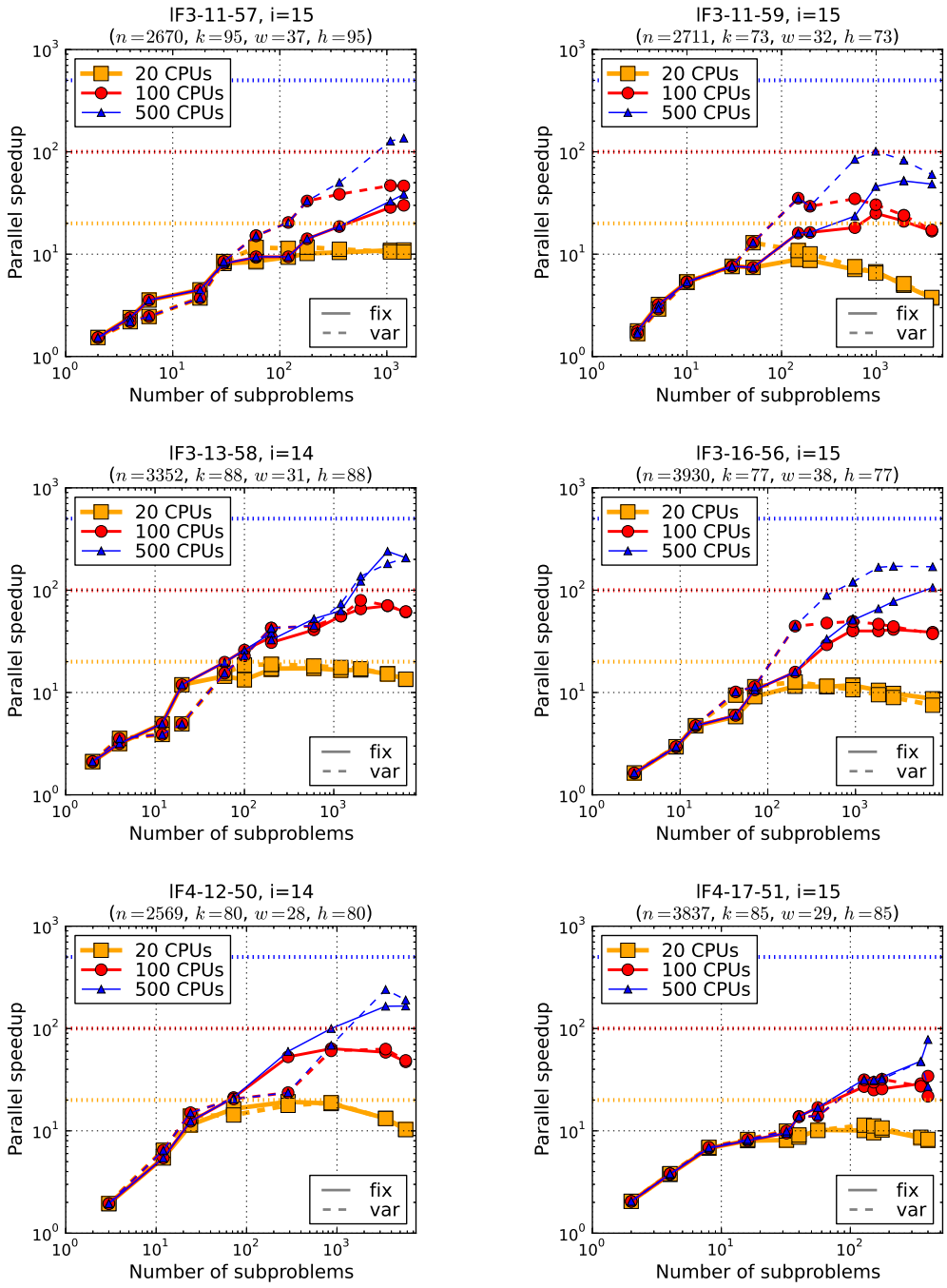


Figure 4.21: Parallel **speedup** plots for select **haplotyping** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.

$d = 11$). Finally, the best speedups with 500 CPUs are just over 200, for instance 207 for largeFam3-13-38 with $i = 14$ and $d = 13$, or 206 and 241 for largeFam4-12-50 at $d = 7$ with $i = 13, 14$, respectively.

Overall, however, speedups are somewhat lower than what we saw for linkage instances in Section 4.6.4.1, in particular with 500 CPUs. We see two main reasons for these weaker results. First, in many cases the instances with low parallel performance are relatively easy and have short sequential runtimes T_{seq}. LargeFam3-13-58 with $i = 18$ or largeFam3-11-59 with $i = 15, 16$, for instance, take only between 1 and 2 ½ hours sequentially – thus attempting to run with 500 CPUs would put subproblem complexity at under well one minute, at which point overhead from the grid system and the effect of centralized preprocessing in the master host have a substantial impact (cf. also Amdahl's Law, Section 4.2.5). Secondly, in several other cases with weaker parallel results, we note that the number of parallel subproblems is relatively low even for the higher cutoff depths we experimented with (for instance, 2496 subproblems for largeFam3-15-53, $i = 18$ at $d = 13$). As we've seen in previous analysis, the parallel scheme tends to work best if the number of subproblems is about a factor of 10 larger the number of CPUs, which is not the case for a number of instances in Tables 4.11 through 4.14, including largeFam3-15-53. The next paragraph investigates this aspect more broadly.

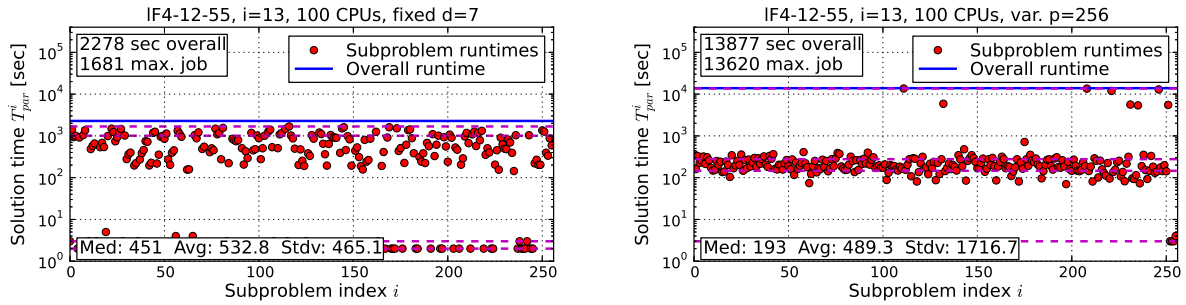
Number of Subproblems vs. CPU Count. Regarding the number of subproblems p in relation to the number of CPUs, earlier analysis (Sect. 4.6.3.4, e.g.) suggested a trade-off where p is about 10 times the CPU count, matching a "rule of thumb" reported by other researchers [105]. As a reminder, the choice of p needs to to balance two conflicting aspects: first, to allow sufficient parallel granularity to compensate for some longer-running subproblems with multiple smaller ones, which suggests a higher cutoff depth; second, to avoid introducing unnecessary overhead from the sequential preprocessing, grid processing delays, and parallel redundancies (cf. Section 4.5), which advocates for a lower cutoff depth.

Figures 4.20 and 4.21 provide a convenient illustration of this trade-off. It is most notably for 20 and 100 CPUs, where performance begins to deteriorate once the number of subproblems grows too large, inducing disproportionate overhead. We can furthermore affirm this general behavior and, more specifically, the particular suggested rule of thumb by consulting Tables 4.11 through 4.14. Consider, for instance, largeFam3-16-56 for both $i = 15, 16$. In Table 4.14 the best speedups for 20, 100, and 500 CPUs are obtained with $p = 205$, 934, and 7582 subproblems, respectively. Similarly, for largeFam4-12-50, we get the highest speedups for 20, 100, and 500 CPUs, respectively, at $p = 288$, 804, and 3456 subproblems ($p = 804$ at $d = 6$ not included in Table 4.14 – cf. Appendix Table B.14, page 320).

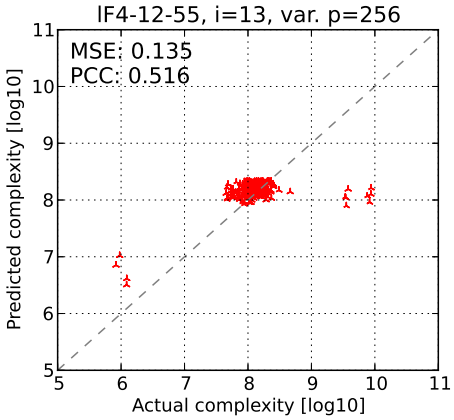
Fixed-depth vs. Variable-depth. As before, we observe different outcomes in the comparison of fixed-depth and variable-depth parallelization. In a few cases, especially for lower values of d, a fixed-depth cutoff yields better runtimes and higher speedups – we will discuss largeFam4-12-55 as an example of this. However, we note that in most cases, particularly with many CPUs, variable-depth parallelization significantly outperforms the fixed-depth scheme; here a good example is largeFam3-16-56, also analyzed below.

LargeFam4-12-55. Parallel runtime and speedup for instance largeFam4-12-55 ($n = 2926$ variables and induced width $w = 28$) is detailed in Table 4.12 and 4.14. For i-bound 13 (sequential runtime over 29 hours) the fixed-depth scheme is overall significantly faster than the variable-depth for all but the highest depth value and variable-depth equivalent – 2278 seconds vs. 13877 seconds, $d = 7$ with 100 CPUs, for instance. Just like above, we look at the last row (with "unlimited" CPUs) to recognize that the performance of the variable-depth scheme is dominated by the largest subproblem in many of these cases – 13628 seconds for $d = 7$, for instance.

This is confirmed by the detailed subproblem results plotted in Figure 4.22: the scatter plot in Figure 4.22b clearly shows a handful of vastly underestimated outliers that dominate the parallel execution in Figure 4.22a (right). However, in Figure 4.22a we also see that, with



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 7$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 256$.

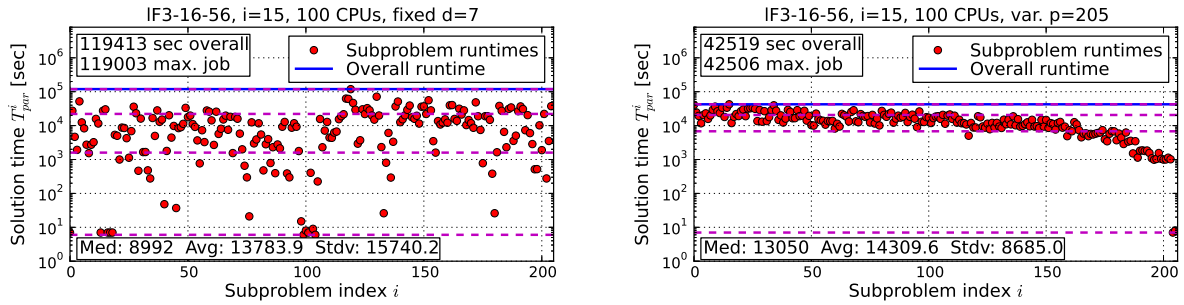


(b) Scatter plot of actual vs. predicted subproblem complexity for variable-depth parallel run with $p = 256$ subproblems.

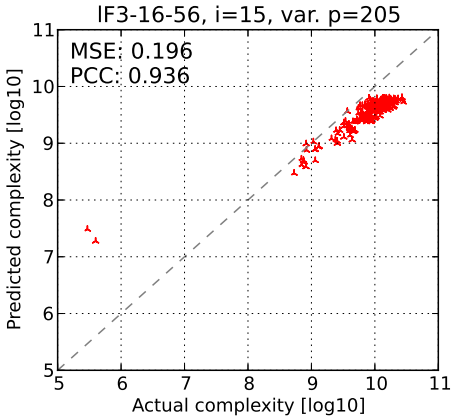
Figure 4.22: Performance details of fixed-depth and variable-depth parallel scheme on largeFam4-12-55 instance ($i = 13$) with $d = 7$ and corresponding $p = 256$ subproblems, respectively.

the exception of these outliers, the vast majority of subproblems in the variable-depth run is more balanced than the fixed-depth run on the left of Figure 4.22a.

LargeFam3-15-56. One of several more positive examples is largeFam3-16-56 ($n = 3930$, $w = 38$), which has a sequential runtime of almost 22 days ($i = 15$). Figure 4.23a details the subproblem runtimes for fixed-depth parallelization with cutoff $d = 7$ (left) and the variable-depth scheme with the corresponding subproblem count $p = 205$ (right). With 100 CPUs, the variable-depth parallel scheme finished in just under 12 hours, while the fixed-depth scheme takes over 33 hours. Figure 4.23b shows that in this case the subproblem complexity estimates of the variable-depth run are a lot more accurate and, crucially, don't exhibit any



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 7$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 205$.



(b) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run with $p = 205$ subproblems.

Figure 4.23: Performance details of fixed-depth and variable-depth parallel scheme on largeFam3-16-56 instance ($i = 15$) with $d = 7$ and corresponding $p = 205$ subproblems, respectively.

outliers. This is also the reason why the subproblem runtimes on the right of Figure 4.23a appear to be a lot more balanced (note that the variance is half that of fixed-depth on the left).

Overall, we definitely see impressive improvements by the variable-depth scheme over the fixed-depth one, which is captured by the summary rows at the bottom of Tables 4.13 and 4.14. For $d = 7$, for instance, the variable-depth performance is at least 10% and 50% better in 51 and 28 cases, while fixed-depth leads only in 11 and 8 cases, respectively. Similarly, at depth $d = 11$, we see 10% and 50% better performance by variable-depth parallelization in 74 and 30 cases, respectively, with only 5 and 1 for fixed-depth.

Summary. In summary, results of our parallel scheme on haplotyping instances were mixed, with some evidence of good performance, but also a number of less convincing cases. These weaker results, however, prove helpful in understanding the parallel scheme better. We have noted that large-scale parallelism (i.e. hundreds of CPUs) is rather wasteful if the problem instance is relatively simple and only takes a few hours sequentially. In the same context, we have confirmed the "rule of thumb" regarding the choice of subproblem count, which should be several times the number of CPUs – our experiments (and report from others) suggest that this factor should be around 10. With respect to fixed-depth versus variable-depth parallelization, we have reconfirmed the dependence of the variable-depth scheme on accurate subproblem complexity estimates, where significant outliers can reduce the overall performance considerably. That being said, in case of haplotyping instances we have found a very impressive advantage for variable-depth parallelization, especially for higher subproblem counts that facilitate a large number of CPUs.

4.6.4.3 Overall Analysis of Protein Side-Chain Prediction Problems

Table 4.15 shows runtime results of running the two parallel schemes, as before with 20, 100, 500, and "unlimited" CPUs, on side-chain prediction instances while Table 4.16 lists the corresponding speedups. Also as before, Figures 4.24 and 4.25 plot runtime and speedup, respectively, for a subset of problem instances.

Impact of Large Domain Size. Side-chain prediction problems are special because of their very large variable domains, with a maximum domain size of $k = 81$. As a consequence of this the mini-bucket heuristic can only be compiled with a relatively low i-bound of 3 – $i = 4$ and higher would quickly exceed the 2 GB memory limit in our experiments. Secondly, even relatively low parallel cutoff depths d already yield a significant number of subproblems, which limits the experiments we can conduct in practice (cf. Section 4.6.2 and Table 4.4). For instance, pdb1hd2 has 3777 subproblems with $d = 2$, but setting $d = 3$ would yield over 66

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var
				(p=9)		*(p=81)*		*(p=511)*							
pdb1a6m			20	109236	109236	96811	**29713**	51456	**8839**						
n=124	3	198326	100	109236	109236	96811	**29713**	51456	**8839**						
k=81			500	109236	109236	96811	**29713**	51456	**8839**						
w=15			∞	*109236*	*109236*	*96811*	***29713***	*51456*	***8839***						
h=34															
				(p=9)		*(p=54)*		*(p=784)*		*(p=15081)*					
pdb1duw			20	261878	261878	185941	**144524**	148576	**34290**	66316	**40886**				
n=241	3	627106	100	261878	261878	185941	**144524**	148576	**13294**	51977	**8190**				
k=81			500	261878	261878	185941	**144524**	148576	**13294**	51977	**3998**				
w=9			∞	*261878*	*261878*	*185941*	***144524***	*148576*	***13294***	*51977*	***3998***				
h=32															
				(p=66)		*(p=1046)*		*(p=11321)*							
pdb1e5k			20	20322	20322	**6876**	7630	10653	10712						
n=154	3	112654	100	20322	20322	5994	**2024**	2299	2153						
k=81			500	20322	20322	5994	**2024**	2034	**783**						
w=12			∞	*20322*	*20322*	*5994*	***2024***	*2034*	***783***						
h=43															
				(p=81)		*(p=6534)*									
pdb1f9i			20	27995	27995	23496	**21220**								
n=103	3	68804	100	27995	27995	8752	**4249**								
k=81			500	27995	27995	8752	**2587**								
w=10			∞	*27995*	*27995*	*8752*	***2587***								
h=24															
				(p=27)		*(p=118)*		*(p=5281)*							
pdb1ft5			20	39764	39764	29982	**8248**	8302	8469						
n=172	3	81118	100	39764	39764	29982	**8248**	4478	**1715**						
k=81			500	39764	39764	29982	**8248**	4478	**802**						
w=14			∞	*39764*	*39764*	*29982*	***8248***	*4478*	***802***						
h=33															
				(p=79)		*(p=3777)*									
pdb1hd2			20	58967	58967	15426	**6470**								
n=126	3	101550	100	58967	58967	15426	**2275**								
k=81			500	58967	58967	15426	**2275**								
w=12			∞	*58967*	*58967*	*15426*	***2275***								
h=27															
				(p=9)		*(p=42)*		*(p=293)*		*(p=654)*		*(p=1588)*		*(p=2597)*	
pdb1huw			20	478239	478239	477785	**402748**	467632	**41642**	462167	**36305**	446255	**31297**	367056	**31646**
n=152	3	545249	100	478239	478239	477785	**402748**	467632	**41642**	462167	**34051**	446255	**18483**	367056	**12750**
k=81			500	478239	478239	477785	**402748**	467632	**41642**	462167	**34051**	446255	**18483**	367056	**12750**
w=15			∞	*478239*	*478239*	*477785*	***402748***	*467632*	***41642***	*462167*	***34051***	*446255*	***18483***	*367056*	***12750***
h=43															
				(p=27)		*(p=215)*		*(p=752)*		*(p=3241)*					
pdb1kao			20	252879	252879	213134	**64745**	145683	**32176**	63832	**18172**				
n=148	3	716795	100	252879	252879	213134	**55749**	145683	**25927**	63832	**6126**				
k=81			500	252879	252879	213134	**55749**	145683	**25927**	63832	**6126**				
w=15			∞	*252879*	*252879*	*213134*	***55749***	*145683*	***25927***	*63832*	***6126***				
h=41															
				(p=6)		*(p=48)*		*(p=336)*		*(p=3812)*					
pdb1nfp			20	328980	328980	292628	**73365**	194064	**38568**	101131	**42531**				
n=204	3	354720	100	328980	328980	292628	**73365**	194064	**27180**	101131	**8752**				
k=81			500	328980	328980	292628	**73365**	194064	**27180**	101131	**6768**				
w=18			∞	*328980*	*328980*	*292628*	***73365***	*194064*	***27180***	*101131*	***6768***				
h=38															
				(p=8)		*(p=109)*		*(p=908)*		*(p=1336)*					
pdb1rss			20	392069	392069	110936	**57202**	37791	**31715**	33834	**24441**				
n=115	3	378579	100	392069	392069	110904	**57202**	37654	**25702**	33706	**24266**				
k=81			500	392069	392069	110904	**57202**	37625	**25702**	33689	**24266**				
w=12			∞	*392069*	*392069*	*110904*	***57202***	*37625*	***25702***	*33689*	***24266***				
h=35															
				(p=27)		*(p=1842)*		*(p=67760)*							
pdb1vhh			20	233763	233763	**52565**	231663	92605	**69612**						
n=133	3	944633	100	233763	233763	**22967**	231663	20965	**13970**						
k=81			500	233763	233763	**21751**	231663	15746	**3921**						
w=14			∞	*233763*	*233763*	***21751***	*231663*	*14878*	***3133***						
h=35															
Better by 10%				0x	0x	5x	39x	0x	33x	0x	20x	0x	4x	0x	4x
Better by 50%				0x	0x	4x	30x	0x	28x	0x	16x	0x	4x	0x	4x

Table 4.15: Subset of parallel **runtime** results on **side-chain prediction instances**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var
				(p=9)		*(p=81)*		*(p=511)*							
pdb1a6m			20	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
n=124 k=81 w=15 h=34	3	198326	100	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
			500	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
			∞	*1.82*	*1.82*	*2.05*	***6.67***	*3.85*	***22.44***						
				(p=9)		*(p=54)*		*(p=784)*		*(p=15081)*					
pdb1duw			20	2.39	2.39	3.37	**4.34**	4.22	**18.29**	9.46	**15.34**				
n=241 k=81 w=9 h=32	3	627106	100	2.39	2.39	3.37	**4.34**	4.22	**47.17**	12.07	**76.57**				
			500	2.39	2.39	3.37	**4.34**	4.22	**47.17**	12.07	**156.85**				
			∞	*2.39*	*2.39*	*3.37*	***4.34***	*4.22*	***47.17***	*12.07*	***156.85***				
				(p=66)		*(p=1046)*		*(p=11321)*							
pdb1e5k			20	5.54	5.54	**16.38**	14.76	10.57	10.52						
n=154 k=81 w=12 h=43	3	112654	100	5.54	5.54	18.79	**55.66**	49.00	52.32						
			500	5.54	5.54	18.79	**55.66**	55.39	**143.87**						
			∞	*5.54*	*5.54*	*18.79*	***55.66***	*55.39*	***143.87***						
				(p=81)		*(p=6534)*									
pdb1f9i			20	2.46	2.46	2.93	**3.24**								
n=103 k=81 w=10 h=24	3	68804	100	2.46	2.46	7.86	**16.19**								
			500	2.46	2.46	7.86	**26.60**								
			∞	*2.46*	*2.46*	*7.86*	***26.60***								
				(p=27)		*(p=118)*		*(p=5281)*							
pdb1ft5			20	2.04	2.04	2.71	**9.83**	9.77	9.58						
n=172 k=81 w=14 h=33	3	81118	100	2.04	2.04	2.71	**9.83**	18.11	**47.30**						
			500	2.04	2.04	2.71	**9.83**	18.11	**101.14**						
			∞	*2.04*	*2.04*	*2.71*	***9.83***	*18.11*	***101.14***						
				(p=79)		*(p=3777)*									
pdb1hd2			20	1.72	1.72	6.58	**15.70**								
n=126 k=81 w=12 h=27	3	101550	100	1.72	1.72	6.58	**44.64**								
			500	1.72	1.72	6.58	**44.64**								
			∞	*1.72*	*1.72*	*6.58*	***44.64***								
				(p=9)		*(p=42)*		*(p=293)*		*(p=654)*		*(p=1588)*		*(p=2597)*	
pdb1huw			20	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**15.02**	1.22	**17.42**	1.49	**17.23**
n=152 k=81 w=15 h=43	3	545249	100	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**16.01**	1.22	**29.50**	1.49	**42.76**
			500	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**16.01**	1.22	**29.50**	1.49	**42.76**
			∞	*1.14*	*1.14*	*1.14*	***1.35***	*1.17*	***13.09***	*1.18*	***16.01***	*1.22*	***29.50***	*1.49*	***42.76***
				(p=27)		*(p=215)*		*(p=752)*		*(p=32241)*					
pdb1kao			20	2.83	2.83	3.36	**11.07**	4.92	**22.28**	11.23	**39.45**				
n=148 k=81 w=15 h=41	3	716795	100	2.83	2.83	3.36	**12.86**	4.92	**27.65**	11.23	**117.01**				
			500	2.83	2.83	3.36	**12.86**	4.92	**27.65**	11.23	**117.01**				
			∞	*2.83*	*2.83*	*3.36*	***12.86***	*4.92*	***27.65***	*11.23*	***117.01***				
				(p=6)		*(p=48)*		*(p=336)*		*(p=3812)*					
pdb1nfp			20	1.08	1.08	1.21	**4.84**	1.83	**9.20**	3.51	**8.34**				
n=204 k=81 w=18 h=38	3	354720	100	1.08	1.08	1.21	**4.84**	1.83	**13.05**	3.51	**40.53**				
			500	1.08	1.08	1.21	**4.84**	1.83	**13.05**	3.51	**52.41**				
			∞	*1.08*	*1.08*	*1.21*	***4.84***	*1.83*	***13.05***	*3.51*	***52.41***				
				(p=8)		*(p=109)*		*(p=908)*		*(p=1336)*					
pdb1rss			20	0.97	0.97	3.41	**6.62**	10.02	**11.94**	11.19	**15.49**				
n=115 k=81 w=12 h=35	3	378579	100	0.97	0.97	3.41	**6.62**	10.05	**14.73**	11.23	**15.60**				
			500	0.97	0.97	3.41	**6.62**	10.06	**14.73**	11.24	**15.60**				
			∞	*0.97*	*0.97*	*3.41*	***6.62***	*10.06*	***14.73***	*11.24*	***15.60***				
				(p=27)		*(p=1842)*		*(p=67760)*							
pdb1vhh			20	4.04	4.04	**17.97**	4.08	10.20	**13.57**						
n=133 k=81 w=14 h=35	3	944633	100	4.04	4.04	**41.13**	4.08	45.06	**67.62**						
			500	4.04	4.04	**43.43**	4.08	59.99	**240.92**						
			∞	*4.04*	*4.04*	***43.43***	*4.08*	*63.49*	***301.51***						
Better by 10%				0x	0x	5x	39x	0x	33x	0x	20x	0x	4x	0x	4x
Better by 50%				0x	0x	4x	30x	0x	28x	0x	16x	0x	4x	0x	4x

Table 4.16: Subset of parallel **speedup** results on **side-chain prediction instances**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

thousand subproblems, which leads to thrashing with our current implementation (because of the massive number of temporary files involved, for instance). pdb1vhh is similar, with over 67 thousand subproblems for $d = 3$ – several attempts and a laborious, manual tweaking of the Condor system actually enabled us to produce a successful parallel run at this scale with the current system. Adapting to these situations more generally would require a major re-engineering of our parallel scheme.

General Performance. We observe mixed, somewhat inconsistent results in Tables 4.15 and 4.16. For 20 CPUs we still see acceptable behavior, with speedups of 17 or 18 on a number of instances (pdb1duw at $d = 3$, e.g.); with 100 CPUs the best speedup is 76 (pdb1duw), but we also see a number of values in the 40s (pdb1ft5, pdb1hd2, and pdb1duw, for instance). For 500 CPUs, however, results are less convincing: the best speedup of 241 is achieved on pdb1vhh, in the experiment over 60 thousand subproblems that required extensive manual tweaking to run. Many other instances that did not receive this special treatment see notably worse performance – e.g., pdb1nfp only reaches speedup 52 with 500 CPUs.

In this context, however, we observe that in almost all cases, fixed-depth or variable-depth and with 100 or 500 CPUs in particular, the parallel runtime is dominated by the longest-running subproblem, as implied by the last row (in italic) of each table field. This issue goes to back to the large variable domains in this class of problems. More specifically, here it is often the case that complex subproblems split very unevenly, in the most extreme case yielding one similarly complex one and many very simple ones. The variable-depth parallelization scheme is designed to address this in principle, but due to the large variable domain size it reaches its subproblem target count p before it can establish a sufficiently balanced parallelization frontier. The following example illustrates this.

Pdb1nfp. Consider the problem instance pdb1nfp with sequential runtime over 4 days. At depth $d = 4$ the fixed-depth scheme manages a very bad speedup of 3.51 across all CPU

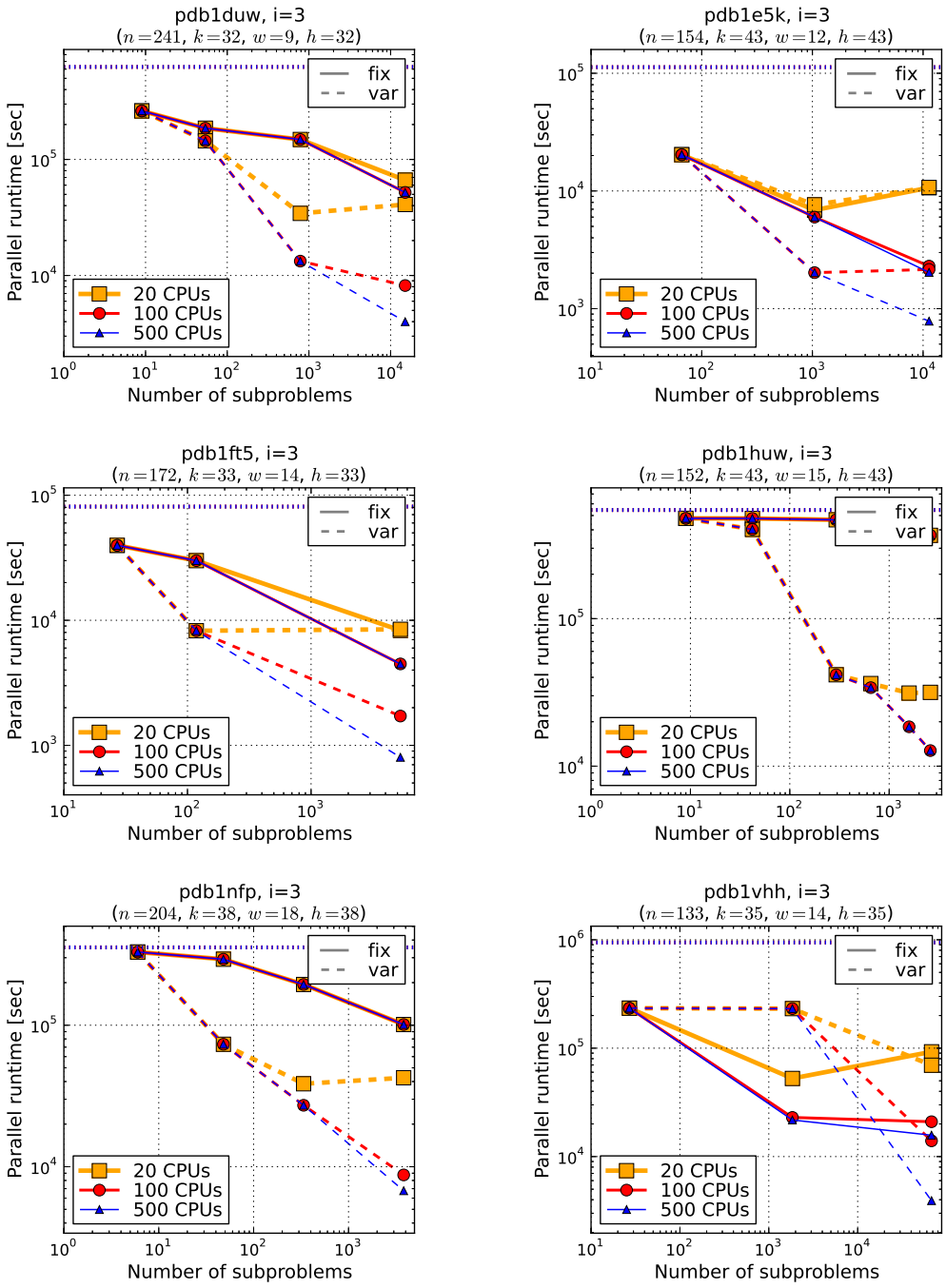


Figure 4.24: Parallel **runtime** plots for select **side-chain prediction** problems. Shown is the runtime using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). The instance's sequential solution time T_{seq} is indicated by the dashed horizontal line.

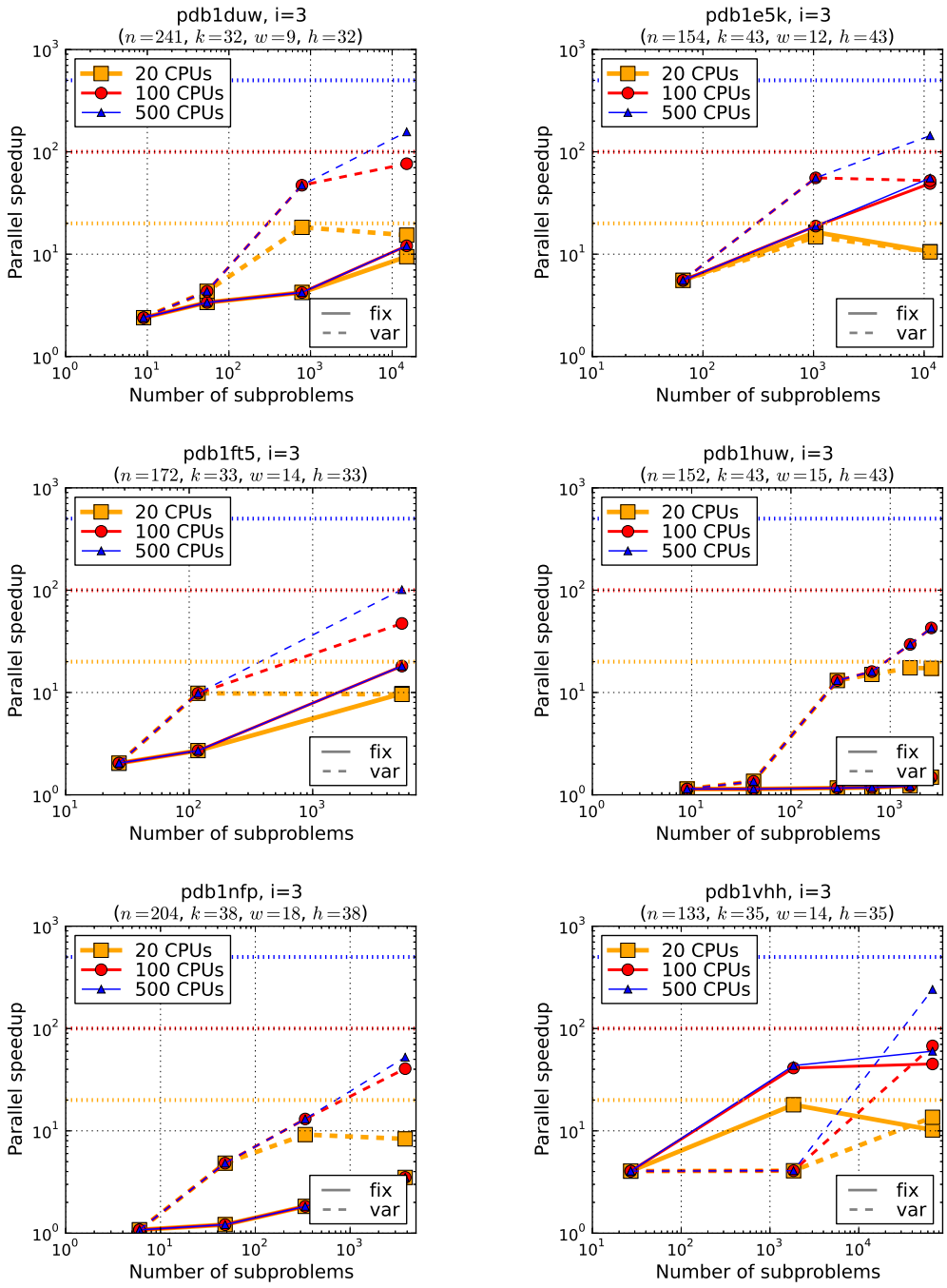
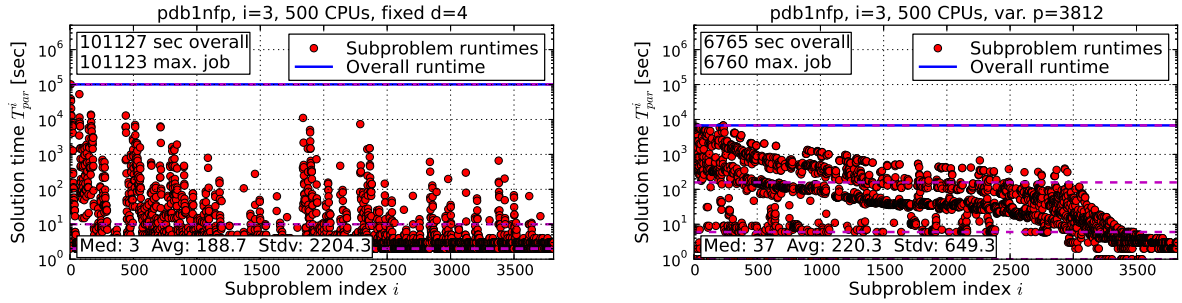
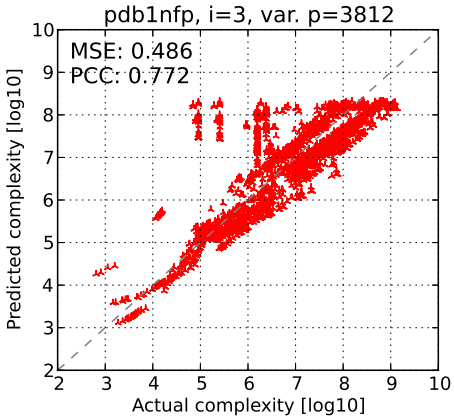


Figure 4.25: Parallel **speedup** plots for select **side-chain prediction** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 4$ using 500 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 3812$.



(b) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run with $p = 3812$ subproblems.

Figure 4.26: Performance details of fixed-depth and variable-depth parallel scheme on pdb1nfp instance ($i = 3$) with $d = 4$ and corresponding $p = 3812$ subproblems, respectively.

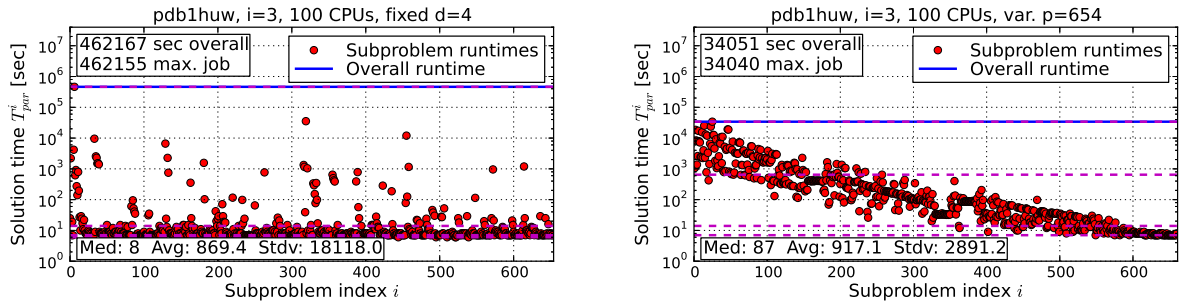
counts, while variable-depth parallelization with the corresponding $p = 3812$ subproblems allows a speedup of 52.41 with 500 CPUs – in both cases performance is bottlenecked by the longest-running subproblem, as indicated by the identical result of "unlimited" CPUs. For a more detailed analysis, Figure 4.26a shows the runtimes of individual subproblems for the fixed-depth (left) and variable-depth (right) run. In both cases we see a significant number of very small subproblems – the dashed 80 percentile line is just above 10 and 100 seconds runtime, respectively. As in earlier experiments we also see that the variable-depth scheme produces a significantly more balanced parallelization frontier (even though it is still fairly unbalanced in itself). Most interestingly, however, we note that the variable-depth scheme actually correctly identifies the hardest subproblems (also see the scatter plot in

Figure 4.26b) – i.e., these subproblems would be broken apart next if we were to allow a larger parallelization frontier and thereby more subproblem splits. And in fact, the results of our manually facilitated run on pdb1vhh with over 60 thousand subproblems (see above) suggest that this is indeed the case.

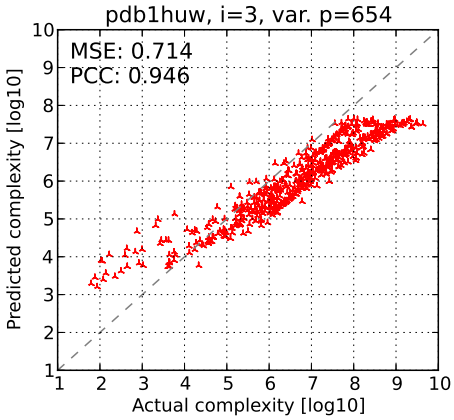
Fixed-depth vs. Variable Depth. Looking at the parallel performance in Table 4.15 and the exemplary plots in Figures 4.24 and 4.25, we see a very strong advantage for the variable-depth in almost all cases. Specifically, due to the search space unbalancedness discussed above the fixed-depth scheme does very poorly on instances from this problem class – even when running with 500 CPUs, the parallel speedup rarely exceeds 20. Within the constraints of our setup, as outlined above, the variable-depth scheme performs a lot better. The following example illustrates.

Pdb1huw. Figure 4.27a shows detailed plots of the subproblem runtimes of pdb1huw when running fixed-depth and variable-depth parallelization with $d = 4$ and the corresponding $p = 654$. We observe that the variable-depth scheme is able to reduce the size of the hardest subproblem, and thereby the overall running time, by a factor of more than 13, from 462155 to 34040 seconds. The variable-depth scheme also yields a drastically reduced standard deviation in subproblem runtime, its parallel resource utilization is about 18%, vs. just over 1% for fixed-depth (cf. Table B.21). Figure 4.27b plots the results of subproblem complexity estimation from the variable-depth run; we see a very high degree of correlation between actual and predicted complexity (correlation coefficient 0.95).

Overall, the last two summary rows of Table 4.15 exhibit superior performance by the variable-depth scheme in the vast majority of instances. The only exception is pdb1vhh at $d = 2$, where the variable-depth scheme is dominated by a few complex subproblems, which were, as in earlier examples, underestimated by the complexity prediction. For $d = 3$ and above, however, the variable-depth scheme is superior by a large margin for all instances.



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 4$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 654$.



(b) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run with $p = 654$ subproblems.

Figure 4.27: Performance details of fixed-depth and variable-depth parallel scheme on pdb1huw instance ($i = 3$) with $d = 4$ and corresponding $p = 654$ subproblems, respectively.

At $d = 3$, for instance, it outperforms the fixed-depth variant by 10% and 50% in 33 and 28 cases, respectively (out of 36 at that level).

Summary. The problem class of side-chain prediction instances turned out to be very tough for our parallel search scheme for two reasons. First, the problem search spaces are typically very unbalanced. Second, the large variable domain size means that even with accurate subproblem complexity estimates a very large parallelization frontier would be required to reach these complex subproblems and break them apart – something that is not easily achieved with our current implementation for technical reasons. Consequently, parallel speedup performance is relatively poor, especially for a high number of CPUs. The exception

was in one parallel run where we manually facilitated a very large parallelization frontier, yielding very good speedup results. In general, however, we saw the variable-depth scheme, aided by reliable subproblem complexity estimates, perform very admirably within these given adverse constraints. In particular, it typically far surpasses the fixed-depth variant across all choices of cutoff depth and corresponding subproblem count.

4.6.4.4 Overall Analysis of Grid Network Problems

Table 4.17 shows parallel runtimes of parallel AOBB on grid network instances for a subset of fixed depths d and corresponding variable-depth subproblem count and Table 4.18 has the corresponding parallel speedup results. In contrast to the other problem classes discussed above, instances of this type have strictly binary domains, so we ran slightly higher cutoff depths to obtain a suitable number of subproblems. In addition to the full tables, Figures 4.28 and 4.29 plot the parallel runtime and speedup, respectively, for a subset of instances.

General performance. Results in Tables 4.17 and 4.18 span a range of outcomes, although generally not as good as results observed, for instance, for grid and haplotyping problems in Sections 4.6.4.1 and 4.6.4.2. Using 20 CPUs, the best speedup we obtain is around 16 for 75-15-1 at depth $d = 13$. Many other instances, however, don't exceed a speedup of 10 with 20 CPUs, which is somewhat disappointing. Similarly, most instances' speedups with 100 CPUs and barely exceed 50 (75-25-1 is the best again with 58). Finally, results remain fairly weak with 500 CPUs. The best speedup of 165 for 75-25-7 at $d = 15$ is more of an exception, as the speedup for most other instances remains well below 100. We can identify a number of reasons these disappointing results:

- First, in spite of running with higher cutoff depths, the subproblem count is still relatively low in some cases. In particular for higher CPU counts the number of subproblems does often not meet the "rule of thumb" we described earlier, according

instance	i	T_{seq}	#cpu	Cutoff depth d = 5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var	15 fix	15 var
				(p=16)		*(p=64)*		*(p=192)*		*(p=192)*		*(p=768)*		*(p=2112)*	
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	11642	11562	5929	5776	**5425**	6458	**5548**	6458	4948	5313	4965	4970
			100	11642	11562	4944	**3940**	3831	3880	3870	3880	**1968**	2810	1643	**1343**
			500	11642	11562	4944	**3940**	3831	3638	3870	3638	**1791**	2810	1643	**1343**
			∞	*11642*	*11562*	*4944*	***3940***	*3831*	*3638*	*3870*	*3638*	***1791***	*2810*	*1643*	***1343***
			(p=8)		*(p=32)*		*(p=96)*		*(p=192)*		*(p=576)*		*(p=1584)*		
	14	15402	20	3630	3737	1564	1433	**1479**	1671	**1296**	1726	**962**	1160	1014	999
			100	3630	3737	1413	**1227**	1211	1253	**987**	1264	**651**	861	652	**325**
			500	3630	3737	1413	**1227**	1211	1253	**987**	1227	**651**	839	649	**226**
			∞	*3630*	*3737*	*1413*	***1227***	*1211*	*1253*	***987***	*1227*	***651***	*839*	*649*	***219***
			(p=6)		*(p=24)*		*(p=48)*		*(p=144)*		*(p=576)*		*(p=1536)*		
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	40537	40453	31861	31583	21832	21646	**13118**	14594	**15733**	17700	19745	19197
			100	40537	40453	31861	31583	21832	21646	9671	10481	**3841**	7952	6938	**5964**
			500	40537	40453	31861	31583	21832	21646	9671	10481	**3841**	7952	4791	4793
			∞	*40537*	*40453*	*31861*	*31583*	*21832*	*21646*	*9671*	*10481*	***3841***	*7952*	*4600*	*4628*
			(p=6)		*(p=24)*		*(p=48)*		*(p=144)*		*(p=576)*		*(p=1536)*		
	15	33656	20	13049	13139	5443	5486	4807	4403	4467	4662	4683	4977	5255	5115
			100	13049	13139	5443	5486	3505	**3103**	**1448**	1720	1116	1220	1579	**1134**
			500	13049	13139	5443	5486	3505	**3103**	**1448**	1720	602	567	850	**601**
			∞	*13049*	*13139*	*5443*	*5486*	*3505*	***3103***	***1448***	*1720*	*602*	*567*	*718*	***601***
			(p=24)		*(p=72)*		*(p=216)*		*(p=504)*		*(p=2016)*		*(p=3360)*		
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	66859	**47596**	30640	28910	26206	28019	28611	29296	30050	30460	30582	31325
			100	66859	**47596**	29325	28910	10908	11472	7645	**6944**	7254	6699	6726	6722
			500	66859	**47596**	29325	28910	10908	11472	6568	6351	3077	**2080**	2062	**1805**
			∞	*66859*	***47596***	*29325*	*28910*	*10908*	*11472*	*6568*	*6351*	*3077*	***1465***	*2002*	***1242***
			(p=24)		*(p=72)*		*(p=216)*		*(p=504)*		*(p=2014)*		*(p=3325)*		
	18	21694	20	**6467**	10548	**2890**	10958	**2618**	6484	**2718**	6508	3204	3244	3455	3492
			100	**6467**	10548	**2890**	10603	**1236**	6384	**752**	6164	1049	1130	908	838
			500	**6467**	10548	**2890**	10603	**1236**	6384	**752**	6164	**713**	804	702	**309**
			∞	***6467***	*10548*	***2890***	*10603*	***1236***	*6384*	***752***	*6164*	***692***	*804*	*702*	***220***
			(p=16)		*(p=32)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1280)*		
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	8736	8789	5770	5738	5244	5724	**5058**	5874	5299	5252	5061	5177
			100	8736	8789	5480	5738	3223	2986	**2465**	2787	1624	1602	1352	1324
			500	8736	8789	5480	5738	3223	2986	**2465**	2762	**1314**	1534	1029	**913**
			∞	*8736*	*8789*	*5480*	*5738*	*3223*	*2986*	***2465***	*2762*	***1314***	*1534*	*1029*	***913***
			(p=16)		*(p=48)*		*(p=160)*		*(p=480)*		*(p=960)*		*(p=1216)*		
	18	26855	20	**4676**	5301	2382	2334	**2344**	2628	2444	2351	2788	2553	**2522**	2915
			100	**4676**	5301	2053	2148	1039	1021	838	827	941	989	**619**	1138
			500	**4676**	5301	2053	2148	1039	**901**	705	702	710	748	**487**	780
			∞	***4676***	*5301*	*2053*	*2148*	*1039*	***901***	*705*	*702*	*696*	*748*	***487***	*731*
			(p=24)		*(p=96)*		*(p=288)*		*(p=640)*		*(p=1280)*		*(p=3840)*		
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	**4412**	6656	**3092**	3808	**2878**	3210	3092	3378	3231	3230	3560	3587
			100	**4412**	6656	**1437**	2317	1078	1153	**687**	971	908	936	763	781
			500	**4412**	6656	**1437**	2317	997	997	**392**	631	490	511	**216**	266
			∞	***4412***	*6656*	***1437***	*2317*	*997*	*997*	***392***	*631*	*434*	*445*	***155***	*199*
			(p=24)		*(p=96)*		*(p=288)*		*(p=640)*		*(p=1280)*		*(p=3840)*		
	20	8053	20	**1520**	2106	**1182**	1345	1258	1338	1502	1521	1800	1803	2968	2974
			100	**1520**	2106	**568**	977	**382**	555	**339**	411	386	411	634	639
			500	**1520**	2106	**568**	977	**326**	517	**159**	304	**123**	157	175	178
			∞	***1520***	*2106*	***568***	*977*	***326***	*517*	***159***	*304*	***102***	*141*	***88***	*106*
			(p=32)		*(p=128)*		*(p=128)*		*(p=384)*		*(p=1152)*		*(p=4608)*		
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	55099	**48079**	43719	40998	42048	40998	35060	38131	35096	36198	35958	36103
			100	47702	48079	24974	25174	24827	25174	13222	12451	11125	10299	**7550**	8627
			500	47702	48079	24974	25174	24827	25174	11941	11978	7401	7283	3551	3758
			∞	*47702*	*48079*	*24974*	*25174*	*24827*	*25174*	*11941*	*11978*	*7298*	*7283*	***3240***	*3613*
			(p=32)		*(p=128)*		*(p=128)*		*(p=384)*		*(p=1152)*		*(p=4608)*		
	12	64758	20	**19267**	31349	16756	**15218**	16417	15218	**13599**	16307	13668	14763	14336	14423
			100	**19267**	29308	10632	10537	10545	10537	**6731**	10472	**5464**	6589	**3328**	4761
			500	**19267**	29308	10632	10537	10545	10537	**6298**	10472	**4097**	6044	**1925**	4086
			∞	***19267***	*29308*	*10632*	*10537*	*10545*	*10537*	***6298***	*10472*	***4097***	*6044*	***1925***	*4086*
			(p=24)		*(p=96)*		*(p=240)*		*(p=960)*		*(p=3840)*		*(p=7680)*		
	16	59609	20	13651	13862	9075	**8135**	7040	7012	7981	8498	9006	9027	9427	9637
			100	13651	13862	7829	7779	**4194**	5802	3517	**2756**	2113	2281	1992	2094
			500	13651	13862	7829	7779	**4194**	5802	3231	**2331**	1923	**1044**	972	886
			∞	*13651*	*13862*	*7829*	*7779*	***4194***	*5802*	*3231*	***2331***	*1923*	***1000***	*868*	*879*
			(p=24)		*(p=96)*		*(p=240)*		*(p=960)*		*(p=3840)*		*(p=7680)*		
75-26-9 n=675 k=2 w=39 h=124	18	66533	20	19383	**11794**	11634	11731	8726	8184	8786	8801	9414	9549	10178	10276
			100	19383	**11794**	10856	11731	**4912**	8184	4078	**2375**	2370	**2007**	2186	2140
			500	19383	**11794**	10856	11731	**4912**	8184	3787	**1424**	1633	**853**	1014	**596**
			∞	*19383*	***11794***	*10856*	*11731*	***4912***	*8184*	*3787*	***1424***	*1553*	***780***	*935*	***304***
			(p=24)		*(p=96)*		*(p=240)*		*(p=640)*		*(p=2560)*		*(p=5120)*		
	20	5708	20	2366	**1597**	1472	**1199**	885	909	1164	1148	2146	2163	3416	3446
			100	2366	**1597**	1316	1199	588	**332**	356	**278**	464	460	723	731
			500	2366	**1597**	1316	1199	588	**332**	225	**166**	159	**127**	184	191
			∞	*2366*	***1597***	*1316*	*1199*	*588*	***332***	*225*	***166***	*114*	***73***	*104*	***77***
Better by 10%				20x	13x	12x	9x	17x	8x	27x	10x	20x	9x	12x	19x
Better by 50%				12x	4x	10x	0x	9x	3x	11x	3x	5x	6x	5x	8x

Table 4.17: Subset of parallel **runtime** results on **grid instances**. Each entry lists, from top to bottom, the runtime with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	Cutoff depth d = 5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var	15 fix	15 var
				(p=16)		*(p=64)*		*(p=192)*		*(p=192)*		*(p=768)*		*(p=2112)*	
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	6.69	6.74	13.15	13.49	**14.37**	12.07	**14.05**	12.07	15.75	14.67	15.70	15.68
			100	6.69	6.74	15.76	**19.78**	20.34	20.09	20.14	20.09	**39.60**	27.74	47.44	**58.03**
			500	6.69	6.74	15.76	**19.78**	20.34	21.42	20.14	21.42	**43.52**	27.74	47.44	**58.03**
			∞	*6.69*	*6.74*	*15.76*	***19.78***	*20.34*	*21.42*	*20.14*	*21.42*	***43.52***	*27.74*	*47.44*	***58.03***
				(p=8)		*(p=32)*		*(p=96)*		*(p=192)*		*(p=576)*		*(p=1584)*	
	14	15402	20	4.24	4.12	9.85	10.75	**10.41**	9.22	**11.88**	8.92	**16.01**	13.28	15.19	15.42
			100	4.24	4.12	10.90	**12.55**	12.72	12.29	**15.60**	12.19	**23.66**	17.89	23.62	**47.39**
			500	4.24	4.12	10.90	**12.55**	12.72	12.29	**15.60**	12.55	**23.66**	18.36	23.73	**68.15**
			∞	*4.24*	*4.12*	*10.90*	***12.55***	*12.72*	*12.29*	***15.60***	*12.55*	***23.66***	*18.36*	*23.73*	***70.33***
				(p=6)		*(p=24)*		*(p=48)*		*(p=144)*		*(p=576)*		*(p=1536)*	
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	2.57	2.57	3.27	3.29	4.77	4.81	**7.93**	7.13	**6.61**	5.88	5.27	5.42
			100	2.57	2.57	3.27	3.29	4.77	4.81	10.76	9.93	**27.09**	13.08	15.00	**17.44**
			500	2.57	2.57	3.27	3.29	4.77	4.81	10.76	9.93	**27.09**	13.08	21.72	21.71
			∞	*2.57*	*2.57*	*3.27*	*3.29*	*4.77*	*4.81*	*10.76*	*9.93*	***27.09***	*13.08*	*22.62*	*22.48*
				(p=6)		*(p=24)*		*(p=48)*		*(p=144)*		*(p=576)*		*(p=1536)*	
	15	33656	20	2.58	2.56	6.18	6.13	7.00	7.64	7.53	7.22	7.19	6.76	6.40	6.58
			100	2.58	2.56	6.18	6.13	9.60	**10.85**	**23.24**	19.57	30.16	27.59	21.31	**29.68**
			500	2.58	2.56	6.18	6.13	9.60	**10.85**	**23.24**	19.57	55.91	59.36	39.60	**56.00**
			∞	*2.58*	*2.56*	*6.18*	*6.13*	*9.60*	***10.85***	***23.24***	*19.57*	*55.91*	*59.36*	*46.87*	***56.00***
				(p=24)		*(p=72)*		*(p=216)*		*(p=504)*		*(p=2016)*		*(p=3360)*	
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	4.45	**6.25**	9.71	10.29	11.35	10.61	10.39	10.15	9.90	9.76	9.72	9.49
			100	4.45	**6.25**	10.14	10.29	27.26	25.92	38.90	**42.83**	40.99	44.39	44.21	44.24
			500	4.45	**6.25**	10.14	10.29	27.26	25.92	45.28	46.82	96.65	**142.97**	144.22	**164.75**
			∞	*4.45*	***6.25***	*10.14*	*10.29*	*27.26*	*25.92*	*45.28*	*46.82*	*96.65*	***202.99***	*148.54*	***239.43***
				(p=24)		*(p=72)*		*(p=216)*		*(p=504)*		*(p=2014)*		*(p=3325)*	
	18	21694	20	**3.35**	2.06	**7.51**	1.98	**8.29**	3.35	**7.98**	3.33	6.77	6.69	6.28	6.21
			100	**3.35**	2.06	**7.51**	2.05	**17.55**	3.40	**28.85**	3.52	20.68	19.20	23.89	25.89
			500	**3.35**	2.06	**7.51**	2.05	**17.55**	3.40	**28.85**	3.52	**30.43**	26.98	30.90	**70.21**
			∞	***3.35***	*2.06*	***7.51***	*2.05*	***17.55***	*3.40*	***28.85***	*3.52*	***31.35***	*26.98*	*30.90*	***98.61***
				(p=16)		*(p=32)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1280)*	
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	5.38	5.35	8.14	8.19	8.96	8.21	**9.29**	8.00	8.87	8.95	9.28	9.08
			100	5.38	5.35	8.57	8.19	14.58	15.74	**19.06**	16.86	28.93	29.33	34.75	35.49
			500	5.38	5.35	8.57	8.19	14.58	15.74	**19.06**	17.01	**35.76**	30.63	45.66	**51.46**
			∞	*5.38*	*5.35*	*8.57*	*8.19*	*14.58*	*15.74*	***19.06***	*17.01*	***35.76***	*30.63*	*45.66*	***51.46***
				(p=16)		*(p=48)*		*(p=160)*		*(p=480)*		*(p=960)*		*(p=1216)*	
	18	26855	20	**5.74**	5.07	11.27	11.51	**11.46**	10.22	10.99	11.42	9.63	10.52	**10.65**	9.21
			100	**5.74**	5.07	13.08	12.50	25.85	26.30	32.05	32.47	28.54	27.15	**43.38**	23.60
			500	**5.74**	5.07	13.08	12.50	25.85	**29.81**	38.09	38.25	37.82	35.90	**55.14**	34.43
			∞	***5.74***	*5.07*	*13.08*	*12.50*	*25.85*	***29.81***	*38.09*	*38.25*	*38.58*	*35.90*	***55.14***	*36.74*
				(p=24)		*(p=96)*		*(p=288)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	**5.73**	3.80	**8.17**	6.64	**8.78**	7.87	8.17	7.48	7.82	7.82	7.10	7.05
			100	**5.73**	3.80	**17.59**	10.91	23.45	21.92	**36.79**	26.03	27.83	27.00	33.12	32.36
			500	**5.73**	3.80	**17.59**	10.91	25.35	25.35	**64.47**	40.05	51.58	49.46	**117.01**	95.02
			∞	***5.73***	*3.80*	***17.59***	*10.91*	*25.35*	*25.35*	***64.47***	*40.05*	*58.24*	*56.80*	***163.06***	*127.01*
				(p=24)		*(p=96)*		*(p=288)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
	20	8053	20	**5.30**	3.82	**6.81**	5.99	6.40	6.02	5.36	5.29	4.47	4.47	2.71	2.71
			100	**5.30**	3.82	**14.18**	8.24	**21.08**	14.51	**23.76**	19.59	20.86	19.59	12.70	12.60
			500	**5.30**	3.82	**14.18**	8.24	**24.70**	15.58	**50.65**	26.49	**65.47**	51.29	46.02	45.24
			∞	***5.30***	*3.82*	***14.18***	*8.24*	***24.70***	*15.58*	***50.65***	*26.49*	***78.95***	*57.11*	***91.51***	*75.97*
				(p=32)		*(p=128)*		*(p=128)*		*(p=384)*		*(p=1152)*		*(p=4608)*	
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	3.62	**4.15**	4.56	4.87	4.74	4.87	5.69	5.23	5.68	5.51	5.55	5.52
			100	4.18	4.15	7.99	7.92	8.03	7.92	15.09	16.02	17.93	19.37	**26.42**	23.12
			500	4.18	4.15	7.99	7.92	8.03	7.92	16.70	16.65	26.95	27.39	56.17	53.08
			∞	*4.18*	*4.15*	*7.99*	*7.92*	*8.03*	*7.92*	*16.70*	*16.65*	*27.33*	*27.39*	***61.56***	*55.21*
				(p=32)		*(p=128)*		*(p=128)*		*(p=384)*		*(p=1152)*		*(p=4608)*	
	12	64758	20	**3.36**	2.07	3.86	**4.26**	3.94	4.26	**4.76**	3.97	4.74	4.39	4.52	4.49
			100	**3.36**	2.21	6.09	6.15	6.14	6.15	**9.62**	6.18	**11.85**	9.83	**19.46**	13.60
			500	**3.36**	2.21	6.09	6.15	6.14	6.15	**10.28**	6.18	**15.81**	10.71	**33.64**	15.85
			∞	***3.36***	*2.21*	*6.09*	*6.15*	*6.14*	*6.15*	***10.28***	*6.18*	***15.81***	*10.71*	***33.64***	*15.85*
				(p=24)		*(p=96)*		*(p=240)*		*(p=960)*		*(p=3840)*		*(p=7680)*	
	16	59609	20	4.37	4.30	6.57	**7.33**	8.47	8.50	7.47	7.01	6.62	6.60	6.32	6.19
			100	4.37	4.30	7.61	7.66	**14.21**	10.27	16.95	**21.63**	28.21	26.13	29.92	28.47
			500	4.37	4.30	7.61	7.66	**14.21**	10.27	18.45	**25.57**	31.00	**57.10**	61.33	67.28
			∞	*4.37*	*4.30*	*7.61*	*7.66*	***14.21***	*10.27*	*18.45*	***25.57***	*31.00*	***59.61***	*68.67*	*67.81*
				(p=24)		*(p=96)*		*(p=240)*		*(p=960)*		*(p=3840)*		*(p=7680)*	
75-26-9 n=675 k=2 w=39 h=124	18	66533	20	3.43	**5.64**	5.72	5.67	7.62	8.13	7.57	7.56	7.07	6.97	6.54	6.47
			100	3.43	**5.64**	6.13	5.67	**13.54**	8.13	16.32	**28.01**	28.07	**33.15**	30.44	31.09
			500	3.43	**5.64**	6.13	5.67	**13.54**	8.13	17.57	**46.72**	40.74	**78.00**	65.61	**111.63**
			∞	*3.43*	***5.64***	*6.13*	*5.67*	***13.54***	*8.13*	*17.57*	***46.72***	*42.84*	***85.30***	*71.16*	***218.86***
				(p=24)		*(p=96)*		*(p=240)*		*(p=640)*		*(p=2560)*		*(p=5120)*	
	20	5708	20	2.41	**3.57**	3.88	**4.76**	6.45	6.28	4.90	4.97	2.66	2.64	1.67	1.66
			100	2.41	**3.57**	4.34	4.76	9.71	**17.19**	16.03	**20.53**	12.30	12.41	7.89	7.81
			500	2.41	**3.57**	4.34	4.76	9.71	**17.19**	25.37	**34.39**	35.90	**44.94**	31.02	29.88
			∞	*2.41*	***3.57***	*4.34*	*4.76*	*9.71*	***17.19***	*25.37*	***34.39***	*50.07*	***78.19***	*54.88*	***74.13***
Better by 10%				20x	13x	12x	9x	17x	8x	27x	10x	20x	9x	12x	19x
Better by 50%				12x	4x	10x	0x	9x	3x	11x	3x	5x	6x	5x	8x

Table 4.18: Subset of parallel **speedup** results on **grid instances**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" parallel cores, with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold. The best value in each row is highlighted in gray.

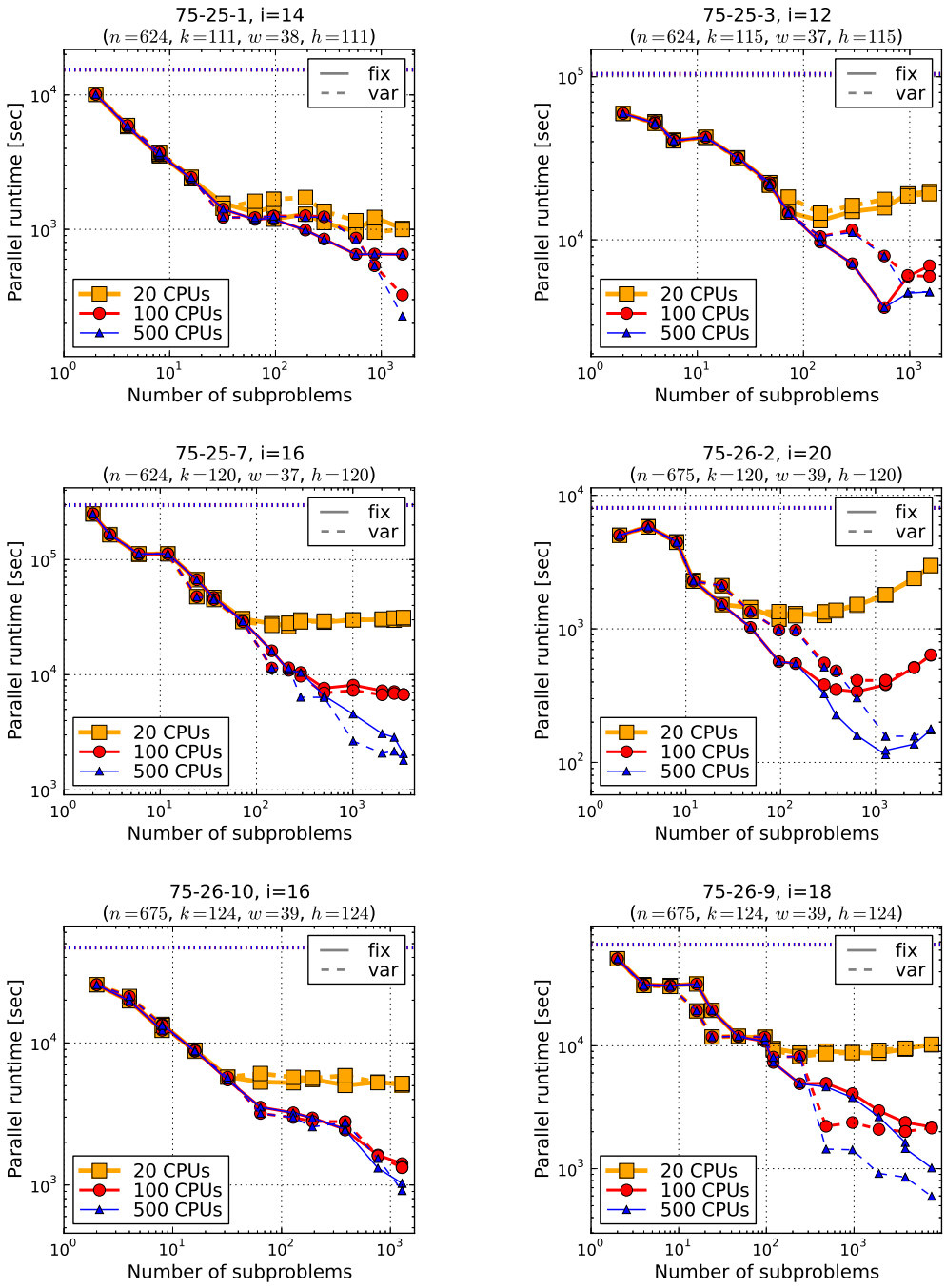


Figure 4.28: Parallel **runtime** plots for select **grid** problems. Shown is the runtime using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). The instance's sequential solution time T_{seq} is indicated by the dashed horizontal line.

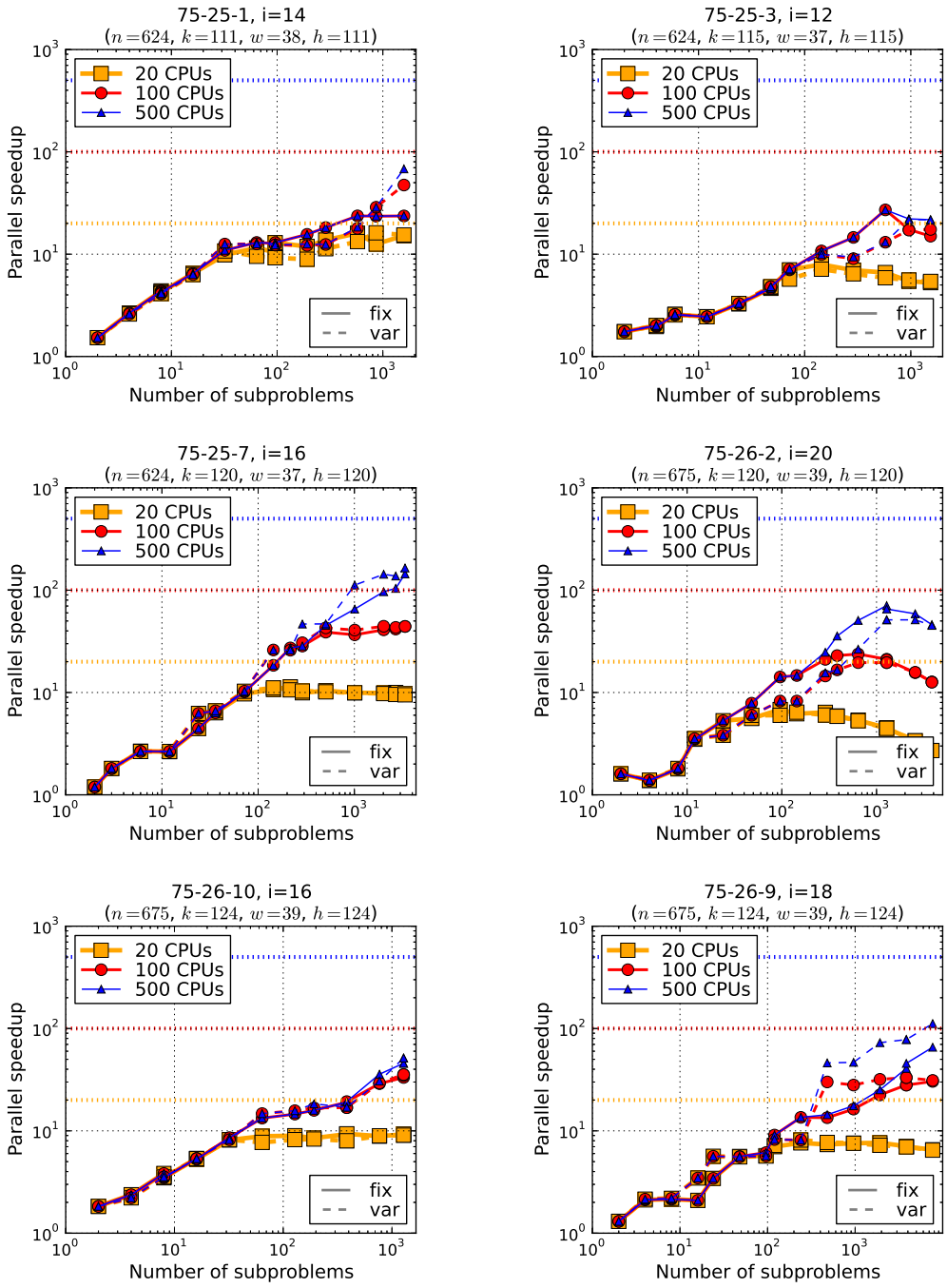


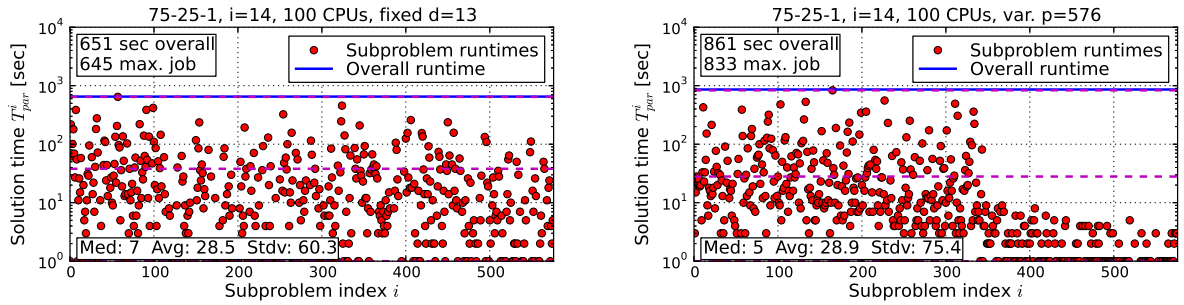
Figure 4.29: Parallel **speedup** plots for select **grid** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.

to which the number of subproblems should be about ten times the CPU count. As a consequence, the parallel performance is often still dominated by the longest-running subproblem, as indicated by comparing to the last, "unlimited" CPU row within each field – see for instance 75-26-10 at $d = 15$, where the 500 CPU speedup almost matches the "unlimited" one.

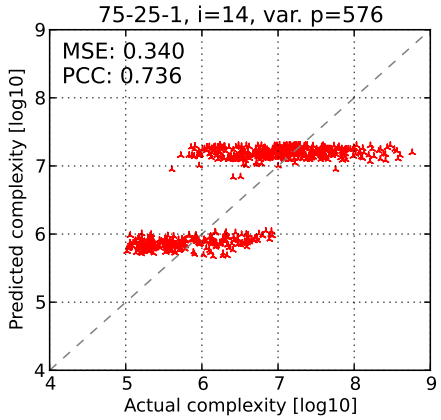
- The examples below will show that the obtained parallelization frontier is often not very balanced, even for variable-depth parallelization. As we demonstrate below, the reason for the relatively poor performance of the variable-depth scheme (in comparison to results on other problem classes) lies again in the quality of the subproblem complexity predictions, which turn out to be fairly inaccurate across most of the grid network instances.

- Most importantly, and in contrast to previously discussed problem classes, experiments on grid networks exhibit a relative large degree of parallel redundancies, as defined in Section 4.5. This section will mention this only briefly, however, with full analysis to follow in Section 4.6.6.

75-25-1. As a first example, we consider problem 75-25-1 with $i = 14$ – detailed subproblem statistics for fixed-depth $d = 13$ and corresponding variable-depth parallelization $p = 576$ are shown in Figure 4.30a. We note that both schemes produce a similarly scattered profile – in fact, the subproblems yielded by the variable-depth scheme have slightly larger standard deviation in subproblem runtime (75 vs. 60) as well as longer maximum subproblem (833 vs. 645 seconds) and therefore overall runtime (861 vs. 651 seconds). Figure 4.30b illustrates the results of the subproblem complexity prediction. The estimation results (vertical axis) can be seen as grouping subproblems into two groups – however, the actual range of complexities (horizontal axes) within each group is a lot more varied than what the estimation suggests. And in fact, the two groups designated by the prediction scheme actually overlap to a large extent, rendering the complexity estimates not very helpful.



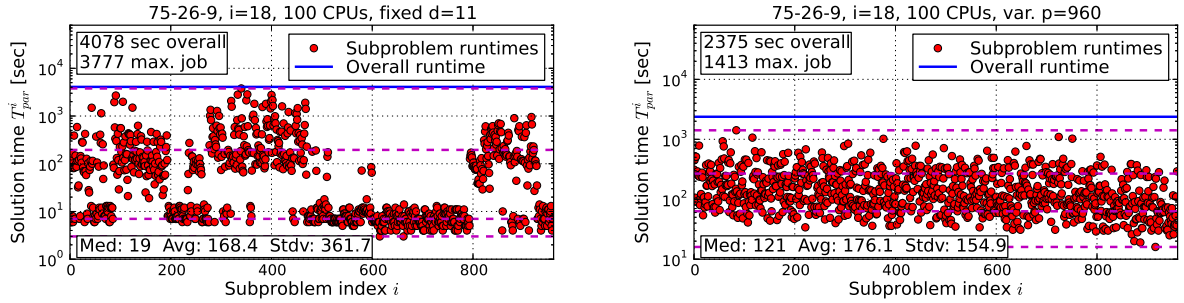
(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 13$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 576$.



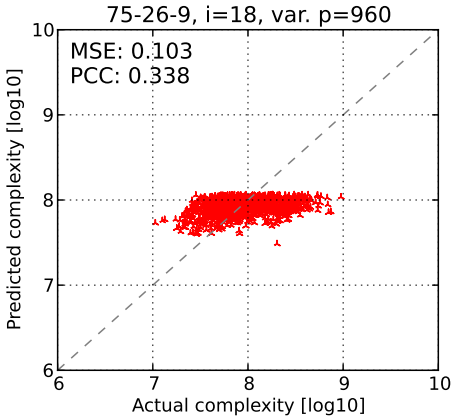
(b) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run with $p = 576$ subproblems.

Figure 4.30: Performance details of fixed-depth and variable-depth parallel scheme on 75-25-1 instance ($i = 14$) with $d = 13$ and corresponding $p = 576$ subproblems, respectively.

75-26-9. One of the few cases where the variable-depth scheme works better than the fixed-depth one is instance 75-26-9 ($i = 18$), as shown in Figure 4.31. As before Figure 4.31a shows subproblem statistics for fixed-depth (left) and variable-depth (right) parallelization. In this instance variable-depth performs a lot better, both in terms of maximum subproblem runtime (1413 vs. 3777 seconds) and overall runtime (2375 vs. 4078 seconds). Notably, the standard deviation over subproblem runtimes is a lot lower as well (155 vs. 362). Figure 4.31b shows the corresponding scatter plot of actual vs. predicted subproblem complexity, which has notably better prediction quality than what we observed for instance 75-25-1 in Figure 4.30b.



(a) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 11$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 960$.



(b) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run with $p = 960$ subproblems.

Figure 4.31: Performance details of fixed-depth and variable-depth parallel scheme on 75-26-9 instance ($i = 18$) with $d = 11$ and corresponding $p = 960$ subproblems, respectively.

Fixed-depth vs. Variable-depth. Given the exposition above, it should not be surprising that the variable-depth scheme does not hold a strong advantage over the fixed-depth variant, as it did for other problem classes. In fact, the latter has an edge over variable-depth performance overall. For instance, at $d = 13$ it is better by 10% and 50% in 20 and 5 cases, respectively, while variable-depth has the advantage in only 9 and 6 cases. Similar results hold for other cutoff depths, with the exception of $d = 15$, where the variable-depth scheme recovers and is superior on average, being better by 10% and 50% in 19 and 8 cases, respectively, versus 12 and 5 for fixed-depth.

Summary. Our results on grid network instances can only be described as sobering, with fairly low parallel speedups throughout. Variable-depth parallelization showed disappointing behavior, compared to previously discussed problem classes, even though it recovers somewhat for deep cutoffs and high subproblem counts. We have attributed this a combination of three factors: First, a relatively low number of subproblems, which is particularly disadvantageous to higher CPU counts. This goes hand in with the second point, poor load balancing. The variable-depth scheme in particular suffers from the often inaccurate subproblem complexity estimates. Lastly, we hinted at the presence of parallel redundancies, which put an inherent limit on the achievable parallel performance. This aspect will be analyzed on its own in Section 4.6.6.

4.6.5 Parallel Resource Utilization

This section will consider the parallel resource utilization, which we defined in Section 4.2.4 as the average processor utilization, relative to the longest-running processor. A value close to 1 (or 100%) indicates that all workers spent about the same time on computation, while a value close to 0 indicates that a majority of parallel cores sat idly throughout most of the overall parallel execution.

Table 4.19 shows a subset of parallel resource utilization values for those problem instances that were used as examples above – the complete set of results is available in Appendix B, Tables B.17 through B.22. Similar to previous tables, for each instance and depth d we give the resource utilization for 20, 100, and 500 CPUs (top to bottom) for fixed-depth parallelization (left) and the corresponding variable-depth run (right). Also as before, for each pair we mark one in bold if it is better by more than 10% (relative) than the other.

We note that the specific problem instances in Table 4.19 were chosen to illustrate certain performance characteristics above. In particular, our goal was generally to highlight one

instance	i	T_{seq}	#cpu	Cutoff depth d 2 fix	2 var	4 fix	4 var	6 fix	6 var	8 fix	8 var	10 fix	10 var	12 fix	12 var
				(p=4)		*(p=8)*		*(p=16)*		*(p=64)*		*(p=96)*		*(p=288)*	
75-25-1 n=624 k=2 w=38 h=111	14	15402	20	0.14	0.14	0.23	0.22	0.35	0.34	**0.63**	0.52	**0.71**	0.50	**0.76**	0.63
			100	0.03	0.03	0.05	0.04	0.07	0.07	0.14	0.14	0.14	0.13	**0.20**	0.14
			500	0.01	0.01	0.01	0.01	0.01	0.01	0.03	0.03	0.03	0.03	**0.04**	0.03
				(p=4)		*(p=16)*		*(p=48)*		*(p=120)*		*(p=480)*		*(p=1920)*	
75-26-9 n=675 k=2 w=39 h=124	18	66533	20	0.13	0.13	0.17	**0.31**	0.54	0.57	0.72	0.79	0.86	**0.96**	0.99	0.98
			100	0.03	0.03	0.03	**0.06**	0.11	0.11	0.19	0.18	0.32	**0.75**	0.58	**0.87**
			500	0.01	0.01	0.01	**0.01**	0.02	0.02	0.04	0.04	0.07	**0.23**	0.13	**0.40**
				(p=4)		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
lF3-15-59 n=3730 k=3 w=31 h=84	19	43307	20	0.13	0.13	0.36	0.39	0.59	**0.77**	0.91	1.00	1.00	1.00	1.00	1.00
			100	0.03	0.03	0.07	0.08	0.13	**0.20**	0.34	**0.70**	0.77	**0.98**	0.99	0.99
			500	0.01	0.01	0.01	0.02	0.03	**0.04**	0.07	**0.18**	0.21	**0.62**	0.90	0.96
				(p=3)		*(p=15)*		*(p=71)*		*(p=470)*		*(p=934)*		*(p=2707)*	
lF3-16-56 n=3930 k=3 w=38 h=77	15	1891710	20	0.08	0.08	0.26	0.27	0.61	**0.78**	0.91	0.97	0.98	0.99	0.99	1.00
			100	0.02	0.02	0.05	0.05	0.14	0.16	0.47	**0.81**	0.67	**0.92**	0.83	**0.99**
			500	0.00	0.00	0.01	0.01	0.03	0.03	0.11	**0.30**	0.17	**0.45**	0.31	**0.77**
				(p=4)		*(p=16)*		*(p=128)*		*(p=512)*		*(p=1024)*		*(p=1792)*	
lF4-12-55 n=2926 k=4 w=28 h=78	13	104837	20	0.19	0.19	**0.54**	0.37	**0.83**	0.39	**0.96**	0.42	**0.98**	0.73	0.99	1.00
			100	0.04	0.04	**0.11**	0.07	**0.44**	0.09	**0.76**	0.10	**0.86**	0.24	0.86	0.87
			500	0.01	0.01	**0.02**	0.01	**0.09**	0.02	**0.27**	0.02	**0.44**	0.05	0.41	0.44
				(p=42)		*(p=654)*		*(p=2597)*							
pdb1huw n=152 k=81 w=15 h=43	3	545249	20	0.06	0.06	0.06	**0.84**	0.07	**1.00**						
			100	0.01	0.01	0.01	**0.18**	0.01	**0.50**						
			500	0.00	0.00	0.00	**0.04**	0.00	**0.10**						
				(p=48)		*(p=3812)*									
pdb1nfp n=204 k=81 w=18 h=38	3	354720	20	0.07	**0.43**	0.36	**1.00**								
			100	0.01	**0.09**	0.07	**0.97**								
			500	0.00	**0.02**	0.01	**0.25**								
				(p=12)		*(p=144)*		*(p=1440)*		*(p=5752)*		*(p=11254)*			
ped19 n=793 k=5 w=25 h=98	16	375110	20	0.18	**0.29**	0.78	**0.89**	0.98	0.99	1.00	1.00	1.00	1.00		
			100	0.04	**0.06**	0.16	**0.24**	0.44	**0.78**	0.95	0.95	1.00	1.00		
			500	0.01	**0.01**	0.03	**0.05**	0.09	**0.22**	0.27	**0.46**	0.69	**0.87**		
				(p=4)		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
ped44 n=811 k=4 w=25 h=65	6	95830	20	0.18	0.18	0.57	0.58	0.90	0.92	0.97	0.99	0.99	0.99	1.00	1.00
			100	0.04	0.04	0.11	0.12	**0.52**	0.35	0.86	0.92	**0.95**	0.86	0.99	1.00
			500	0.01	0.01	0.02	0.02	**0.15**	0.07	**0.54**	0.36	**0.78**	0.43	**0.95**	0.64
				(p=4)		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
ped7 n=1068 k=4 w=32 h=90	6	118383	20	**0.17**	0.10	**0.52**	0.11	0.74	0.79	0.73	**0.98**	0.84	**0.97**	0.94	1.00
			100	**0.03**	0.02	**0.11**	0.02	**0.26**	0.17	0.29	**0.62**	0.37	**0.81**	0.59	**0.95**
			500	**0.01**	0.00	**0.02**	0.00	**0.05**	0.03	0.06	**0.14**	0.09	**0.23**	0.18	**0.34**
Better by 10%				3x	6x	6x	12x	7x	9x	5x	8x	6x	11x	4x	6x
Better by 50%				3x	6x	3x	11x	5x	7x	4x	7x	3x	6x	0x	4x

Table 4.19: Subset of parallel resource utilization results on example instances. Each entry lists, from top to bottom, the average utilization with 20, 100, and (simulated) 500 parallel cores with fixed-depth parallel cutoff on the left ("fix") and variable-depth on the right ("var"). If one scheme is better than the other by more than 10% (relative) its results is marked bold.

positive and one negative example of variable-depth performance performance, which is why the summary results in Table 4.19 are not representative of the full results, which are given in Appendix B.

General Observations. We observe that parallel resource utilization increases as the depth, and with it number of subproblems, grows. This is not surprising given the scheduling approach of the Condor system, which assigns jobs from the queue as workers complete their previous subproblem – a larger number of subproblems allows CPUs that finish early to

remain busy with another subproblem. We also observe that, obviously, a larger number of parallel CPUs requires a larger number of subproblems to approach full utilization of 100% (or close to it).

Utilization, Load Balancing, and Speedup. We can view the parallel resource utilization as an indicator of load balancing, where higher utilization implies better balanced parallel load. In this light we can also make a connection to the overall parallel performance, which is at least partially correlated as follows: as the number of subproblems grows, parallel speedup for a given number of CPUs increases with the parallel resource utilization, since the workload is distributed better across the parallel resources and we expect the overall runtime to decrease (i.e., speedup increases). Once the utilization is at or close to 1, increasing the number of subproblems beyond that level will not improve load balancing and the speedup with it, but it is likely to introduce additional distributed overhead that will hurt parallel runtime. In other words, high resource utilization is a necessary condition for good speedup, but not sufficient.

Overall resource utilization results (cf. Appendix B) are also in line with overall performance and speedup results as seen on the particular problem classes. Namely, the variable-depth scheme yields better resource utilization compared to fixed-depth parallelization for three out of the four problem classes, as we exemplify in the following:

Linkage Problems. For instance, at depth $d = 8$ and the corresponding subproblem count, on linkage instances the variable-depth scheme produces at least 10% better utilization in 35 cases compared to 15 for the fixed-depth scheme, with 15 better by at least 50% vs. 10 for fixed-depth (taken from Tables B.17 and B.18). This matches the results for parallel speedups on linkage problems, where at depth $d = 8$ variable-depth has a 10% and 50% advantage in 49 and 22 cases, respectively, while fixed-depth is better in 24 and 16 cases (cf. Tables 4.9 and 4.10).

Haplotyping Problems. On haplotyping problems, also at cutoff depth $d = 8$, the advantage for variable-depth parallelization is even more pronounced, with 40 cases better by at least 10% and 22 by at least 50% vs. 6 and 4 cases, respectively, for fixed-depth (taken from Tables B.19 and B.20). The respective percent advantages of the variable-depth scheme speedup on haplotyping problems can be found in 51 and 32 cases, respectively, with only 10 and 6 such outcomes for fixed-depth (cf. Tables 4.13 and 4.14).

Side-chain Prediction Problems. As seen with parallel runtimes above, variable-depth parallelization does vastly better on side-chain prediction instances, as well. In fact, for depth $d = 4$ and the corresponding subproblem count, its resource utilization is better then fixed-depth by at least 10% for all 18 cases, by at least 50% in 15 cases, and (not shown in Table B.21) by at least 500% in 9 cases. The corresponding case counts of the parallel speedup results in Table 4.16 are 20 and 16 better by 10% and 50% respectively, for the variable-depth runs, with zero results in favor fixed-depth.

Grid Network Problems. Finally, just as before, results are rather mixed for grid instances, where for most depths the fixed-depth scheme in fact yields similar or slightly better resource utilization numbers. For instance, at depth $d = 10$ it is better by at least 10% in 15 cases (compared to 11 cases for variable-depth) and by at least 15% in 5 cases (compared to 3). Again we relate this to the parallel speedup results (cf. Table 4.18), where fixed-depth had an advantage by 10% and 50% in 20 and 8 cases, respectively, with 9 and 5 in favor of variable-depth.

Summary. Parallel resource utilization is a secondary performance metric that can serve to assess the level of load balancing. As such, we have demonstrated that it is partially correlated to the overall parallel performance and speedup. Namely, high resource utilization is necessary to achieve good speedups, since it captures efficient load balancing across parallel resources. However, it is not a guarantee for good speedups, since further increasing

the subproblem count leaves high resource utilization in tact, but most likely introduces additional, detrimental overhead to the parallel performance.

4.6.6 Parallel Redundancies

In this section we investigate the issue of parallel redundancies, as discussed and analyzed in detail in Section 4.5. Recall that these potential redundancies stem from the conditioning of subproblems in the parallelization process together with the fact that communication between worker hosts is not possible in our parallel model. In particular, optimal solutions to earlier subproblems, that could have facilitated stronger pruning in sequential AOBB, will not be available to guide the pruning in the parallel execution, as laid out in Section 4.5.1. Secondly, some degree of caching of context-unifiable subproblems is lost across subproblems – Section 4.5.2 provided detailed analysis and examples.

In Section 4.5.2 we also derived an expression $SS_{par}(d)$ (Equation 4.3) that captures the size of the underlying parallel search space as a function of the cutoff depth d which constitutes an upper bound on the number of node expansions by parallel AOBB. Note that the value of $SS_{par}(d)$ can be computed ahead of time, as it depends only on a problem's structural parameters.

In line with the analysis of Section 4.5, here we limit ourselves to fixed-depth parallelization. This simplifies the presentation of results, but any of our findings are straightforward to apply to the variable-depth scheme as well.

4.6.6.1 Tightness of Parallel Search Space Bound SS_{par}

To evaluate the practical impact of the aforementioned redundancies we record the sum of node expansions by parallel AOBB across all subproblems for a given cutoff depth d and

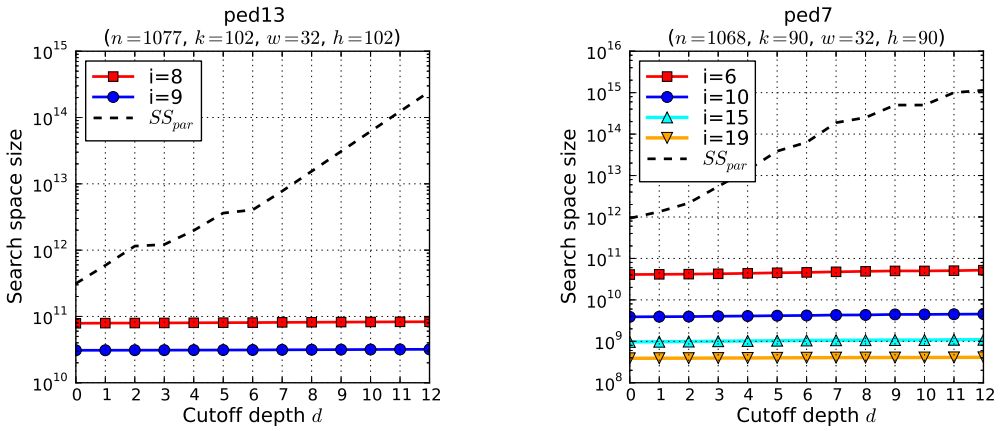


Figure 4.32: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **pedigree instances**.

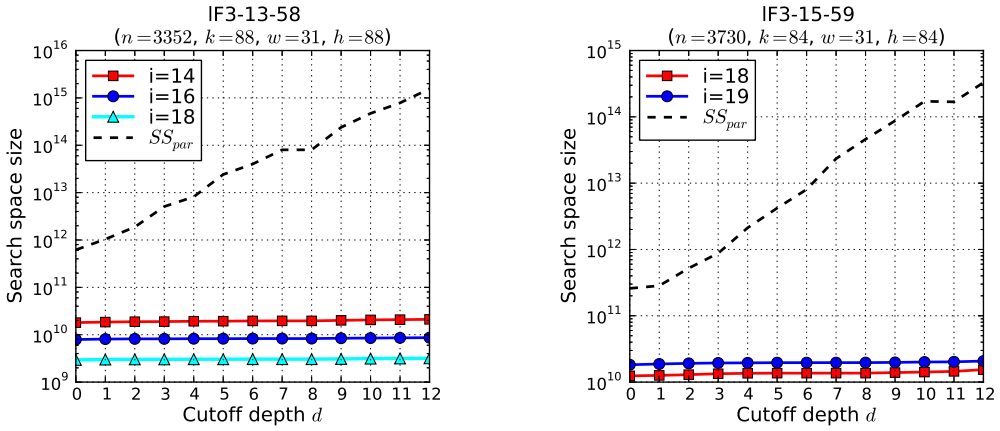


Figure 4.33: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **haplotyping instances**.

denote this measure $N_{par}(d)$. We also compute the respective underlying search space sizes $SS_{par}(d)$ as referenced above. Comparing the sequence of $N_{par}(d)$ and $SS_{par}(d)$ for increasing d will then give us an idea of the impact of redundancies in theory and practice. We note again that $N_{par}(0)$ and $SS_{par}(0)$ actually correspond to the number of node expansions by sequential AOBB and the non-parallel state space bound discussed in Chapter 3 (Section 3.2.1, Eq. 3.1), respectively.

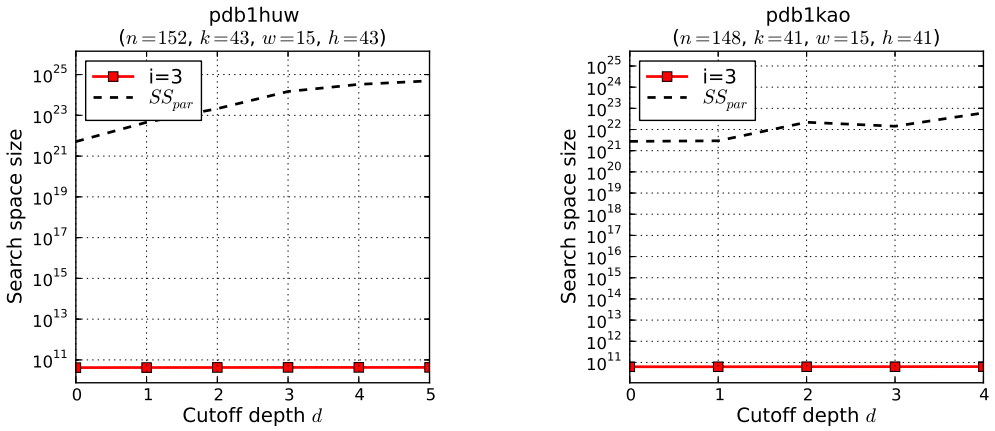


Figure 4.34: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **side-chain prediction instances**.

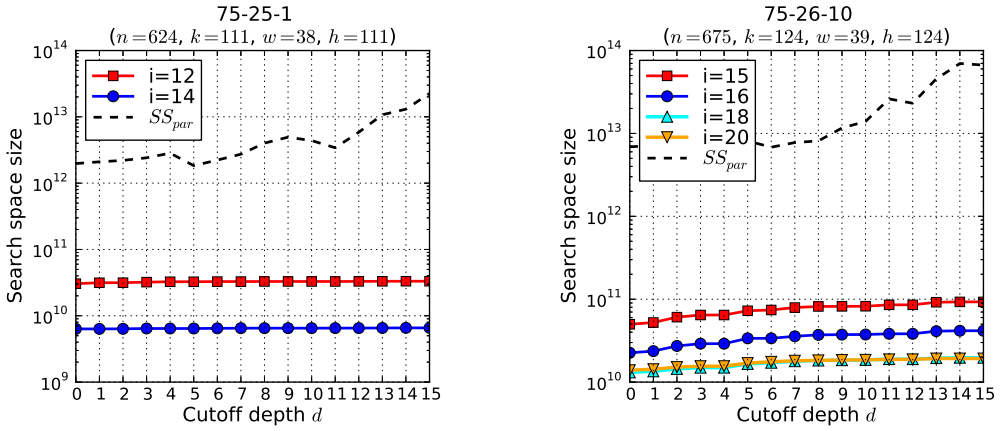


Figure 4.35: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **grid network instances**.

To that end Figures 4.32 through 4.35 plot the comparison of $SS_{par}(d)$ and $N_{par}(d)$ for two instances each of linkage, haplotyping, side-chain prediction, and grid network instances, respectively (results for other instances are very similar). Each plot shows $SS_{par}(d)$ with a dashed line as well as one or more solid line plots of $N_{par}(d)$ for varying mini-bucket i-bound (as indicated by the plot legend).

Upper Bound $SS_{par}(d)$. A number of observations can be made regarding $SS_{par}(d)$ across all instances in Figures 4.32 through 4.35. First, we can again confirm one of the central

premises of Chapter 3, namely that for sequential AOBB (corresponding to $d = 0$) the state space bound is loose by several orders of magnitude – this is most extreme for side-chain prediction instances in Figure 4.34, where the difference is roughly ten orders of magnitude.

Secondly, we observe that for $d > 0$ the size of the underlying parallel search space $SS_{par}(d)$ does indeed generally grow exponentially – note the logarithmic vertical scale. This signifies the marked impact that the loss of caching across subproblem instances has on the underlying parallel search space. Recall from the analysis in Section 4.5.2 that we expect the value of $SS_{par}(d)$ to decrease again eventually, since $SS_{par}(h) = SS_{par}(0)$, where h is the height of the guiding pseudo tree – however, we don't see this for the cutoff depths we consider here, which are relatively low compared to the height of the guiding pseudo trees.

Behavior of $N_{par}(d)$. In contrast, the actual number of explored nodes $N_{par}(d)$ grows far slower than exponentially, if at all, and the upper bound $SS_{par}(d)$, i.e., the size of the underlying search space, becomes exceedingly lose for bounding the explored search space. In fact, on the logarithmic scale of the plots the increase in node expansions $N_{par}(d)$ is in many cases not clearly discernible. The most notable exception is the grid instance 75-26-10 (Figure 4.35), where a growth of $N_{par}(d)$ is visible on the log scale, albeit still slower than the upper bound (Section 4.6.6.2 will investigate this in more depth). All in all, we take these results as a confirmation that the pruning power of AOBB with the mini-bucket heuristic is able to largely compensate for the fast-growing underlying parallel search space.

Impact of Mini-bucket i-bound. As noted, Figures 4.32 through 4.35 include results for various mini-bucket i-bounds used with parallel AOBB, letting us compare parallel performance from this angle as well. We know that higher i-bounds typically yield a more accurate heuristic function, so it is not surprising to see this manifested in fewer node expansions across subproblems for parallel AOBB as well. The most prominent example is pedigree7 (Figure 4.32, right), where the $i = 19$ expands approximately two orders of magnitude fewer nodes

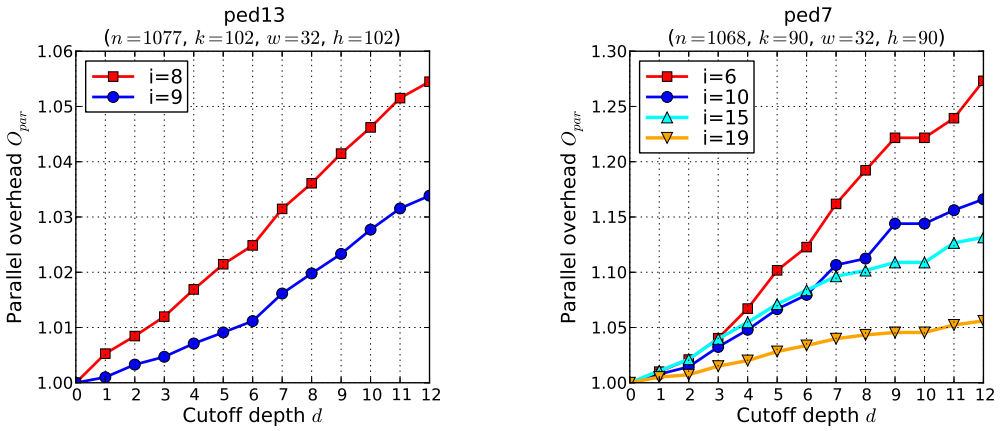


Figure 4.36: Parallel overhead $O_{par}(d)$ of the parallel scheme on **pedigree instances**, relative the number of node expansions of sequential AOBB.

than the $i = 6$. Similarly, for the grid instance 75-26-10 in Figure 4.35 (right) the difference between $i = 15$ and $i = 20$ is almost one order of magnitude in expanded number of nodes.

4.6.6.2 Parallel Overhead O_{par}

To better analyze the behavior of $N_{par}(d)$ we consider the metric of parallel overhead O_{par}, defined in Section 4.2.4 as the ratio N_{par}/N_{seq} of nodes expanded overall by the parallel algorithm compared to sequential AOBB. Analogous to the analysis of $SS_{par}(d)$ here we consider the overhead as a function of the cutoff depth d through $O_{par}(d) = N_{par}(d)/N_{seq} = N_{par}(d)/N_{par}(0)$, since $N_{par}(0) = N_{seq}$. Note that the absence of parallel overhead translates to a value of 1.0.

Figures 4.36 through 4.39 plot parallel overhead $O_{par}(d)$ as a function of d for instances from the four problem classes. As before, each plot contains results for more than one mini-bucket i-bound, if available.

Overview of Results. Results with respect to parallel overhead are twofold. On linkage, haplotyping, and side-chain prediction instances in Figures 4.36, 4.37, and 4.38, respectively, we observe low overhead values close to 1.0 – pedigree7 with $i = 6$ here sees the highest

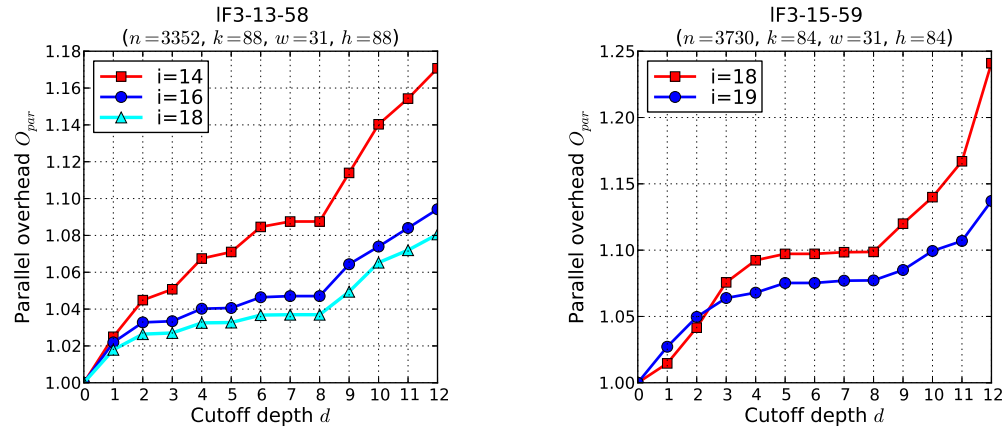


Figure 4.37: Parallel overhead $O_{par}(d)$ of the parallel scheme on **haplotyping instances**, relative the number of node expansions of sequential AOBB.

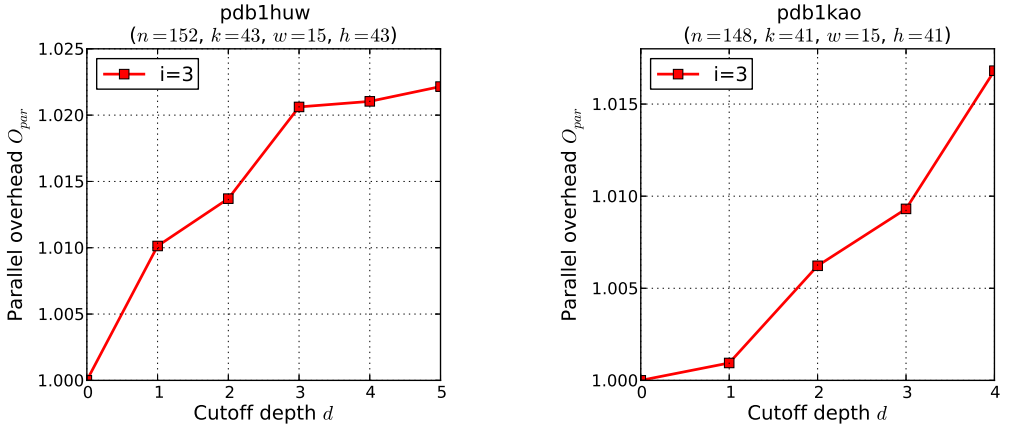


Figure 4.38: Parallel overhead $O_{par}(d)$ of the parallel scheme on **side-chain prediction instances**, relative the number of node expansions of sequential AOBB.

overhead of just under 1.3 at $d = 12$, but several other instances don't exceed 1.1 across the range of cutoff depths evaluated. Notably, the parallel overhead (and thus the number of explored nodes $N_{par}(d)$) also appears to grow linearly with d – in stark contrast to the exponential growth of the underlying search space $SS_{par}(d)$.

Parallel Overhead on Grid Networks. We observe slightly different results for grid networks in Figure 4.39. Namely, the parallel overhead is considerably higher than in the other three problem classes, with 75-25-3 and 75-27-2 reaching values of 3.5 and 3.0, respectively,

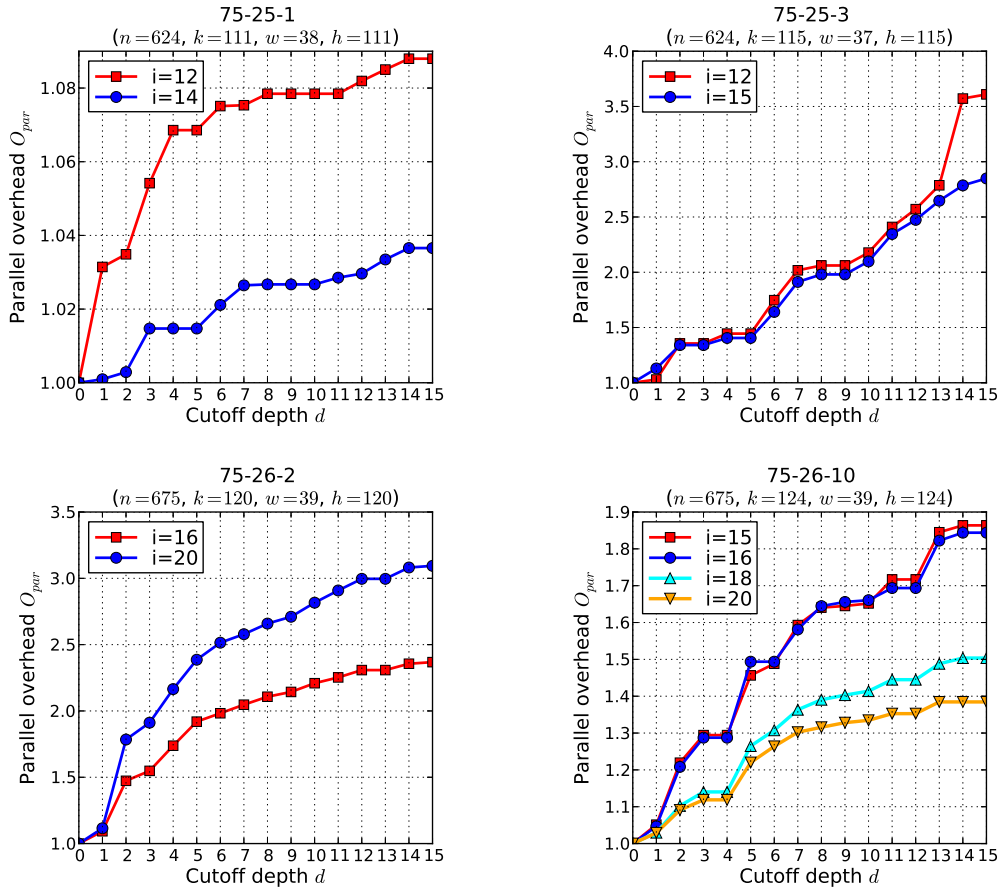


Figure 4.39: Parallel overhead $O_{par}(d)$ of the parallel scheme on **grid network instances**, relative the number of node expansions of sequential AOBB.

for deeper cutoff depths d. To make the impact on parallel performance more explicit, we can formulate the following straightforward proposition.

PROPOSITION **4.1.** *Assuming a parallel overhead of o and parallel execution on c CPUs, the parallel speedup the system can achieve is bounded by c/o – even with perfect load balancing and under the assumption of zero communication overhead.*

Note that the parallel overhead o is typically not known until after the parallel scheme finishes, yet it is useful to apply Proposition 4.1 to reason about reduced speedups after the fact. In the case of grid instance 75-25-3 with $i = 15$, for instance, with an parallel overhead of about 3.0 at depth $d = 14$ the best speedup we can hope for with 100 CPUs would be around 33 – which is fairly close to the speedup of 29.7 we observed in Table 4.18.

Impact of Mini-bucket i-bound. When we compare the plots for different i-bounds within each Figure against each other, we note that higher i-bounds and thus stronger mini-bucket heuristics tend to reduce the parallel overhead and its growth. As before, this is particularly evident for pedigree7, which has at $d = 12$ a maximum overhead of close to 1.30 using i-bound 6, but only overhead 1.05 with $i = 19$. Other pronounced example are largeFam3-13-58, which sees a maximum overhead of 1.18 for $i = 14$ and only 1.08 for $i = 18$, and the grid instance 75-26-10 with maximum overhead close to 1.9 for $i = 15$ and 1.4 for $i = 20$, respectively. And while the effect is not as pronounced for all instances, it makes intuitive sense that a stronger heuristic allows AOBB to combat the structural overhead more efficiently.

4.6.6.3 Parallel Redundancies Summary

We have investigated the practical implications of the redundancies introduced by the parallelization process, as described in Section 4.5. Through computing the size of the underlying parallel search space, we have shown that in theory the degree of redundancy can be considerable and, in particular, grows exponentially in the cutoff depth for many problem instances. However, contrasting this with experiments recording the actual number of node expansions by parallel AOBB showed that the impact in practice is far less pronounced, rendering the aforementioned parallel search space bounds exceedingly lose. Namely, for three out of the four problem classes evaluated, the parallel overhead is very close to the optimum 1.0, only growing linearly and very slowly in d. For grid instances, however, we mostly found the overhead to be considerably higher, sometimes reaching values of 3.0 or 3.5. This, in spite of the overhead's linear-only growth, imposes a decided limit on the achievable parallel speedup (cf. Proposition 4.1), which matches the relatively weak parallel performance we have observed on grid networks in Section 4.6.4.4. Finally, we have also confirmed that a stronger

mini-bucket heuristic, with a higher i-bound, is generally more effective at suppressing this overhead.

4.6.7 Parallel Scaling

This section addresses the question how parallel performance scales with the number of CPUs. Note that this information was already contained in Section 4.6.4, as part of the parallel speedup tables 4.9 and 4.10 for pedigrees, 4.13 and 4.14 for haplotyping, 4.16 for side-chain prediction, and 4.18 for grid networks. However, parallel performance scaling is a common metric in the field of parallel and distributed computing, hence we highlight the results here once more.

Figures 4.40 through 4.43 plot the parallel speedup of fixed-depth and variable-depth parallelization as a function of the CPU count. Shown are a selection of problem instances from each problem class – the same set of instances that was highlighted in earlier analysis, in fact. Besides our earlier results for 20, 100, and 500 CPUs, each plot also includes simulated speedups for 50, 200, 300, and 400 CPUs (cf. Section 4.6.1.2 for simulation details).

Results in Section 4.6.4 and throughout this chapter have suggested a "rule of thumb" that targets a subproblem count roughly at ten times the number of CPUs, which also matches the experience of other researchers. Each plot entry is thus a cross section of an instance's row in the full speedup table as follows: to obtain the speedup value for c CPUs, take the parallel run that has a subproblem count closest to $10c$ and use it as a basis for simulation. For instance, the 20-CPU speedup of the fixed-depth scheme on largeFam3-15-59 as shown in Figure 4.41 (left) is taken to be 17.77 from the $d = 7$ column in Table 4.14, which corresponds to $p = 240$ subproblems. Similarly, the 50-CPU speedup is simulated from the set of subproblems at $d = 8$ (not shown in Table 4.14, cf. Appendix B) which in case of largeFam3-15-59 has $p = 476$ subproblems.

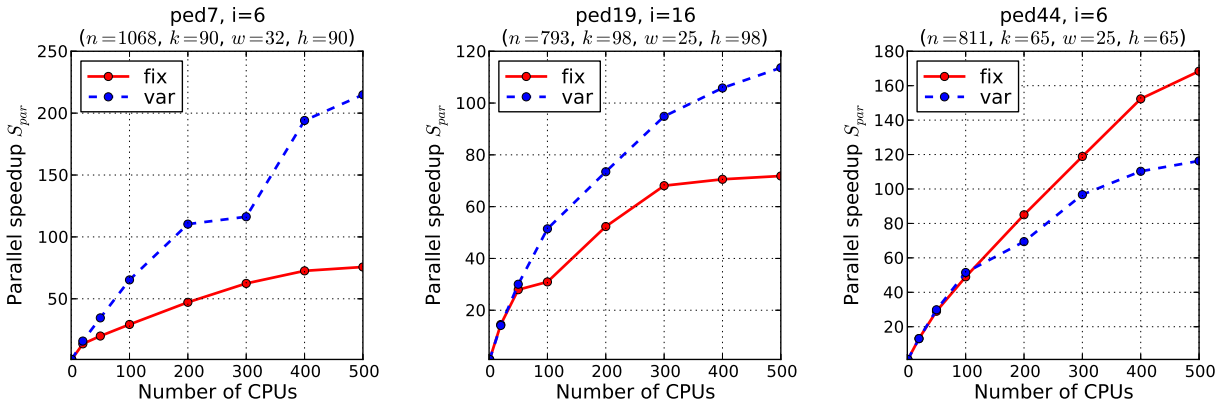


Figure 4.40: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **pedigree instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

Pedigree Linkage Instances. Figure 4.40 show scaling results for three pedigree instances, reflecting what we pointed out in Section 4.6.4.1 already. Both pedigree7 and pedigree19 are instances where the subproblem complexity estimation works well, leading to a more balanced parallel cutoff, better load balancing, and ultimately higher speedups for the variable-depth scheme. Pedigree7 in particular sees a very nice speedup of over 200 with 500 CPUs. Pedigree44, on the other hand, was one of the few pedigree examples where the variable-depth scheme fails to improve performance (and arguably decreases it) because of an outlier in the subproblem complexity estimates, and we see this mirrored in the plot in the plot on the right of 4.40.

LargeFam Haplotyping Instances. Scaling results for three haplotyping problems are depicted in Figure 4.41. We observe similar behavior to linkage instances: the variable-depth scheme and its subproblem estimates work well on largeFam3-15-59 and largeFam3-16-56 and achieve relatively high speedup values. Variable-depth performance on largeFam4-12-55 initially suffers from outlier subproblems, as illustrated earlier in Figure 4.22, but resolves this at 200 CPUs / 2000 subproblems and catches up to the fixed-depths scheme. (We note that we haven't conducted any runs with more than 3000 subproblems, which is why the plot entries for 400 and 500 CPUs left out.)

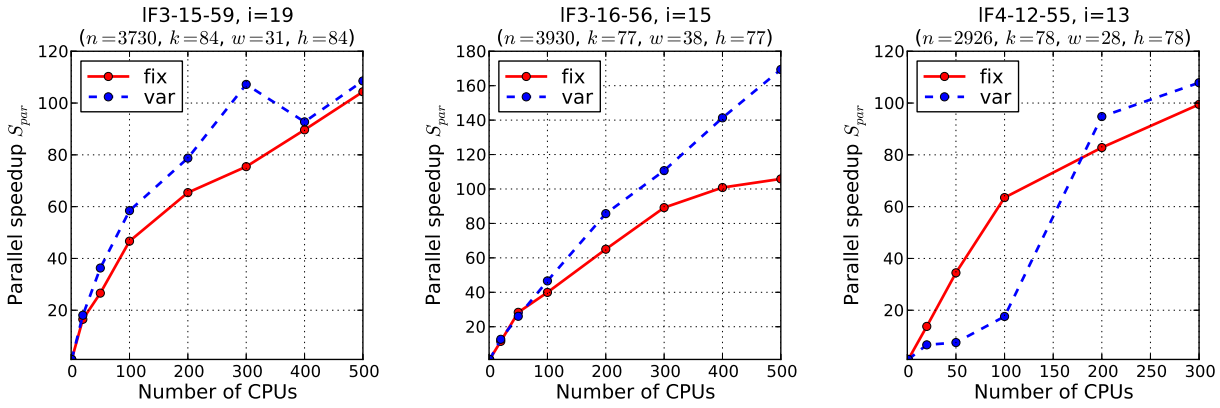


Figure 4.41: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **haplotyping instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

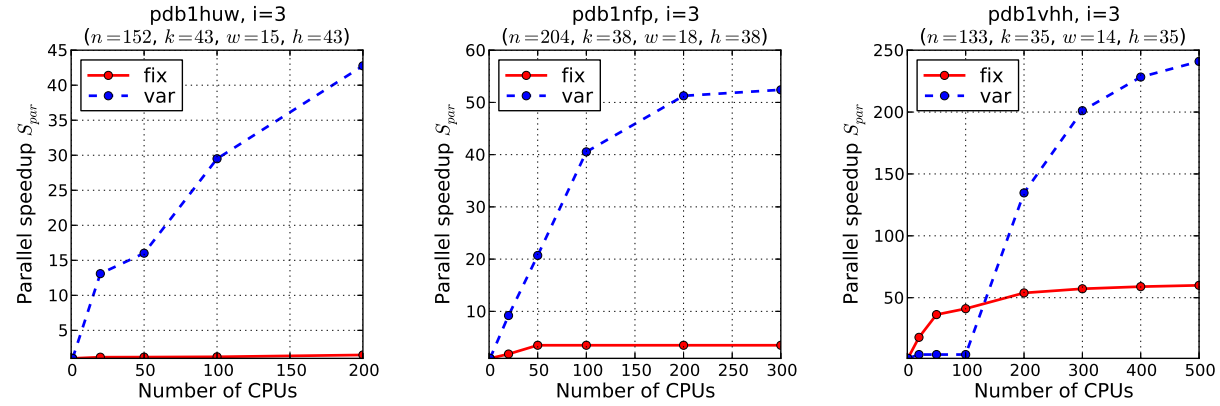


Figure 4.42: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **side-chain prediction instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

Pdb Protein Side-chain Prediction Instances. We plot scaling results for three side-chain prediction instances in Figure 4.42. As mentioned in Section 4.6.4.3, the combination of large domain size and very unbalanced search spaces makes effective load balancing very hard on these problems, at least without generating tens of thousands of problems. The one problem where we manually facilitated this process, pdb1vhh, is blocked by another complexity outlier early on but does indeed see very good speedups eventually in Figure 4.42 (right). And in all three cases shown the complexity prediction can alleviate some of the load imbalance, leading the variable-depth scheme to vastly outperform the fixed-depth one.

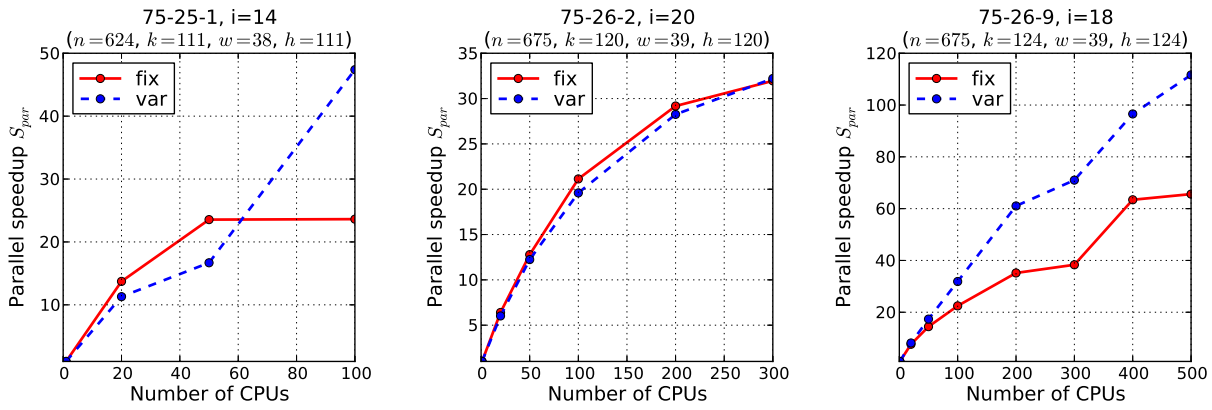


Figure 4.43: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **grid network instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

Grid Network Instances. Finally, Figure 4.43 show scaling results for three grid network instances. Section 4.6.4.4 explained how the performance on this class of instances is negatively impacted by a number of factors, including the implications of Amdahl's Law and parallel redundancies introduced by the conditioning process. Consequently, we find the scaling results in this section similarly sobering, in particular for network 75-26-2 (Fig. 4.43, middle). We note that 75-26-9 was in fact one of the few grid network examples where variable-depth parallelization did better, which is reflected here, too.

4.6.8 Summary of Empirical Evaluation and Analysis

We have conducted an in-depth experimental evaluation of parallelized AOBB, both in its fixed-depth as well as variable-depth incarnation, which uses the complexity estimation model proposed in Chapter 3 in order to improve parallel load balancing. We have considered four different problem classes which we characterized in Section 4.6.2, with experiments conducted on a variety of problem configurations. We have furthermore investigated a number of metrics and performance measures in Sections 4.6.3 through 4.6.7, from parallel runtime

and speedup to resource utilization and parallel overhead. In the following we put these results into a common, overall context.

4.6.8.1 Performance Considerations

The central element of parallel AOBB, and the distinguishing detail of its different implementations, is the choice of parallelization frontier. As stated previously and demonstrated in our experiments, it determines whether the parallel scheme can spread the overall workload evenly across the available parallel CPUs on the computational grid. This choice can be thought of along two dimensions: its size, i.e., how many subproblems to generate, and its shape, i.e., which particular subproblems to choose. The following sections elaborate and put our reported empirical results into this context.

Number of Subproblems. Regarding the size of the parallelization frontier, we can identify two conflicting motivations:

- Intuitively, it is desirable to have a large number of subproblems for the following reasons:
  - Trivially, large-scale parallel computations to solve harder and harder problem instances, with more and more CPUs, require an ever-increasing number of parallel subproblems.
  - Branch-and-Bound search spaces are often inherently unbalanced and due to their discrete nature a perfectly balanced parallel cutoff is unrealistic. Hence we aim to have a sufficiently large number of subproblems (beyond the number of parallel CPUs), so that a longer-running one can be compensated for by several smaller ones on another CPU. We have confirmed this experimentally and provided an intuitive formalization in form of the **parallel resource utilization** as detailed

in Section 4.6.5, in the sense that good parallel speedup necessitates high parallel resource utilization.

- At the same time, however, it is very easy to get to a point where too large a number of subproblems hurts the parallel performance:
  - First, as subproblems get smaller and smaller, the impact of **distributed processing overhead** like communication delays gets more noticeable. We have observed this in many cases of relatively simple problems in Section 4.6.4, where sequential AOBB takes only an hour or two – increasing the subproblem count more and more eventually leads to degraded parallel performance.
  - Second, somewhat related to the first point, generating many subproblems can take considerable time. Since it occurs in the master host and is non-parallelizable, it can seriously constrain the achievable parallel performance. This relationship is captured by **Amdahl's Law** (cf. Section 4.2.5); our experiments in Section 4.6.4 have yielded some indication of this, again mostly on simpler problem instances.
  - Third, following the theoretical analysis of Section 4.5, Section 4.6.6 has demonstrated that parallel AOBB can indeed suffer from a certain degree of **parallel redundancies** in practice. In particular, our experiments found that this redundancy appears to grow linearly with the cutoff depth. Even though far from the exponential growth typical of the underlying parallel search space, this suggests not pushing the parallel cutoff deeper than absolutely necessary.
  - Although more of a technical than a conceptual challenge, any implementation has to take various practical limitations into account, such as fixed, bounded network capacity or, in our case, a limit on how many subproblems the master process can reasonably handle before encountering unexpected behavior of the underlying operating system and file system, for instance.

Thus deciding on a parallelization frontier means finding the right trade-off between these two conflicting motivations: enough subproblems to facilitate parallelization on the given number of CPUs with efficient load balancing, but not too many to incur significant performance penalties as outlined above.

Subproblem Balancedness. The second performance consideration, the shape of the parallel cutoff through the particular choice of subproblems, is inherently intertwined with the issue of subproblem count discussed above. Namely, a small and balanced parallelization frontier can be superior to a larger, but unbalanced one. At the same time, we can sometimes compensate for an unbalanced parallel cutoff by simply increasing the number of subproblems, again as discussed above. This dichotomy is at the heart of the two different parallel AOBB implementations we've considered.

- *Fixed-depth* parallelization generates all subproblems at the same depth d, thus ignoring the inherent unbalancedness of branch-and-bound search spaces. Our experiments have shown that indeed in almost all cases this yields a very inhomogeneous, i.e., unbalanced parallelization frontier. The fixed-depth approach thus solely relies on generating a sufficiently large parallelization frontier to allow balanced parallel load and good parallel performance.

- The *variable-depth* scheme, in contrast, explicitly aims to generate a parallel cutoff with a more balanced set of subproblems by employing estimates of subproblem complexities and adapts the parallelization frontier accordingly. It should be clear that the scheme depends to a large extent on the accuracy of these estimates. If working as intended, however, our experiments have shown that a more balanced frontier can supersede the need to increase the number of subproblems.

With these considerations in mind, the following section will put into context the performance of parallel AND/OR Branch-and-Bound in general, as well as our two specific implementations, for the different problem classes we considered.

4.6.8.2 Empirical Results in Context

Given the above analysis regarding the variety of performance trade-offs, we can recapitulate the results of our experiments as follows:

Linkage and Haplotyping Instances. Problems from these two classes yielded the best results overall, which is facilitated by a number of things. First, our experiments have shown that parallel AOBB suffers from only small degrees of parallel overhead on these classes, with values of O_{par} fairly close to the optimum of 1.0. This allows us to establish sufficiently deep parallelization frontiers that enable good load balancing and high resource utilization, a prerequisite for high parallel speedups, as explained above. Second, we have found the complexity estimates within these two classes to be fairly reliable, which enables variable-depth parallel AOBB in particular to sufficiently balance the size of the parallel subproblems. This results in the parallel-depth scheme generally outperforming the fixed-depth one, especially for high subproblem counts and large number of CPUs.

Cases with weaker results were generally either too simple (such that distributed processing overhead and the implications of Amdahl's Law become a concern) or, specifically in the context of the variable-dept scheme, saw one or a handful of subproblems with vastly underestimated complexity that would turn out to dominate the overall performance.

Side-chain Prediction Instances. These problems are unique because of their large variable domains sizes and their generally very unbalanced search spaces. In our parallelization context this is a problematic combination since the number of subproblems grows very rapidly as the master process performs its conditioning operations, yet most of these

subproblems are very simple. This fact, together with technical limitations of our current implementation, makes it hard to achieve efficient load balancing and good parallel speedups, especially for large number of CPUs. Notably, however, in one example where we manually worked around said technical limitations we did end up with very good parallel performance (a future, improved version of the system might be able to work around these technical issues more generally). Secondly, within these given constraints we saw the variable-depth scheme drastically outperform the fixed-depth version, thanks to rather accurate subproblem complexity estimates.

Grid Network Instances. These problems showed relatively weak performance in our tests, both for the fixed-depth and variable-depth scheme. First, the subproblem complexity estimates turn out to be not accurate enough, which causes the variable-depth scheme to lose its advantage over fixed-depth that we've seen in other problem classes. Secondly, with all-binary variables, even the increased cutoff depths we experimented with often didn't yield a sufficiently large number of subproblems to achieve good load balancing. Thirdly, and most crucially, Section 4.6.6 demonstrated considerable degree of parallel overhead introduced in the conditioning process, sometimes reducing the theoretically achievable speedup by a factor of 3 or 3.5.

4.7 Conclusion to Chapter 4

This chapter presented a principled approach to parallelizing AND/OR Branch-and-Bound on a grid of computers, with the goal of pushing the boundaries of feasibility for exact inference. In contrast to many shared-memory or message-passing approaches, our assumed distributed framework is very general and resembles a general grid computing environment – i.e., our proposed scheme operates on a set of loosely coupled, independent computer systems, with one host acting as a "master" system, which is the only point of communication

for the remaining "workers," This model allows deployment in a wide range of network configurations and is to our knowledge, the only such implementation.

The master host explores a central search space over partial assignments, which serve as the conditioning set for the parallel subproblems and imply a "parallelization frontier." We have described two methods for choosing this frontier, one based on placing the parallel cutoff at a fixed-depth within the master search space, the other using the complexity estimates as developed in Chapter 3 to find a balanced set of subproblems at varying depth within the master process' search space.

The parallel scheme's properties were evaluated in-depth. We discussed the distributed overhead associated with any parallel system and laid out how it manifests itself in our case. We furthermore analyzed in detail the redundancies inherent to our specific problem setting of parallelizing AND/OR Branch-and-Bound graph search. In particular, we showed two things: first, how the lack of communication between workers can impact the pruning (due to unavailability of subproblem solutions as bounds for pruning); second and more importantly, how the theoretical upper bound on the number of explored node, the underlying parallel state space, grows with increased parallel cutoff depth, because caching of context-unifiable subproblems is lost across parallel processes. Overall, we have thus clearly demonstrated that parallelizing AND/OR Branch-and-Bound is far from embarrassingly parallel.

Experimental performance of the proposed parallel AOBB schemes was assessed and analyzed in an extensive empirical evaluation over a wide range of problem instances from four different classes, with a variety of algorithm parameters (e.g., mini-bucket i-bound). Running on linkage and haplotyping problems yielded generally positive results. Speedups were often relatively close to the theoretical limit, especially for small-scale and medium-scale parallelism with 20 and 100 CPUs, respectively. Large-scale parallel performance results on 500 CPUs are still good and further decrease parallel runtime; they are however not as strong in terms of the parallel speedup obtained relative to the theoretical maximum, in

particular for simpler problem instances where the implications of Amdahl's Law and things like grid communication delay become an issue at large scale. Either way, the variable-depth scheme was mostly able to outperform the fixed-depth one by a good margin, thanks to its better load balancing and avoidance of bottlenecks in the form of single, overly complex subproblems – in the few examples where that was not the case, it was commonly due to one such vastly underestimated parallel subproblem that would end up dominating the overall runtime.

In contrast, running side-chain prediction and grid network problem proved illustrative in highlighting the limitations of the proposed parallel scheme, both conceptually and in practice. On grid networks results were not as strong, the observed speedups were still substantial but generally far lower than the theoretical maximum suggested by the parallel CPU count. On the one hand, subproblem complexity estimates turned out to be rather unreliable, which caused bad variable-depth performance, often worse than the fixed-depth scheme. More importantly, however, instances in this problem class actually exhibit a fair degree of redundancies in the parallel search space, which immediately reduces the achievable parallel speedup. These redundancies were identified and quantified during theoretical analysis of the approach, but have not been a significant factor for the other problem classes.

Finally, for side-chain prediction instances a combination of large variable domain sizes and very unbalanced search spaces implies that a very large number of parallel subproblems is needed for efficient load balancing. However, our current implementation does not generally support this due to technical limitations. With these constraints in mind, however, we still found variable-depth parallelization to greatly outperform the fixed-depth scheme.

Overall, we are confident in the potential of the suggested parallel AND/OR implementation. Far from embarrassingly parallel, it succeeded in solving a large number of very complex problem instances several orders of magnitude faster than the already very advanced and award-winning sequential AND/OR Branch-and-Bound.

4.7.1 Integration with Superlink-Online SNP

As mentioned above, some of the initial motivation for this work was the wish to develop an advanced haplotyping component for the Superlink Online system (recall that maximum likelihood haplotyping can be translated to an MPE query over a suitably generated Bayesian network) [39, 105]. At the same time, this objective also determined some, if not most of the choices regarding the parallel architecture (grid-based, master-worker organization, with no shared memory or worker-to-worker message passing, using the Condor grid management system).

This goal was achieved in early 2012 as part of the release of Superlink-Online SNP, which is available at `http://cbl-hap.cs.technion.ac.il/superlink-snp/`. Besides enabling analysis of dense SNP data, this improved version of Superlink Online also includes parallel AOBB to enable haplotyping on previously infeasible input pedigree instances. Specifically, the deployed algorithm uses variable-depth parallelization as described above, based on an earlier instance of a regression model learned just from haplotyping instances. It runs on a shared cluster of computers at the Technion Institute of Technology in Haifa, Israel with up to 120 parallel cores – the target subproblem count is thus set to 1200.

A considerable amount of time was spent on the integration of parallel AOBB with the existing workflow of the Superlink Online system, including, but not limited to, preprocessing of the pedigree data, proper error handling, and (most irritatingly) cross-system and cross-platform compatibility of the executable binary files. The result of our efforts, the Superlink-Online SNP system, has been described in a recent journal article entitled "A System for Exact and Approximate Genetic Linkage Analysis of SNP Data in Large Pedigrees" in *Bioinformatics* (accepted for publication in November 2012).

4.7.2 Open Questions & Possible Future Work

There are a number of open research issues that could be addressed in the future. Conceptually, we can distinguish two principal directions:

First, the proposed scheme can be extended and improved within the framework discussed above. Possible questions to ask include how the variable ordering impacts parallel performance – specifically, can we find variable orderings that are "more suitable" in the sense that they minimize the structural redundancies resulting from the loss of (some of the) caching across subproblems? For instance, we might want variables that appear in the context of many other variables to appear close to the root in the guiding pseudo tree (recall that redundancies are caused by out-of-context variables that are part of the conditioning set). Alternatively, variables that only have relevance to a few other variables might be acceptable as part of the parallel conditioning (i.e., close to the root of the pseudo tree) if those other variables are also conditioned on – in that case proper caching can be applied within the master process.

Similarly within the existing framework, we can aim to make the variable-depth parallel scheme in particular more robust to inaccurate subproblem complexity estimates, which we have shown to be the limiting element in a number of cases. For instance, when additional parallel resources are available, the master process could decide to break up long-running subproblems into smaller pieces and submit those to the grid as well, in the hope that they might finish faster than the existing single job (which could be kept active regardless).

Second, we can consider moving away from our current model of parallelism, which is very widely applicable at the expense of its restrictiveness in terms of parallel communication. As discussed in the introductory sections of this chapter, there are a whole range of options to consider. A first step might be to allow workers to send updates back to the master host, as well as receive messages from it, at runtime. For instance, this could be used to exchange

live bounding information. A second, more direct approach is clearly to make the workers aware of each other and allow them to communicate directly, in a truly distributed fashion.

In comparing these two possible distributed approaches, many aspects can be considered. For one, we recognize that allowing workers to communicate directly could be more flexible in principle, but it also requires a more permissive network topology (which might be prohibitive for geographically distributed computing resources with firewall systems in-between). Moreover, in a naïve implementation the amount of communication would grow quadratically with the number of parallel CPUs, when it is only linear if communication is channeled through the master host.

4.8 Overview of Earlier Work

For completeness we include here a brief summary of some earlier work we conducted in the context of parallelizing AOBB. In particular, our initial investigation into parallelizing AOBB was based on somewhat more dynamic parallelism, where subproblems are generated "on-demand" instead of statically ahead of time. Full details are described in [90, 91, 92], the following outlines and discusses the central ideas.

4.8.1 Parallel Design

We assume the same setup as in earlier sections of this chapter, i.e. a computational grid with c independent, parallel processors. The parallelization process then proceeds as follows:

- The master generates the first c subproblems through depth-first search of the conditioning space and sends them to the grid worker hosts, one per worker. At this point the master pauses its exploration of the conditioning space.

  - The selection of parallel subproblems is dependent on an estimate of a subproblem's runtime and a "target" subproblem complexity T, provided as input to the algorithm (e.g., targeting 30 minutes): a subproblem is chosen for parallelization and submitted to the grid when its estimated complexity is below the threshold T, otherwise it is conditioned further.

- Upon receipt of a subproblem solution from a worker, the master processes it and immediately generates another subproblem for the newly idle worker, continuing with the previously halted exploration of the conditioning space.

- This continues until no further subproblems can be generated, at which point the master waits for the remaining subproblem solutions.

The on-demand generation of subproblems has a number of advantages. For instance, solution costs of solved subproblems can be propagated in a branch-and-bound fashion within the conditioning search space. Subsequent subproblems can then possibly profit from tighter cost bounds. In addition, this design allows the master to adjust its complexity predictions in an online fashion by incorporating information from solved subproblems. The following section elaborates.

4.8.1.1 Subproblem Complexity Assessment

The work presented in [90, 92] uses an earlier approach of subproblem complexity modeling that is slightly different from the general linear regression approach presented earlier in this thesis. Building on insight gained from general heuristic search [87], the underlying assumption is that the complexity of the subproblem below a given node n is captured by the expression $N(n) = b(n)^{D(n)}$, where $b(n)$ is the *effective branching factor* and $D(n)$ is the *average leaf node depth* of the subproblem below n.

Effective Branching Factor. We assume that for each problem instance there is a "true" underlying effective branching factor b. Given a set of previously solved sample subproblems $\{n_1, \ldots, n_m\}$ from the same instance, we note that $N(n_j)$ and $D(n_j)$ are easily counted, based on which $b(n_j) = \sqrt[D(n_j)]{N(n_j)}$ can be computed after the fact. For a new subproblem n' we then use the average $\bar{b} = \frac{1}{m} \sum_{j=1}^{m} b(n_j)$ as an estimate for the branching factor $b(n')$.

Average leaf node depth. We model the average depth of a given subproblem search space below root node n as a function of the upper and lower cost bound $U(n)$ and $L(n)$, respectively, the height of the subproblem pseudo tree $h(n)$, as well as the *average increment* denoted $inc(n)$. The following expands on these properties:

- The subproblem cost bounds $U(n)$ and $L(n)$ are provided by the mini-bucket heuristic and the branch-and-bound logic, respectively, and are known before the subproblem is explored. The difference $U(n) - L(n)$ then measures the "constrainedness" of the subproblem below n – intuitively, subproblems with a large gap between upper and lower cost bound require more effort, while a small value of $U(n) - L(n)$ tendentially signifies a highly constrained subproblem in which large parts of the underlying search space will be pruned.

- $inc(n)$ is meant to capture the average value of function instantiations from one level in the search space to the next, which can alternatively be called the cost increment. In the context of the subproblem cost bounds, the idea is that larger increment values require fewer steps to bridge the gap $U(n) - L(n)$ – empirical motivation and a more detailed derivation is given in [90, 92].

Based on this, the model for the average leaf node depth proposed in [90, 92] is the following:

$$D(n) \;=\; \frac{U(n) - L(n)}{inc(n)} \cdot \sqrt{h(n)}\,, \tag{4.4}$$

where the factor $\sqrt{h(n)}$ was empirically motivated. As with the branching factor $b(n)$ above, the average increment can be determined after a subproblem is solved by $inc(n) = (U(n) - L(n)) \cdot \sqrt{h(n)} \cdot D(n)^{-1}$, where $U(n)$, $L(n)$, and $h(n)$ are know ahead of time, and $D(n)$ is computed during subproblem exploration by recording and averaging the depth of all encountered leaf nodes.

To estimate the average leaf node depth of a new subproblem instance n', we in turn estimate its average increment $inc(n')$ as the arithmetic mean $\widehat{inc} = \frac{1}{m}\sum_{j=1}^{m} inc(n_j)$ of the average increments of previously solved subproblems $\{n_1, \ldots, n_m\}$ (i.e., we assume an underlying average increment value). With that and the new subproblem's upper and lower cost bound as well as pseudo tree height, we can compute an estimate of the average leaf depth using Equation 4.4.

Overall estimation scheme. To compute a complexity estimate for a new subproblem n' we simply combine the two elements above as follows:

$$\hat{N}(n') \;=\; \hat{b}^{\frac{U(n')-L(n')}{\widehat{inc}} \cdot \sqrt{h(n')}} \tag{4.5}$$

Here $\hat{b}$ and $\widehat{inc}$ are the arithmetic means of effective branching degrees and average increments, respectively, as computed across previously solved subproblems (these values can thus be updated every time a new subproblem solution is received by the master process).

Note that this approach to complexity estimation is in fact somewhat similar to, albeit more restricted than, the general regression approach described earlier in this thesis (cf. Section 3). Namely, modulo log transformation it can be regarded as a linear regression model with exactly one feature $(U(n) - L(n)) \cdot \sqrt{h(n)}$, with feature weight $\lambda = \frac{1}{\widehat{inc}}$.

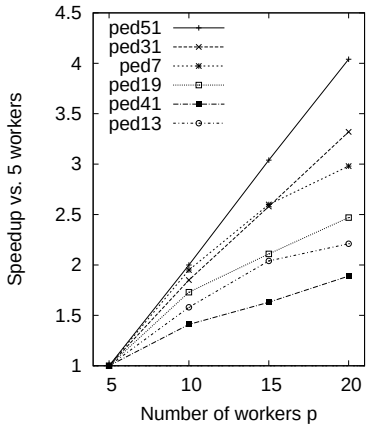


Figure 4.44: Speedup results of early dynamic parallel scheme on select pedigree instances, relative to 5 parallel workers (taken form [92]).

4.8.2 Limitations

At this early stage of our research as described in this section, experimental evaluation was only conducted on a handful of pedigree linkage instances. Secondly, a lack of computational resources limited the experimental setup to 20 parallel CPUs. Speedup results ranged from good to fair. Figure 4.44 reproduces one of the plots from [92]. Shown is the parallel speedup, relative to 5 CPUs, for up to 20 CPUs on the set of pedigree instances that we experimented with at that time. Comparison with the more recent results presented in this chapter is not straightforward, since the baseline in Figure 4.44 was chosen as the parallel runtime with 5 CPUs. We do note, however, that some of the instances (e.g., ped41) improve by less than a factor of 2 when going from 5 CPUs to 20 CPUs, which is quite inefficient at this small scale of parallelism.

Other metrics like parallel overhead were not directly considered at that point in time, see [90, 92] for more details. More importantly, however, our work on this system made a number of crucial limitations evident:

- Subsequent experiments with larger numbers of CPUs have demonstrated severe scaling issues – for instance, the master host is prone to being bottlenecked and getting blocked by the constant stream of grid input and output.

- At this early stage of our work no appropriate initialization techniques for the online estimation scheme had been developed. The starting estimates for branching degree and average increment were based on a short AOBB-based probe of the search space, results of which exhibited significant variance in terms of usefulness.

- The on-demand parallelization approach is not very suitable for a shared grid environment – a new subproblem is generated whenever an existing one finishes, but in the meantime other users might have scheduled many jobs in the shared parallel queue, thereby delaying the execution of new subproblems.

- Simulation of parallel runs, as employed throughout earlier parts of this chapter, is not easily attainable, since solution costs from earlier subproblems are reused as bounds for later ones, which depends very much on the actual order of subproblem execution.

All of the above points were at odds with a potential deployment of parallel AOBB within Superlink Online – additionally, they made systematic empirical evaluation very challenging and at times frustrating. This is why our subsequent efforts, presented comprehensively in Chapters 3 and 4, have focused on more static parallelization schemes and a more general, more reliable approach to subproblem prediction. However, in future research certain elements like refining complexity models in an online fashion might be worth revisiting.

Chapter 5

Conclusion

This thesis has presented a number of different contributions in the area of combinatorial optimization over graphical models. Our work comprises significant extensions and in-depth analysis of AND/OR Branch-and-Bound, a state-of-the art search algorithm for these typically NP-hard problems. The research we report on has pushed the boundaries of AOBB and combinatorial optimization in general along three distinct dimensions. We provide a brief summary for each of these in the following, open questions and future research directions were discussed in the respective chapters.

First, we significantly enhanced the applicability of AOBB for approximate inference by restoring and improving its anytime performance. Based on our analysis of the shortcomings of AOBB in this context, we have developed Breadth-Rotating AOBB, which combines ideas from breadth-first search with the principles of depth-first branch-and-bound while still maintaining favorable complexity guarantees. Extensive empirical evaluation has shown great and very robust results, both with regards to solution time and solution quality, on a large set of problem instances. We have also performed comparison against the leading local

search solver, inherently designed for anytime performance, and observed very competitive, if not better, anytime performance.

Second, we have investigated the issue of runtime complexity of AOBB. Known asymptotic bounds based on structural parameters like the induced width are often very loose in practice, due to determinism in the problem specification and the algorithm's pruning power. Instead we have developed simple complexity models based on statistical regression learning. Based on a set of 35 problem or subproblem features that we devised, we proceeded to learn a number of complexity models on and across different classes of problem instances. Through in-depth experimental evaluation of these models we demonstrated good predictive performance in the majority of cases across a variety of scenarios. Further analysis also considered the informativeness of the different problem features and confirmed that structural properties are less helpful than dynamic ones that incorporate, for instance, information derived from the problem's cost functions.

Third, we returned to exact inference, where we have pushed AOBB's ability to find and prove optimal solutions by several orders of magnitude through the introduction of distributed processing on a grid of computer. Our analysis has shown this to be far from embarrassingly parallel and therefore inherently hard to parallelize. On top of the detrimental effect of search space redundancies introduced in the parallelization process, the pruning power of AOBB as well as determinism in the problem's function tables render the choice of parallel subproblems and subsequent efficient load balancing, a prerequisite for good parallel performance, very challenging. We have proposed two implementations of parallel AOBB, one which chooses subproblems at a fixed depth in the overall search space, and one which tries to employ the complexity estimation developed previously to create a more balanced set of subproblems at varying depths. Detailed, large-scale experiments on instances from several problem classes have highlighted the various performance considerations and trade-offs to be made. In that context, we were able to demonstrate very good parallel performance

on two problem classes, with good load balancing and relatively high speedups. Two other problem classes each exhibit distinct properties that make them inherently less suitable for our parallel implementation, for conceptual and technical reasons, respectively – performance improvements over the sequential algorithm were not as impressive as in the other experiments, but still notable. Separately, we have also found that the variable-depth parallelization scheme has the ability to substantially outperform its fixed-depth sibling, provided that the underlying complexity estimates are sufficiently accurate.

The viability of our contributions has been further validated in practice as follows: First, Breadth-Rotating AOBB has recently won 1st places in all three relevant tracks of the PASCAL 2011 Probabilistic Inference Challenge, edging out a variety of other powerful solvers. Second, parallel AOBB with learning-based variable-depth load balancing, as described in Chapter 4, has been successfully integrated into Superlink Online-SNP, an online platform for large-scale genetic analysis used by geneticists and medical researchers worldwide.

Bibliography

[1] D. Allen and A. Darwiche. RC_Link: Genetic linkage analysis using Bayesian networks. *International Journal of Approximate Reasoning*, 48(2):499–525, 2008.

[2] D. Allouche, S. de Givry, and T. Schiex. Towards Parallel Non Serial Dynamic Programming for Solving Hard Weighted CSP. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, pages 53–60, 2010.

[3] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, Nov. 2002.

[5] S. Arnborg. Efficient Algorithms for Combinatorial Problems with Bounded Decomposability - A Survey. *BIT*, 25(1):2–23, 1985.

[6] F. Bacchus, S. Dalmao, and T. Pitassi. Value Elimination: Bayesian Interence via Backtracking Search. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, pages 20–28, 2003.

[7] R. J. Bayardo Jr. and D. P. Miranker. On the Space-Time Trade-off in Solving Constraint Satisfaction Problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 558–562, 1995.

[8] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *23rd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

[9] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] M. S. Boddy and T. Dean. Solving Time-Dependent Planning Problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 979–984, 1989.

[11] H. L. Bodlaender. Treewidth: Algorithmic Techniques and Results. In *22nd International Symposium on Mathematical Foundations of Computer Science*, pages 19–36, 1997.

[12] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth and minimum elimination tree-height. Technical report, Utrecht University, 1991.

[13] M. Campbell, A. J. Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[14] T. Cazenave and N. Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In *6th International Conference on Computers and Games*, pages 72–80, 2008.

[15] D. J. Challou, M. L. Gini, and V. Kumar. Parallel Search Algorithms for Robot Motion Planning. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 46–51, 1993.

[16] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. In *6th International Conference on Computers and Games*, pages 60–71, 2008.

[17] A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1427–1429, 2006.

[18] P. C. Chen. Heuristic Sampling: A Method for Predicting the Performance of Tree Searching Programs. *SIAM J. Comput.*, 21(2):295–315, 1992.

[19] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-Based Work Stealing in Parallel Constraint Programming. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 226–241, 2009.

[20] G. Chu and P. J. Stuckey. PMiniSAT: A Parallelization of MiniSAT 2.0. Technical report, SAT-race 2008 solver descriptions, 2008.

[21] G. Cornuéjols, M. Karamanov, and Y. Li. Early Estimates of the Size of Branch-and-Bound Trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.

[22] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, Feb. 2001.

[23] A. Darwiche, R. Dechter, A. Choi, V. Gogate, and L. Otten. UAI 2008 Probabilistic Inference Evaluation. `http://graphmod.ics.uci.edu/uai08/`, 2008.

[24] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[26] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):279–312, 1990.

[27] R. Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[28] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[29] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, Feb. 2007.

[30] R. Dechter and J. Pearl. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[31] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[32] R. Dechter and I. Rish. Mini-buckets: a general scheme for bounded inference. *Journal of the ACM*, 50(2):107–153, 2003.

[33] N. R. Draper and H. Smith. *Applied Regression Analysis*. Wiley-Interscience, 3rd edition, 1998.

[34] M. Drozdowski. *Scheduling for Parallel Processing*. Springer, 2009.

[35] G. Elidan and A. Globerson. UAI 2010 Approximate Inference Challenge. `http://www.cs.huji.ac.il/project/UAI10/`, 2010.

[36] G. Elidan, A. Globerson, and U. Heinemann. PASCAL 2011 Probabilistic Inference Challenge. `http://www.cs.huji.ac.il/project/PASCAL/`, 2012.

[37] M. P. Evett, J. A. Hendler, A. Mahanti, and D. S. Nau. PRA*: Massively Parallel Heuristic Search. *J. Parallel Distrib. Comput.*, 25(2):133–143, 1995.

[38] C. Ferguson and R. E. Korf. Distributed Tree Search and Its Application to Alpha-Beta Pruning. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 128–132, 1988.

[39] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 59:41–60, 2005.

[40] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(suppl 1):S189–S198, Jan. 2002.

[41] M. Fishelson and D. Geiger. Optimizing exact genetic linkage computations. *Journal of Computational Biology*, 11(2-3):263–275, Jan. 2004.

[42] N. Flerova, R. Marinescu, and R. Dechter. Preliminary Empirical Evaluation of Anytime Weighted AND/OR Best-First Search for MAP. In *Proceedings of 4th NIPS workshop on Discrete Optimization in Machine Learning*, 2012.

[43] I. Foster and C. Kesselmann. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 1998.

[44] E. C. Freuder and M. J. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1076–1078, 1985.

[45] M. R. Garey, D. S. Johnson, and R. Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[46] B. Gendron and T. G. Crainic. Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[47] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman and Hall/CRC, 2006.

[48] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[49] G. Gottlob, Zoltán Miklós, and T. Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM*, 56(6), 2009.

[50] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[51] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[52] A. Grama and V. Kumar. Parallel Search Algorithms for Discrete Optimization Problems. *INFORMS Journal on Computing*, 7(4):365–385, 1995.

[53] A. Grama and V. Kumar. State of the Art in Parallel Search Techniques for Discrete Optimization Problems. *IEEE Trans. Knowl. Data Eng.*, 11(1):28–35, 1999.

[54] Y. Hamadi and C. M. Wintersteiger. Seven Challenges in Parallel SAT Solving. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, 2012.

[55] E. A. Hansen and R. Zhou. Anytime Heuristic Search. *J. Artif. Intell. Res. (JAIR)*, 28:267–297, 2007.

[56] E. A. Hansen and S. Zilberstein. Monitoring the Progress of Anytime Problem-Solving. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference*, pages 1229–1234, 1996.

[57] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[58] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 607–613, 1995.

[59] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2009.

[60] F. Hutter, H. H. Hoos, and T. Stützle. Efficient Stochastic Local Search for MPE Solving. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 169–174, 2005.

[61] A. Ihler, N. Flerova, R. Dechter, and L. Otten. Join-graph based cost-shifting schemes. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, 2012.

[62] P. Jégou and C. Terrioux. Decomposition and Good Recording for Solving Max-CSPs. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 196–200, 2004.

[63] B. Jurkowiak, C. M. Li, and G. Utard. A Parallelization Scheme Based on Work Stealing for a Class of SAT Solvers. *J. Autom. Reasoning*, 34(1):73–101, 2005.

[64] K. Kask and R. Dechter. Stochastic local search for Bayesian network. In *Proceedings of the 7th International Workshop on Artificial Intelligence and Statistics*, 1999.

[65] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, June 2001.

[66] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166(1-2):165–193, Aug. 2005.

[67] K. Kask, A. Gelfand, L. Otten, and R. Dechter. Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 2011.

[68] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial intelligence - Volume 2*, pages 1014–1019, 2006.

[69] U. Kjaerulff. Triangulation of Graphs – Algorithms Giving Small Total State Space. Technical report, Dept. of Mathematics and Computer Science, Aalborg University, 1990.

[70] D. E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[71] R. E. Korf. Improved Limited Discrepancy Search. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference*, pages 286–291, 1996.

[72] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[73] V. Kumar, A. Grama, and N. R. Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *J. Parallel Distrib. Comput.*, 22(1):60–79, 1994.

[74] V. Kumar and V. N. Rao. Parallel depth first search. Part II. Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[75] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[76] L. Lelis, S. Zilles, and R. C. Holte. Fast and Accurate Predictions of IDA*'s Performance. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, 2012.

[77] L. H. Lelis, S. Zilles, and R. C. Holte. Predicting the size of IDA*s search tree. *Artificial Intelligence*, 196:53–76, 2013.

[78] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[79] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Advances in Neural Information Processing Systems 16*, 2003.

[80] R. Lüling and B. Monien. Load Balancing for Distributed Branch and Bound Algorithms. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 543–548, 1992.

[81] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[82] R. Marinescu and R. Dechter. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16-17):1457–1491, Nov. 2009.

[83] R. Marinescu and R. Dechter. Memory intensive AND/OR search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16-17):1492–1524, Nov. 2009.

[84] R. Marinescu, K. Kask, and R. Dechter. Systematic vs. Non-systematic Algorithms for Solving the MPE Task. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, pages 394–402, 2003.

[85] P. Meseguer. Interleaved Depth-First Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1382–1387, 1997.

[86] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.

[87] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc., 1997.

[88] L. Otten and R. Dechter. Bounding Search Space Size via (Hyper)tree Decompositions. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 452–459, 2008.

[89] L. Otten and R. Dechter. Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 576–581, 2008.

[90] L. Otten and R. Dechter. Load Balancing for Parallel Branch and Bound. In *Proceedings of 10th Workshop on Preferences and Soft Constraints*, 2010.

[91] L. Otten and R. Dechter. Towards Parallel Search for Optimization in Graphical Models. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, 2010.

[92] L. Otten and R. Dechter. Finding Most Likely Haplotypes in General Pedigrees through Parallel Search with Dynamic Load Balancing. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 26–37, 2011.

[93] S. J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Trans. Knowl. Data Eng.*, 22(10):1345–1359, 2010.

[94] J. D. Park. Using Weighted MAX-SAT Engines to Solve MPE. In *Proceedings of the 18th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence*, pages 682–687, 2002.

[95] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley, 1984.

[96] J. Pearl. *Probabilistic reasoning in intelligent systems.* Morgan Kaufmann Publishers Inc., 1988.

[97] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[98] C. Powley, C. Ferguson, and R. E. Korf. Depth-First Heuristic Search on a SIMD Machine. *Artificial Intelligence*, 60(2):199–242, 1993.

[99] P. Prosser and C. Unsworth. Limited discrepancy search revisited. *J. Exp. Algorithmics*, 16, 2011.

[100] N. Robertson and P. D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, 7(3):309–322, 1986.

[101] J. L. Rodgers and W. A. Nicewander. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician*, 42(1):59–66, 1988.

[102] G. A. F. Seber and A. J. Lee. *Linear Regression Analysis.* John Wiley & Sons, Inc., 2nd edition, 2003.

[103] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating Hard Satisfiability Problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[104] S. E. Shimony. Finding MAPs for Belief Networks is NP-Hard. *Artificial Intelligence*, 68(2):399–410, 1994.

[105] M. Silberstein. Building an Online Domain-Specific Computing Service over Non-dedicated Grid and Cloud Resources: The Superlink-Online Experience. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 174–183, 2011.

[106] M. Silberstein, A. Tzemach, N. Dovgolevsky, M. Fishelson, A. Schuster, and D. Geiger. Online system for faster multipoint linkage analysis via parallel execution on thousands of personal computers. *The American Journal of Human Genetics*, 78(6):922–935, 2006.

[107] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., 2002.

[108] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[109] J. T. Thayer and W. Ruml. Anytime Heuristic Search: Frameworks and Algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search*, 2010.

[110] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58(1):267–288, 1996.

[111] S. Tschöke, R. Lüling, and B. Monie. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 182–189, 1995.

[112] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 2nd edition, 2000.

[113] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media / Yahoo Press, 2nd edition, 2010.

[114] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

[115] C. Yanover, O. Schueler-Furman, and Y. Weiss. Minimizing and learning energy functions for side-chain prediction. *Journal of Computational Biology*, 15(7):899–911, Sept. 2008.

[116] W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *J. Artif. Intell. Res. (JAIR)*, 38:85–133, 2010.

[117] W. Yeoh and M. Yokoo. Distributed Problem Solving. *AI Magazine*, 33(3):53–65, 2012.

[118] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowl. Data Eng.*, 10(5):673–385, 1998.

[119] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.

[120] R. Zivan and A. Meisels. Dynamic Ordering for Asynchronous Backtracking on DisCSPs. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, pages 32–46, 2005.

Appendices

A Additional Complexity Estimation Results

A.1 Complexity Estimation per Instance

- Figure A.1 shows results of per-instance learning on eight pedigree linkage problem instances, using 5-fold cross validation.
- Figure A.2 shows results of per-instance learning on eight largeFam haplotyping problem instances, using 5-fold cross validation.
- Figure A.3 shows results of per-instance learning on eight pdb protein side-chain prediction problem instances, using 5-fold cross validation.
- Figure A.4 shows results of per-instance learning on eight grid network problem instances, using 5-fold cross validation.

A.2 Complexity Estimation per Problem Class

- Figure A.5 shows results of per-class learning on eight pedigree linkage problem instances, using the respective other instances of the same class for model training.
- Figure A.6 shows results of per-class learning on eight largeFam haplotyping problem instances, using the respective other instances of the same class for model training.
- Figure A.7 shows results of per-class learning on eight pdb protein side-chain prediction problem instances, using the respective other instances of the same class for model training.
- Figure A.8 shows results of per-class learning on eight grid network problem instances, using the respective other instances of the same class for model training.

A.3 Complexity Estimation across Problem Classes

- Figure A.9 shows results of cross-class learning on eight pedigree linkage problem instances, using instances from all classes for model training.
- Figure A.10 shows results of cross-class learning on eight largeFam haplotyping problem instances, using instances from all classes for model training.
- Figure A.11 shows results of cross-class learning on eight pdb protein side-chain prediction problem instances, using instances from all classes for model training.
- Figure A.12 shows results of cross-class learning on eight grid network problem instances, using instances from all classes for model training.

A.4 Complexity Estimation for Unseen Problem Class

- Figure A.13 shows results of unseen-class learning on eight pedigree linkage problem instances, using instances from all non-pedigree classes for model training.
- Figure A.14 shows results of unseen-class learning on eight largeFam haplotyping problem instances, using instances from all non-largeFam classes for model training.
- Figure A.15 shows results of unseen-class learning on eight pdb protein side-chain prediction problem instances, using instances from all non-pdb classes for model training.
- Figure A.16 shows results of unseen-class learning on eight grid network problem instances, using instances from all non-grid classes for model training.

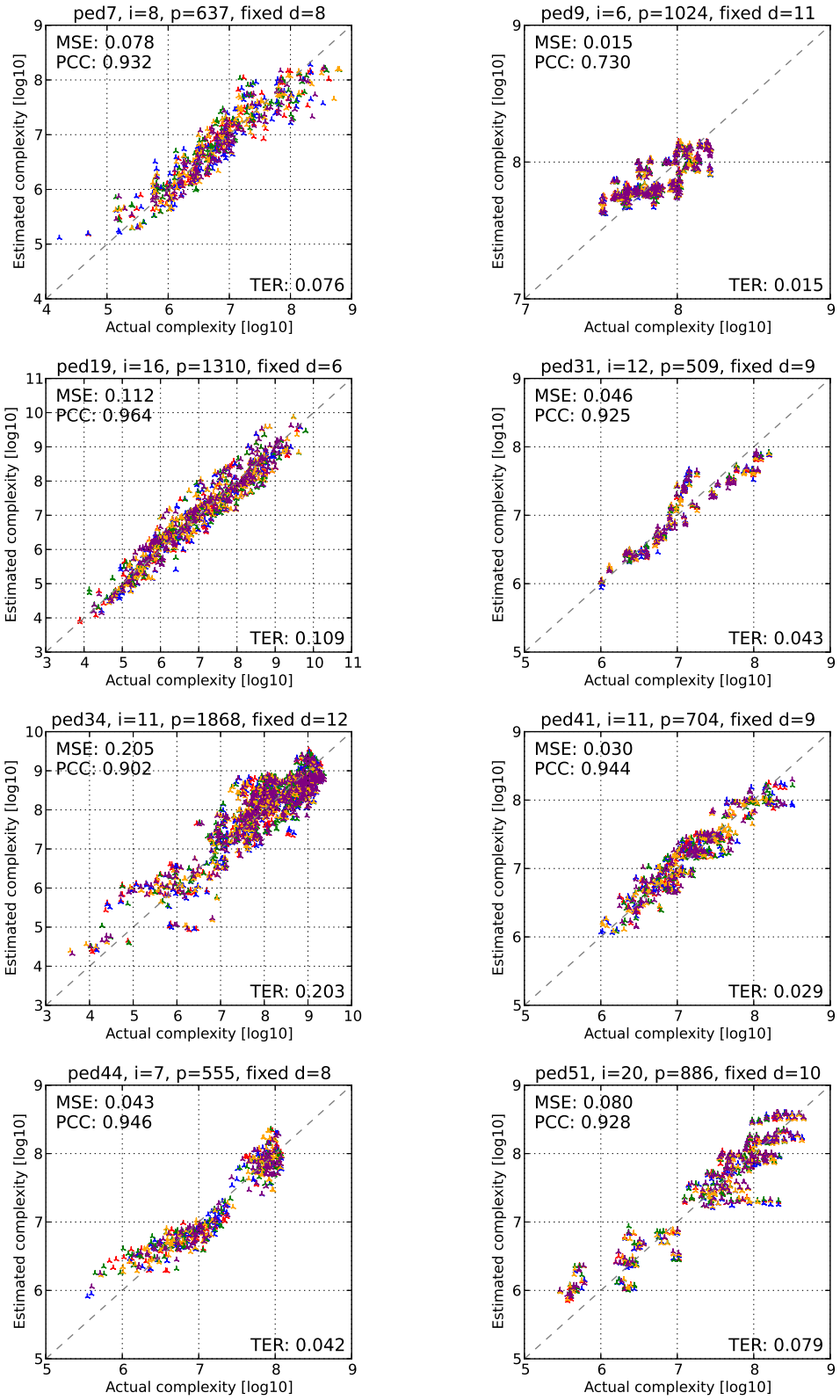


Figure A.1: Per-instance estimation results on pedigree linkage instances, using 5-fold cross validation.

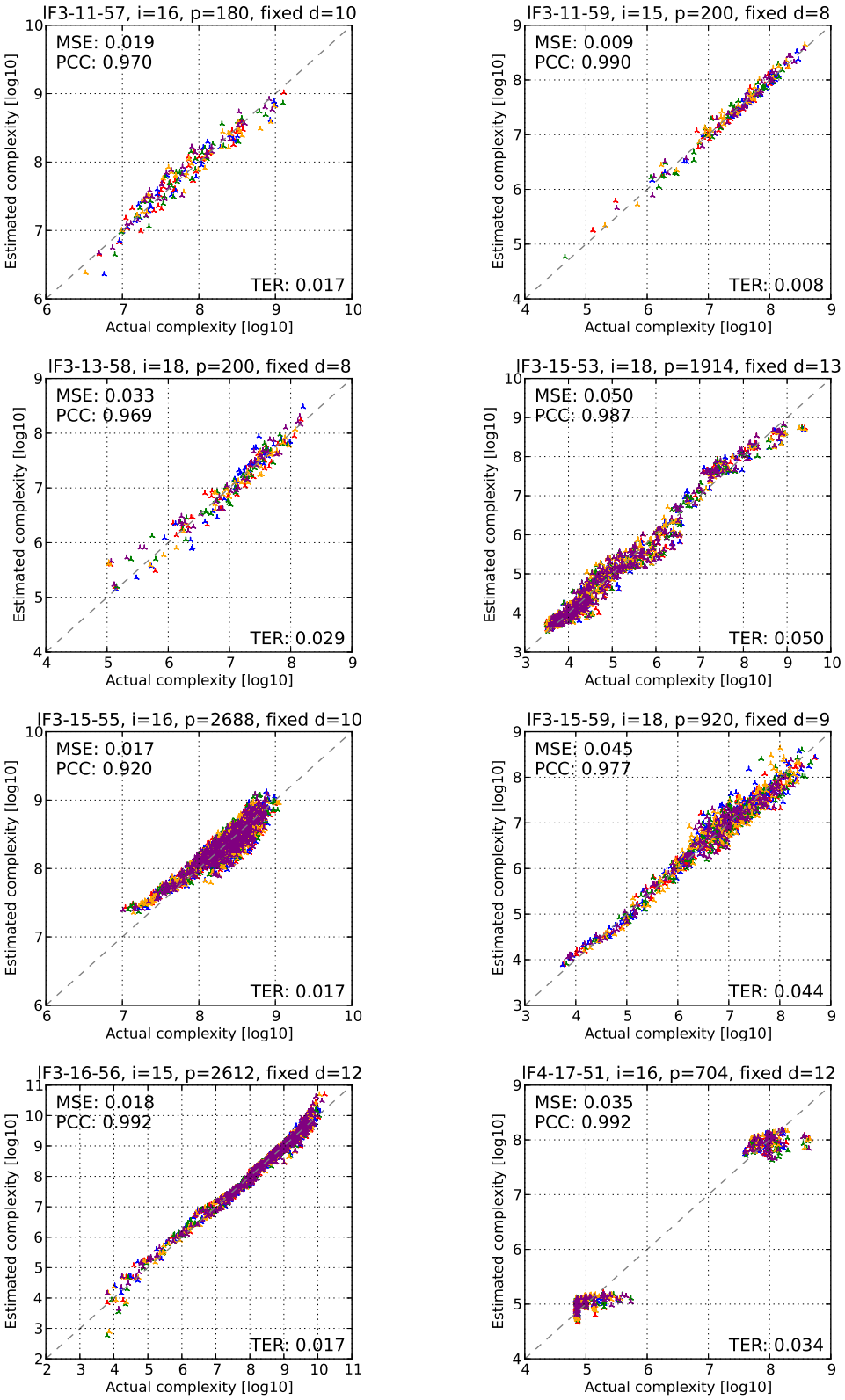


Figure A.2: Per-instance estimation results on largeFam haplotyping instances, using 5-fold cross validation.

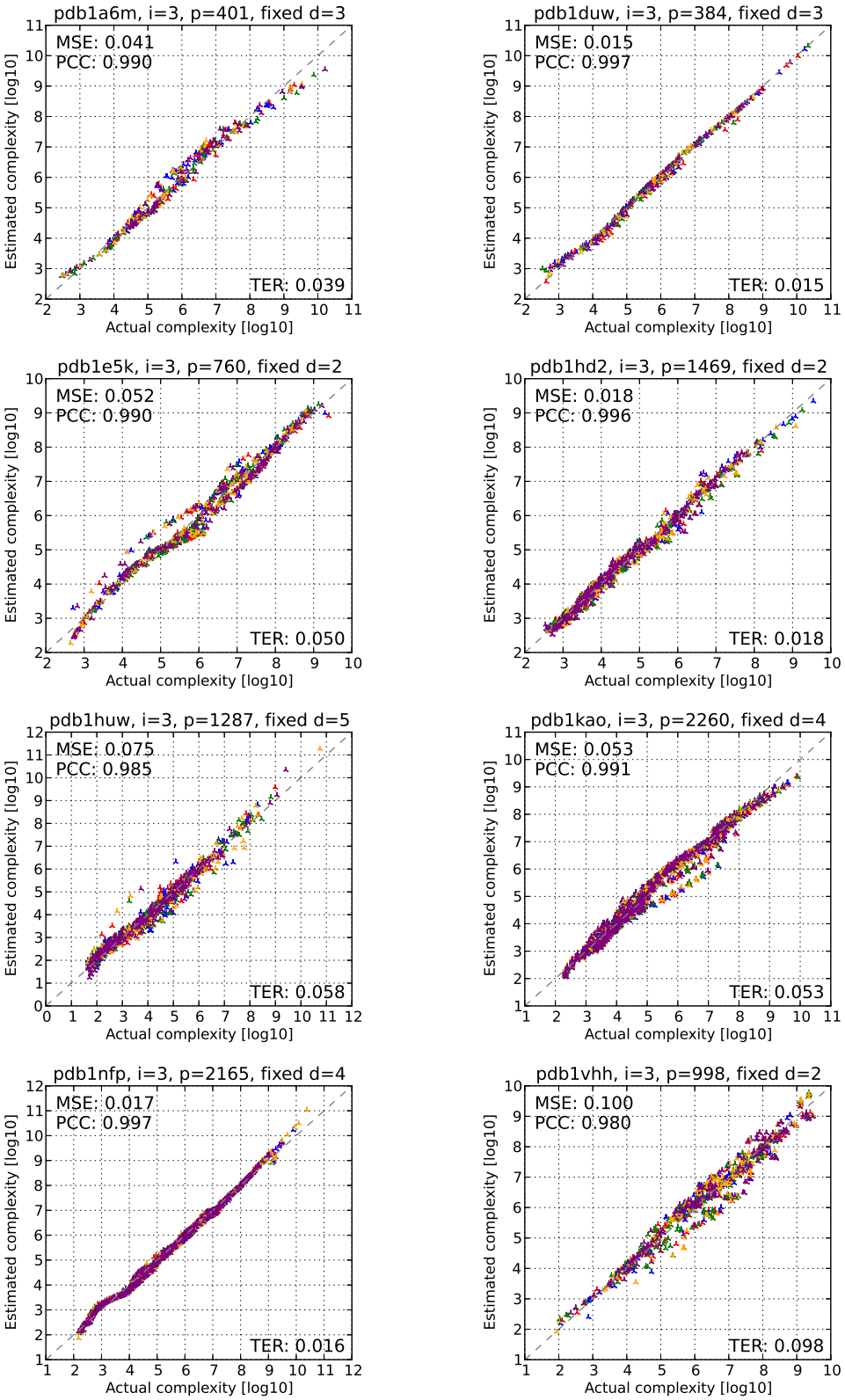


Figure A.3: Per-instance estimation results on pdb side-chain prediction instances, using 5-fold cross validation.

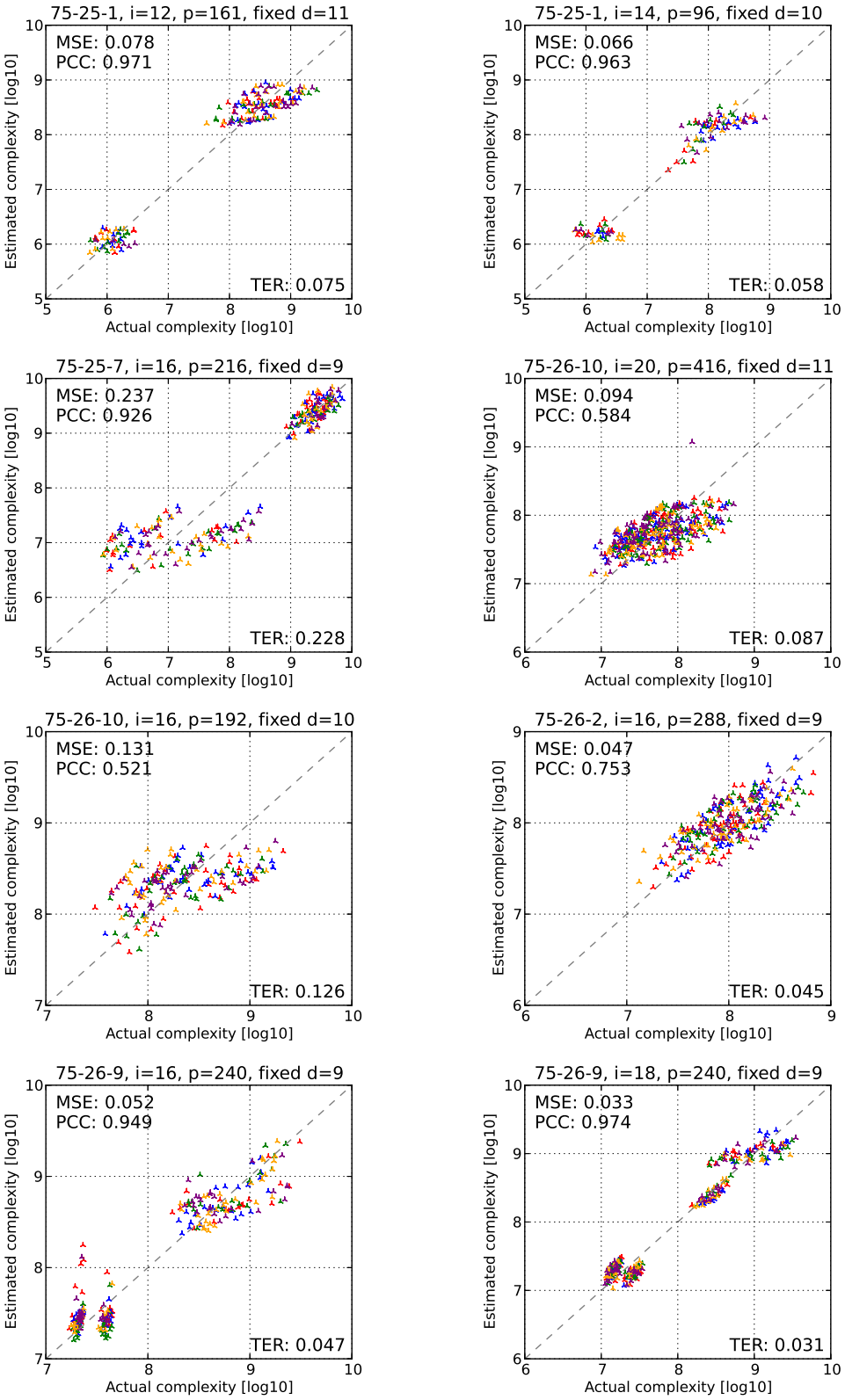


Figure A.4: Per-instance estimation results on grid network instances, using 5-fold cross validation.

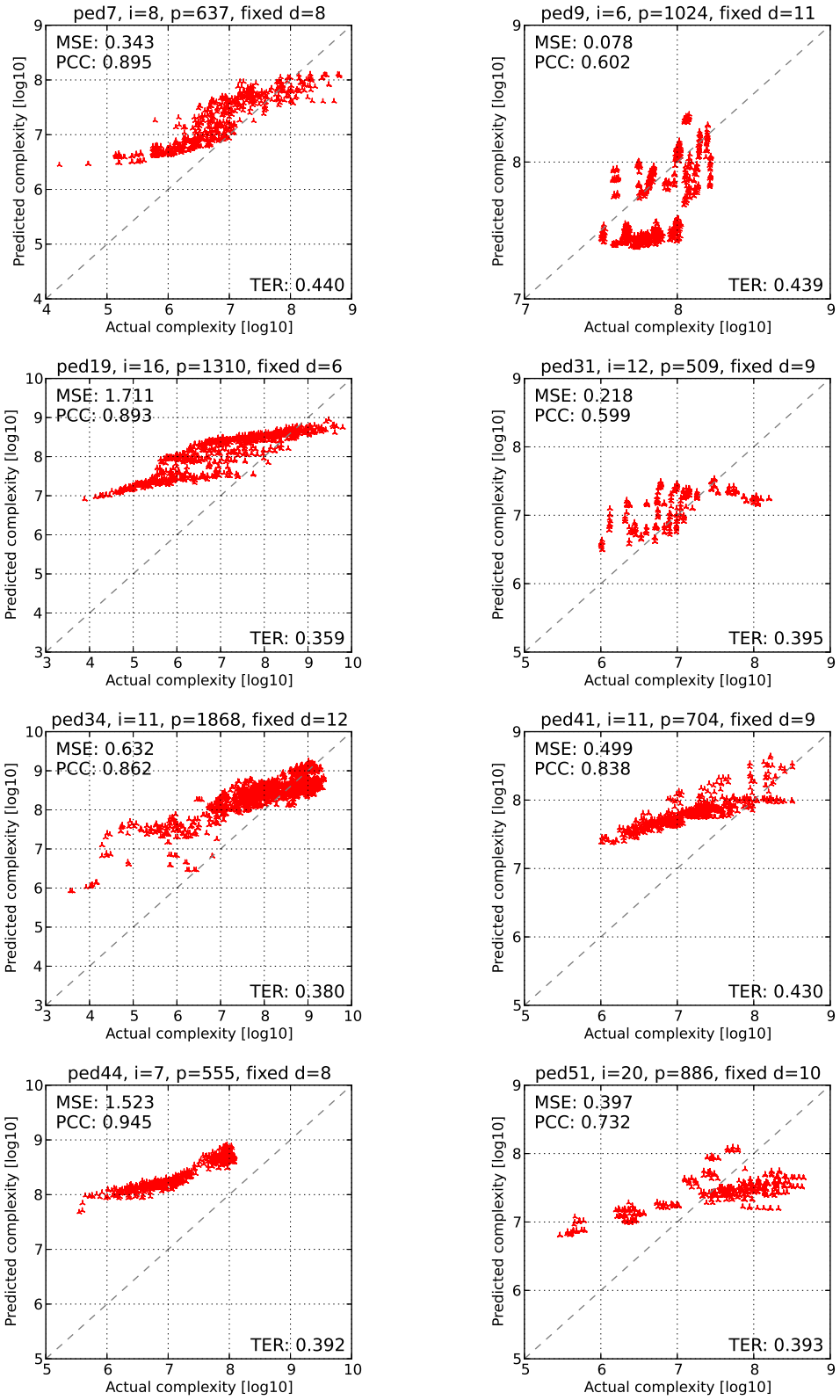


Figure A.5: Per-class estimation results on pedigree linkage instances, using the respective other instances of the same class for model training.

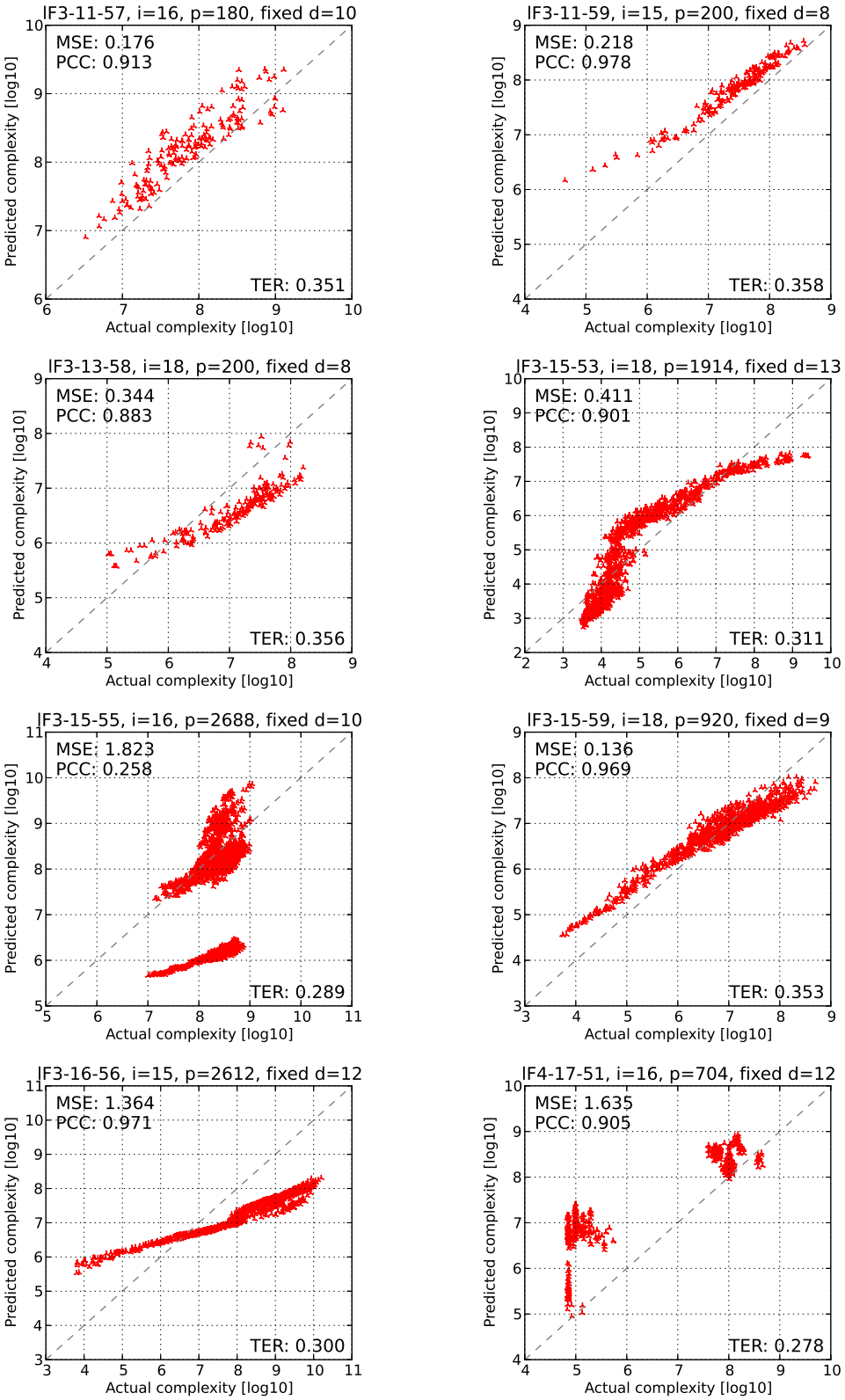


Figure A.6: Per-class estimation results on largeFam haplotype instances, using the respective other instances of the same class for model training.

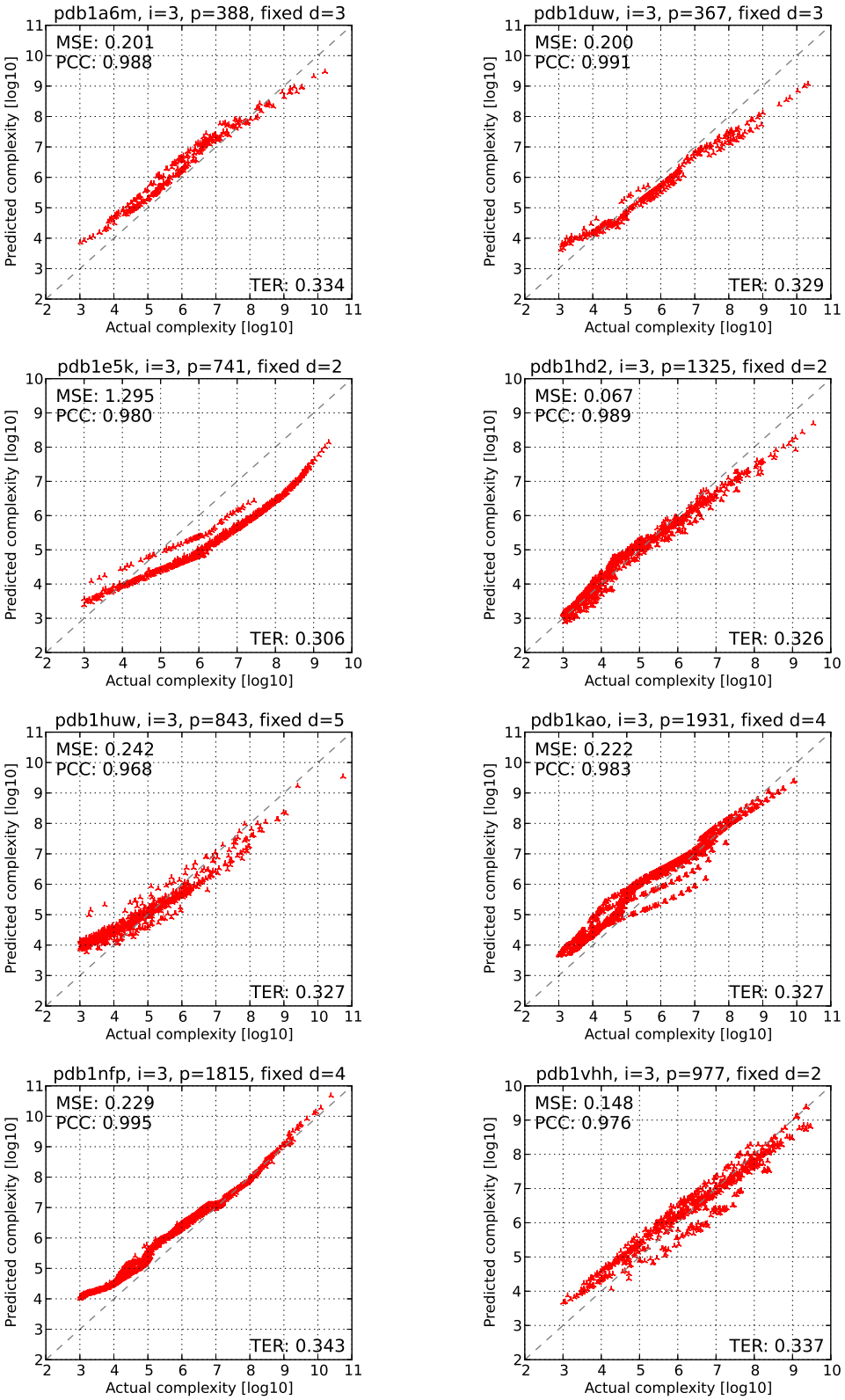


Figure A.7: Per-class estimation results on pdb side-chain prediction instances, using the respective other instances of the same class for model training.

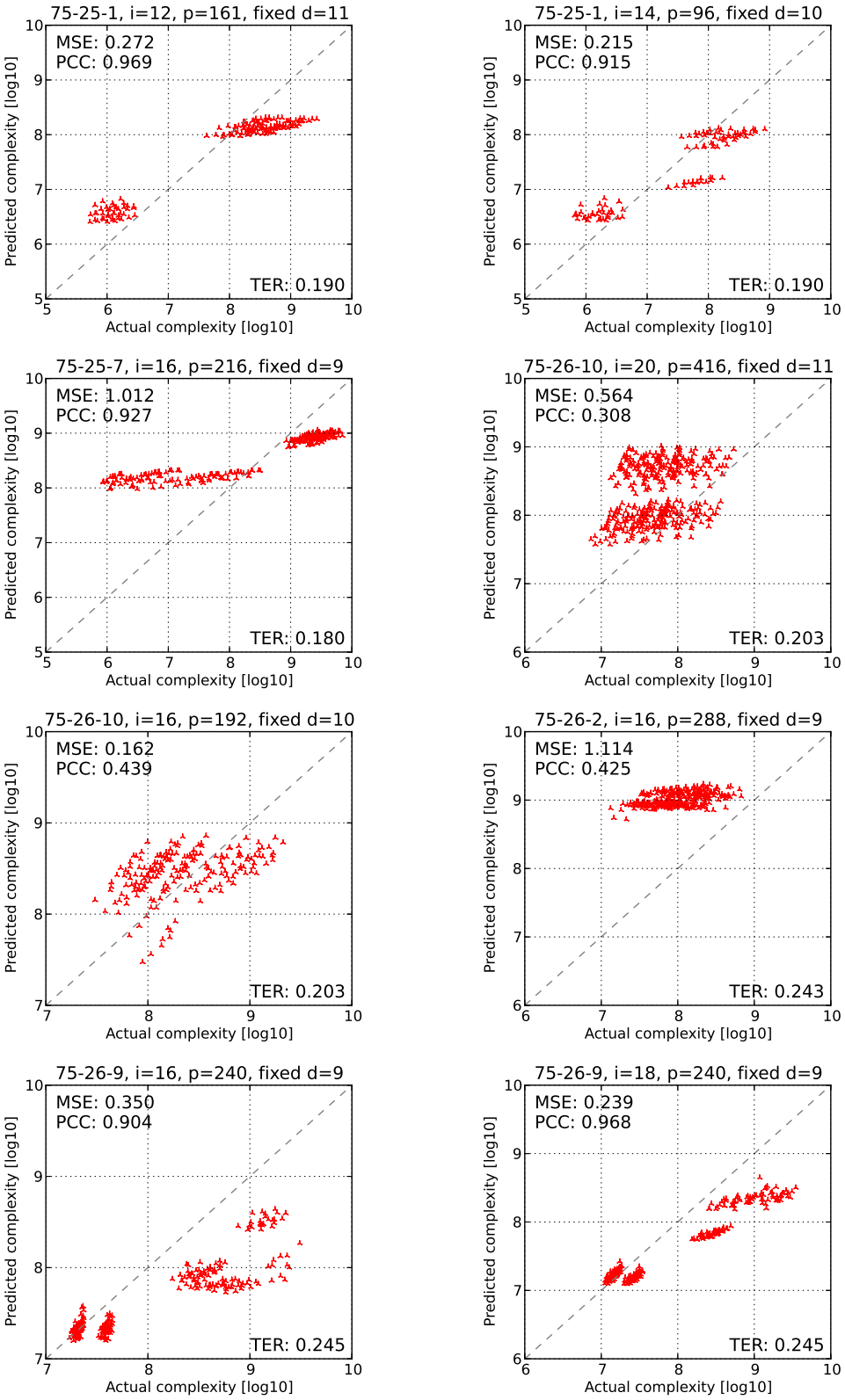


Figure A.8: Per-class estimation results on grid network instances, using the respective other instances of the same class for model training.

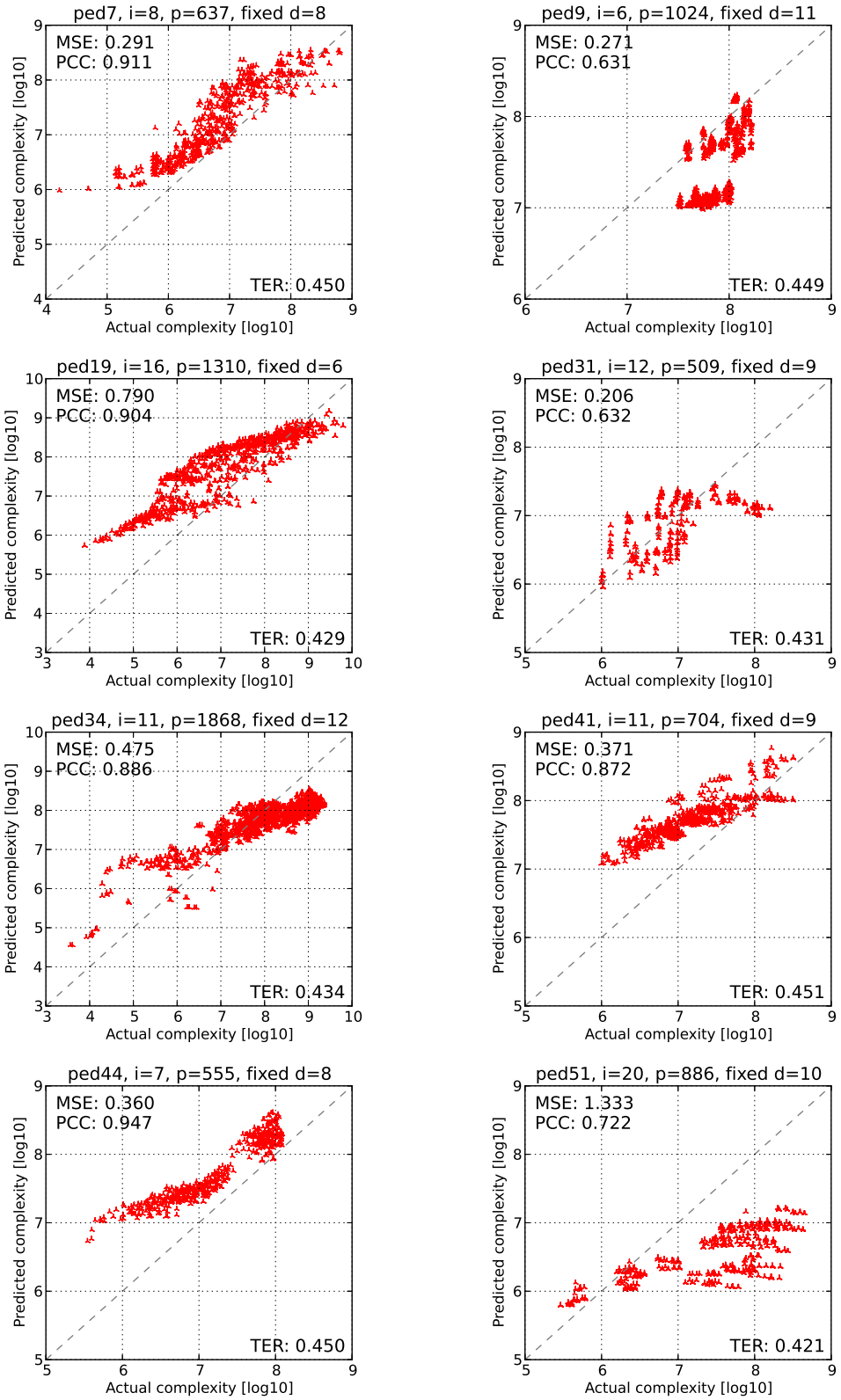


Figure A.9: Cross-class estimation results on pedigree linkage instances, using instances from all classes for model training.

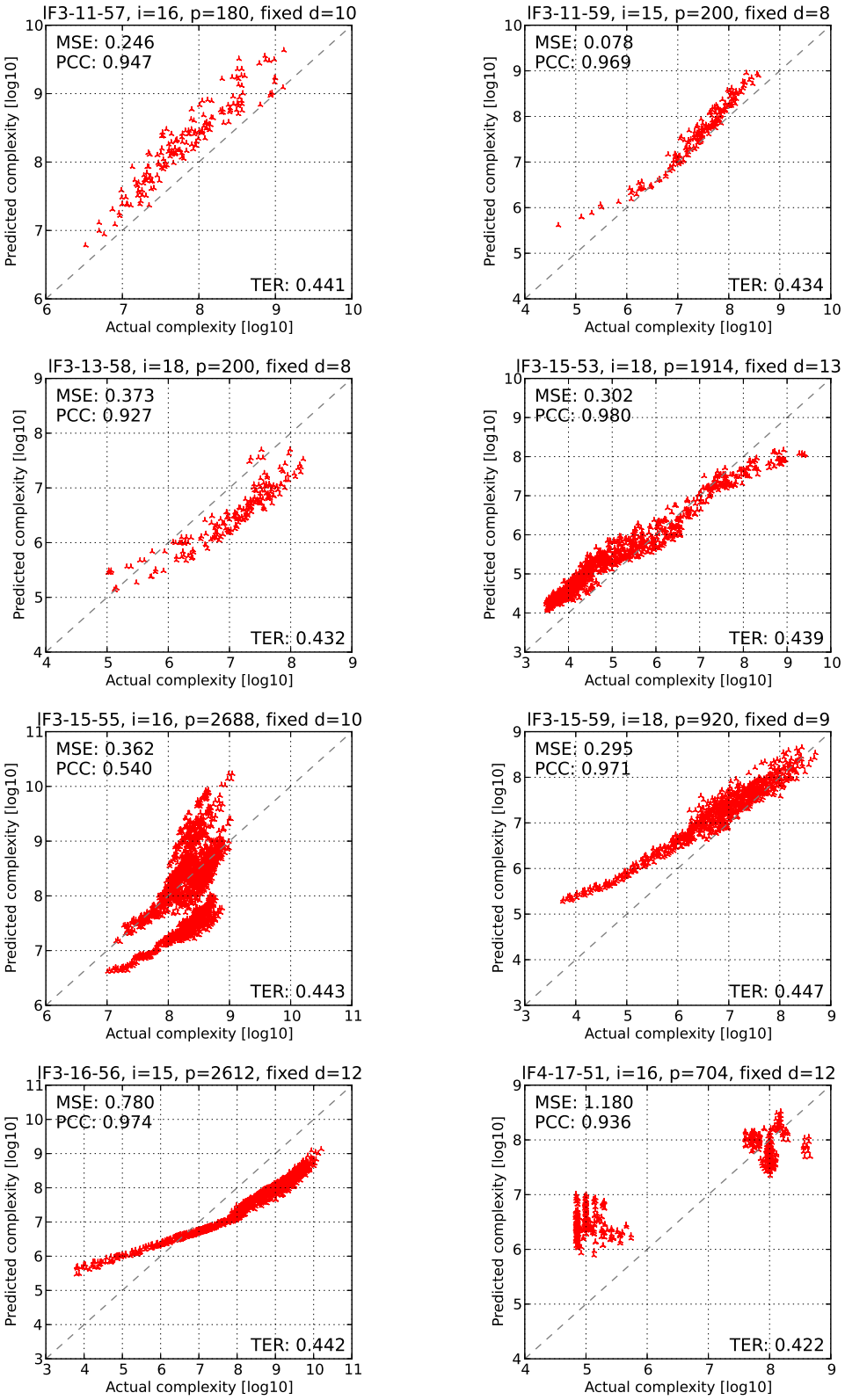


Figure A.10: Cross-class estimation results on largeFam haplotype instances, using instances from all classes for model training.

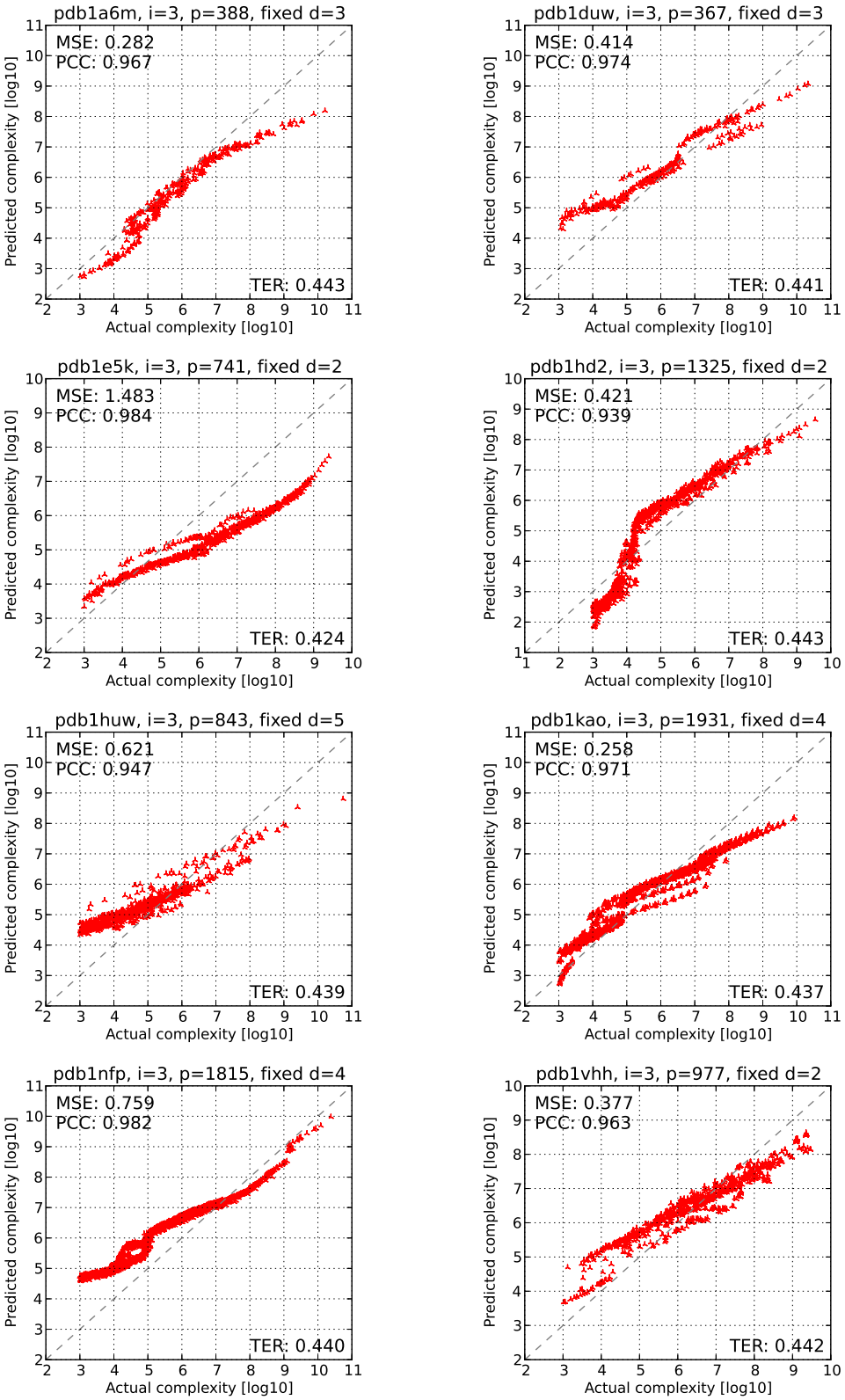


Figure A.11: Cross-class estimation results on pdb side-chain prediction instances, using instances from all classes for model training.

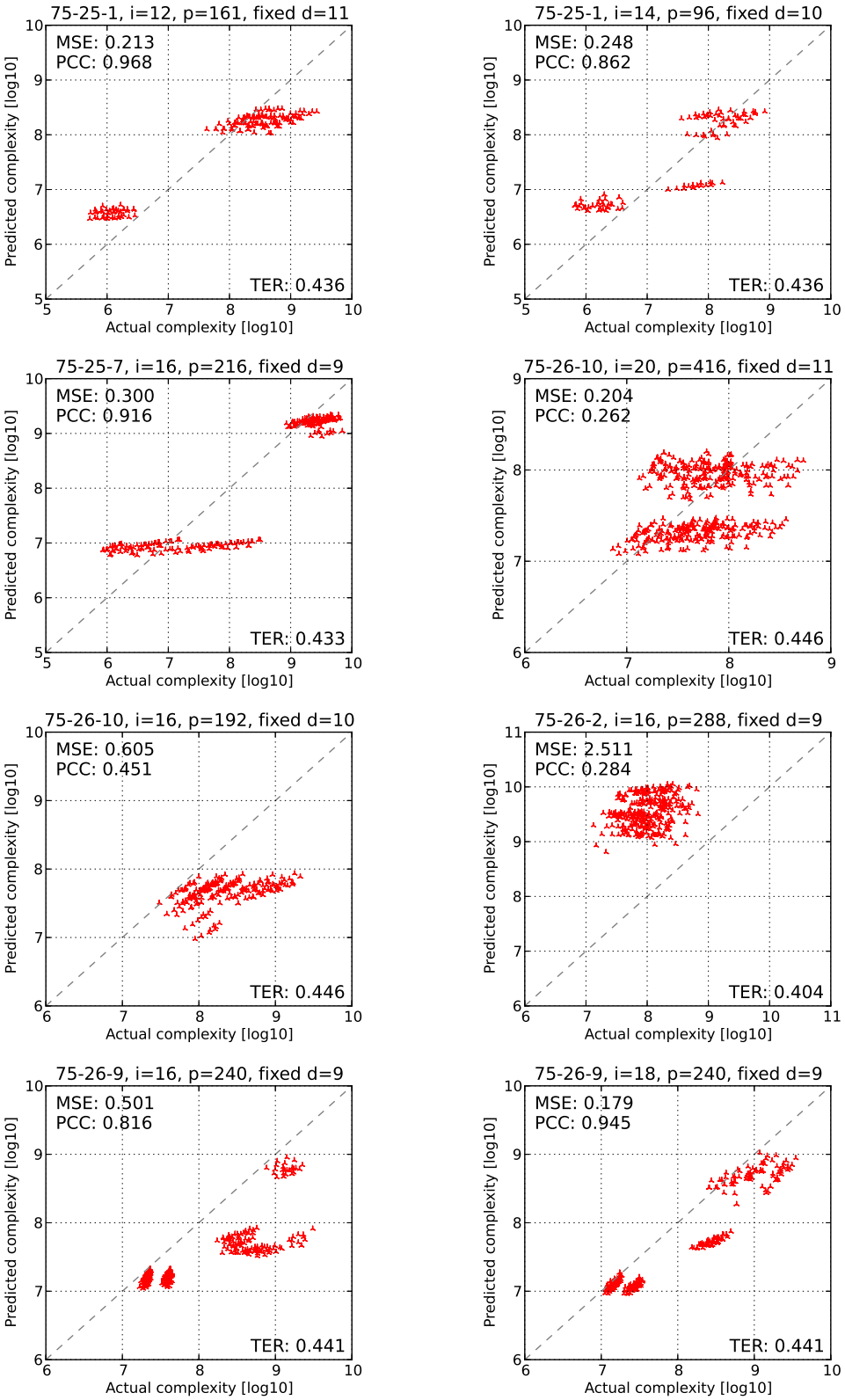


Figure A.12: Cross-class estimation results on grid network instances, using instances from all classes for model training.

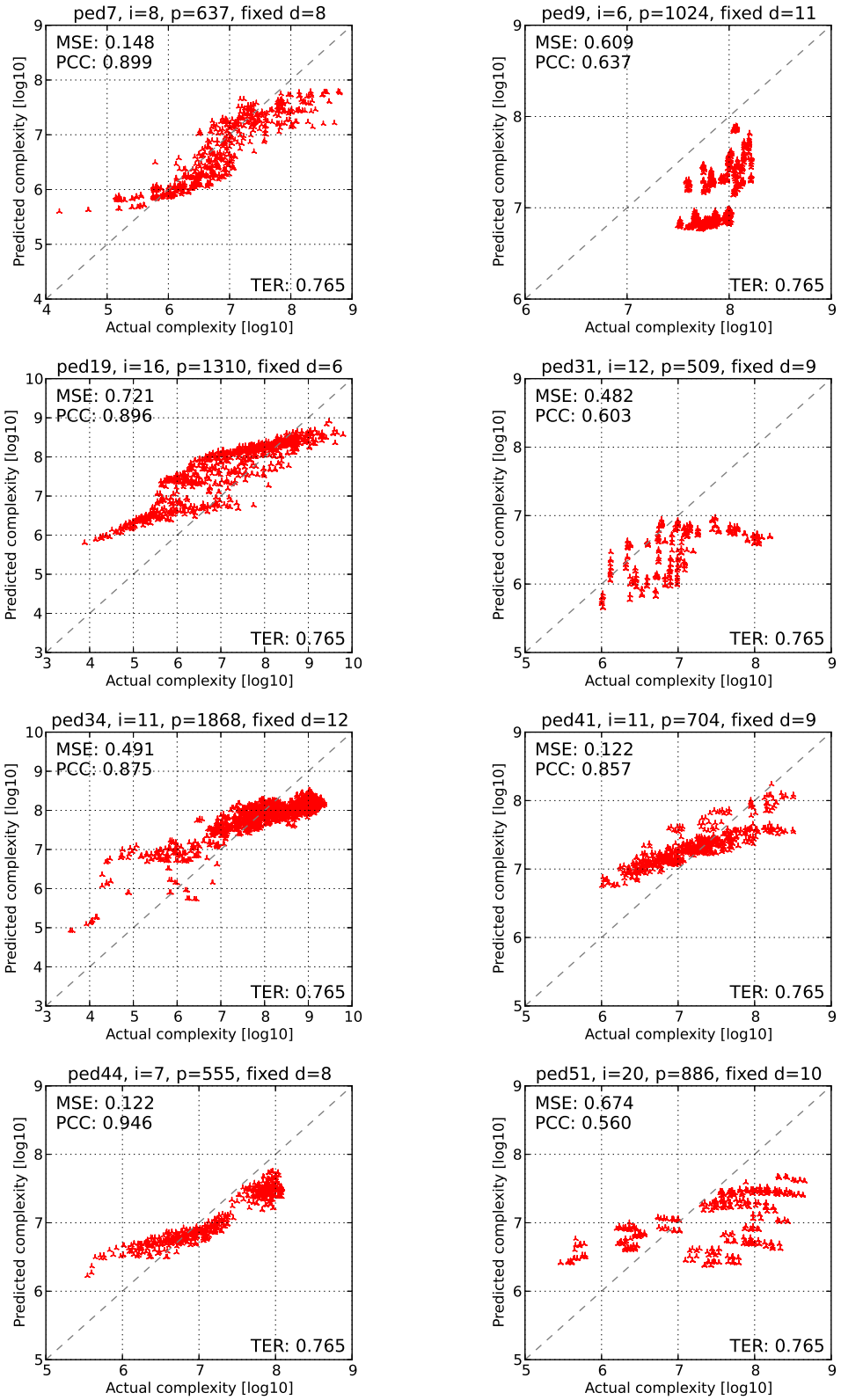


Figure A.13: Unseen-class estimation results on pedigree linkage instances, using instances from all other classes for model training.

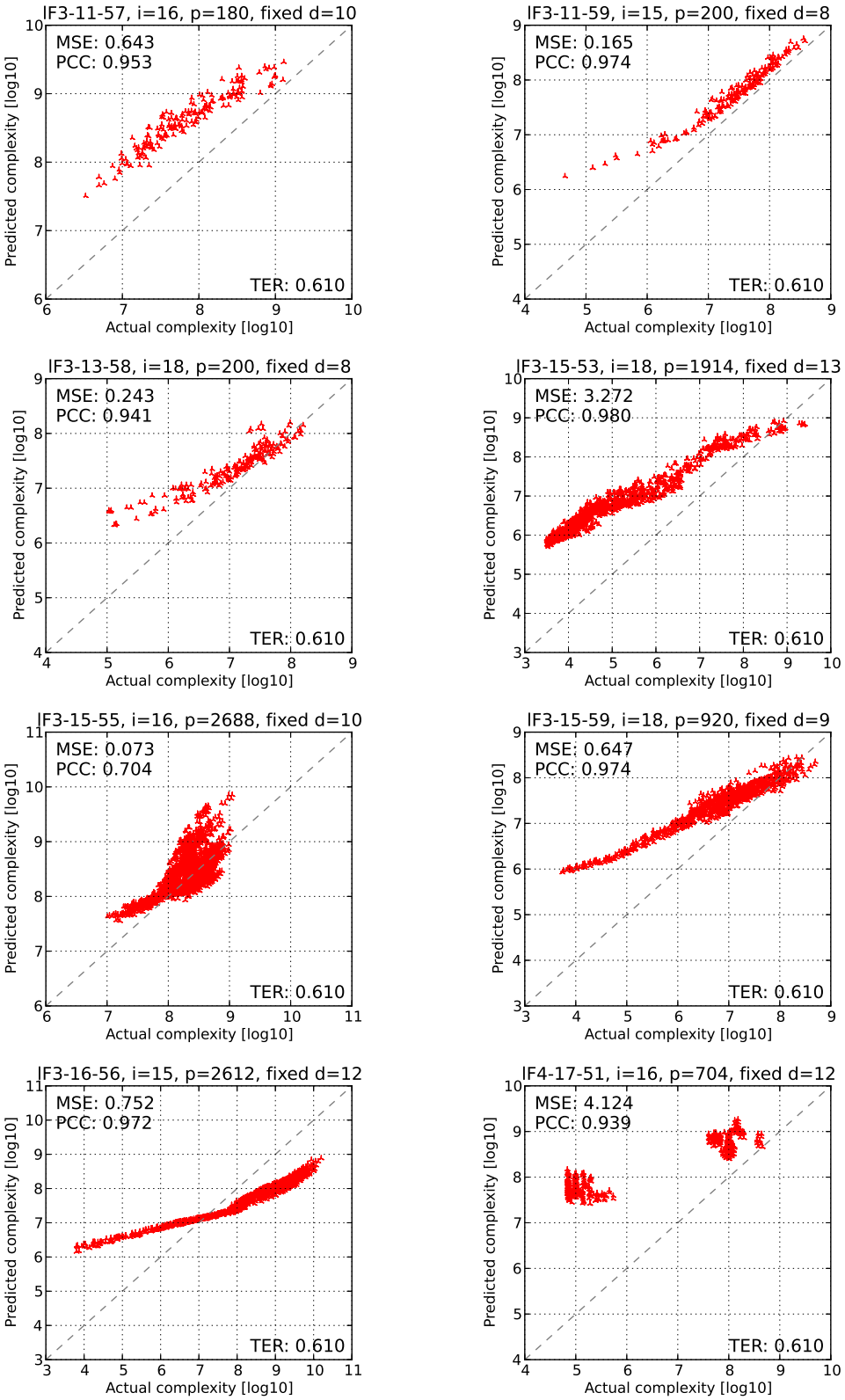


Figure A.14: Unseen-class estimation results on largeFam haplotype instances, using instances from all other classes for model training.

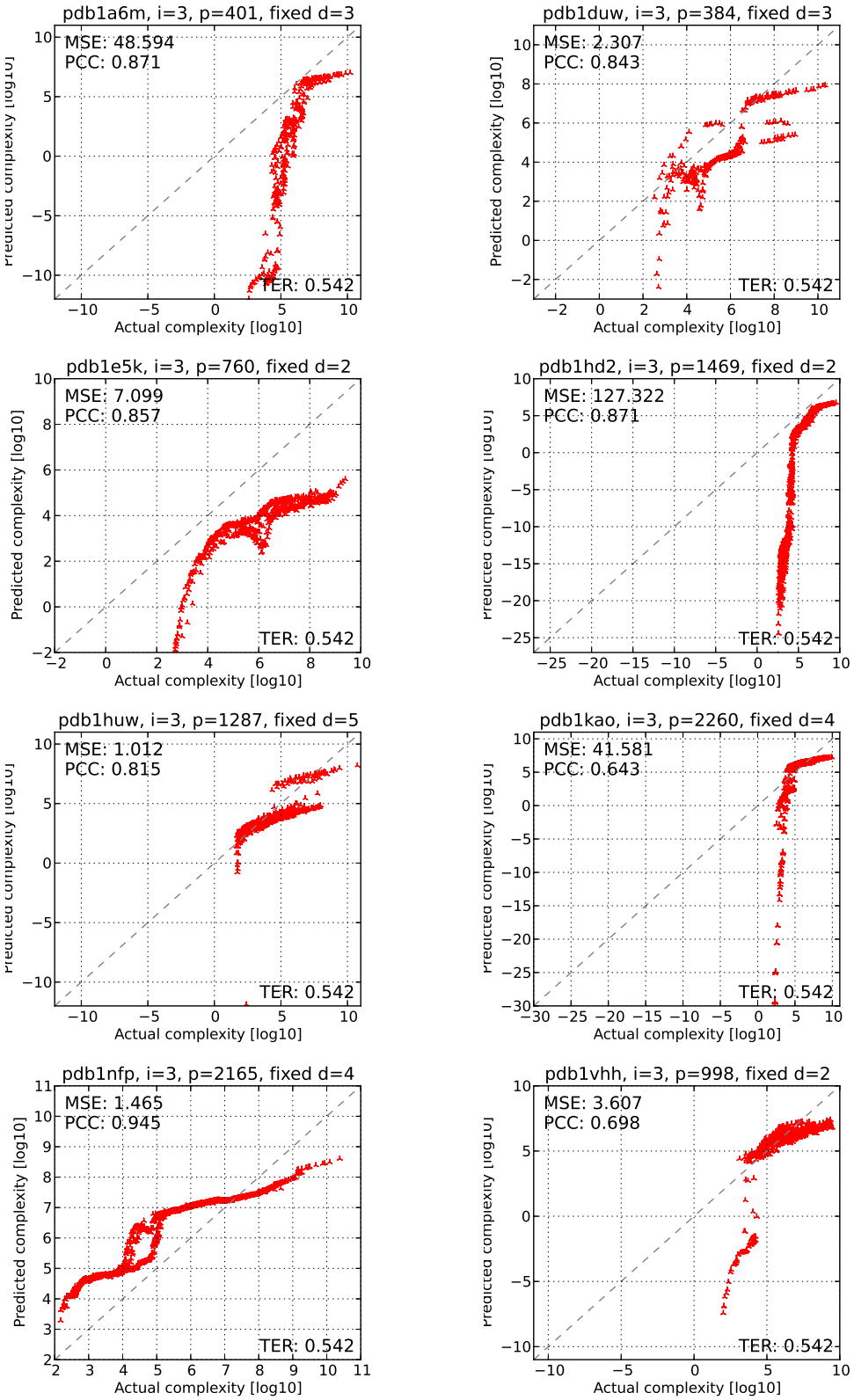


Figure A.15: Unseen-class estimation results on pdb side-chain prediction instances, using instances from all other classes for model training.

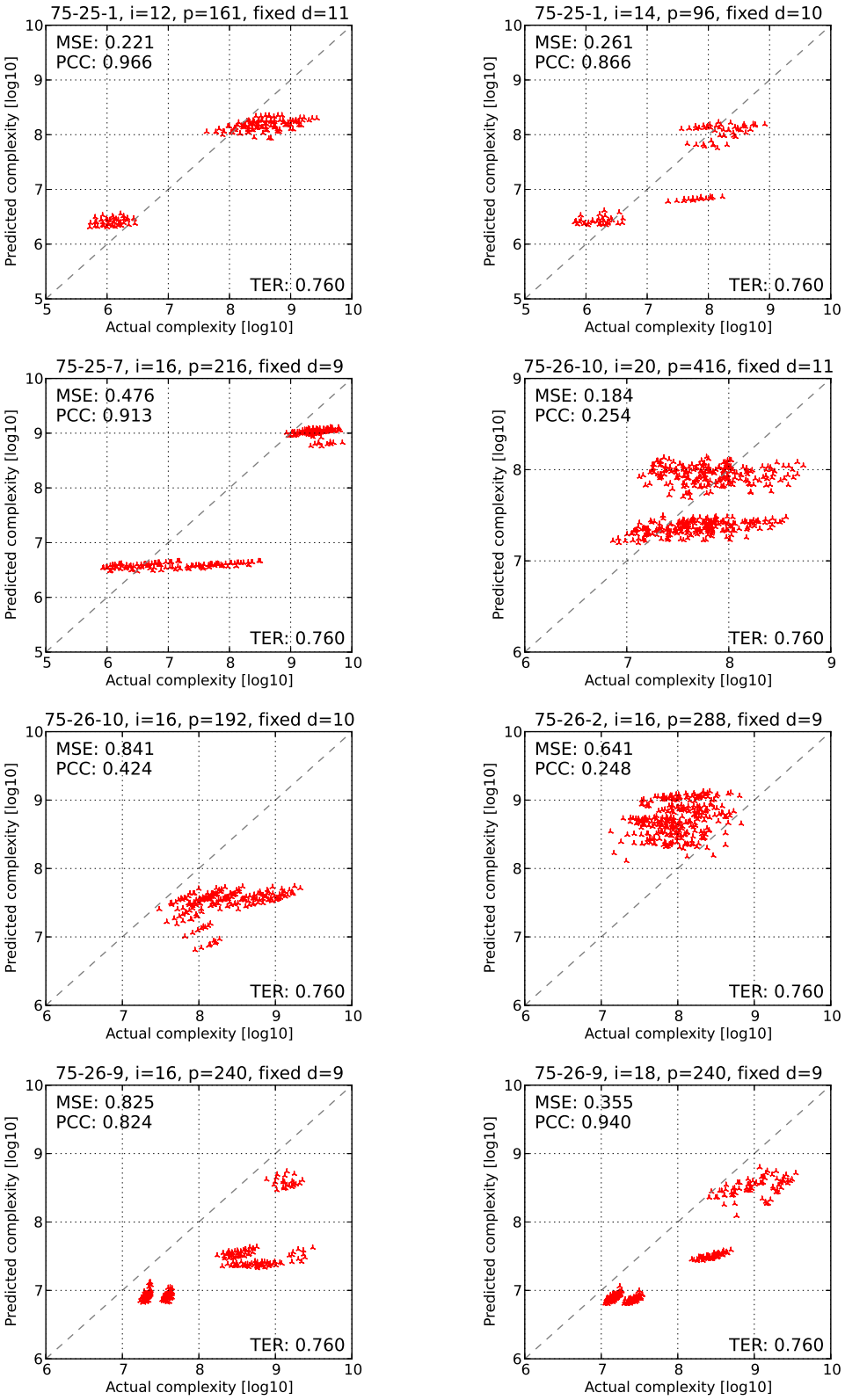


Figure A.16: Unseen-class estimation results on grid network instances, using instances from all other classes for model training.

B Complete Parallel Result Tables

This section contains the complete tables of parallel results, a subset of which was presented in Section 4.6. The experimental setup is describe more in-depth in Section 4.6.1, details regarding the problem instances are given in Section 4.6.2.

As a reminder, for each instance and cutoff depth d, we first run parallel AOBB with fixed-depth parallel cutoff (Algorithm 4.1). The resulting number of subproblems p is then used for the variable-depth scheme (Algorithm 4.2). In all cases, if one scheme is better than the other by more than 10% (relative) its results is marked bold in the following tables. The best entry in each row is highlighted in gray.

B.1 Parallel Preprocessing Times

Preprocessing includes the time needed for mini-bucket heuristic computation as well as determining the parallelization frontier.

- Table B.1 lists preprocessing times on pedigree linkage analysis problem instances.
- Table B.2 lists preprocessing times on largeFam haplotyping problem instances.
- Table B.3 lists preprocessing times on pdb side-chain prediction problem instances.
- Table B.4 lists preprocessing times on grid network problem instances.

B.2 Parallel Runtime Results

- Tables B.5 and B.6 (pages 311 and 312) list full parallel runtime results on pedigree linkage analysis problem instances.

- Tables B.7 and B.8 (pages 313 and 314) list full parallel runtime results on largeFam haplotyping problem instances.
- Table B.9 (page 315) lists full parallel runtime results on protein side-chain prediction problem instances.
- Table B.10 (page 316) lists full parallel runtime results on grid network problem instances.

B.3 Parallel Speedup Results

- Tables B.11 and B.12 (pages 317 and 318) list full parallel speedup results on pedigree linkage analysis problem instances.
- Tables B.13 and B.14 (pages 319 and 320) list full parallel speedup results on largeFam haplotyping problem instances.
- Table B.15 (page 321) lists full parallel speedup results on protein side-chain prediction problem instances.
- Table B.16 (page 322) lists full parallel speedup results on grid network problem instances.

B.4 Average Resource Utilization Results

- Tables B.17 and B.18 (pages 323 and 324) list full parallel resource utilization results on pedigree linkage analysis problem instances.
- Tables B.19 and B.20 (pages 325 and 326) list full parallel resource utilization results on largeFam haplotyping problem instances.

- Table B.21 (page 327) lists full parallel resource utilization results on protein side-chain prediction problem instances.
- Table B.22 (page 328) lists full parallel resource utilization results on grid network problem instances.

instance	n	m	k	w	h	i	T_{seq}	Cutoff depth d																					
								1		2		3		4		5		6		7		8		9		10		11		12		13	
								fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
ped13	1077	1077	3	32	102	8	252654	0	1	1	0	0	0	0	0	0	1	1	1	1	1	2	3	5	5	10	9	19	19	37	39	65	58
						9	102385	0	0	0	0	0	0	0	0	0	0	1	0	2	1	3	3	6	5	13	9	25	19	53	37	61	55
ped19	793	793	5	25	98	16	375110	41	41	41	41	41	41	45	42	44	42	54	50	74	61	111	89	153	109	144	141	198	181				
ped20	437	437	5	22	60	3	5136	0	0	0	0	0	0	0	0	1	0	1	1	1	1	3	2	7	9	18	18						
						4	2185	0	0	0	0	0	0	0	1	0	0	1	0	1	1	2	2	7	8	16	17						
ped31	1183	1183	5	30	85	10	1258519	1	1	0	0	1	0	0	0	1	0	1	0	1	1	3	3	5	5	10	9	19	19	38	37	73	75
						11	433029	0	0	0	0	0	1	1	0	1	1	1	1	2	1	4	3	8	5	13	10	25	20	50	40	77	78
						12	16238	0	0	1	0	1	1	0	1	1	1	1	1	2	2	3	3	5	5	9	9	18	19	34	36	66	70
ped33	798	798	4	28	98	4	6010	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1	1	3	2	4	4	8	9	12	9
ped34	1160	1160	5	31	102	10	962006	0	0	0	1	0	0	0	0	1	0	1	1	1	1	2	2	4	5	8	9	12	11	23	21	41	41
						11	350574	1	1	0	0	1	0	0	0	0	0	1	1	1	1	2	2	4	4	8	8	13	11	24	21	40	40
						12	96122	0	0	0	1	1	0	1	1	1	1	1	1	2	1	3	2	6	4	10	9	15	11	29	21	53	39
ped39	1272	1272	5	21	76	4	6632	0	0	0	0	0	0	0	0	0	0	2	1	3	3	6	6	11	9	19	19	39	38				
						5	2202	0	0	1	0	0	0	1	0	0	1	1	1	2	3	6	6	10	9	19	18	37	36				
ped41	1062	1062	5	33	100	9	25607	0	0	0	0	0	0	0	1	0	1	1	1	2	2	3	3	6	6	11	10	18	17	34	35	65	NA
						10	46819	0	0	0	0	1	0	1	0	1	1	2	1	2	2	4	2	8	5	17	11	27	16	57	34	65	NA
						11	27583	0	0	1	0	0	0	1	0	0	0	1	1	2	2	3	3	5	5	10	11	18	18	34	37	65	NA
ped44	811	811	4	25	65	5	207136	0	0	0	0	0	0	0	0	1	0	0	0	2	2	3	2	7	6	12	12	25	24	49	49	99	98
						6	95830	1	1	0	0	0	0	0	0	1	0	1	0	4	2	7	3	12	6	24	11	49	24	48	48	96	96
ped50	514	514	6	17	47	3	4135	0	0	0	0	0	0	2	0	1	1	5	6	15	14	41	37										
						4	1780	0	0	0	0	0	0	1	0	1	2	5	5	14	14	39	37										
ped51	1152	1152	5	39	98	20	101788	21	21	21	21	21	21	21	21	21	21	21	21	22	22	23	23	25	25	30	29	36	37	52	52	82	84
						21	164817	43	43	44	43	43	43	43	43	43	43	44	44	45	44	47	45	49	48	55	51	71	59	94	75	107	108
ped7	1068	1068	4	32	90	6	118383	0	0	0	0	0	0	0	1	0	1	1	1	4	5	6	6	12	12	15	12	26	24	40	36	70	72
						7	93380	0	0	0	0	0	0	1	0	1	0	2	1	6	5	9	6	17	13	23	13	38	26	61	40	110	85
						8	30717	0	0	1	0	0	1	1	0	1	1	1	1	4	4	6	5	11	12	16	12	27	25	42	38	72	80
ped9	1118	1118	7	27	100	6	101172	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	2	2	3	5	5	10	9	17	19	35	36
						7	58657	1	0	0	0	0	1	0	0	0	1	1	1	1	1	2	1	3	2	7	5	12	8	27	16	47	34
						8	41061	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	2	3	2	5	4	9	8	16	17	33	33

Table B.1: Preprocessing times of fixed-depth and variable-depth parallel AOBB on **linkage instances**.

								Cutoff depth d																									
instance	n	m	k	w	h	i	T_{seq}	1		2		3		4		5		6		7		8		9		10		11		12		13	
								fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
IF3-11-57	2670	2670	3	37	95	15	121311	3	3	4	4	3	3	4	4	4	4	4	4	5	5	5	5	5	5	7	6	8	9	14	18	23	23
						16	35820	6	6	6	6	6	6	6	7	6	6	7	6	7	7	7	7	8	8	9	8	11	10	17	20	24	24
						17	18312	15	15	15	15	15	16	15	16	16	15	16	15	16	16	16	16	17	17	18	17	20	20	26	28	33	32
IF3-11-59	2711	2711	3	32	73	14	35457	4	4	4	4	3	4	4	4	4	4	4	4	5	5	7	6	10	11	15	15	25	27	36	27	58	51
						15	8523	5	5	4	5	4	5	5	5	4	4	5	5	6	6	7	6	10	11	15	15	24	27	34	27	53	49
						16	3023	11	11	10	11	11	11	11	11	11	11	11	12	12	12	13	13	16	17	20	22	30	33	39	33	59	55
IF3-13-58	3352	3352	3	31	88	14	46464	1	1	1	1	1	1	1	1	2	2	2	2	3	3	4	3	8	8	14	14	25	24	45	47	78	76
						16	20270	4	4	4	4	4	4	4	5	5	5	5	5	6	6	7	6	10	10	16	17	27	25	47	47	80	74
						18	7647	21	21	20	20	21	20	21	20	21	21	21	21	22	23	24	23	26	27	33	33	44	41	63	62	96	89
IF3-15-53	3384	3384	3	32	108	17	345544	7	7	7	7	7	7	7	7	7	7	8	7	8	8	10	10	12	13	16	17	24	24	39	38	61	55
						18	98346	15	15	16	15	15	15	15	16	16	15	16	15	16	16	18	18	19	19	23	23	29	29	40	39	51	55
IF3-15-59	3730	3730	3	31	84	18	28613	8	8	8	8	8	8	8	8	8	8	9	9	11	11	13	13	20	19	30	30	52	53	95	95	180	183
						19	43307	14	14	13	13	13	13	14	13	13	14	14	14	15	16	18	19	23	25	36	36	56	58	100	103	183	190
IF3-16-56	3930	3930	3	38	77	15	1891710			8	8	8	8	8	9	8	8	9	8	11	11	14	14	21	22	28	22	41	35	59	46	115	107
						16	489614	21	21	21	21	21	21	21	21	22	22	22	22	24	23	27	27	34	34	39	34	52	47	76	59	123	113
IF4-12-50	2569	2569	4	28	80	13	57842	7	7	7	7	7	6	7	7	9	8	12	11	25	28	48	43										
						14	33676	5	5	5	5	5	4	6	5	6	7	12	10	24	27	47	42										
IF4-12-55	2926	2926	4	28	78	13	104837	3	3	2	3	3	3	3	3	3	4	4	4	6	6	8	10	13	17	18	17	27	30	35	30	51	54
						14	25905	8	8	8	7	8	8	8	8	8	8	8	9	9	10	11	13	17	18	19	18	26	32	35	32	48	58
IF4-17-51	3837	3837	4	29	85	15	10607	8	8	8	8	8	8	8	8	8	8	8	8	9	8	9	8	10	10	11	11	12	11	15	15	17	16
						16	66103	13	13	14	13	13	14	14	14	14	13	14	14	14	14	15	15	17	18	19	19	21	20	26	26	30	27

Table B.2: Preprocessing times of fixed-depth and variable-depth parallel AOBB on **haplotyping instances**.

								Cutoff depth d												
instance	n	m	k	w	h	i	T_{seq}	1		2		3		4		5		6	
								fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
pdb1a6m	124	521	81	15	34	3	198326	3	3	3	3	7	4						
pdb1duw	241	743	81	9	32	3	627106	4	4	3	4	4	4	19	15				
pdb1e5k	154	587	81	12	43	3	112654	4	4	10	4	11	10						
pdb1f9i	103	387	81	10	24	3	68804	2	2	40	4								
pdb1ft5	172	645	81	14	33	3	81118	15	15	16	16	53	20						
pdb1hd2	126	448	81	12	27	3	101550	6	6	33	8								
pdb1huw	152	587	81	15	43	3	545249	9	9	8	9	9	9	10	9	11	10	13	11
pdb1kao	148	568	81	15	41	3	716795	6	6	6	6	6	6	8	7				
pdb1nfp	204	791	81	18	38	3	354720	3	3	3	3	3	3	6	6				
pdb1rss	115	448	81	12	35	3	378579	16	16	16	17	16	17	15	17				
pdb1vhh	133	556	81	14	35	3	944633	12	12	12	13	50	53						

Table B.3: Preprocessing times of fixed-depth and variable-depth parallel AOBB on **side-chain prediction instances**.

instance	n	m	k	w	h	i	T_{seq}	Cutoff depth d																											
								1		2		3		4		5		6		7		8		9		10		11		12		13		14		15	
								fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
75-25-1	624	626	2	38	111	12	77941	0	0	1	0	0	0	0	1	1	1	0	0	0	0	1	2	2	2	2	2	3	2	4	4	5	6	9	9	15	15
						14	15402	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0	2	1	2	2	2	2	3	4	4	7	7	10	12
75-25-3	624	626	2	37	115	12	104037	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	2	3	2	4	4	7	8	12	11
						15	33656	1	1	0	1	1	1	0	0	1	0	0	0	1	1	1	0	1	0	1	1	2	1	2	2	4	4	7	7	11	11
75-25-7	624	626	2	37	120	16	297377	1	1	1	1	0	0	1	0	1	1	1	1	1	1	1	2	3	3	3	3	5	5	8	8	13	20	22	27	31	36
						18	21694	2	2	2	2	3	3	2	2	3	2	2	2	2	3	3	3	4	4	5	4	6	6	9	11	14	19	21	27	30	35
75-26-10	675	677	2	39	124	16	46985	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	2	1	1	2	2	3	3	4	3	6	6	10	11	12	11
						18	26855	2	2	2	3	2	3	2	3	2	2	3	2	3	2	3	3	3	3	4	4	5	5	6	5	8	9	11	13	15	13
75-26-2	675	677	2	39	120	16	25274	1	1	1	1	1	1	1	0	1	1	1	1	2	2	2	2	3	4	4	4	6	6	11	12	15	12	24	25	37	38
						20	8053	7	7	7	7	7	7	7	7	8	7	8	8	8	8	9	8	9	9	10	10	12	12	16	19	21	19	29	31	41	44
75-26-6	675	677	2	39	133	10	199460	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	2	1	3	3	4	4	6	6	10	10	17	18	33	37
						12	64758	0	0	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	2	2	3	3	5	5	9	9	16	17	30	38
75-26-9	675	677	2	39	124	16	59609	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	2	3	2	4	4	6	7	11	13	21	27	31	27	50	53
						18	66533	2	2	2	3	2	2	2	2	2	2	2	2	3	2	3	3	3	3	5	5	8	9	13	17	22	33	33	33	54	64
						20	5708	8	8	9	8	8	8	8	8	8	8	8	8	9	9	9	9	10	10	11	10	12	12	16	18	22	28	29	28	43	47

Table B.4: Preprocessing times of fixed-depth and variable-depth parallel AOBB on **grid instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
ped13 n=1077 k=3 w=32 h=102	8	252654	20	133286	139596	**69461**	136558	**34982**	137281	**19125**	70897	**18615**	35261	**16290**	24717	**15655**	21913	15228	**13822**	14545	14071	14074	13883	14859	13985	14734	14486	14680	14520
			100	133286	139596	**69461**	136558	**34982**	137281	**19125**	70897	**11684**	35261	**5945**	18836	**5808**	19106	**4867**	11027	**4205**	5157	**3603**	3979	3768	**3047**	3713	3620	3712	**3277**
			500	133286	139596	**69461**	136558	**34982**	137281	**19125**	70897	**11684**	35261	**5945**	18836	**5041**	19106	**3339**	11027	**2430**	5157	**1741**	2816	2008	1871	1516	1608	1525	**1176**
			∞	*133286*	*139596*	***69461***	*136558*	***34982***	*137281*	***19125***	*70897*	***11684***	*35261*	***5945***	*18836*	***5041***	*19106*	***3339***	*11027*	***2430***	*5157*	***1658***	*2816*	*1774*	*1871*	***1139***	*1399*	*1140*	*1125*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
	9	102385	20	53547	54036	30925	31659	**15869**	23811	**10539**	11993	**9440**	12041	**5806**	9706	**5214**	6891	**5111**	5691	5327	5244	5223	5573	5547	5319	5656	5574	5905	5747
			100	53547	54036	30925	31659	**15869**	23811	**10539**	11993	**7148**	12041	**3481**	9706	**3318**	6891	**1941**	3237	1829	1934	**1337**	2082	1648	**1436**	1463	1546	1532	1458
			500	53547	54036	30925	31659	**15869**	23811	**10539**	11993	**7148**	12041	**3481**	9706	**3318**	6891	**1682**	3202	1523	1564	**835**	1764	857	880	824	841	850	831
			∞	*53547*	*54036*	*30925*	*31659*	***15869***	*23811*	***10539***	*11993*	***7148***	*12041*	***3481***	*9706*	***3318***	*6891*	***1682***	*3202*	*1523*	*1564*	***807***	*1764*	***739***	*822*	*752*	*765*	*776*	*770*
ped19 n=793 k=5 w=25 h=98	16	375110		*(p=4)*		*(p=12)*		*(p=48)*		*(p=144)*		*(p=288)*		*(p=1440)*		*(p=2880)*		*(p=5752)*		*(p=7672)*		*(p=11254)*		*(p=14968)*					
			20	180120	184438	119407	**73960**	66165	**41630**	30976	**28027**	25924	26562	27290	28435	29811	31468	35187	37384	39868	41288	47882	48784	54845	56214				
			100	180120	184438	119407	**73960**	66165	**39990**	30976	**20593**	17294	**12489**	12125	**7294**	8340	7807	7492	7947	8141	8360	9705	9887	11147	11401				
			500	180120	184438	119407	**73960**	66165	**39990**	30976	**20593**	17294	**12489**	11930	**5158**	6716	**3474**	5222	**3301**	3454	**2983**	2912	**2381**	3061	**2465**				
			∞	*180120*	*184438*	*119407*	***73960***	*66165*	***39990***	*30976*	***20593***	*17294*	***12489***	*11930*	***5158***	*6716*	***2738***	*5079*	***2639***	*3170*	***2595***	*2709*	***1934***	*2687*	***1638***				
				(p=2)		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
ped20 n=437 k=5 w=22 h=60	3	5136	20	2461	2461	1361	1392	739	717	454	423	420	416	395	433	468	455	514	505	804	817	1164	1167						
			100	2461	2461	1361	1392	739	717	383	381	231	235	**111**	238	**132**	163	127	133	169	176	249	252						
			500	2461	2461	1361	1392	739	717	383	381	231	235	**95**	238	**71**	134	**52**	133	**46**	54	67	72						
			∞	*2461*	*2461*	*1361*	*1392*	*739*	*717*	*383*	*381*	*231*	*235*	***95***	*238*	***71***	*134*	***41***	*133*	***27***	*48*	***32***	*43*						
				(p=2)		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
	4	2185	20	571	571	229	227	**118**	192	**81**	90	83	78	79	75	114	112	149	149	395	401	733	728						
			100	571	571	229	227	**118**	192	**51**	90	30	31	**21**	24	28	**24**	34	32	86	88	161	160						
			500	571	571	229	227	**118**	192	**51**	90	30	31	**15**	24	12	12	12	11	25	25	46	47						
			∞	*571*	*571*	*229*	*227*	***118***	*192*	***51***	*90*	*30*	*31*	***15***	*24*	*12*	*12*	***10***	*11*	*11*	*12*	*20*	*20*						
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	10	1258519	20	711530	711530	360625	358220	187151	170419	104694	103211	106669	**91276**	103639	**89404**	92364	**83492**	85986	81762	85658	81048	81513	80922	80859	80611	82833	81644	84114	84426
			100	711530	711530	360625	358220	187151	170419	104694	103211	**60463**	84793	48472	49703	**33396**	36791	30712	**26508**	24505	**18669**	19653	**17786**	17498	16711	17082	16784	17462	17430
			500	711530	711530	360625	358220	187151	170419	104694	103211	**60463**	84793	48472	49703	**28510**	36791	23569	**17423**	12522	**9981**	8172	**5668**	5756	**4111**	4607	**3929**	4246	3971
			∞	*711530*	*711530*	*360625*	*358220*	*187151*	*170419*	*104694*	*103211*	***60463***	*84793*	*48472*	*49703*	***28510***	*36791*	*23569*	***17423***	*12522*	***9981***	*7100*	***5118***	*4008*	***2969***	*2033*	***1584***	*1374*	***1032***
ped31 n=1183 k=5 w=30 h=85				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	11	433029	20	195317	195317	107463	108489	54365	54458	30376	29491	33632	**28823**	31225	**26680**	22498	24712	22809	24087	23045	24312	24048	24254	23950	24723	25193	25315	26527	26400
			100	195317	195317	107463	108489	54365	54458	30376	29491	**19267**	28823	15695	15780	8873	9263	7333	7668	**4877**	5923	5748	**5069**	5393	5218	5367	5144	5698	5405
			500	195317	195317	107463	108489	54365	54458	30376	29491	**19267**	28823	15695	15780	8873	9263	7333	7668	4073	**3382**	2384	**1820**	1746	**1372**	1473	1360	1674	1526
			∞	*195317*	*195317*	*107463*	*108489*	*54365*	*54458*	*30376*	*29491*	***19267***	*28823*	*15695*	*15780*	*8873*	*9263*	*7333*	*7668*	*4073*	***3382***	*2141*	***1820***	*1082*	***834***	*1083*	***818***	*985*	***834***
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	12	16238	20	8097	8097	3758	4046	2355	**2098**	1298	1224	1342	1472	1463	**1126**	1406	**1028**	1086	**929**	1091	**938**	1055	1037	1165	1172	1444	1447	1974	1971
			100	8097	8097	3758	4046	2355	**2098**	1298	1224	**820**	1472	1176	**811**	784	**471**	437	**393**	356	**262**	287	**233**	276	256	325	319	452	451
			500	8097	8097	3758	4046	2355	**2098**	1298	1224	**820**	1472	1176	**811**	771	**471**	406	393	225	225	139	131	106	**80**	115	**95**	160	148
			∞	*8097*	*8097*	*3758*	*4046*	*2355*	***2098***	*1298*	*1224*	***820***	*1472*	*1176*	***811***	*771*	***471***	*406*	*393*	*225*	*225*	*127*	*131*	*89*	***74***	*70*	***63***	*92*	*88*
ped33 n=798 k=4 w=28 h=98				*(p=2)*		*(p=3)*		*(p=6)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=1536)*		*(p=1536)*	
	4	6010	20	3975	3975	3071	3238	1668	1521	1645	1521	841	**712**	649	**486**	541	**426**	478	467	474	**406**	467	434	535	490	594	597	610	597
			100	3975	3975	3071	3238	1668	1521	1645	1521	841	**712**	519	**367**	365	**323**	252	**173**	236	**123**	173	**112**	176	**119**	159	**139**	172	**139**
			500	3975	3975	3071	3238	1668	1521	1645	1521	841	**712**	519	**367**	365	**323**	252	**173**	209	**123**	124	**91**	108	**71**	87	**59**	88	**59**
			∞	*3975*	*3975*	*3071*	*3238*	*1668*	*1521*	*1645*	*1521*	*841*	***712***	*519*	***367***	*365*	***323***	*252*	***173***	*209*	***123***	*124*	***91***	*101*	***71***	*77*	***49***	*78*	***49***
				(p=3)		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=952)*		*(p=1896)*		*(p=3752)*	
	10	962006	20	490747	490747	424691	424270	280891	275499	175178	**144147**	145176	143540	145741	**122690**	104354	101243	109138	**93405**	98912	95149	98912	97309	102778	97530	97118	96468	97005	96170
			100	490747	490747	424691	424270	280891	275499	175178	**144147**	144552	143540	115446	**75577**	75563	75573	42110	39354	32808	**24178**	27212	**21134**	27151	**21039**	21438	21187	20390	19432
			500	490747	490747	424691	424270	280891	275499	175178	**144147**	144552	143540	115446	**75577**	75563	75573	41663	39354	24752	**20572**	13890	**11203**	11491	10730	6670	6136	5260	4882
			∞	*490747*	*490747*	*424691*	*424270*	*280891*	*275499*	*175178*	***144147***	*144552*	*143540*	*115446*	***75577***	*75563*	*75573*	*41663*	*39354*	*24752*	***20572***	*13680*	***11203***	*11033*	*10730*	*5773*	*5926*	*3260*	*3044*
ped34 n=1160 k=5 w=31 h=102				*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=720)*		*(p=956)*		*(p=1912)*		*(p=3808)*	
	11	350574	20	271792	271792	216931	217171	**156741**	218420	**103935**	155893	**96190**	108730	95723	95475	**85044**	95135	80649	74447	72536	71327	74526	76593	80137	76424	76817	76397	76963	76469
			100	271792	271792	216931	217171	**156741**	218420	**103935**	155893	95443	95987	79842	79099	51909	51305	30073	29976	23264	23301	19172	18735	19242	18813	16714	16145	16234	15655
			500	271792	271792	216931	217171	**156741**	218420	**103935**	155893	95443	95987	79842	79099	51909	51305	30042	29976	16019	16121	10423	**9238**	9443	**8312**	5282	5692	4505	**3877**
			∞	*271792*	*271792*	*216931*	*217171*	***156741***	*218420*	***103935***	*155893*	*95443*	*95987*	*79842*	*79099*	*51909*	*51305*	*30042*	*29976*	*16019*	*16121*	*10423*	***9238***	*9326*	***8312***	*4666*	***4194***	*2499*	***2155***
				(p=3)		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=948)*		*(p=1896)*		*(p=3728)*	
	12	96122	20	42753	43233	26241	26328	23300	23569	20456	**15283**	15305	15227	15829	15743	15980	**13462**	13996	13785	12204	12681	12819	13081	13294	13104	12979	13421	13568	13891
			100	42753	43233	26241	26328	23300	23569	20456	**15283**	15283	15227	13298	**9028**	9383	9107	5402	5648	4086	**3706**	3489	**2971**	3439	**2883**	2887	2972	2955	2956
			500	42753	43233	26241	26328	23300	23569	20456	**15283**	15283	15227	13298	**9028**	9383	9107	5390	5648	3302	**2841**	2100	**1298**	1718	**1089**	1014	988	**898**	1119
			∞	*42753*	*43233*	*26241*	*26328*	*23300*	*23569*	*20456*	***15283***	*15283*	*15227*	*13298*	***9028***	*9383*	*9107*	*5390*	*5648*	*3302*	***2841***	*2096*	***1298***	*1715*	***1089***	*916*	***625***	***623***	*972*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
ped39 n=1272 k=5 w=21 h=76	4	6632	20	3070	3070	2690	2731	2661	2546	1494	1503	736	**610**	697	728	834	**727**	709	**566**	666	**556**	672	677	928	937				
			100	3070	3070	2690	2731	2661	2546	1494	1503	669	**606**	596	571	591	**504**	516	**318**	353	**237**	252	**190**	265	**234**				
			500	3070	3070	2690	2731	2661	2546	1494	1503	669	**606**	596	571	568	**485**	491	**304**	311	**202**	181	**129**	160	**133**				
			∞	*3070*	*3070*	*2690*	*2731*	*2661*	*2546*	*1494*	*1503*	*669*	***606***	*596*	*571*	*568*	***485***	*491*	***304***	*309*	***200***	*169*	***121***	*141*	***116***				
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
	5	2202	20	1030	1030	793	867	750	756	409	**359**	420	**368**	421	**292**	406	**268**	422	**306**	414	**329**	443	452	709	723				
			100	1030	1030	793	867	750	756	409	**359**	345	336	317	292	307	**166**	278	**141**	204	**113**	156	**122**	203	**176**				
			500	1030	1030	793	867	750	756	409	**359**	345	336	312	292	296	**156**	253	**111**	166	**110**	103	**70**	109	**76**				
			∞	*1030*	*1030*	*793*	*867*	*750*	*756*	*409*	***359***	*345*	*336*	*312*	*292*	*296*	***156***	*252*	***109***	*161*	***110***	*94*	***70***	*87*	***65***				
Better by 10%				0x	0x	4x	4x	16x	8x	16x	16x	18x	14x	14x	21x	15x	21x	10x	20x	6x	27x	7x	26x	1x	25x	1x	10x	2x	9x
Better by 50%				0x	0x	4x	4x	12x	4x	7x	3x	10x	0x	13x	6x	8x	9x	8x	8x	3x	7x	5x	3x	0x	4x	0x	1x	1x	1x

Table B.5: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8556)*	
ped41 n=1062 k=5 w=33 h=100	9	25607	20	20030	20030	16193	**14319**	9067	**4598**	4737	**2882**	2872	**2162**	2193	2186	1997	2085	2445	**2118**	2262	2269	2427	2457	2603	2674	2972	2982	3542	NA
			100	20030	20030	16193	**14319**	9067	**4598**	4565	**2436**	2410	**1256**	1307	**793**	848	809	827	**576**	508	503	545	520	**591**	877	805	**666**	764	NA
			500	20030	20030	16193	**14319**	9067	**4598**	4565	**2436**	2410	**1256**	1272	**793**	817	**611**	650	**340**	347	**265**	251	211	**337**	604	446	**225**	264	NA
			∞	*20030*	*20030*	*16193*	***14319***	*9067*	***4598***	*4565*	***2436***	*2410*	***1256***	*1272*	***793***	*817*	***611***	*650*	***340***	*347*	***265***	*222*	***172***	***301***	*553*	*401*	***148***	*184*	*NA*
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8576)*	
	10	46819	20	38982	39015	33309	**17287**	17767	**9186**	9679	**4851**	5031	**3577**	3613	3331	3212	3299	3307	3199	2951	3014	3059	3237	3177	3250	3507	3538	4104	NA
			100	38982	39015	33309	**17287**	17767	**9186**	9573	**4851**	4707	**2909**	2762	**1711**	1633	**1478**	1483	**1175**	977	996	**904**	1032	920	**783**	**751**	866	1034	NA
			500	38982	39015	33309	**17287**	17767	**9186**	9573	**4851**	4707	**2909**	2728	**1711**	1595	**1345**	1274	1175	808	785	**705**	999	600	580	519	504	642	NA
			∞	*38982*	*39015*	*33309*	***17287***	*17767*	***9186***	*9573*	***4851***	*4707*	***2909***	*2728*	***1711***	*1595*	***1345***	*1274*	*1175*	*808*	*785*	***694***	*999*	*570*	*574*	*514*	***435***	*567*	*NA*
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8460)*	
	11	27583	20	23060	23060	20222	**10169**	10263	**6033**	6101	**3877**	3263	**2031**	2189	**1866**	2067	**1771**	2051	**1784**	2002	**1782**	1988	1964	2209	2036	2268	2266	2796	NA
			100	23060	23060	20222	**10169**	10263	**6033**	6039	**3877**	3066	**1580**	1639	**1099**	1208	**821**	1010	**553**	852	**455**	**548**	794	909	**660**	548	**491**	643	NA
			500	23060	23060	20222	**10169**	10263	**6033**	6039	**3877**	3066	**1580**	1615	**1099**	1183	**821**	900	**497**	688	**284**	**292**	607	672	660	259	**179**	279	NA
			∞	*23060*	*23060*	*20222*	***10169***	*10263*	***6033***	*6039*	***3877***	*3066*	***1580***	*1615*	***1099***	*1183*	***821***	*900*	***497***	*688*	***284***	***270***	*581*	*642*	*660*	*212*	***117***	*216*	*NA*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
ped44 n=811 k=4 w=25 h=65	5	207136	20	112293	112293	65724	65934	**35680**	65496	**20396**	31287	17026	18552	12719	13425	14321	14295	17214	16269	18655	18339	20630	20412	25054	23501	29114	**25629**	34070	**28744**
			100	112293	112293	65724	65934	**35680**	65496	**20396**	31287	**11906**	18552	**3415**	10659	3553	3463	4040	**3600**	3942	3959	4281	4551	5113	5551	5919	5481	6919	**5992**
			500	112293	112293	65724	65934	**35680**	65496	**20396**	31287	**11906**	18552	**3415**	10659	**2080**	2622	**1554**	1868	**1033**	1279	**1000**	1405	**1182**	2167	**1270**	1584	1489	1569
			∞	*112293*	*112293*	*65724*	*65934*	***35680***	*65496*	***20396***	*31287*	***11906***	*18552*	***3415***	*10659*	***2080***	*2622*	***1371***	*1868*	***604***	*1041*	***366***	*1002*	***564***	*1840*	***737***	*921*	***202***	*840*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
	6	95830	20	52643	52643	26776	26836	15267	15340	9716	9481	9503	**6399**	6741	6811	7308	7330	7959	7947	9084	8749	10103	9763	10586	10828	12418	12472	16551	**14725**
			100	52643	52643	26776	26836	15267	15340	9716	9481	5939	5968	**2344**	3586	1852	**1636**	1799	1700	1957	1861	2126	2276	2204	2273	2545	2543	3400	**3035**
			500	52643	52643	26776	26836	15267	15340	9716	9481	5939	5968	**1659**	3586	**1016**	1352	**583**	886	546	571	**536**	905	**525**	794	**569**	824	771	**699**
			∞	*52643*	*52643*	*26776*	*26836*	*15267*	*15340*	*9716*	*9481*	*5939*	*5968*	***1659***	*3586*	***1016***	*1352*	***459***	*886*	***269***	*425*	***433***	*774*	***235***	*680*	***103***	*696*	***135***	*229*
				(p=2)		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14401)*											
ped50 n=514 k=6 w=17 h=47	3	4135	20	2376	2376	1485	1477	594	587	423	**345**	367	344	465	451	825	820	1734	1730										
			100	2376	2376	1485	1477	594	587	345	345	254	**131**	198	**106**	200	**180**	381	378										
			500	2376	2376	1485	1477	594	587	345	345	236	**88**	159	**88**	125	**55**	127	**111**										
			∞	*2376*	*2376*	*1485*	*1477*	*594*	*587*	*345*	*345*	*236*	***84***	*153*	***88***	*109*	***42***	*89*	***53***										
				(p=2)		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14400)*											
	4	1780	20	499	499	272	255	77	76	75	**67**	125	127	273	277	652	648	1551	1549										
			100	499	499	272	255	77	76	42	41	38	**30**	62	61	143	142	342	340										
			500	499	499	272	255	77	76	42	41	27	**20**	27	**19**	42	41	101	99										
			∞	*499*	*499*	*272*	*255*	*77*	*76*	*42*	*41*	*27*	***20***	*22*	***17***	*26*	***19***	*46*	*42*										
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4064)*		*(p=7968)*	
ped51 n=1152 k=5 w=39 h=98	20	101788	20	52571	52571	27299	27269	13531	13725	8261	7697	8536	**6638**	7051	**6225**	6362	5859	6404	5885	6354	6090	6573	6570	7678	7596	9899	9866	13946	14039
			100	52571	52571	27299	27269	13531	13725	8261	7697	5692	5530	3457	**2658**	2714	**2441**	2340	**1687**	1616	1711	**1578**	2208	1831	**1560**	2186	2025	2957	2884
			500	52571	52571	27299	27269	13531	13725	8261	7697	5692	5530	3457	**2658**	2714	**2441**	1772	**1163**	914	894	**704**	1525	732	729	852	**748**	**762**	944
			∞	*52571*	*52571*	*27299*	*27269*	*13531*	*13725*	*8261*	*7697*	*5692*	*5530*	*3457*	***2658***	*2714*	***2441***	*1772*	***1163***	*914*	*894*	***681***	*1440*	*699*	*649*	*689*	*675*	***294***	*828*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	21	164817	20	80553	80553	43197	42435	21290	21434	11542	11279	14055	**10122**	**9030**	10008	9568	9656	8727	9508	9155	9629	10221	10488	12359	12656	16982	17114	25880	25922
			100	80553	80553	43197	42435	21290	21434	11542	11279	7985	8106	4867	4868	3288	3357	**2349**	2950	**1911**	2631	2132	2253	2886	**2595**	3537	3500	5300	5294
			500	80553	80553	43197	42435	21290	21434	11542	11279	7985	8106	4867	4868	3288	3357	2349	**1860**	**1210**	1606	**908**	1571	1004	**666**	**1050**	1165	1573	1485
			∞	*80553*	*80553*	*43197*	*42435*	*21290*	*21434*	*11542*	*11279*	*7985*	*8106*	*4867*	*4868*	*3288*	*3357*	*2349*	***1860***	***1210***	*1606*	***908***	*1571*	*638*	***567***	*845*	*778*	*852*	***740***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*	
ped7 n=1068 k=4 w=32 h=90	6	118383	20	60912	56966	**35387**	58872	**20683**	58988	**12338**	58121	**10239**	23958	9031	8515	8704	**7469**	9654	**7319**	8689	**7582**	8705	**7582**	8689	**7735**	8236	7693	8348	8154
			100	60912	56966	**35387**	58872	**20683**	58988	**11956**	58121	**6634**	23958	**5122**	7690	4768	**2555**	4860	**2306**	4045	**1814**	3929	**1814**	3145	**1745**	2644	**1649**	2646	**1719**
			500	60912	56966	**35387**	58872	**20683**	58988	**11956**	58121	**6634**	23958	**4984**	7690	4442	**2387**	4359	**2086**	3276	**1301**	3294	**1301**	2176	**892**	1764	**943**	1564	**551**
			∞	*60912*	*56966*	***35387***	*58872*	***20683***	*58988*	***11956***	*58121*	***6634***	*23958*	***4984***	*7690*	*4442*	***2387***	*4359*	***2086***	*3268*	***1301***	*3256*	***1301***	*2102*	***892***	*1740*	***876***	*1346*	***499***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*			
	7	93380	20	46685	48986	**25119**	47316	**15964**	51644	**7989**	51318	**6230**	26604	**5015**	26061	4992	5357	5909	**5366**	6103	**5461**	6061	**5461**	5642	5524	5924	5706	6138	6099
			100	46685	48986	**25119**	47316	**15964**	51644	**7989**	51318	**4928**	25173	**2947**	25320	2667	**2136**	3002	**1997**	2769	**1505**	2819	**1505**	2217	**1277**	2156	**1268**	2075	**1349**
			500	46685	48986	**25119**	47316	**15964**	51644	**7989**	51318	**4928**	25173	**2947**	25320	2667	**1827**	2615	**1368**	2348	**1180**	2381	**1180**	1610	**506**	1433	**513**	1366	**501**
			∞	*46685*	*48986*	***25119***	*47316*	***15964***	*51644*	***7989***	*51318*	***4928***	*25173*	***2947***	*25320*	*2667*	***1827***	*2615*	***1368***	*2348*	***1180***	*2381*	***1180***	*1608*	***445***	*1421*	***394***	*1312*	***372***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1276)*		*(p=2552)*		*(p=3816)*		*(p=7588)*			
	8	30717	20	17113	17113	**8913**	18311	**5432**	18331	**2976**	18357	**2301**	10245	**2344**	10390	2300	2125	2204	**1938**	2634	**1786**	2196	**1786**	2110	1936	2109	2069	2550	2503
			100	17113	17113	**8913**	18311	**5432**	18331	**2976**	18357	**2257**	9883	**1276**	9856	1229	**989**	1146	**1004**	1306	**944**	1075	**944**	964	**753**	916	**601**	974	**583**
			500	17113	17113	**8913**	18311	**5432**	18331	**2976**	18357	**2257**	9883	**1256**	9856	1157	**989**	1090	**854**	1104	**833**	1008	**833**	750	**537**	682	**345**	700	**273**
			∞	*17113*	*17113*	***8913***	*18311*	***5432***	*18331*	***2976***	*18357*	***2257***	*9883*	***1256***	*9856*	*1157*	***989***	*1090*	***854***	*1094*	***833***	*1008*	***833***	*739*	***497***	*648*	***294***	*644*	***216***
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
	6	101172	20	51735	52172	**27626**	52049	**17242**	25293	**10362**	17818	7231	7356	7283	7356	**6174**	7215	6232	6557	5972	5986	5945	6098	6631	6378	6648	6718	7218	7192
			100	51735	52172	**27626**	52049	**17242**	25293	**10362**	17818	**5377**	7016	**5438**	7016	**2613**	5093	**1559**	2678	**1558**	1826	1273	1294	1407	1340	1390	1405	**1491**	1721
			500	51735	52172	**27626**	52049	**17242**	25293	**10362**	17818	**5377**	7016	**5438**	7016	**2613**	5093	**1406**	2678	**1012**	1826	**395**	931	467	467	**345**	384	**346**	879
			∞	*51735*	*52172*	***27626***	*52049*	***17242***	*25293*	***10362***	*17818*	***5377***	*7016*	***5438***	*7016*	***2613***	*5093*	***1406***	*2678*	***1012***	*1826*	***395***	*931*	*467*	*467*	***170***	*321*	***128***	*752*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
ped9 n=1118 k=7 w=27 h=100	7	58657	20	30956	30531	15391	15640	10090	10195	**5383**	6071	5181	4791	5148	4791	3931	3976	3644	3746	3558	3629	3516	3561	3744	3765	3827	3933	4124	4183
			100	30956	30531	15391	15640	10090	10195	**5383**	6071	3017	2923	2957	2923	**1457**	2185	1051	1128	884	943	806	763	**825**	1048	816	834	876	889
			500	30956	30531	15391	15640	10090	10195	**5383**	6071	3017	2923	2957	2923	**1457**	2185	**753**	1128	**454**	567	**234**	356	**250**	708	**221**	413	227	241
			∞	*30956*	*30531*	*15391*	*15640*	*10090*	*10195*	***5383***	*6071*	*3017*	*2923*	*2957*	*2923*	***1457***	*2185*	***753***	*1128*	***454***	*567*	***234***	*356*	***153***	*677*	***99***	*413*	***93***	*132*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
	8	41061	20	20842	20842	10995	10923	7032	6913	4827	4411	3670	**3213**	3634	**3213**	**2783**	3532	**2560**	3277	2412	2407	2439	2455	2635	2573	2736	2749	3059	3011
			100	20842	20842	10995	10923	7032	6913	4827	4411	**2239**	3157	**2259**	3157	**1322**	1883	**746**	1826	**626**	976	541	583	577	551	578	574	647	637
			500	20842	20842	10995	10923	7032	6913	4827	4411	**2239**	3157	**2259**	3157	**1322**	1883	**604**	1826	**420**	927	**255**	465	**191**	282	155	151	168	167
			∞	*20842*	*20842*	*10995*	*10923*	*7032*	*6913*	*4827*	*4411*	***2239***	*3157*	***2259***	*3157*	***1322***	*1883*	***604***	*1826*	***420***	*927*	***255***	*465*	***153***	*282*	***78***	*108*	***70***	*93*
Better by 10%				0x	0x	16x	12x	20x	12x	24x	14x	21x	22x	24x	20x	15x	27x	14x	29x	14x	18x	21x	14x	12x	16x	11x	18x	9x	15x
Better by 50%				0x	0x	16x	8x	16x	12x	20x	12x	15x	13x	17x	9x	3x	5x	8x	14x	7x	9x	15x	6x	9x	7x	4x	12x	5x	9x

Table B.6: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1		2		3		4		5		6		7		8		9		10		11		12		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
				(p=2)		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
	15	121311	20	79143	79143	50557	55478	**34103**	49291	**27035**	32568	14789	14313	14956	14313	14431	**10438**	14247	**10438**	12991	**10664**	11939	**10363**	11669	10866	11132	11539	11038	11574
			100	79143	79143	50557	55478	**34103**	49291	**27035**	32568	14789	14313	14956	14313	13039	**8015**	12864	**8015**	12868	**5947**	8580	**3668**	6518	**3148**	4237	**2594**	4038	**2611**
			500	79143	79143	50557	55478	**34103**	49291	**27035**	32568	14789	14313	14956	14313	13039	**8015**	12864	**8015**	12868	**5947**	8580	**3668**	6501	**2419**	3675	**951**	3150	**891**
			∞	*79143*	*79143*	*50557*	*55478*	***34103***	*49291*	***27035***	*32568*	*14789*	*14313*	*14956*	*14313*	*13039*	***8015***	*12864*	***8015***	*12868*	***5947***	*8580*	***3668***	*6501*	***2419***	*3675*	***915***	*3099*	***877***
IF3-11-57				*(p=2)*		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
n=2670			20	25066	25066	16854	16593	**9703**	13630	7526	**5650**	4689	4728	4896	4728	**3535**	3902	**3527**	3902	3453	**2984**	3500	**2993**	3002	2943	3724	3464	3508	3542
k=3	16	35820	100	25066	25066	16854	16593	**9703**	13630	7526	**5650**	**4214**	4728	4317	4728	3383	3415	3527	3415	3446	**2272**	2235	**1615**	1733	**1018**	1252	**934**	1156	**802**
w=37			500	25066	25066	16854	16593	**9703**	13630	7526	**5650**	**4214**	4728	4317	4728	3383	3415	3527	3415	3446	**2272**	2235	**1615**	1733	**970**	1081	**617**	935	**392**
h=95			∞	*25066*	*25066*	*16854*	*16593*	***9703***	*13630*	*7526*	***5650***	***4214***	*4728*	*4317*	*4728*	*3383*	*3415*	*3527*	*3415*	*3446*	***2272***	*2235*	***1615***	*1733*	***970***	*1081*	***617***	*935*	***384***
				(p=2)		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
	17	18312	20	12656	12656	8700	**6848**	**5413**	7022	3933	**3248**	2285	2436	2382	2436	1933	1848	2002	1848	2047	**1527**	1673	1605	1978	**1772**	2398	2470	2815	2760
			100	12656	12656	8700	**6848**	**5413**	7022	3933	**3248**	2285	2436	2382	2436	1933	1848	1927	1848	1858	**970**	1158	**725**	954	**531**	769	**544**	788	**585**
			500	12656	12656	8700	**6848**	**5413**	7022	3933	**3248**	2285	2436	2382	2436	1933	1848	1927	1848	1858	**970**	1158	**725**	925	**531**	561	**375**	565	**202**
			∞	*12656*	*12656*	*8700*	***6848***	***5413***	*7022*	*3933*	***3248***	*2285*	*2436*	*2382*	*2436*	*1933*	*1848*	*1927*	*1848*	*1858*	***970***	*1158*	***725***	*925*	***531***	*561*	***375***	*549*	***176***
				(p=3)		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=600)*		*(p=1000)*		*(p=2000)*		*(p=2000)*		*(p=4000)*	
	14	35457	20	21976	21976	10447	11342	6309	5960	6388	5960	4790	**3789**	3861	**2761**	3214	**2866**	3333	**2936**	3539	3404	4081	**3665**	4308	4157	4298	4157	4838	4856
			100	21976	21976	10447	11342	6309	5960	6388	5960	4790	**3789**	3861	**2484**	2147	**1220**	1547	**832**	1161	**722**	1234	**773**	1309	**871**	1006	**871**	1315	**1019**
			500	21976	21976	10447	11342	6309	5960	6388	5960	4790	**3789**	3861	**2484**	2147	**1220**	1547	**770**	1128	**341**	657	**229**	736	**304**	694	**304**	709	**259**
			∞	*21976*	*21976*	*10447*	*11342*	*6309*	*5960*	*6388*	*5960*	*4790*	***3789***	*3861*	***2484***	*2147*	***1220***	*1547*	***770***	*1128*	***341***	*603*	***229***	*638*	***242***	*694*	***242***	*576*	***129***
				(p=3)		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=596)*		*(p=992)*		*(p=1962)*		*(p=1962)*		*(p=3886)*	
IF3-11-59			20	4787	5090	**2644**	2924	1597	1590	1593	1590	1117	1127	1149	**659**	959	**786**	980	**846**	1202	1130	1265	1306	1704	1658	1716	1658	2284	2259
n=2711	15	8523	100	4787	5090	**2644**	2924	1597	1590	1593	1590	1117	1127	1149	**655**	530	**242**	523	**290**	469	**245**	340	**280**	404	**356**	407	**356**	508	495
k=3			500	4787	5090	**2644**	2924	1597	1590	1593	1590	1117	1127	1149	**655**	530	**242**	523	**290**	362	**101**	186	**84**	162	**103**	164	**103**	176	**142**
w=32			∞	*4787*	*5090*	***2644***	*2924*	*1597*	*1590*	*1593*	*1590*	*1117*	*1127*	*1149*	***655***	*530*	***242***	*523*	***290***	*362*	***101***	*186*	***78***	*154*	***67***	*164*	***67***	*122*	***78***
h=73				*(p=3)*		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=600)*		*(p=1000)*		*(p=1999)*		*(p=1999)*		*(p=3992)*	
	16	3023	20	1792	1861	892	883	739	**494**	498	494	442	**368**	682	**345**	446	412	486	473	788	835	1106	1127	1819	1783	1816	1783	3061	3017
			100	1792	1861	892	883	739	**494**	498	494	372	368	528	**246**	166	157	179	**157**	209	**187**	259	250	395	389	402	389	664	654
			500	1792	1861	892	883	739	**494**	498	494	372	368	528	**246**	163	157	149	157	143	**71**	102	**80**	129	**112**	133	**112**	190	183
			∞	*1792*	*1861*	*892*	*883*	*739*	***494***	*498*	*494*	*372*	*368*	*528*	***246***	*163*	*157*	*149*	*157*	*143*	***71***	*91*	***62***	*79*	***65***	*85*	***65***	*90*	***80***
				(p=2)		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=600)*		*(p=1200)*		*(p=2000)*		*(p=4000)*		*(p=6400)*	
	14	46464	20	22027	22027	14647	**13049**	**9384**	11947	**3902**	9402	3220	3007	3489	**2507**	2754	**2464**	2713	**2464**	2695	2562	2811	2645	2795	2727	3053	3062	3459	3441
			100	22027	22027	14647	**13049**	**9384**	11947	**3902**	9402	**2350**	3007	**1789**	2034	1337	**1089**	1510	**1089**	1134	1044	832	833	708	**584**	661	657	760	751
			500	22027	22027	14647	**13049**	**9384**	11947	**3902**	9402	**2350**	3007	**1789**	2034	1222	**1089**	1423	**1089**	**884**	1030	737	**630**	380	**340**	**194**	256	224	225
			∞	*22027*	*22027*	*14647*	***13049***	***9384***	*11947*	***3902***	*9402*	***2350***	*3007*	***1789***	*2034*	*1222*	***1089***	*1423*	***1089***	***884***	*1030*	*737*	***630***	*380*	***340***	***145***	*256*	***130***	*180*
				(p=2)		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=600)*		*(p=1200)*		*(p=1998)*		*(p=3990)*		*(p=6390)*	
IF3-13-58			20	12387	12387	7511	7516	5073	5097	2318	2303	1648	**1361**	1400	**1226**	1244	1152	1283	**1152**	1366	1389	1336	1334	1575	1567	2123	2139	2829	2825
n=3352	16	20270	100	12387	12387	7511	7516	5073	5097	2318	2303	1478	1361	1104	**813**	824	**508**	858	**508**	807	**606**	367	340	425	**340**	468	470	636	630
k=3			500	12387	12387	7511	7516	5073	5097	2318	2303	1478	1361	1104	**813**	800	**508**	833	**508**	742	**490**	256	247	302	**179**	147	140	207	190
w=31			∞	*12387*	*12387*	*7511*	*7516*	*5073*	*5097*	*2318*	*2303*	*1478*	*1361*	*1104*	***813***	*800*	***508***	*833*	***508***	*742*	***490***	*247*	*247*	*282*	***179***	*110*	*106*	*116*	*111*
h=88				*(p=2)*		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=591)*		*(p=1181)*		*(p=1958)*		*(p=3858)*		*(p=6121)*	
	18	7647	20	3870	3870	2367	2391	1705	1597	**912**	1090	785	**488**	633	**513**	707	**605**	657	605	1049	1024	1653	1652	2502	2483	4521	4522	6933	6918
			100	3870	3870	2367	2391	1705	1597	**912**	1090	588	**488**	326	336	294	**210**	288	**210**	346	**235**	387	364	566	538	967	964	1472	1464
			500	3870	3870	2367	2391	1705	1597	**912**	1090	588	**488**	326	336	262	**210**	258	**210**	319	**115**	168	**119**	177	**148**	265	250	389	375
			∞	*3870*	*3870*	*2367*	*2391*	*1705*	*1597*	***912***	*1090*	*588*	***488***	*326*	*336*	*262*	***210***	*258*	***210***	*319*	***115***	*143*	***94***	*103*	*100*	*105*	*100*	*129*	*120*
				(p=2)		*(p=4)*		*(p=12)*		*(p=16)*		*(p=34)*		*(p=46)*		*(p=78)*		*(p=201)*		*(p=358)*		*(p=632)*		*(p=1093)*		*(p=1927)*		*(p=2831)*	
	17	345544	20	197187	197187	98772	103108	99673	100130	98117	96794	94280	**59566**	81379	**36691**	79765	**32938**	39599	**24845**	34001	**24934**	30659	**24702**	28754	**24071**	26633	**20627**	26657	**21190**
			100	197187	197187	98772	103108	99673	100130	98117	96794	94280	**59566**	81379	**33495**	79765	**32938**	39599	**16260**	32364	**12653**	22044	**12651**	16594	**8995**	12595	**5473**	12624	**5411**
IF3-15-53			500	197187	197187	98772	103108	99673	100130	98117	96794	94280	**59566**	81379	**33495**	79765	**32938**	39599	**16260**	32364	**12271**	22044	**12423**	16594	**7324**	12595	**4766**	12230	**4727**
n=3384			∞	*197187*	*197187*	*98772*	*103108*	*99673*	*100130*	*98117*	*96794*	*94280*	***59566***	*81379*	***33495***	*79765*	***32938***	*39599*	***16260***	*32364*	***12271***	*22044*	***12423***	*16594*	***7324***	*12595*	***4766***	*12230*	***4727***
k=3				*(p=2)*		*(p=4)*		*(p=12)*		*(p=16)*		*(p=32)*		*(p=44)*		*(p=68)*		*(p=165)*		*(p=284)*		*(p=526)*		*(p=912)*		*(p=1572)*		*(p=2496)*	
w=32			20	54511	54511	28358	29935	28557	29872	28534	27602	26907	**24321**	23377	**9942**	23792	**9719**	12093	**7328**	10796	**7007**	10373	**7669**	10158	**7481**	9615	**8337**	9946	9127
h=108	18	98346	100	54511	54511	28358	29935	28557	29872	28534	27602	26907	**24321**	23377	**9942**	23792	**9719**	12093	**4088**	9803	**4136**	6835	**3689**	5403	**2788**	4532	**2121**	4702	**2592**
			500	54511	54511	28358	29935	28557	29872	28534	27602	26907	**24321**	23377	**9942**	23792	**9719**	12093	**4088**	9803	**4136**	6835	**3511**	5347	**2788**	4485	**1873**	4020	**1894**
			∞	*54511*	*54511*	*28358*	*29935*	*28557*	*29872*	*28534*	*27602*	*26907*	***24321***	*23377*	***9942***	*23792*	***9719***	*12093*	***4088***	*9803*	***4136***	*6835*	***3511***	*5347*	***2788***	*4485*	***1873***	*4020*	***1856***
Better by 10%				0x	0x	4x	8x	16x	4x	12x	8x	6x	18x	3x	26x	1x	31x	1x	32x	2x	35x	0x	32x	0x	33x	2x	25x	1x	22x
Better by 50%				0x	0x	0x	0x	0x	0x	4x	0x	0x	5x	0x	19x	0x	20x	0x	20x	0x	28x	0x	18x	0x	22x	1x	15x	0x	16x

Table B.7: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
				(p=2)		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=942)*		*(p=1855)*		*(p=3633)*		*(p=7098)*		*(p=13781)*	
IF3-15-59 n=3730 k=3 w=31 h=84	18	28613	20	15023	15023	7910	7946	**5293**	6734	**2935**	4828	2840	**2068**	2595	**1633**	1791	**1610**	1950	**1713**	2045	1959	2558	2417	3227	3302	4945	5002	8120	8209
			100	15023	15023	7910	7946	**5293**	6734	**2935**	4828	2055	2068	1735	**1323**	893	**585**	934	**476**	824	**462**	766	**526**	730	707	1081	1080	1773	1792
			500	15023	15023	7910	7946	**5293**	6734	**2935**	4828	2055	2068	1735	**1323**	842	**535**	818	**392**	636	**462**	416	**177**	319	**211**	358	**299**	518	508
			∞	*15023*	*15023*	*7910*	*7946*	***5293***	*6734*	***2935***	*4828*	*2055*	*2068*	*1735*	***1323***	*842*	***535***	*818*	***392***	*636*	***462***	*356*	***163***	*250*	***127***	*206*	***141***	*253*	***229***
				(p=2)		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=936)*		*(p=1830)*		*(p=3571)*		*(p=6964)*		*(p=13482)*	
	19	43307	20	29588	29588	15858	15694	10234	10164	5909	5470	3684	3417	3649	**2845**	2626	2398	2744	2501	2852	2854	3482	3505	4734	4775	7222	7238	11914	11913
			100	29588	29588	15858	15694	10234	10164	5909	5470	3684	3417	3434	**2247**	1485	**1079**	1494	**723**	1296	**658**	928	**741**	1042	1008	1540	1536	2534	2541
			500	29588	29588	15858	15694	10234	10164	5909	5470	3684	3417	3434	**2247**	1485	**1079**	1414	**573**	1113	**417**	692	**260**	508	**291**	415	399	660	667
			∞	*29588*	*29588*	*15858*	*15694*	*10234*	*10164*	*5909*	*5470*	*3684*	*3417*	*3434*	***2247***	*1485*	***1079***	*1414*	***573***	*1113*	***417***	*675*	***260***	*442*	***177***	*311*	***189***	*317*	***251***
						(p=3)		*(p=9)*		*(p=15)*		*(p=43)*		*(p=71)*		*(p=205)*		*(p=470)*		*(p=934)*		*(p=934)*		*(p=1827)*		*(p=2707)*		*(p=7582)*	
			20			1163230	1155580	643626	639982	401282	398939	325905	**200608**	207721	**165795**	164502	**149029**	166386	164442	**160338**	177599	**161219**	177599	**180036**	198701	193080	211770	**217768**	251809
	15	1891710	100			1163230	1155580	643626	639982	401282	398939	316651	**186789**	178618	165795	119413	**42519**	64570	**39617**	46136	**38277**	47356	**38277**	47363	**40579**	45790	42911	48752	50451
IF3-16-56 n=3930 k=3 w=38 h=77			500			1163230	1155580	643626	639982	401282	398939	316651	**186789**	178618	165795	119016	**42519**	56746	**21260**	35911	**15754**	37114	**15754**	28697	**11309**	24390	**11051**	17870	**11166**
			∞			*1163230*	*1155580*	*643626*	*639982*	*401282*	*398939*	*316651*	***186789***	*178618*	*165795*	*119016*	***42519***	*56746*	***21260***	*35893*	***15754***	*37069*	***15754***	*26721*	***10225***	*22845*	***7938***	*15239*	***4262***
				(p=2)		*(p=3)*		*(p=9)*		*(p=15)*		*(p=42)*		*(p=70)*		*(p=201)*		*(p=455)*		*(p=900)*		*(p=900)*		*(p=1766)*		*(p=2629)*		*(p=7122)*	
			20	383172	383172	322266	323359	182770	**125290**	126100	**85253**	89282	**56806**	54618	54536	53173	**47562**	55996	52947	56246	58857	54609	58857	**60079**	67647	**66004**	73700	**78744**	90965
	16	489614	100	383172	383172	322266	323359	182770	**125290**	126100	**85253**	81351	**56806**	43689	**36589**	33623	**26361**	20474	**13229**	19126	**12956**	16459	**12956**	15158	**13653**	17377	**14807**	19498	18292
			500	383172	383172	322266	323359	182770	**125290**	126100	**85253**	81351	**56806**	43689	**36589**	33425	**26361**	17488	**8406**	12942	**6514**	13210	**6514**	10045	**5321**	9416	**4234**	8158	**4118**
			∞	*383172*	*383172*	*322266*	*323359*	*182770*	***125290***	*126100*	***85253***	*81351*	***56806***	*43689*	***36589***	*33425*	***26361***	*17488*	***8406***	*12595*	***6514***	*13185*	***6514***	*9849*	***5321***	*8945*	***3930***	*6573*	***1345***
				(p=3)		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
			20	32678	32678	10281	10182	4810	4863	3352	3070	3413	3408	3249	3252	4245	4149	5011	5059										
	13	57842	100	32678	32678	10281	10182	4810	4863	2603	2663	1190	**1052**	**827**	962	908	861	1069	1051										
IF4-12-50 n=2569 k=4 w=28 h=80			500	32678	32678	10281	10182	4810	4863	2603	2663	1103	**899**	566	527	555	**281**	**299**	741										
			∞	*32678*	*32678*	*10281*	*10182*	*4810*	*4863*	*2603*	*2663*	*1103*	***899***	*566*	*527*	*536*	***236***	***155***	*717*										
				(p=3)		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
			20	17309	17309	6207	**5210**	2946	**2423**	**2048**	2359	1750	1897	1823	1790	2542	2551	3252	3272										
	14	33676	100	17309	17309	6207	**5210**	2720	**2238**	1592	1644	**637**	1425	530	556	575	535	712	691										
			500	17309	17309	6207	**5210**	2720	**2238**	1592	1644	**564**	1425	**337**	491	204	**140**	203	**177**										
			∞	*17309*	*17309*	*6207*	***5210***	*2720*	***2238***	*1592*	*1644*	***564***	*1425*	***328***	*491*	*151*	***101***	*135*	***99***										
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=1024)*		*(p=1792)*		*(p=1792)*		*(p=3072)*	
			20	53594	53594	28796	28568	**16287**	27758	**11110**	15603	**8247**	15781	**7920**	15870	**7623**	15799	**7218**	15737	**7279**	9773	**7364**	9773	7746	7590	7578	7590	8110	7913
	13	104837	100	53594	53594	28796	28568	**16287**	27758	**11110**	15603	**3732**	13666	**2974**	13923	**2278**	13877	**1822**	13412	**1651**	5958	**1686**	5958	1953	**1764**	1779	1764	2004	**1721**
IF4-12-55 n=2926 k=4 w=28 h=78			500	53594	53594	28796	28568	**16287**	27758	**11110**	15603	**3732**	13666	**2974**	13636	**1689**	13628	**1029**	12986	**672**	5540	**666**	5540	813	**712**	764	712	887	863
			∞	*53594*	*53594*	*28796*	*28568*	***16287***	*27758*	***11110***	*15603*	***3732***	*13666*	***2974***	*13636*	***1689***	*13628*	***1029***	*12986*	***672***	*5540*	***581***	*5540*	***606***	*691*	*679*	*691*	*730*	*760*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=768)*		*(p=1536)*		*(p=1536)*		*(p=3072)*	
			20	13300	13300	7086	6693	**3595**	6882	**2125**	6797	**2103**	3599	2145	1975	1968	**1788**	2018	1844	2006	1931	2052	1931	2386	2296	2307	2296	2986	2991
	14	25905	100	13300	13300	7086	6693	**3595**	6882	**2125**	6797	**1181**	3599	**1159**	1975	699	759	691	**536**	574	**474**	583	**474**	589	**498**	575	**498**	717	**648**
			500	13300	13300	7086	6693	**3595**	6882	**2125**	6797	**1181**	3599	**1159**	1975	**566**	759	438	**352**	281	**216**	357	**216**	**230**	341	**235**	341	375	**233**
			∞	*13300*	*13300*	*7086*	*6693*	***3595***	*6882*	***2125***	*6797*	***1181***	*3599*	***1159***	*1975*	***566***	*759*	*438*	***352***	*255*	***216***	*357*	***216***	***164***	*341*	***173***	*341*	*329*	***191***
				(p=2)		*(p=4)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=40)*		*(p=56)*		*(p=128)*		*(p=152)*		*(p=176)*		*(p=352)*		*(p=400)*	
			20	5182	5182	2812	2785	2819	2785	1565	1538	1322	1287	1307	**1065**	1223	1165	1039	1052	1053	**933**	1106	**950**	1039	994	1244	1225	1312	1285
	15	10607	100	5182	5182	2812	2785	2819	2785	1565	1538	1322	1287	1118	1065	766	773	**627**	760	392	**336**	423	**355**	413	**332**	367	389	**310**	487
IF4-17-51 n=3837 k=4 w=29 h=85			500	5182	5182	2812	2785	2819	2785	1565	1538	1322	1287	1118	1065	766	773	**627**	760	333	336	344	337	317	332	221	224	**136**	391
			∞	*5182*	*5182*	*2812*	*2785*	*2819*	*2785*	*1565*	*1538*	*1322*	*1287*	*1118*	*1065*	*766*	*773*	***627***	*760*	*333*	*336*	*344*	*337*	*317*	*332*	*221*	*224*	***136***	*391*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=80)*		*(p=112)*		*(p=256)*		*(p=304)*		*(p=352)*		*(p=704)*		*(p=800)*	
			20	33524	33524	**15988**	28639	**15508**	29154	**8712**	25678	**7281**	28848	**6623**	15067	**4805**	15050	5056	**4473**	4224	3934	4158	4111	3922	4213	3964	4162	4049	4086
	16	66103	100	33524	33524	**15988**	28639	**15508**	29154	**8712**	25678	**7281**	28848	**6610**	15067	**3719**	15050	3205	3399	1733	1860	1786	1808	1848	1992	1156	1194	1125	1065
			500	33524	33524	**15988**	28639	**15508**	29154	**8712**	25678	**7281**	28848	**6610**	15067	**3719**	15050	3205	3399	1668	1620	**1584**	1805	1641	1633	850	918	**606**	951
			∞	*33524*	*33524*	***15988***	*28639*	***15508***	*29154*	***8712***	*25678*	***7281***	*28848*	***6610***	*15067*	***3719***	*15050*	*3205*	*3399*	*1668*	*1620*	***1584***	*1805*	*1641*	*1633*	*850*	*918*	***606***	*951*
Better by 10%				0x	0x	4x	4x	16x	8x	17x	4x	15x	12x	14x	13x	10x	20x	9x	19x	5x	17x	7x	17x	5x	14x	3x	9x	7x	10x
Better by 50%				0x	0x	4x	0x	12x	0x	12x	0x	15x	5x	11x	4x	8x	8x	6x	12x	3x	8x	3x	10x	1x	8x	1x	5x	5x	6x

Table B.8: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var
pdb1a6m				*(p=9)*		*(p=81)*		*(p=511)*							
n=124	3	198326	20	109236	109236	96811	**29713**	51456	**8839**						
k=81			100	109236	109236	96811	**29713**	51456	**8839**						
w=15			500	109236	109236	96811	**29713**	51456	**8839**						
h=34			∞	*109236*	*109236*	*96811*	***29713***	*51456*	***8839***						
pdb1duw				*(p=9)*		*(p=54)*		*(p=784)*		*(p=15081)*					
n=241	3	627106	20	261878	261878	185941	**144524**	148576	**34290**	66316	**40886**				
k=81			100	261878	261878	185941	**144524**	148576	**13294**	51977	**8190**				
w=9			500	261878	261878	185941	**144524**	148576	**13294**	51977	**3998**				
h=32			∞	*261878*	*261878*	*185941*	***144524***	*148576*	***13294***	*51977*	***3998***				
pdb1e5k				*(p=66)*		*(p=1046)*		*(p=11321)*							
n=154	3	112654	20	20322	20322	**6876**	7630	10653	10712						
k=81			100	20322	20322	5994	**2024**	2299	2153						
w=12			500	20322	20322	5994	**2024**	2034	**783**						
h=43			∞	*20322*	*20322*	*5994*	***2024***	*2034*	***783***						
pdb1f9i				*(p=81)*		*(p=6534)*									
n=103	3	68804	20	27995	27995	23496	**21220**								
k=81			100	27995	27995	8752	**4249**								
w=10			500	27995	27995	8752	**2587**								
h=24			∞	*27995*	*27995*	*8752*	***2587***								
pdb1ft5				*(p=27)*		*(p=118)*		*(p=5281)*							
n=172	3	81118	20	39764	39764	29982	**8248**	8302	8469						
k=81			100	39764	39764	29982	**8248**	4478	**1715**						
w=14			500	39764	39764	29982	**8248**	4478	**802**						
h=33			∞	*39764*	*39764*	*29982*	***8248***	*4478*	***802***						
pdb1hd2				*(p=79)*		*(p=3777)*									
n=126	3	101550	20	58967	58967	15426	**6470**								
k=81			100	58967	58967	15426	**2275**								
w=12			500	58967	58967	15426	**2275**								
h=27			∞	*58967*	*58967*	*15426*	***2275***								
pdb1huw				*(p=9)*		*(p=42)*		*(p=293)*		*(p=654)*		*(p=1588)*		*(p=2597)*	
n=152	3	545249	20	478239	478239	477785	**402748**	467632	**41642**	462167	**36305**	446255	**31297**	367056	**31646**
k=81			100	478239	478239	477785	**402748**	467632	**41642**	462167	**34051**	446255	**18483**	367056	**12750**
w=15			500	478239	478239	477785	**402748**	467632	**41642**	462167	**34051**	446255	**18483**	367056	**12750**
h=43			∞	*478239*	*478239*	*477785*	***402748***	*467632*	***41642***	*462167*	***34051***	*446255*	***18483***	*367056*	***12750***
pdb1kao				*(p=27)*		*(p=215)*		*(p=752)*		*(p=3241)*					
n=148	3	716795	20	252879	252879	213134	**64745**	145683	**32176**	63832	**18172**				
k=81			100	252879	252879	213134	**55749**	145683	**25927**	63832	**6126**				
w=15			500	252879	252879	213134	**55749**	145683	**25927**	63832	**6126**				
h=41			∞	*252879*	*252879*	*213134*	***55749***	*145683*	***25927***	*63832*	***6126***				
pdb1nfp				*(p=6)*		*(p=48)*		*(p=336)*		*(p=3812)*					
n=204	3	354720	20	328980	328980	292628	**73365**	194064	**38568**	101131	**42531**				
k=81			100	328980	328980	292628	**73365**	194064	**27180**	101131	**8752**				
w=18			500	328980	328980	292628	**73365**	194064	**27180**	101131	**6768**				
h=38			∞	*328980*	*328980*	*292628*	***73365***	*194064*	***27180***	*101131*	***6768***				
pdb1rss				*(p=8)*		*(p=109)*		*(p=908)*		*(p=1336)*					
n=115	3	378579	20	392069	392069	110936	**57202**	37791	**31715**	33834	**24441**				
k=81			100	392069	392069	110904	**57202**	37654	**25702**	33706	**24266**				
w=12			500	392069	392069	110904	**57202**	37625	**25702**	33689	**24266**				
h=35			∞	*392069*	*392069*	*110904*	***57202***	*37625*	***25702***	*33689*	***24266***				
pdb1vhh				*(p=27)*		*(p=1842)*		*(p=67760)*							
n=133	3	944633	20	233763	233763	**52565**	231663	92605	**69612**						
k=81			100	233763	233763	**22967**	231663	20965	**13970**						
w=14			500	233763	233763	**21751**	231663	15746	**3921**						
h=35			∞	*233763*	*233763*	***21751***	*231663*	*14878*	***3133***						
Better by 10%				0x	0x	5x	39x	0x	33x	0x	20x	0x	4x	0x	4x
Better by 50%				0x	0x	4x	30x	0x	28x	0x	16x	0x	4x	0x	4x

Table B.9: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **side-chain prediction instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var	14 fix	14 var	15 fix	15 var
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=192)*		*(p=192)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2112)*	
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	48170	48170	26651	26500	20957	20332	11334	11562	11642	11562	8139	7966	5929	5776	**5588**	6625	**5425**	6458	**5207**	6458	**5548**	6458	**5167**	5976	4948	5313	5170	4815	4965	4970
			100	48170	48170	26651	26500	20957	20332	11334	11562	11642	11562	8139	7966	4944	**3940**	3833	3915	3831	3880	3978	3880	3870	3880	**3230**	3978	**1968**	2810	**1715**	1907	1643	**1343**
			500	48170	48170	26651	26500	20957	20332	11334	11562	11642	11562	8139	7966	4944	**3940**	3833	3846	3831	3638	3978	3638	3870	3638	**3147**	3865	**1791**	2810	**1617**	1813	1643	**1343**
			∞	*48170*	*48170*	*26651*	*26500*	*20957*	*20332*	*11334*	*11562*	*11642*	*11562*	*8139*	*7966*	*4944*	***3940***	*3833*	*3846*	*3831*	*3638*	*3978*	*3638*	*3870*	*3638*	***3147***	*3865*	***1791***	*2810*	***1617***	*1813*	*1643*	***1343***
				(p=2)		*(p=4)*		*(p=8)*		*(p=8)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=96)*		*(p=96)*		*(p=192)*		*(p=288)*		*(p=576)*		*(p=864)*		*(p=1584)*	
	14	15402	20	10087	10087	5803	5922	3521	3737	3546	3737	3630	3737	2376	2437	1564	1433	**1337**	1617	**1479**	1671	**1190**	1671	**1296**	1726	**1122**	1363	**962**	1160	1227	**953**	1014	999
			100	10087	10087	5803	5922	3521	3737	3546	3737	3630	3737	2376	2437	1413	**1227**	1184	1223	1211	1253	1187	1253	**987**	1264	**847**	1238	**651**	861	655	**536**	652	**325**
			500	10087	10087	5803	5922	3521	3737	3546	3737	3630	3737	2376	2437	1413	**1227**	1184	1223	1211	1253	1187	1253	**987**	1227	**847**	1238	**651**	839	655	**536**	649	**226**
			∞	*10087*	*10087*	*5803*	*5922*	*3521*	*3737*	*3546*	*3737*	*3630*	*3737*	*2376*	*2437*	*1413*	***1227***	*1184*	*1223*	*1211*	*1253*	*1187*	*1253*	***987***	*1227*	***847***	*1238*	***651***	*839*	*655*	***536***	*649*	***219***
				(p=2)		*(p=4)*		*(p=4)*		*(p=6)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=48)*		*(p=72)*		*(p=144)*		*(p=288)*		*(p=576)*		*(p=960)*		*(p=1536)*	
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	59546	59546	51966	51773	52978	51773	40868	40453	40537	40453	42494	42532	31861	31583	22438	21646	21832	21646	**14803**	18266	**13118**	14594	14937	16206	**15733**	17700	19130	18620	19745	19197
			100	59546	59546	51966	51773	52978	51773	40868	40453	40537	40453	42494	42532	31861	31583	22438	21646	21832	21646	14803	14537	9671	10481	**7135**	11508	**3841**	7952	5975	6055	6938	**5964**
			500	59546	59546	51966	51773	52978	51773	40868	40453	40537	40453	42494	42532	31861	31583	22438	21646	21832	21646	14803	14537	9671	10481	**7135**	11090	**3841**	7952	4706	4778	4791	4793
			∞	*59546*	*59546*	*51966*	*51773*	*52978*	*51773*	*40868*	*40453*	*40537*	*40453*	*42494*	*42532*	*31861*	*31583*	*22438*	*21646*	*21832*	*21646*	*14803*	*14537*	*9671*	*10481*	***7135***	*11090*	***3841***	*7952*	*4706*	*4649*	*4600*	*4628*
				(p=2)		*(p=4)*		*(p=4)*		*(p=6)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=48)*		*(p=72)*		*(p=144)*		*(p=288)*		*(p=576)*		*(p=960)*		*(p=1536)*	
	15	33656	20	25356	25356	16330	16188	16447	16188	13225	13139	13049	13139	8356	8489	5443	5486	4916	**4403**	4807	4403	4275	4509	4467	4662	4443	4857	4683	4977	4909	4958	5255	5115
			100	25356	25356	16330	16188	16447	16188	13225	13139	13049	13139	8356	8489	5443	5486	3563	**3103**	3505	**3103**	**2313**	2716	**1448**	1720	**1047**	1332	1116	1220	1077	1073	1579	**1134**
			500	25356	25356	16330	16188	16447	16188	13225	13139	13049	13139	8356	8489	5443	5486	3563	**3103**	3505	**3103**	**2313**	2716	**1448**	1720	**1001**	1332	602	567	481	504	850	**601**
			∞	*25356*	*25356*	*16330*	*16188*	*16447*	*16188*	*13225*	*13139*	*13049*	*13139*	*8356*	*8489*	*5443*	*5486*	*3563*	***3103***	*3505*	***3103***	***2313***	*2716*	***1448***	*1720*	***1001***	*1332*	*602*	*567*	*481*	*475*	*718*	***601***
				(p=2)		*(p=3)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=36)*		*(p=72)*		*(p=144)*		*(p=216)*		*(p=288)*		*(p=504)*		*(p=1008)*		*(p=2016)*		*(p=2688)*		*(p=3360)*	
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	249495	249495	165094	164069	111418	112110	112534	112086	66859	**47596**	46908	44937	30640	28910	27762	26814	26206	28019	29913	28771	28611	29296	29922	29912	30050	30460	29728	31082	30582	31325
			100	249495	249495	165094	164069	111418	112110	112534	112086	66859	**47596**	46908	44937	29325	28910	16135	**11426**	10908	11472	10455	9703	7645	**6944**	8116	**7303**	7254	6699	7135	6873	6726	6722
			500	249495	249495	165094	164069	111418	112110	112534	112086	66859	**47596**	46908	44937	29325	28910	16135	**11426**	10908	11472	10455	**6374**	6568	6351	4549	**2653**	3077	**2080**	2858	**2163**	2062	**1805**
			∞	*249495*	*249495*	*165094*	*164069*	*111418*	*112110*	*112534*	*112086*	*66859*	***47596***	*46908*	*44937*	*29325*	*28910*	*16135*	***11426***	*10908*	*11472*	*10455*	***6374***	*6568*	*6351*	*4549*	***2571***	*3077*	***1465***	*2613*	***1196***	*2002*	***1242***
				(p=2)		*(p=3)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=36)*		*(p=72)*		*(p=144)*		*(p=216)*		*(p=288)*		*(p=504)*		*(p=1008)*		*(p=2014)*		*(p=2661)*		*(p=3325)*	
	18	21694	20	21374	21374	16704	16317	9641	10539	**9654**	10686	**6467**	10548	**5033**	10933	**2890**	10958	**2688**	10990	**2618**	6484	**2658**	6951	**2718**	6508	2989	2962	3204	3244	3292	3399	3455	3492
			100	21374	21374	16704	16317	9641	10539	**9654**	10686	**6467**	10548	**5033**	10631	**2890**	10603	**1836**	10623	**1236**	6384	**1221**	6210	**752**	6164	**775**	1657	1049	1130	**786**	1132	908	838
			500	21374	21374	16704	16317	9641	10539	**9654**	10686	**6467**	10548	**5033**	10631	**2890**	10603	**1836**	10588	**1236**	6384	**1221**	6210	**752**	6164	**600**	1595	**713**	804	**601**	709	702	**309**
			∞	*21374*	*21374*	*16704*	*16317*	*9641*	*10539*	***9654***	*10686*	***6467***	*10548*	***5033***	*10631*	***2890***	*10603*	***1836***	*10588*	***1236***	*6384*	***1221***	*6210*	***752***	*6164*	***600***	*1595*	***692***	*804*	*601*	*623*	*702*	***220***
				(p=2)		*(p=4)*		*(p=8)*		*(p=8)*		*(p=16)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=192)*		*(p=384)*		*(p=384)*		*(p=768)*		*(p=1280)*		*(p=1280)*	
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	25695	25695	19776	21223	12259	13268	13515	13268	8736	8789	8896	8789	5770	5738	**5304**	6089	5244	5724	5534	5651	**5058**	5874	**5021**	5874	5299	5252	5079	5177	5061	5177
			100	25695	25695	19776	21223	12259	13268	13515	13268	8736	8789	8896	8789	5480	5738	3529	**3187**	3223	2986	2963	2804	**2465**	2787	**2434**	2787	1624	1602	1413	1324	1352	1324
			500	25695	25695	19776	21223	12259	13268	13515	13268	8736	8789	8896	8789	5480	5738	3529	**3187**	3223	2986	2963	**2562**	**2465**	2762	**2434**	2762	**1314**	1534	1027	**913**	1029	**913**
			∞	*25695*	*25695*	*19776*	*21223*	*12259*	*13268*	*13515*	*13268*	*8736*	*8789*	*8896*	*8789*	*5480*	*5738*	*3529*	***3187***	*3223*	*2986*	*2963*	***2562***	***2465***	*2762*	***2434***	*2762*	***1314***	*1534*	*1027*	***913***	*1029*	***913***
				(p=2)		*(p=4)*		*(p=8)*		*(p=8)*		*(p=16)*		*(p=24)*		*(p=48)*		*(p=80)*		*(p=160)*		*(p=240)*		*(p=480)*		*(p=480)*		*(p=960)*		*(p=1216)*		*(p=1216)*	
	18	26855	20	17937	17937	12189	11268	7411	7498	7442	7498	**4676**	5301	**3891**	4787	2382	2334	2531	2566	**2344**	2628	2399	**2158**	2444	2351	2350	2351	2788	2553	**2510**	2915	**2522**	2915
			100	17937	17937	12189	11268	7411	7498	7442	7498	**4676**	5301	**3891**	4787	2053	2148	1272	1187	1039	1021	910	900	838	827	**702**	827	941	989	**1019**	1138	**619**	1138
			500	17937	17937	12189	11268	7411	7498	7442	7498	**4676**	5301	**3891**	4787	2053	2148	1272	1187	1039	**901**	894	900	705	702	702	702	710	748	745	780	**487**	780
			∞	*17937*	*17937*	*12189*	*11268*	*7411*	*7498*	*7442*	*7498*	***4676***	*5301*	***3891***	*4787*	*2053*	*2148*	*1272*	*1187*	*1039*	***901***	*894*	*900*	*705*	*702*	*702*	*702*	*696*	*748*	*714*	*731*	***487***	*731*
				(p=2)		*(p=4)*		*(p=8)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=144)*		*(p=288)*		*(p=384)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*	
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	16045	16045	**11223**	12553	**8918**	11024	**6758**	10812	**4412**	6656	**3476**	4101	**3092**	3808	3187	3024	**2878**	3210	3194	3303	3092	3378	3177	3230	3231	3230	3374	3382	3560	3587
			100	16045	16045	**11223**	12553	**8918**	11024	**6758**	10812	**4412**	6656	**2557**	4101	**1437**	2317	**1440**	2253	1078	1153	**847**	1019	**687**	971	**669**	936	908	936	775	753	763	781
			500	16045	16045	**11223**	12553	**8918**	11024	**6758**	10812	**4412**	6656	**2557**	4101	**1437**	2317	**1440**	2253	997	997	**566**	1003	**392**	631	**240**	511	490	511	526	**326**	**216**	266
			∞	*16045*	*16045*	***11223***	*12553*	***8918***	*11024*	***6758***	*10812*	***4412***	*6656*	***2557***	*4101*	***1437***	*2317*	***1440***	*2253*	*997*	*997*	***566***	*1003*	***392***	*631*	***205***	*445*	*434*	*445*	*488*	***326***	***155***	*199*
				(p=2)		*(p=4)*		*(p=8)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=144)*		*(p=288)*		*(p=384)*		*(p=640)*		*(p=1280)*		*(p=2560)*		*(p=3840)*			
	20	8053	20	5016	5016	5831	5794	4489	4443	2304	2274	**1520**	2106	1444	1344	**1182**	1345	1299	1253	1258	1338	1364	1377	1502	1521	1794	1803	1800	1803	2390	2385	2968	2974
			100	5016	5016	5831	5794	4489	4443	2304	2274	**1520**	2106	**1025**	1344	**568**	977	**549**	976	**382**	555	**351**	484	**339**	411	381	411	386	411	515	511	634	639
			500	5016	5016	5831	5794	4489	4443	2304	2274	**1520**	2106	**1025**	1344	**568**	977	**549**	976	**326**	517	**226**	484	**159**	304	**114**	157	**123**	157	**137**	157	175	178
			∞	*5016*	*5016*	*5831*	*5794*	*4489*	*4443*	*2304*	*2274*	***1520***	*2106*	***1025***	*1344*	***568***	*977*	***549***	*976*	***326***	*517*	***226***	*484*	***159***	*304*	***95***	*141*	***102***	*141*	***76***	*95*	***88***	*106*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=128)*		*(p=128)*		*(p=256)*		*(p=384)*		*(p=576)*		*(p=1152)*		*(p=2304)*		*(p=4608)*	
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	282186	282186	164678	164547	84949	93045	74973	74584	55099	**48079**	52792	**43415**	43719	40998	42329	40998	42048	40998	39350	36861	35060	38131	35660	36487	35096	36198	35891	38091	35958	36103
			100	282186	282186	164678	164547	84949	93045	74973	74584	47702	48079	41044	41612	24974	25174	24715	25174	24827	25174	16118	15174	13222	12451	13743	13219	11125	10299	**8482**	9918	**7550**	8627
			500	282186	282186	164678	164547	84949	93045	74973	74584	47702	48079	41044	41612	24974	25174	24715	25174	24827	25174	15411	15174	11941	11978	12119	12187	7401	7283	5890	5970	3551	3758
			∞	*282186*	*282186*	*164678*	*164547*	*84949*	*93045*	*74973*	*74584*	*47702*	*48079*	*41044*	*41612*	*24974*	*25174*	*24715*	*25174*	*24827*	*25174*	*15411*	*15174*	*11941*	*11978*	*12119*	*12187*	*7298*	*7283*	*5745*	*5606*	***3240***	*3613*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=128)*		*(p=128)*		*(p=256)*		*(p=384)*		*(p=576)*		*(p=1152)*		*(p=2304)*		*(p=4608)*	
	12	64758	20	87736	87736	66080	66050	36419	36030	30343	30469	**19267**	31349	**19555**	29465	16756	**15218**	17624	**15218**	16417	15218	**15708**	17567	**13599**	16307	**14367**	17184	13668	14763	14155	15305	14336	14423
			100	87736	87736	66080	66050	36419	36030	30343	30469	**19267**	29308	**17150**	29465	10632	10537	11009	10537	10545	10537	**7903**	10594	**6731**	10472	**6573**	7941	**5464**	6589	**5477**	6802	**3328**	4761
			500	87736	87736	66080	66050	36419	36030	30343	30469	**19267**	29308	**17150**	29465	10632	10537	11009	10537	10545	10537	**7708**	10594	**6298**	10472	**6090**	7690	**4097**	6044	**4895**	6097	**1925**	4086
			∞	*87736*	*87736*	*66080*	*66050*	*36419*	*36030*	*30343*	*30469*	***19267***	*29308*	***17150***	*29465*	*10632*	*10537*	*11009*	*10537*	*10545*	*10537*	***7708***	*10594*	***6298***	*10472*	***6090***	*7690*	***4097***	*6044*	***4884***	*6068*	***1925***	*4086*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=120)*		*(p=240)*		*(p=480)*		*(p=960)*		*(p=1920)*		*(p=3840)*		*(p=3840)*		*(p=7680)*	
	16	59609	20	43299	43299	25348	25603	24417	23879	23900	**13973**	13651	13862	**8939**	13627	9075	**8135**	8075	7896	7040	7012	7779	7735	7981	8498	8562	8915	9006	9027	8924	9027	9427	9637
			100	43299	43299	25348	25603	24417	23879	23900	**13973**	13651	13862	**8939**	13627	7829	7779	**5956**	6642	**4194**	5802	4010	3837	3517	**2756**	2671	**2202**	2113	2281	2168	2281	**1992**	2094
			500	43299	43299	25348	25603	24417	23879	23900	**13973**	13651	13862	**8939**	13627	7829	7779	**5956**	6642	**4194**	5802	3793	3837	3231	**2331**	2386	**1341**	1923	**1044**	1389	**1044**	972	886
			∞	*43299*	*43299*	*25348*	*25603*	*24417*	*23879*	*23900*	***13973***	*13651*	*13862*	***8939***	*13627*	*7829*	*7779*	***5956***	*6642*	***4194***	*5802*	*3793*	*3837*	*3231*	***2331***	*2380*	***1341***	*1923*	***1000***	*1334*	***1000***	***868***	*879*
75-26-9 n=675 k=2 w=39 h=124				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=120)*		*(p=240)*		*(p=480)*		*(p=960)*		*(p=1920)*		*(p=3840)*		*(p=3840)*		*(p=7680)*	
	18	66533	20	50994	50994	31458	30934	30951	30367	31796	**19235**	19383	**11794**	11843	11948	11634	11731	9490	9228	8726	8184	9053	8634	8786	8801	8710	9202	9414	9549	9490	9549	10178	10276
			100	50994	50994	31458	30934	30951	30367	31796	**19235**	19383	**11794**	11843	11948	10856	11731	**7318**	8083	**4912**	8184	4947	**2215**	4078	**2375**	2963	**2088**	2370	**2007**	2378	**2007**	2186	2140
			500	50994	50994	31458	30934	30951	30367	31796	**19235**	19383	**11794**	11843	11948	10856	11731	**7318**	8083	**4912**	8184	4628	**1443**	3787	**1424**	2644	**915**	1633	**853**	1458	**853**	1014	**596**
			∞	*50994*	*50994*	*31458*	*30934*	*30951*	*30367*	*31796*	***19235***	*19383*	***11794***	*11843*	*11948*	*10856*	*11731*	***7318***	*8083*	***4912***	*8184*	*4628*	***1443***	*3787*	***1424***	*2639*	***879***	*1553*	***780***	*1444*	***780***	*935*	***304***
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=120)*		*(p=240)*		*(p=320)*		*(p=640)*		*(p=1280)*		*(p=2560)*		*(p=2560)*		*(p=5120)*	
	20	5708	20	3488	3488	2164	2160	2021	2026	2867	**2344**	2366	**1597**	1612	1607	1472	**1199**	1180	**811**	885	909	1027	958	1164	1148	1455	1514	2146	2163	2136	2163	3416	3446
			100	3488	3488	2164	2160	2021	2026	2867	**2344**	2366	**1597**	1612	1607	1316	1199	979	**523**	588	**332**	374	**331**	356	**278**	347	336	464	460	457	460	723	731
			500	3488	3488	2164	2160	2021	2026	2867	**2344**	2366	**1597**	1612	1607	1316	1199	979	**523**	588	**332**	351	331	225	**166**	181	**103**	159	**127**	154	**127**	184	191
			∞	*3488*	*3488*	*2164*	*2160*	*2021*	*2026*	*2867*	***2344***	*2366*	***1597***	*1612*	*1607*	*1316*	*1199*	*979*	***523***	*588*	***332***	*351*	*331*	*225*	***166***	*170*	***84***	*114*	***73***	*121*	***73***	*104*	***77***
Better by 10%				0x	0x	4x	0x	4x	0x	8x	12x	20x	13x	23x	1x	12x	9x	19x	15x	17x	8x	20x	9x	27x	10x	31x	11x	20x	9x	13x	17x	12x	19x
Better by 50%				0x	0x	0x	0x	0x	0x	4x	8x	12x	4x	15x	0x	10x	0x	10x	3x	9x	3x	8x	5x	11x	3x	8x	8x	5x	6x	0x	5x	5x	8x

Table B.10: Parallel runtime with 20, 100, 500, and "unlimited" CPUs, on **grid instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
ped13 n=1077 k=3 w=32 h=102	8	252654	20	1.90	1.81	**3.64**	1.85	**7.22**	1.84	**13.21**	3.56	**13.57**	7.17	**15.51**	10.22	**16.14**	11.53	16.59	**18.28**	17.37	17.96	17.95	18.20	17.00	18.07	17.15	17.44	17.21	17.40
			100	1.90	1.81	**3.64**	1.85	**7.22**	1.84	**13.21**	3.56	**21.62**	7.17	**42.50**	13.41	**43.50**	13.22	**51.91**	22.91	**60.08**	48.99	**70.12**	63.50	67.05	**82.92**	68.05	69.79	68.06	**77.10**
			500	1.90	1.81	**3.64**	1.85	**7.22**	1.84	**13.21**	3.56	**21.62**	7.17	**42.50**	13.41	**50.12**	13.22	**75.67**	22.91	**103.97**	48.99	**145.12**	89.72	125.82	135.04	166.66	157.12	165.67	**214.84**
			∞	*1.90*	*1.81*	***3.64***	*1.85*	***7.22***	*1.84*	***13.21***	*3.56*	***21.62***	*7.17*	***42.50***	*13.41*	***50.12***	*13.22*	***75.67***	*22.91*	***103.97***	*48.99*	***152.38***	*89.72*	*142.42*	*135.04*	***221.82***	*180.60*	*221.63*	*224.58*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
	9	102385	20	1.91	1.89	3.31	3.23	**6.45**	4.30	**9.71**	8.54	**10.85**	8.50	**17.63**	10.55	**19.64**	14.86	**20.03**	17.99	19.22	19.52	19.60	18.37	18.46	19.25	18.10	18.37	17.34	17.82
			100	1.91	1.89	3.31	3.23	**6.45**	4.30	**9.71**	8.54	**14.32**	8.50	**29.41**	10.55	**30.86**	14.86	**52.75**	31.63	55.98	52.94	**76.58**	49.18	62.13	**71.30**	69.98	66.23	66.83	70.22
			500	1.91	1.89	3.31	3.23	**6.45**	4.30	**9.71**	8.54	**14.32**	8.50	**29.41**	10.55	**30.86**	14.86	**60.87**	31.98	67.23	65.46	**122.62**	58.04	119.47	116.35	124.25	121.74	120.45	123.21
			∞	*1.91*	*1.89*	*3.31*	*3.23*	***6.45***	*4.30*	***9.71***	*8.54*	***14.32***	*8.50*	***29.41***	*10.55*	***30.86***	*14.86*	***60.87***	*31.98*	*67.23*	*65.46*	***126.87***	*58.04*	***138.55***	*124.56*	*136.15*	*133.84*	*131.94*	*132.97*
ped19 n=793 k=5 w=25 h=98	16	375110		*(p=4)*		*(p=12)*		*(p=48)*		*(p=144)*		*(p=288)*		*(p=1440)*		*(p=2880)*		*(p=5752)*		*(p=7672)*		*(p=11254)*		*(p=14968)*					
			20	2.08	2.03	3.14	**5.07**	5.67	**9.01**	12.11	**13.38**	14.47	14.12	13.75	13.19	12.58	11.92	10.66	10.03	9.41	9.09	7.83	7.69	6.84	6.67				
			100	2.08	2.03	3.14	**5.07**	5.67	**9.38**	12.11	**18.22**	21.69	**30.04**	30.94	**51.43**	44.98	48.05	50.07	47.20	46.08	44.87	38.65	37.94	33.65	32.90				
			500	2.08	2.03	3.14	**5.07**	5.67	**9.38**	12.11	**18.22**	21.69	**30.04**	31.44	**72.72**	55.85	**107.98**	71.83	**113.64**	108.60	**125.75**	128.82	**157.54**	122.54	**152.17**				
			∞	*2.08*	*2.03*	*3.14*	***5.07***	*5.67*	***9.38***	*12.11*	***18.22***	*21.69*	***30.04***	*31.44*	***72.72***	*55.85*	***137.00***	*73.86*	***142.14***	*118.33*	***144.55***	*138.47*	***193.96***	*139.60*	***229.00***				
ped20 n=437 k=5 w=22 h=60	3	5136		*(p=2)*		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
			20	2.09	2.09	3.77	3.69	6.95	7.16	11.31	12.14	12.23	12.35	13.00	11.86	10.97	11.29	9.99	10.17	6.39	6.29	4.41	4.40						
			100	2.09	2.09	3.77	3.69	6.95	7.16	13.41	13.48	22.23	21.86	**46.27**	21.58	**38.91**	31.51	40.44	38.62	30.39	29.18	20.63	20.38						
			500	2.09	2.09	3.77	3.69	6.95	7.16	13.41	13.48	22.23	21.86	**54.06**	21.58	**72.34**	38.33	**98.77**	38.62	**111.65**	95.11	76.66	71.33						
			∞	*2.09*	*2.09*	*3.77*	*3.69*	*6.95*	*7.16*	*13.41*	*13.48*	*22.23*	*21.86*	***54.06***	*21.58*	***72.34***	*38.33*	***125.27***	*38.62*	***190.22***	*107.00*	***160.50***	*119.44*						
	4	2185		*(p=2)*		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
			20	3.83	3.83	9.54	9.63	**18.52**	11.38	**26.98**	24.28	26.33	28.01	27.66	29.13	19.17	19.51	14.66	14.66	5.53	5.45	2.98	3.00						
			100	3.83	3.83	9.54	9.63	**18.52**	11.38	**42.84**	24.28	72.83	70.48	**104.05**	91.04	78.04	**91.04**	64.26	68.28	25.41	24.83	13.57	13.66						
			500	3.83	3.83	9.54	9.63	**18.52**	11.38	**42.84**	24.28	72.83	70.48	**145.67**	91.04	182.08	182.08	182.08	198.64	87.40	87.40	47.50	46.49						
			∞	*3.83*	*3.83*	*9.54*	*9.63*	***18.52***	*11.38*	***42.84***	*24.28*	*72.83*	*70.48*	***145.67***	*91.04*	*182.08*	*182.08*	***218.50***	*198.64*	*198.64*	*182.08*	*109.25*	*109.25*						
ped31 n=1183 k=5 w=30 h=85				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	10	1258519	20	1.77	1.77	3.49	3.51	6.72	7.38	12.02	12.19	11.80	**13.79**	12.14	**14.08**	13.63	**15.07**	14.64	15.39	14.69	15.53	15.44	15.55	15.56	15.61	15.19	15.41	14.96	14.91
			100	1.77	1.77	3.49	3.51	6.72	7.38	12.02	12.19	**20.81**	14.84	25.96	25.32	**37.68**	34.21	40.98	**47.48**	51.36	**67.41**	64.04	**70.76**	71.92	75.31	73.68	74.98	72.07	72.20
			500	1.77	1.77	3.49	3.51	6.72	7.38	12.02	12.19	**20.81**	14.84	25.96	25.32	**44.14**	34.21	53.40	**72.23**	100.50	**126.09**	154.00	**222.04**	218.64	**306.13**	273.18	**320.32**	296.40	316.93
			∞	*1.77*	*1.77*	*3.49*	*3.51*	*6.72*	*7.38*	*12.02*	*12.19*	***20.81***	*14.84*	*25.96*	*25.32*	***44.14***	*34.21*	*53.40*	***72.23***	*100.50*	***126.09***	*177.26*	***245.90***	*314.00*	***423.89***	*619.05*	***794.52***	*915.95*	***1219.50***
	11	433029		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	2.22	2.22	4.03	3.99	7.97	7.95	14.26	14.68	12.88	**15.02**	13.87	**16.23**	19.25	17.52	18.99	17.98	18.79	17.81	18.01	17.85	18.08	17.52	17.19	17.11	16.32	16.40
			100	2.22	2.22	4.03	3.99	7.97	7.95	14.26	14.68	**22.48**	15.02	27.59	27.44	48.80	46.75	59.05	56.47	**88.79**	73.11	75.34	**85.43**	80.29	82.99	80.68	84.18	76.00	80.12
			500	2.22	2.22	4.03	3.99	7.97	7.95	14.26	14.68	**22.48**	15.02	27.59	27.44	48.80	46.75	59.05	56.47	106.32	**128.04**	181.64	**237.93**	248.01	**315.62**	293.98	318.40	258.68	283.77
			∞	*2.22*	*2.22*	*4.03*	*3.99*	*7.97*	*7.95*	*14.26*	*14.68*	***22.48***	*15.02*	*27.59*	*27.44*	*48.80*	*46.75*	*59.05*	*56.47*	*106.32*	***128.04***	*202.26*	***237.93***	*400.21*	***519.22***	*399.84*	***529.38***	*439.62*	***519.22***
	12	16238		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	2.01	2.01	4.32	4.01	6.90	**7.74**	12.51	13.27	12.10	11.03	11.10	**14.42**	11.55	**15.80**	14.95	**17.48**	14.88	**17.31**	15.39	15.66	13.94	13.85	11.25	11.22	8.23	8.24
			100	2.01	2.01	4.32	4.01	6.90	**7.74**	12.51	13.27	**19.80**	11.03	13.81	**20.02**	20.71	**34.48**	37.16	**41.32**	45.61	**61.98**	56.58	**69.69**	58.83	63.43	49.96	50.90	35.92	36.00
			500	2.01	2.01	4.32	4.01	6.90	**7.74**	12.51	13.27	**19.80**	11.03	13.81	**20.02**	21.06	**34.48**	40.00	41.32	72.17	72.17	116.82	123.95	153.19	**202.97**	141.20	**170.93**	101.49	109.72
			∞	*2.01*	*2.01*	*4.32*	*4.01*	*6.90*	***7.74***	*12.51*	*13.27*	***19.80***	*11.03*	*13.81*	***20.02***	*21.06*	***34.48***	*40.00*	*41.32*	*72.17*	*72.17*	*127.86*	*123.95*	*182.45*	***219.43***	*231.97*	***257.75***	*176.50*	*184.52*
ped33 n=798 k=4 w=28 h=98				*(p=2)*		*(p=3)*		*(p=6)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=1536)*		*(p=1536)*	
	4	6010	20	1.51	1.51	1.96	1.86	3.60	3.95	3.65	3.95	7.15	**8.44**	9.26	**12.37**	11.11	**14.11**	12.57	12.87	12.68	**14.80**	12.87	13.85	11.23	12.27	10.12	10.07	9.85	10.07
			100	1.51	1.51	1.96	1.86	3.60	3.95	3.65	3.95	7.15	**8.44**	11.58	**16.38**	16.47	**18.61**	23.85	**34.74**	25.47	**48.86**	34.74	**53.66**	34.15	**50.50**	37.80	**43.24**	34.94	**43.24**
			500	1.51	1.51	1.96	1.86	3.60	3.95	3.65	3.95	7.15	**8.44**	11.58	**16.38**	16.47	**18.61**	23.85	**34.74**	28.76	**48.86**	48.47	**66.04**	55.65	**84.65**	69.08	**101.86**	68.30	**101.86**
			∞	*1.51*	*1.51*	*1.96*	*1.86*	*3.60*	*3.95*	*3.65*	*3.95*	*7.15*	***8.44***	*11.58*	***16.38***	*16.47*	***18.61***	*23.85*	***34.74***	*28.76*	***48.86***	*48.47*	***66.04***	*59.50*	***84.65***	*78.05*	***122.65***	*77.05*	***122.65***
ped34 n=1160 k=5 w=31 h=102				*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=952)*		*(p=1896)*		*(p=3752)*	
	10	962006	20	1.96	1.96	2.27	2.27	3.42	3.49	5.49	**6.67**	6.63	6.70	6.60	**7.84**	9.22	9.50	8.81	**10.30**	9.73	10.11	9.73	9.89	9.36	9.86	9.91	9.97	9.92	10.00
			100	1.96	1.96	2.27	2.27	3.42	3.49	5.49	**6.67**	6.66	6.70	8.33	**12.73**	12.73	12.73	22.85	24.44	29.32	**39.79**	35.35	**45.52**	35.43	**45.72**	44.87	45.41	47.18	49.51
			500	1.96	1.96	2.27	2.27	3.42	3.49	5.49	**6.67**	6.66	6.70	8.33	**12.73**	12.73	12.73	23.09	24.44	38.87	**46.76**	69.26	**85.87**	83.72	89.66	144.23	156.78	182.89	197.05
			∞	*1.96*	*1.96*	*2.27*	*2.27*	*3.42*	*3.49*	*5.49*	***6.67***	*6.66*	*6.70*	*8.33*	***12.73***	*12.73*	*12.73*	*23.09*	*24.44*	*38.87*	***46.76***	*70.32*	***85.87***	*87.19*	*89.66*	*166.64*	*162.34*	*295.09*	*316.03*
	11	350574		*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=720)*		*(p=956)*		*(p=1912)*		*(p=3808)*	
			20	1.29	1.29	1.62	1.61	**2.24**	1.61	**3.37**	2.25	**3.64**	3.22	3.66	3.67	**4.12**	3.69	4.35	4.71	4.83	4.92	4.70	4.58	4.37	4.59	4.56	4.59	4.56	4.58
			100	1.29	1.29	1.62	1.61	**2.24**	1.61	**3.37**	2.25	3.67	3.65	4.39	4.43	6.75	6.83	11.66	11.70	15.07	15.05	18.29	18.71	18.22	18.63	20.97	21.71	21.60	22.39
			500	1.29	1.29	1.62	1.61	**2.24**	1.61	**3.37**	2.25	3.67	3.65	4.39	4.43	6.75	6.83	11.67	11.70	21.88	21.75	33.63	**37.95**	37.13	**42.18**	66.37	61.59	77.82	**90.42**
			∞	*1.29*	*1.29*	*1.62*	*1.61*	***2.24***	*1.61*	***3.37***	*2.25*	*3.67*	*3.65*	*4.39*	*4.43*	*6.75*	*6.83*	*11.67*	*11.70*	*21.88*	*21.75*	*33.63*	***37.95***	*37.59*	***42.18***	*75.13*	***83.59***	*140.29*	***162.68***
	12	96122		*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=948)*		*(p=1896)*		*(p=3728)*	
			20	2.25	2.22	3.66	3.65	4.13	4.08	4.70	**6.29**	6.28	6.31	6.07	6.11	6.02	**7.14**	6.87	6.97	7.88	7.58	7.50	7.35	7.23	7.34	7.41	7.16	7.08	6.92
			100	2.25	2.22	3.66	3.65	4.13	4.08	4.70	**6.29**	6.29	6.31	7.23	**10.65**	10.24	10.55	17.79	17.02	23.52	**25.94**	27.55	**32.35**	27.95	**33.34**	33.29	32.34	32.53	32.52
			500	2.25	2.22	3.66	3.65	4.13	4.08	4.70	**6.29**	6.29	6.31	7.23	**10.65**	10.24	10.55	17.83	17.02	29.11	**33.83**	45.77	**74.05**	55.95	**88.27**	94.79	97.29	**107.04**	85.90
			∞	*2.25*	*2.22*	*3.66*	*3.65*	*4.13*	*4.08*	*4.70*	***6.29***	*6.29*	*6.31*	*7.23*	***10.65***	*10.24*	*10.55*	*17.83*	*17.02*	*29.11*	***33.83***	*45.86*	***74.05***	*56.05*	***88.27***	*104.94*	***153.80***	***154.29***	*98.89*
ped39 n=1272 k=5 w=21 h=76				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
	4	6632	20	2.16	2.16	2.47	2.43	2.49	2.60	4.44	4.41	9.01	**10.87**	9.52	9.11	7.95	**9.12**	9.35	**11.72**	9.96	**11.93**	9.87	9.80	7.15	7.08				
			100	2.16	2.16	2.47	2.43	2.49	2.60	4.44	4.41	9.91	**10.94**	11.13	11.61	11.22	**13.16**	12.85	**20.86**	18.79	**27.98**	26.32	**34.91**	25.03	**28.34**				
			500	2.16	2.16	2.47	2.43	2.49	2.60	4.44	4.41	9.91	**10.94**	11.13	11.61	11.68	**13.67**	13.51	**21.82**	21.32	**32.83**	36.64	**51.41**	41.45	**49.86**				
			∞	*2.16*	*2.16*	*2.47*	*2.43*	*2.49*	*2.60*	*4.44*	*4.41*	*9.91*	***10.94***	*11.13*	*11.61*	*11.68*	***13.67***	*13.51*	***21.82***	*21.46*	***33.16***	*39.24*	***54.81***	*47.04*	***57.17***				
	5	2202		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
			20	2.14	2.14	2.78	2.54	2.94	2.91	5.38	**6.13**	5.24	**5.98**	5.23	**7.54**	5.42	**8.22**	5.22	**7.20**	5.32	**6.69**	4.97	4.87	3.11	3.05				
			100	2.14	2.14	2.78	2.54	2.94	2.91	5.38	**6.13**	6.38	6.55	6.95	7.54	7.17	**13.27**	7.92	**15.62**	10.79	**19.49**	14.12	**18.05**	10.85	**12.51**				
			500	2.14	2.14	2.78	2.54	2.94	2.91	5.38	**6.13**	6.38	6.55	7.06	7.54	7.44	**14.12**	8.70	**19.84**	13.27	**20.02**	21.38	**31.46**	20.20	**28.97**				
			∞	*2.14*	*2.14*	*2.78*	*2.54*	*2.94*	*2.91*	*5.38*	***6.13***	*6.38*	*6.55*	*7.06*	*7.54*	*7.44*	***14.12***	*8.74*	***20.20***	*13.68*	***20.02***	*23.43*	***31.46***	*25.31*	***33.88***				
Better by 10%				0x	0x	4x	4x	16x	8x	16x	16x	18x	14x	14x	21x	15x	21x	10x	20x	6x	27x	7x	26x	1x	25x	1x	10x	2x	9x
Better by 50%				0x	0x	4x	4x	12x	4x	7x	3x	10x	0x	13x	6x	8x	9x	8x	8x	3x	7x	5x	3x	0x	4x	0x	1x	1x	1x

Table B.11: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8556)*	
ped41 n=1062 k=5 w=33 h=100	9	25607	20	1.28	1.28	1.58	**1.79**	2.82	**5.57**	5.41	**8.89**	8.92	**11.84**	11.68	11.71	12.82	12.28	10.47	**12.09**	11.32	11.29	10.55	10.42	9.84	9.58	8.62	8.59	7.23	NA
			100	1.28	1.28	1.58	**1.79**	2.82	**5.57**	5.61	**10.51**	10.63	**20.39**	19.59	**32.29**	30.20	31.65	30.96	**44.46**	50.41	50.91	46.99	49.24	**43.33**	29.20	31.81	**38.45**	33.52	NA
			500	1.28	1.28	1.58	**1.79**	2.82	**5.57**	5.61	**10.51**	10.63	**20.39**	20.13	**32.29**	31.34	**41.91**	39.40	**75.31**	73.80	**96.63**	102.02	**121.36**	**75.99**	42.40	57.41	**113.81**	97.00	NA
			∞	*1.28*	*1.28*	*1.58*	***1.79***	*2.82*	***5.57***	*5.61*	***10.51***	*10.63*	***20.39***	*20.13*	***32.29***	*31.34*	***41.91***	*39.40*	***75.31***	*73.80*	***96.63***	*115.35*	***148.88***	***85.07***	*46.31*	*63.86*	***173.02***	*139.17*	*NA*
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8576)*	
	10	46819	20	1.20	1.20	1.41	**2.71**	2.64	**5.10**	4.84	**9.65**	9.31	**13.09**	12.96	14.06	14.58	14.19	14.16	14.64	15.87	15.53	15.31	14.46	14.74	14.41	13.35	13.23	11.41	NA
			100	1.20	1.20	1.41	**2.71**	2.64	**5.10**	4.89	**9.65**	9.95	**16.09**	16.95	**27.36**	28.67	**31.68**	31.57	**39.85**	47.92	47.01	**51.79**	45.37	50.89	**59.79**	**62.34**	54.06	45.28	NA
			500	1.20	1.20	1.41	**2.71**	2.64	**5.10**	4.89	**9.65**	9.95	**16.09**	17.16	**27.36**	29.35	**34.81**	36.75	39.85	57.94	59.64	**66.41**	46.87	78.03	80.72	90.21	**92.89**	72.93	NA
			∞	*1.20*	*1.20*	*1.41*	***2.71***	*2.64*	***5.10***	*4.89*	***9.65***	*9.95*	***16.09***	*17.16*	***27.36***	*29.35*	***34.81***	*36.75*	*39.85*	*57.94*	*59.64*	***67.46***	*46.87*	*82.14*	*81.57*	*91.09*	***107.63***	*82.57*	*NA*
				(p=3)		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8460)*	
	11	27583	20	1.20	1.20	1.36	**2.71**	2.69	**4.57**	4.52	**7.11**	8.45	**13.58**	12.60	**14.78**	13.34	**15.57**	13.45	**15.46**	13.78	**15.48**	13.87	14.04	12.49	13.55	12.16	12.17	9.87	NA
			100	1.20	1.20	1.36	**2.71**	2.69	**4.57**	4.57	**7.11**	9.00	**17.46**	16.83	**25.10**	22.83	**33.60**	27.31	**49.88**	32.37	**60.62**	**50.33**	34.74	30.34	**41.79**	50.33	**56.18**	42.90	NA
			500	1.20	1.20	1.36	**2.71**	2.69	**4.57**	4.57	**7.11**	9.00	**17.46**	17.08	**25.10**	23.32	**33.60**	30.65	**55.50**	40.09	**97.12**	**94.46**	45.44	41.05	41.79	106.50	**154.09**	98.86	NA
			∞	*1.20*	*1.20*	*1.36*	***2.71***	*2.69*	***4.57***	*4.57*	***7.11***	*9.00*	***17.46***	*17.08*	***25.10***	*23.32*	***33.60***	*30.65*	***55.50***	*40.09*	***97.12***	***102.16***	*47.48*	*42.96*	*41.79*	*130.11*	***235.75***	*127.70*	*NA*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
ped44 n=811 k=4 w=25 h=65	5	207136	20	1.84	1.84	3.15	3.14	**5.81**	3.16	**10.16**	6.62	12.17	11.17	16.29	15.43	14.46	14.49	12.03	12.73	11.10	11.29	10.04	10.15	8.27	8.81	7.11	**8.08**	6.08	**7.21**
			100	1.84	1.84	3.15	3.14	**5.81**	3.16	**10.16**	6.62	**17.40**	11.17	**60.65**	19.43	58.30	59.81	51.27	**57.54**	52.55	52.32	48.38	45.51	40.51	37.32	35.00	37.79	29.94	**34.57**
			500	1.84	1.84	3.15	3.14	**5.81**	3.16	**10.16**	6.62	**17.40**	11.17	**60.65**	19.43	**99.58**	79.00	**133.29**	110.89	**200.52**	161.95	**207.14**	147.43	**175.24**	95.59	**163.10**	130.77	139.11	132.02
			∞	*1.84*	*1.84*	*3.15*	*3.14*	***5.81***	*3.16*	***10.16***	*6.62*	***17.40***	*11.17*	***60.65***	*19.43*	***99.58***	*79.00*	***151.08***	*110.89*	***342.94***	*198.98*	***565.95***	*206.72*	***367.26***	*112.57*	***281.05***	*224.90*	***1025.43***	*246.59*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
	6	95830	20	1.82	1.82	3.58	3.57	6.28	6.25	9.86	10.11	10.08	**14.98**	14.22	14.07	13.11	13.07	12.04	12.06	10.55	10.95	9.49	9.82	9.05	8.85	7.72	7.68	5.79	**6.51**
			100	1.82	1.82	3.58	3.57	6.28	6.25	9.86	10.11	16.14	16.06	**40.88**	26.72	51.74	**58.58**	53.27	56.37	48.97	51.49	45.08	42.10	43.48	42.16	37.65	37.68	28.19	**31.57**
			500	1.82	1.82	3.58	3.57	6.28	6.25	9.86	10.11	16.14	16.06	**57.76**	26.72	**94.32**	70.88	**164.37**	108.16	175.51	167.83	**178.79**	105.89	**182.53**	120.69	**168.42**	116.30	124.29	**137.10**
			∞	*1.82*	*1.82*	*3.58*	*3.57*	*6.28*	*6.25*	*9.86*	*10.11*	*16.14*	*16.06*	***57.76***	*26.72*	***94.32***	*70.88*	***208.78***	*108.16*	***356.25***	*225.48*	***221.32***	*123.81*	***407.79***	*140.93*	***930.39***	*137.69*	***709.85***	*418.47*
				(p=2)		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14401)*											
ped50 n=514 k=6 w=17 h=47	3	4135	20	1.74	1.74	2.78	2.80	6.96	7.04	9.78	**11.99**	11.27	12.02	8.89	9.17	5.01	5.04	2.38	2.39										
			100	1.74	1.74	2.78	2.80	6.96	7.04	11.99	11.99	16.28	**31.56**	20.88	**39.01**	20.68	**22.97**	10.85	10.94										
			500	1.74	1.74	2.78	2.80	6.96	7.04	11.99	11.99	17.52	**46.99**	26.01	**46.99**	33.08	**75.18**	32.56	**37.25**										
			∞	*1.74*	*1.74*	*2.78*	*2.80*	*6.96*	*7.04*	*11.99*	*11.99*	*17.52*	***49.23***	*27.03*	***46.99***	*37.94*	***98.45***	*46.46*	***78.02***										
				(p=2)		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14400)*											
	4	1780	20	3.57	3.57	6.54	6.98	23.12	23.42	23.73	**26.57**	14.24	14.02	6.52	6.43	2.73	2.75	1.15	1.15										
			100	3.57	3.57	6.54	6.98	23.12	23.42	42.38	43.41	46.84	**59.33**	28.71	29.18	12.45	12.54	5.20	5.24										
			500	3.57	3.57	6.54	6.98	23.12	23.42	42.38	43.41	65.93	**89.00**	65.93	**93.68**	42.38	43.41	17.62	17.98										
			∞	*3.57*	*3.57*	*6.54*	*6.98*	*23.12*	*23.42*	*42.38*	*43.41*	*65.93*	***89.00***	*80.91*	***104.71***	*68.46*	***93.68***	*38.70*	*42.38*											
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4064)*		*(p=7968)*	
ped51 n=1152 k=5 w=39 h=98	20	101788	20	1.94	1.94	3.73	3.73	7.52	7.42	12.32	13.22	11.92	**15.33**	14.44	**16.35**	16.00	17.37	15.89	17.30	16.02	16.71	15.49	15.49	13.26	13.40	10.28	10.32	7.30	7.25
			100	1.94	1.94	3.73	3.73	7.52	7.42	12.32	13.22	17.88	18.41	29.44	**38.29**	37.50	**41.70**	43.50	**60.34**	62.99	59.49	**64.50**	46.10	55.59	**65.25**	46.56	50.27	34.42	35.29
			500	1.94	1.94	3.73	3.73	7.52	7.42	12.32	13.22	17.88	18.41	29.44	**38.29**	37.50	**41.70**	57.44	**87.52**	111.37	113.86	**144.59**	66.75	139.05	139.63	119.47	**136.08**	**133.58**	107.83
			∞	*1.94*	*1.94*	*3.73*	*3.73*	*7.52*	*7.42*	*12.32*	*13.22*	*17.88*	*18.41*	*29.44*	***38.29***	*37.50*	***41.70***	*57.44*	***87.52***	*111.37*	*113.86*	***149.47***	*70.69*	*145.62*	*156.84*	*147.73*	*150.80*	***346.22***	*122.93*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
	21	164817	20	2.05	2.05	3.82	3.88	7.74	7.69	14.28	14.61	11.73	**16.28**	**18.25**	16.47	17.23	17.07	18.89	17.33	18.00	17.12	16.13	15.71	13.34	13.02	9.71	9.63	6.37	6.36
			100	2.05	2.05	3.82	3.88	7.74	7.69	14.28	14.61	20.64	20.33	33.86	33.86	50.13	49.10	**70.16**	55.87	**86.25**	62.64	77.31	73.15	57.11	**63.51**	46.60	47.09	31.10	31.13
			500	2.05	2.05	3.82	3.88	7.74	7.69	14.28	14.61	20.64	20.33	33.86	33.86	50.13	49.10	70.16	**88.61**	**136.21**	102.63	**181.52**	104.91	164.16	**247.47**	**156.97**	141.47	104.78	110.99
			∞	*2.05*	*2.05*	*3.82*	*3.88*	*7.74*	*7.69*	*14.28*	*14.61*	*20.64*	*20.33*	*33.86*	*33.86*	*50.13*	*49.10*	*70.16*	***88.61***	***136.21***	*102.63*	***181.52***	*104.91*	*258.33*	***290.68***	*195.05*	*211.85*	*193.45*	***222.73***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*	
ped7 n=1068 k=4 w=32 h=90	6	118383	20	1.94	2.08	**3.35**	2.01	**5.72**	2.01	**9.59**	2.04	**11.56**	4.94	13.11	13.90	13.60	**15.85**	12.26	**16.17**	13.62	**15.61**	13.60	**15.61**	13.62	**15.30**	14.37	15.39	14.18	14.52
			100	1.94	2.08	**3.35**	2.01	**5.72**	2.01	**9.90**	2.04	**17.84**	4.94	**23.11**	15.39	24.83	**46.33**	24.36	**51.34**	29.27	**65.26**	30.13	**65.26**	37.64	**67.84**	44.77	**71.79**	44.74	**68.87**
			500	1.94	2.08	**3.35**	2.01	**5.72**	2.01	**9.90**	2.04	**17.84**	4.94	**23.75**	15.39	26.65	**49.59**	27.16	**56.75**	36.14	**90.99**	35.94	**90.99**	54.40	**132.72**	67.11	**125.54**	75.69	**214.85**
			∞	*1.94*	*2.08*	***3.35***	*2.01*	***5.72***	*2.01*	***9.90***	*2.04*	***17.84***	*4.94*	***23.75***	*15.39*	*26.65*	***49.59***	*27.16*	***56.75***	*36.22*	***90.99***	*36.36*	***90.99***	*56.32*	***132.72***	*68.04*	***135.14***	*87.95*	***237.24***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*	
	7	93380	20	2.00	1.91	**3.72**	1.97	**5.85**	1.81	**11.69**	1.82	**14.99**	3.51	**18.62**	3.58	18.71	17.43	15.80	**17.40**	15.30	**17.10**	15.41	**17.10**	16.55	16.90	15.76	16.37	15.21	15.31
			100	2.00	1.91	**3.72**	1.97	**5.85**	1.81	**11.69**	1.82	**18.95**	3.71	**31.69**	3.69	35.01	**43.72**	31.11	**46.76**	33.72	**62.05**	33.13	**62.05**	42.12	**73.12**	43.31	**73.64**	45.00	**69.22**
			500	2.00	1.91	**3.72**	1.97	**5.85**	1.81	**11.69**	1.82	**18.95**	3.71	**31.69**	3.69	35.01	**51.11**	35.71	**68.26**	39.77	**79.14**	39.22	**79.14**	58.00	**184.55**	65.16	**182.03**	68.36	**186.39**
			∞	*2.00*	*1.91*	***3.72***	*1.97*	***5.85***	*1.81*	***11.69***	*1.82*	***18.95***	*3.71*	***31.69***	*3.69*	*35.01*	***51.11***	*35.71*	***68.26***	*39.77*	***79.14***	*39.22*	***79.14***	*58.07*	***209.84***	*65.71*	***237.01***	*71.17*	***251.02***
				(p=2)		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1276)*		*(p=1276)*		*(p=2552)*		*(p=3816)*		*(p=7588)*	
	8	30717	20	1.79	1.79	**3.45**	1.68	**5.65**	1.68	**10.32**	1.67	**13.35**	3.00	**13.10**	2.96	13.36	14.46	13.94	**15.85**	11.66	**17.20**	13.99	**17.20**	14.56	15.87	14.56	14.85	12.05	12.27
			100	1.79	1.79	**3.45**	1.68	**5.65**	1.68	**10.32**	1.67	**13.61**	3.11	**24.07**	3.12	24.99	**31.06**	26.80	**30.59**	23.52	**32.54**	28.57	**32.54**	31.86	**40.79**	33.53	**51.11**	31.54	**52.69**
			500	1.79	1.79	**3.45**	1.68	**5.65**	1.68	**10.32**	1.67	**13.61**	3.11	**24.46**	3.12	26.55	**31.06**	28.18	**35.97**	27.82	**36.88**	30.47	**36.88**	40.96	**57.20**	45.04	**89.03**	43.88	**112.52**
			∞	*1.79*	*1.79*	***3.45***	*1.68*	***5.65***	*1.68*	***10.32***	*1.67*	***13.61***	*3.11*	***24.46***	*3.12*	*26.55*	***31.06***	*28.18*	***35.97***	*28.08*	***36.88***	*30.47*	***36.88***	*41.57*	***61.80***	*47.40*	***104.48***	*47.70*	***142.21***
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
	6	101172	20	1.96	1.94	**3.66**	1.94	**5.87**	4.00	**9.76**	5.68	13.99	13.75	13.89	13.75	**16.39**	14.02	16.23	15.43	16.94	16.90	17.02	16.59	15.26	15.86	15.22	15.06	14.02	14.07
			100	1.96	1.94	**3.66**	1.94	**5.87**	4.00	**9.76**	5.68	**18.82**	14.42	**18.60**	14.42	**38.72**	19.86	**64.90**	37.78	**64.94**	55.41	79.48	78.19	71.91	75.50	72.79	72.01	**67.86**	58.79
			500	1.96	1.94	**3.66**	1.94	**5.87**	4.00	**9.76**	5.68	**18.82**	14.42	**18.60**	14.42	**38.72**	19.86	**71.96**	37.78	**99.97**	55.41	256.13	108.67	216.64	216.64	**293.25**	263.47	**292.40**	115.10
			∞	*1.96*	*1.94*	***3.66***	*1.94*	***5.87***	*4.00*	***9.76***	*5.68*	***18.82***	*14.42*	***18.60***	*14.42*	***38.72***	*19.86*	***71.96***	*37.78*	***99.97***	*55.41*	***256.13***	*108.67*	*216.64*	*216.64*	***595.13***	*315.18*	***790.41***	*134.54*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
ped9 n=1118 k=7 w=27 h=100	7	58657	20	1.89	1.92	3.81	3.75	5.81	5.75	**10.90**	9.66	11.32	12.24	11.39	12.24	14.92	14.75	16.10	15.66	16.49	16.16	16.68	16.47	15.67	15.58	15.33	14.91	14.22	14.02
			100	1.89	1.92	3.81	3.75	5.81	5.75	**10.90**	9.66	19.44	20.07	19.84	20.07	**40.26**	26.85	55.81	52.00	66.35	62.20	72.78	76.88	**71.10**	55.97	71.88	70.33	66.96	65.98
			500	1.89	1.92	3.81	3.75	5.81	5.75	**10.90**	9.66	19.44	20.07	19.84	20.07	**40.26**	26.85	**77.90**	52.00	**129.20**	103.45	**250.67**	164.77	**234.63**	82.85	**265.42**	142.03	258.40	243.39
			∞	*1.89*	*1.92*	*3.81*	*3.75*	*5.81*	*5.75*	***10.90***	*9.66*	*19.44*	*20.07*	*19.84*	*20.07*	***40.26***	*26.85*	***77.90***	*52.00*	***129.20***	*103.45*	***250.67***	*164.77*	***383.38***	*86.64*	***592.49***	*142.03*	***630.72***	*444.37*
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
	8	41061	20	1.97	1.97	3.73	3.76	5.84	5.94	8.51	9.31	11.19	**12.78**	11.30	**12.78**	**14.75**	11.63	**16.04**	12.53	17.02	17.06	16.84	16.73	15.58	15.96	15.01	14.94	13.42	13.64
			100	1.97	1.97	3.73	3.76	5.84	5.94	8.51	9.31	**18.34**	13.01	**18.18**	13.01	**31.06**	21.81	**55.04**	22.49	**65.59**	42.07	75.90	70.43	71.16	74.52	71.04	71.53	63.46	64.46
			500	1.97	1.97	3.73	3.76	5.84	5.94	8.51	9.31	**18.34**	13.01	**18.18**	13.01	**31.06**	21.81	**67.98**	22.49	**97.76**	44.29	**161.02**	88.30	**214.98**	145.61	264.91	271.93	244.41	245.87
			∞	*1.97*	*1.97*	*3.73*	*3.76*	*5.84*	*5.94*	*8.51*	*9.31*	***18.34***	*13.01*	***18.18***	*13.01*	***31.06***	*21.81*	***67.98***	*22.49*	***97.76***	*44.29*	***161.02***	*88.30*	***268.37***	*145.61*	***526.42***	*380.19*	***586.59***	*441.52*
Better by 10%				0x	0x	16x	12x	20x	12x	24x	14x	21x	22x	24x	20x	15x	27x	14x	29x	14x	18x	21x	14x	12x	16x	11x	18x	9x	15x
Better by 50%				0x	0x	16x	8x	16x	12x	20x	12x	15x	13x	17x	9x	3x	5x	8x	14x	7x	9x	15x	6x	9x	7x	4x	12x	5x	9x

Table B.12: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1		2		3		4		5		6		7		8		9		10		11		12		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
IF3-11-57 n=2670 k=3 w=37 h=95	15	121311		*(p=2)*		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
			20	1.53	1.53	2.40	2.19	**3.56**	2.46	**4.49**	3.72	8.20	8.48	8.11	8.48	8.41	**11.62**	8.51	**11.62**	9.34	**11.38**	10.16	**11.71**	10.40	11.16	10.90	10.51	10.99	10.48
			100	1.53	1.53	2.40	2.19	**3.56**	2.46	**4.49**	3.72	8.20	8.48	8.11	8.48	9.30	**15.14**	9.43	**15.14**	9.43	**20.40**	14.14	**33.07**	18.61	**38.54**	28.63	**46.77**	30.04	**46.46**
			500	1.53	1.53	2.40	2.19	**3.56**	2.46	**4.49**	3.72	8.20	8.48	8.11	8.48	9.30	**15.14**	9.43	**15.14**	9.43	**20.40**	14.14	**33.07**	18.66	**50.15**	33.01	**127.56**	38.51	**136.15**
			∞	*1.53*	*1.53*	*2.40*	*2.19*	***3.56***	*2.46*	***4.49***	*3.72*	*8.20*	*8.48*	*8.11*	*8.48*	*9.30*	***15.14***	*9.43*	***15.14***	*9.43*	***20.40***	*14.14*	***33.07***	*18.66*	***50.15***	*33.01*	***132.58***	*39.15*	***138.32***
				(p=2)		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
	16	35820	20	1.43	1.43	2.13	2.16	**3.69**	2.63	4.76	**6.34**	7.64	7.58	7.32	7.58	**10.13**	9.18	**10.16**	9.18	10.37	**12.00**	10.23	**11.97**	11.93	12.17	9.62	10.34	10.21	10.11
			100	1.43	1.43	2.13	2.16	**3.69**	2.63	4.76	**6.34**	**8.50**	7.58	8.30	7.58	10.59	10.49	10.16	10.49	10.39	**15.77**	16.03	**22.18**	20.67	**35.19**	28.61	**38.35**	30.99	**44.66**
			500	1.43	1.43	2.13	2.16	**3.69**	2.63	4.76	**6.34**	**8.50**	7.58	8.30	7.58	10.59	10.49	10.16	10.49	10.39	**15.77**	16.03	**22.18**	20.67	**36.93**	33.14	**58.06**	38.31	**91.38**
			∞	*1.43*	*1.43*	*2.13*	*2.16*	***3.69***	*2.63*	*4.76*	***6.34***	***8.50***	*7.58*	*8.30*	*7.58*	*10.59*	*10.49*	*10.16*	*10.49*	*10.39*	***15.77***	*16.03*	***22.18***	*20.67*	***36.93***	*33.14*	***58.06***	*38.31*	***93.28***
				(p=2)		*(p=4)*		*(p=6)*		*(p=18)*		*(p=30)*		*(p=30)*		*(p=60)*		*(p=60)*		*(p=120)*		*(p=180)*		*(p=360)*		*(p=1080)*		*(p=1440)*	
	17	18312	20	1.45	1.45	2.10	**2.67**	**3.38**	2.61	4.66	**5.64**	8.01	7.52	7.69	7.52	9.47	9.91	9.15	9.91	8.95	**11.99**	10.95	11.41	9.26	**10.33**	7.64	7.41	6.51	6.63
			100	1.45	1.45	2.10	**2.67**	**3.38**	2.61	4.66	**5.64**	8.01	7.52	7.69	7.52	9.47	9.91	9.50	9.91	9.86	**18.88**	15.81	**25.26**	19.19	**34.49**	23.81	**33.66**	23.24	**31.30**
			500	1.45	1.45	2.10	**2.67**	**3.38**	2.61	4.66	**5.64**	8.01	7.52	7.69	7.52	9.47	9.91	9.50	9.91	9.86	**18.88**	15.81	**25.26**	19.80	**34.49**	32.64	**48.83**	32.41	**90.65**
			∞	*1.45*	*1.45*	*2.10*	***2.67***	***3.38***	*2.61*	*4.66*	***5.64***	*8.01*	*7.52*	*7.69*	*7.52*	*9.47*	*9.91*	*9.50*	*9.91*	*9.86*	***18.88***	*15.81*	***25.26***	*19.80*	***34.49***	*32.64*	***48.83***	*33.36*	***104.05***
IF3-11-59 n=2711 k=3 w=32 h=73				*(p=3)*		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=600)*		*(p=1000)*		*(p=2000)*		*(p=2000)*		*(p=4000)*	
	14	35457	20	1.61	1.61	3.39	3.13	5.62	5.95	5.55	5.95	7.40	**9.36**	9.18	**12.84**	11.03	**12.37**	10.64	**12.08**	10.02	10.42	8.69	**9.67**	8.23	8.53	8.25	8.53	7.33	7.30
			100	1.61	1.61	3.39	3.13	5.62	5.95	5.55	5.95	7.40	**9.36**	9.18	**14.27**	16.51	**29.06**	22.92	**42.62**	30.54	**49.11**	28.73	**45.87**	27.09	**40.71**	35.25	**40.71**	26.96	**34.80**
			500	1.61	1.61	3.39	3.13	5.62	5.95	5.55	5.95	7.40	**9.36**	9.18	**14.27**	16.51	**29.06**	22.92	**46.05**	31.43	**103.98**	53.97	**154.83**	48.18	**116.63**	51.09	**116.63**	50.01	**136.90**
			∞	*1.61*	*1.61*	*3.39*	*3.13*	*5.62*	*5.95*	*5.55*	*5.95*	*7.40*	***9.36***	*9.18*	***14.27***	*16.51*	***29.06***	*22.92*	***46.05***	*31.43*	***103.98***	*58.80*	***154.83***	*55.58*	***146.52***	*51.09*	***146.52***	*61.56*	***274.86***
				(p=3)		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=596)*		*(p=992)*		*(p=1962)*		*(p=1962)*		*(p=3886)*	
	15	8523	20	1.78	1.67	**3.22**	2.91	5.34	5.36	5.35	5.36	7.63	7.56	7.42	**12.93**	8.89	**10.84**	8.70	**10.07**	7.09	7.54	6.74	6.53	5.00	5.14	4.97	5.14	3.73	3.77
			100	1.78	1.67	**3.22**	2.91	5.34	5.36	5.35	5.36	7.63	7.56	7.42	**13.01**	16.08	**35.22**	16.30	**29.39**	18.17	**34.79**	25.07	**30.44**	21.10	**23.94**	20.94	**23.94**	16.78	17.22
			500	1.78	1.67	**3.22**	2.91	5.34	5.36	5.35	5.36	7.63	7.56	7.42	**13.01**	16.08	**35.22**	16.30	**29.39**	23.54	**84.39**	45.82	**101.46**	52.61	**82.75**	51.97	**82.75**	48.43	**60.02**
			∞	*1.78*	*1.67*	***3.22***	*2.91*	*5.34*	*5.36*	*5.35*	*5.36*	*7.63*	*7.56*	*7.42*	***13.01***	*16.08*	***35.22***	*16.30*	***29.39***	*23.54*	***84.39***	*45.82*	***109.27***	*55.34*	***127.21***	*51.97*	***127.21***	*69.86*	***109.27***
				(p=3)		*(p=5)*		*(p=10)*		*(p=10)*		*(p=30)*		*(p=50)*		*(p=150)*		*(p=200)*		*(p=600)*		*(p=1000)*		*(p=1999)*		*(p=1999)*		*(p=3992)*	
	16	3023	20	1.69	1.62	3.39	3.42	4.09	**6.12**	6.07	6.12	6.84	**8.21**	4.43	**8.76**	6.78	7.34	6.22	6.39	3.84	3.62	2.73	2.68	1.66	1.70	1.66	1.70	0.99	1.00
			100	1.69	1.62	3.39	3.42	4.09	**6.12**	6.07	6.12	8.13	8.21	5.73	**12.29**	18.21	19.25	16.89	**19.25**	14.46	**16.17**	11.67	12.09	7.65	7.77	7.52	7.77	4.55	4.62
			500	1.69	1.62	3.39	3.42	4.09	**6.12**	6.07	6.12	8.13	8.21	5.73	**12.29**	18.55	19.25	20.29	19.25	21.14	**42.58**	29.64	**37.79**	23.43	**26.99**	22.73	**26.99**	15.91	16.52
			∞	*1.69*	*1.62*	*3.39*	*3.42*	*4.09*	***6.12***	*6.07*	*6.12*	*8.13*	*8.21*	*5.73*	***12.29***	*18.55*	*19.25*	*20.29*	*19.25*	*21.14*	***42.58***	*33.22*	***48.76***	*38.27*	***46.51***	*35.56*	***46.51***	*33.59*	***37.79***
				(p=2)		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=600)*		*(p=1200)*		*(p=2000)*		*(p=4000)*		*(p=6400)*	
	14	46464	20	2.11	2.11	3.17	**3.56**	**4.95**	3.89	**11.91**	4.94	14.43	15.45	13.32	**18.53**	16.87	**18.86**	17.13	**18.86**	17.24	18.14	16.53	17.57	16.62	17.04	15.22	15.17	13.43	13.50
			100	2.11	2.11	3.17	**3.56**	**4.95**	3.89	**11.91**	4.94	**19.77**	15.45	**25.97**	22.84	34.75	**42.67**	30.77	**42.67**	40.97	44.51	55.85	55.78	65.63	**79.56**	70.29	70.72	61.14	61.87
			500	2.11	2.11	3.17	**3.56**	**4.95**	3.89	**11.91**	4.94	**19.77**	15.45	**25.97**	22.84	38.02	**42.67**	32.65	**42.67**	**52.56**	45.11	63.04	**73.75**	122.27	**136.66**	**239.51**	181.50	207.43	206.51
			∞	*2.11*	*2.11*	*3.17*	***3.56***	***4.95***	*3.89*	***11.91***	*4.94*	***19.77***	*15.45*	***25.97***	*22.84*	*38.02*	***42.67***	*32.65*	***42.67***	***52.56***	*45.11*	*63.04*	***73.75***	*122.27*	***136.66***	***320.44***	*181.50*	***357.42***	*258.13*
IF3-13-58 n=3352 k=3 w=31 h=88				*(p=2)*		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=600)*		*(p=1200)*		*(p=1998)*		*(p=3990)*		*(p=6390)*	
	16	20270	20	1.64	1.64	2.70	2.70	4.00	3.98	8.74	8.80	12.30	**14.89**	14.48	**16.53**	16.29	17.60	15.80	**17.60**	14.84	14.59	15.17	15.19	12.87	12.94	9.55	9.48	7.17	7.18
			100	1.64	1.64	2.70	2.70	4.00	3.98	8.74	8.80	13.71	14.89	18.36	**24.93**	24.60	**39.90**	23.62	**39.90**	25.12	**33.45**	55.23	59.62	47.69	**59.62**	43.31	43.13	31.87	32.17
			500	1.64	1.64	2.70	2.70	4.00	3.98	8.74	8.80	13.71	14.89	18.36	**24.93**	25.34	**39.90**	24.33	**39.90**	27.32	**41.37**	79.18	82.06	67.12	**113.24**	137.89	144.79	97.92	106.68
			∞	*1.64*	*1.64*	*2.70*	*2.70*	*4.00*	*3.98*	*8.74*	*8.80*	*13.71*	*14.89*	*18.36*	***24.93***	*25.34*	***39.90***	*24.33*	***39.90***	*27.32*	***41.37***	*82.06*	*82.06*	*71.88*	***113.24***	*184.27*	*191.23*	*174.74*	*182.61*
				(p=2)		*(p=4)*		*(p=12)*		*(p=20)*		*(p=60)*		*(p=100)*		*(p=200)*		*(p=200)*		*(p=591)*		*(p=1181)*		*(p=1958)*		*(p=3858)*		*(p=6121)*	
	18	7647	20	1.98	1.98	3.23	3.20	4.49	4.79	**8.38**	7.02	9.74	**15.67**	12.08	**14.91**	10.82	**12.64**	11.64	12.64	7.29	7.47	4.63	4.63	3.06	3.08	1.69	1.69	1.10	1.11
			100	1.98	1.98	3.23	3.20	4.49	4.79	**8.38**	7.02	13.01	**15.67**	23.46	22.76	26.01	**36.41**	26.55	**36.41**	22.10	**32.54**	19.76	21.01	13.51	14.21	7.91	7.93	5.19	5.22
			500	1.98	1.98	3.23	3.20	4.49	4.79	**8.38**	7.02	13.01	**15.67**	23.46	22.76	29.19	**36.41**	29.64	**36.41**	23.97	**66.50**	45.52	**64.26**	43.20	**51.67**	28.86	30.59	19.66	20.39
			∞	*1.98*	*1.98*	*3.23*	*3.20*	*4.49*	*4.79*	***8.38***	*7.02*	*13.01*	***15.67***	*23.46*	*22.76*	*29.19*	***36.41***	*29.64*	***36.41***	*23.97*	***66.50***	*53.48*	***81.35***	*74.24*	*76.47*	*72.83*	*76.47*	*59.28*	*63.73*
				(p=2)		*(p=4)*		*(p=12)*		*(p=16)*		*(p=34)*		*(p=46)*		*(p=78)*		*(p=201)*		*(p=358)*		*(p=632)*		*(p=1093)*		*(p=1927)*		*(p=2831)*	
	17	345544	20	1.75	1.75	3.50	3.35	3.47	3.45	3.52	3.57	3.67	**5.80**	4.25	**9.42**	4.33	**10.49**	8.73	**13.91**	10.16	**13.86**	11.27	**13.99**	12.02	**14.36**	12.97	**16.75**	12.96	**16.31**
			100	1.75	1.75	3.50	3.35	3.47	3.45	3.52	3.57	3.67	**5.80**	4.25	**10.32**	4.33	**10.49**	8.73	**21.25**	10.68	**27.31**	15.68	**27.31**	20.82	**38.42**	27.44	**63.14**	27.37	**63.86**
			500	1.75	1.75	3.50	3.35	3.47	3.45	3.52	3.57	3.67	**5.80**	4.25	**10.32**	4.33	**10.49**	8.73	**21.25**	10.68	**28.16**	15.68	**27.81**	20.82	**47.18**	27.44	**72.50**	28.25	**73.10**
			∞	*1.75*	*1.75*	*3.50*	*3.35*	*3.47*	*3.45*	*3.52*	*3.57*	*3.67*	***5.80***	*4.25*	***10.32***	*4.33*	***10.49***	*8.73*	***21.25***	*10.68*	***28.16***	*15.68*	***27.81***	*20.82*	***47.18***	*27.44*	***72.50***	*28.25*	***73.10***
				(p=2)		*(p=4)*		*(p=12)*		*(p=16)*		*(p=32)*		*(p=44)*		*(p=68)*		*(p=165)*		*(p=284)*		*(p=526)*		*(p=912)*		*(p=1572)*		*(p=2496)*	
	18	98346	20	1.80	1.80	3.47	3.29	3.44	3.29	3.45	3.56	3.66	**4.04**	4.21	**9.89**	4.13	**10.12**	8.13	**13.42**	9.11	**14.04**	9.48	**12.82**	9.68	**13.15**	10.23	**11.80**	9.89	10.78
			100	1.80	1.80	3.47	3.29	3.44	3.29	3.45	3.56	3.66	**4.04**	4.21	**9.89**	4.13	**10.12**	8.13	**24.06**	10.03	**23.78**	14.39	**26.66**	18.20	**35.27**	21.70	**46.37**	20.92	**37.94**
			500	1.80	1.80	3.47	3.29	3.44	3.29	3.45	3.56	3.66	**4.04**	4.21	**9.89**	4.13	**10.12**	8.13	**24.06**	10.03	**23.78**	14.39	**28.01**	18.39	**35.27**	21.93	**52.51**	24.46	**51.93**
			∞	*1.80*	*1.80*	*3.47*	*3.29*	*3.44*	*3.29*	*3.45*	*3.56*	*3.66*	***4.04***	*4.21*	***9.89***	*4.13*	***10.12***	*8.13*	***24.06***	*10.03*	***23.78***	*14.39*	***28.01***	*18.39*	***35.27***	*21.93*	***52.51***	*24.46*	***52.99***
Better by 10%				0x	0x	4x	8x	16x	4x	12x	8x	6x	18x	3x	26x	1x	31x	1x	32x	2x	35x	0x	32x	0x	33x	2x	25x	1x	22x
Better by 50%				0x	0x	0x	0x	0x	0x	4x	0x	0x	5x	0x	19x	0x	20x	0x	20x	0x	28x	0x	18x	0x	22x	1x	15x	0x	16x

Table B.13: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d																									
				1		2		3		4		5		6		7		8		9		10		11		12		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
				(p=2)		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=942)*		*(p=1855)*		*(p=3633)*		*(p=7098)*		*(p=13781)*	
			20	1.90	1.90	3.62	3.60	**5.41**	4.25	**9.75**	5.93	10.07	**13.84**	11.03	**17.52**	15.98	**17.77**	14.67	**16.70**	13.99	14.61	11.19	11.84	8.87	8.67	5.79	5.72	3.52	3.49
	18	28613	100	1.90	1.90	3.62	3.60	**5.41**	4.25	**9.75**	5.93	13.92	13.84	16.49	**21.63**	32.04	**48.91**	30.63	**60.11**	34.72	**61.93**	37.35	**54.40**	39.20	40.47	26.47	26.49	16.14	15.97
IF3-15-59			500	1.90	1.90	3.62	3.60	**5.41**	4.25	**9.75**	5.93	13.92	13.84	16.49	**21.63**	33.98	**53.48**	34.98	**72.99**	44.99	**61.93**	68.78	**161.66**	89.70	**135.61**	79.92	**95.70**	55.24	56.32
n=3730			∞	*1.90*	*1.90*	*3.62*	*3.60*	***5.41***	*4.25*	***9.75***	*5.93*	*13.92*	*13.84*	*16.49*	***21.63***	*33.98*	***53.48***	*34.98*	***72.99***	*44.99*	***61.93***	*80.37*	***175.54***	*114.45*	***225.30***	*138.90*	***202.93***	*113.09*	***124.95***
k=3				*(p=2)*		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=936)*		*(p=1830)*		*(p=3571)*		*(p=6964)*		*(p=13482)*	
w=31			20	1.46	1.46	2.73	2.76	4.23	4.26	7.33	7.92	11.76	12.67	11.87	**15.22**	16.49	18.06	15.78	17.32	15.18	15.17	12.44	12.36	9.15	9.07	6.00	5.98	3.63	3.64
h=84	19	43307	100	1.46	1.46	2.73	2.76	4.23	4.26	7.33	7.92	11.76	12.67	12.61	**19.27**	29.16	**40.14**	28.99	**59.90**	33.42	**65.82**	46.67	**58.44**	41.56	42.96	28.12	28.19	17.09	17.04
			500	1.46	1.46	2.73	2.76	4.23	4.26	7.33	7.92	11.76	12.67	12.61	**19.27**	29.16	**40.14**	30.63	**75.58**	38.91	**103.85**	62.58	**166.57**	85.25	**148.82**	104.35	108.54	65.62	64.93
			∞	*1.46*	*1.46*	*2.73*	*2.76*	*4.23*	*4.26*	*7.33*	*7.92*	*11.76*	*12.67*	*12.61*	***19.27***	*29.16*	***40.14***	*30.63*	***75.58***	*38.91*	***103.85***	*64.16*	***166.57***	*97.98*	***244.67***	*139.25*	***229.14***	*136.62*	***172.54***
						(p=3)		*(p=9)*		*(p=15)*		*(p=43)*		*(p=71)*		*(p=205)*		*(p=470)*		*(p=934)*		*(p=1827)*		*(p=2707)*		*(p=7582)*			
			20			1.63	1.64	2.94	2.96	4.71	4.74	5.80	**9.43**	9.11	**11.41**	11.50	**12.69**	11.37	11.50	**11.80**	10.65	**11.73**	10.65	**10.51**	9.52	9.80	8.93	**8.69**	7.51
	15	1891710	100			1.63	1.64	2.94	2.96	4.71	4.74	5.97	**10.13**	10.59	11.41	15.84	**44.49**	29.30	**47.75**	41.00	**49.42**	39.95	**49.42**	39.94	**46.62**	41.31	44.08	38.80	37.50
IF3-16-56			500			1.63	1.64	2.94	2.96	4.71	4.74	5.97	**10.13**	10.59	11.41	15.89	**44.49**	33.34	**88.98**	52.68	**120.08**	50.97	**120.08**	65.92	**167.27**	77.56	**171.18**	105.86	**169.42**
n=3930			∞			*1.63*	*1.64*	*2.94*	*2.96*	*4.71*	*4.74*	*5.97*	***10.13***	*10.59*	*11.41*	*15.89*	***44.49***	*33.34*	***88.98***	*52.70*	***120.08***	*51.03*	***120.08***	*70.79*	***185.01***	*82.81*	***238.31***	*124.14*	***443.85***
k=3				*(p=2)*		*(p=3)*		*(p=9)*		*(p=15)*		*(p=42)*		*(p=70)*		*(p=201)*		*(p=455)*		*(p=900)*		*(p=900)*		*(p=1766)*		*(p=2629)*		*(p=7122)*	
w=38			20	1.28	1.28	1.52	1.51	2.68	**3.91**	3.88	**5.74**	5.48	**8.62**	8.96	8.98	9.21	**10.29**	8.74	9.25	8.70	8.32	8.97	8.32	**8.15**	7.24	**7.42**	6.64	**6.22**	5.38
h=77	16	489614	100	1.28	1.28	1.52	1.51	2.68	**3.91**	3.88	**5.74**	6.02	**8.62**	11.21	**13.38**	14.56	**18.57**	23.91	**37.01**	25.60	**37.79**	29.75	**37.79**	32.30	**35.86**	28.18	**33.07**	25.11	26.77
			500	1.28	1.28	1.52	1.51	2.68	**3.91**	3.88	**5.74**	6.02	**8.62**	11.21	**13.38**	14.65	**18.57**	28.00	**58.25**	37.83	**75.16**	37.06	**75.16**	48.74	**92.02**	52.00	**115.64**	60.02	**118.90**
			∞	*1.28*	*1.28*	*1.52*	*1.51*	*2.68*	***3.91***	*3.88*	***5.74***	*6.02*	***8.62***	*11.21*	***13.38***	*14.65*	***18.57***	*28.00*	***58.25***	*38.87*	***75.16***	*37.13*	***75.16***	*49.71*	***92.02***	*54.74*	***124.58***	*74.49*	***364.03***
				(p=3)		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
			20	1.77	1.77	5.63	5.68	12.03	11.89	17.26	18.84	16.95	16.97	17.80	17.79	13.63	13.94	11.54	11.43										
	13	57842	100	1.77	1.77	5.63	5.68	12.03	11.89	22.22	21.72	48.61	**54.98**	**69.94**	60.13	63.70	67.18	54.11	55.04										
IF4-12-50			500	1.77	1.77	5.63	5.68	12.03	11.89	22.22	21.72	52.44	**64.34**	102.19	109.76	104.22	**205.84**	**193.45**	78.06										
n=2569			∞	*1.77*	*1.77*	*5.63*	*5.68*	*12.03*	*11.89*	*22.22*	*21.72*	*52.44*	***64.34***	*102.19*	*109.76*	*107.91*	***245.09***	***373.17***	*80.67*										
k=4				*(p=3)*		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
w=28			20	1.95	1.95	5.43	**6.46**	11.43	**13.90**	**16.44**	14.28	19.24	17.75	18.47	18.81	13.25	13.20	10.36	10.29										
h=80	14	33676	100	1.95	1.95	5.43	**6.46**	12.38	**15.05**	21.15	20.48	**52.87**	23.63	63.54	60.57	58.57	62.95	47.30	48.74										
			500	1.95	1.95	5.43	**6.46**	12.38	**15.05**	21.15	20.48	**59.71**	23.63	**99.93**	68.59	165.08	**240.54**	165.89	**190.26**										
			∞	*1.95*	*1.95*	*5.43*	***6.46***	*12.38*	***15.05***	*21.15*	*20.48*	***59.71***	*23.63*	***102.67***	*68.59*	*223.02*	***333.43***	*249.45*	***340.16***										
				(p=2)		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=1024)*		*(p=1792)*		*(p=1792)*		*(p=3072)*	
			20	1.96	1.96	3.64	3.67	**6.44**	3.78	**9.44**	6.72	**12.71**	6.64	**13.24**	6.61	**13.75**	6.64	**14.52**	6.66	**14.40**	10.73	**14.24**	10.73	13.53	13.81	13.83	13.81	12.93	13.25
	13	104837	100	1.96	1.96	3.64	3.67	**6.44**	3.78	**9.44**	6.72	**28.09**	7.67	**35.25**	7.53	**46.02**	7.55	**57.54**	7.82	**63.50**	17.60	**62.18**	17.60	53.68	**59.43**	58.93	59.43	52.31	**60.92**
IF4-12-55			500	1.96	1.96	3.64	3.67	**6.44**	3.78	**9.44**	6.72	**28.09**	7.67	**35.25**	7.69	**62.07**	7.69	**101.88**	8.07	**156.01**	18.92	**157.41**	18.92	128.95	**147.24**	137.22	147.24	118.19	121.48
n=2926			∞	*1.96*	*1.96*	*3.64*	*3.67*	***6.44***	*3.78*	***9.44***	*6.72*	***28.09***	*7.67*	***35.25***	*7.69*	***62.07***	*7.69*	***101.88***	*8.07*	***156.01***	*18.92*	***180.44***	*18.92*	***173.00***	*151.72*	*154.40*	*151.72*	*143.61*	*137.94*
k=4				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=768)*		*(p=1536)*		*(p=1536)*		*(p=3072)*	
w=28			20	1.95	1.95	3.66	3.87	**7.21**	3.76	**12.19**	3.81	**12.32**	7.20	12.08	13.12	13.16	**14.49**	12.84	14.05	12.91	13.42	12.62	13.42	10.86	11.28	11.23	11.28	8.68	8.66
h=78	14	25905	100	1.95	1.95	3.66	3.87	**7.21**	3.76	**12.19**	3.81	**21.93**	7.20	**22.35**	13.12	37.06	34.13	37.49	**48.33**	45.13	**54.65**	44.43	**54.65**	43.98	**52.02**	45.05	**52.02**	36.13	**39.98**
			500	1.95	1.95	3.66	3.87	**7.21**	3.76	**12.19**	3.81	**21.93**	7.20	**22.35**	13.12	**45.77**	34.13	59.14	**73.59**	92.19	**119.93**	72.56	**119.93**	**112.63**	75.97	**110.23**	75.97	69.08	**111.18**
			∞	*1.95*	*1.95*	*3.66*	*3.87*	***7.21***	*3.76*	***12.19***	*3.81*	***21.93***	*7.20*	***22.35***	*13.12*	***45.77***	*34.13*	*59.14*	***73.59***	*101.59*	***119.93***	*72.56*	***119.93***	***157.96***	*75.97*	***149.74***	*75.97*	*78.74*	***135.63***
				(p=2)		*(p=4)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=40)*		*(p=56)*		*(p=128)*		*(p=152)*		*(p=176)*		*(p=352)*		*(p=400)*	
			20	2.05	2.05	3.77	3.81	3.76	3.81	6.78	6.90	8.02	8.24	8.12	**9.96**	8.67	9.10	10.21	10.08	10.07	**11.37**	9.59	**11.17**	10.21	10.67	8.53	8.66	8.08	8.25
	15	10607	100	2.05	2.05	3.77	3.81	3.76	3.81	6.78	6.90	8.02	8.24	9.49	9.96	13.85	13.72	**16.92**	13.96	27.06	**31.57**	25.08	**29.88**	25.68	**31.95**	28.90	27.27	**34.22**	21.78
IF4-17-51			500	2.05	2.05	3.77	3.81	3.76	3.81	6.78	6.90	8.02	8.24	9.49	9.96	13.85	13.72	**16.92**	13.96	31.85	31.57	30.83	31.47	33.46	31.95	48.00	47.35	**77.99**	27.13
n=3837			∞	*2.05*	*2.05*	*3.77*	*3.81*	*3.76*	*3.81*	*6.78*	*6.90*	*8.02*	*8.24*	*9.49*	*9.96*	*13.85*	*13.72*	***16.92***	*13.96*	*31.85*	*31.57*	*30.83*	*31.47*	*33.46*	*31.95*	*48.00*	*47.35*	***77.99***	*27.13*
k=4				*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=80)*		*(p=112)*		*(p=256)*		*(p=304)*		*(p=352)*		*(p=704)*		*(p=800)*	
w=29			20	1.97	1.97	**4.13**	2.31	**4.26**	2.27	**7.59**	2.57	**9.08**	2.29	**9.98**	4.39	**13.76**	4.39	13.07	**14.78**	15.65	16.80	15.90	16.08	16.85	15.69	16.68	15.88	16.33	16.18
h=85	16	66103	100	1.97	1.97	**4.13**	2.31	**4.26**	2.27	**7.59**	2.57	**9.08**	2.29	**10.00**	4.39	**17.77**	4.39	20.62	19.45	38.14	35.54	37.01	36.56	35.77	33.18	57.18	55.36	58.76	62.07
			500	1.97	1.97	**4.13**	2.31	**4.26**	2.27	**7.59**	2.57	**9.08**	2.29	**10.00**	4.39	**17.77**	4.39	20.62	19.45	39.63	40.80	**41.73**	36.62	40.28	40.48	77.77	72.01	**109.08**	69.51
			∞	*1.97*	*1.97*	***4.13***	*2.31*	***4.26***	*2.27*	***7.59***	*2.57*	***9.08***	*2.29*	***10.00***	*4.39*	***17.77***	*4.39*	*20.62*	*19.45*	*39.63*	*40.80*	***41.73***	*36.62*	*40.28*	*40.48*	*77.77*	*72.01*	***109.08***	*69.51*
Better by 10%				0x	0x	4x	4x	16x	8x	17x	4x	15x	12x	14x	13x	10x	20x	9x	19x	5x	17x	7x	17x	5x	14x	3x	9x	7x	10x
Better by 50%				0x	0x	4x	0x	12x	0x	12x	0x	15x	5x	11x	4x	8x	8x	6x	12x	3x	8x	3x	10x	1x	8x	1x	5x	5x	6x

Table B.14: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var
pdb1a6m				*(p=9)*		*(p=81)*		*(p=511)*							
n=124	3	198326	20	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
k=81			100	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
w=15			500	1.82	1.82	2.05	**6.67**	3.85	**22.44**						
h=34			∞	*1.82*	*1.82*	*2.05*	***6.67***	*3.85*	***22.44***						
pdb1duw				*(p=9)*		*(p=54)*		*(p=784)*		*(p=15081)*					
n=241	3	627106	20	2.39	2.39	3.37	**4.34**	4.22	**18.29**	9.46	**15.34**				
k=81			100	2.39	2.39	3.37	**4.34**	4.22	**47.17**	12.07	**76.57**				
w=9			500	2.39	2.39	3.37	**4.34**	4.22	**47.17**	12.07	**156.85**				
h=32			∞	*2.39*	*2.39*	*3.37*	***4.34***	*4.22*	***47.17***	*12.07*	***156.85***				
pdb1e5k				*(p=66)*		*(p=1046)*		*(p=11321)*							
n=154	3	112654	20	5.54	5.54	**16.38**	14.76	10.57	10.52						
k=81			100	5.54	5.54	18.79	**55.66**	49.00	52.32						
w=12			500	5.54	5.54	18.79	**55.66**	55.39	**143.87**						
h=43			∞	*5.54*	*5.54*	*18.79*	***55.66***	*55.39*	***143.87***						
pdb1f9i				*(p=81)*		*(p=6534)*									
n=103	3	68804	20	2.46	2.46	2.93	**3.24**								
k=81			100	2.46	2.46	7.86	**16.19**								
w=10			500	2.46	2.46	7.86	**26.60**								
h=24			∞	*2.46*	*2.46*	*7.86*	***26.60***								
pdb1ft5				*(p=27)*		*(p=118)*		*(p=5281)*							
n=172	3	81118	20	2.04	2.04	2.71	**9.83**	9.77	9.58						
k=81			100	2.04	2.04	2.71	**9.83**	18.11	**47.30**						
w=14			500	2.04	2.04	2.71	**9.83**	18.11	**101.14**						
h=33			∞	*2.04*	*2.04*	*2.71*	***9.83***	*18.11*	***101.14***						
pdb1hd2				*(p=79)*		*(p=3777)*									
n=126	3	101550	20	1.72	1.72	6.58	**15.70**								
k=81			100	1.72	1.72	6.58	**44.64**								
w=12			500	1.72	1.72	6.58	**44.64**								
h=27			∞	*1.72*	*1.72*	*6.58*	***44.64***								
pdb1huw				*(p=9)*		*(p=42)*		*(p=293)*		*(p=654)*		*(p=1588)*		*(p=2597)*	
n=152	3	545249	20	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**15.02**	1.22	**17.42**	1.49	**17.23**
k=81			100	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**16.01**	1.22	**29.50**	1.49	**42.76**
w=15			500	1.14	1.14	1.14	**1.35**	1.17	**13.09**	1.18	**16.01**	1.22	**29.50**	1.49	**42.76**
h=43			∞	*1.14*	*1.14*	*1.14*	***1.35***	*1.17*	***13.09***	*1.18*	***16.01***	*1.22*	***29.50***	*1.49*	***42.76***
pdb1kao				*(p=27)*		*(p=215)*		*(p=752)*		*(p=3241)*					
n=148	3	716795	20	2.83	2.83	3.36	**11.07**	4.92	**22.28**	11.23	**39.45**				
k=81			100	2.83	2.83	3.36	**12.86**	4.92	**27.65**	11.23	**117.01**				
w=15			500	2.83	2.83	3.36	**12.86**	4.92	**27.65**	11.23	**117.01**				
h=41			∞	*2.83*	*2.83*	*3.36*	***12.86***	*4.92*	***27.65***	*11.23*	***117.01***				
pdb1nfp				*(p=6)*		*(p=48)*		*(p=336)*		*(p=3812)*					
n=204	3	354720	20	1.08	1.08	1.21	**4.84**	1.83	**9.20**	3.51	**8.34**				
k=81			100	1.08	1.08	1.21	**4.84**	1.83	**13.05**	3.51	**40.53**				
w=18			500	1.08	1.08	1.21	**4.84**	1.83	**13.05**	3.51	**52.41**				
h=38			∞	*1.08*	*1.08*	*1.21*	***4.84***	*1.83*	***13.05***	*3.51*	***52.41***				
pdb1rss				*(p=8)*		*(p=109)*		*(p=908)*		*(p=1336)*					
n=115	3	378579	20	0.97	0.97	3.41	**6.62**	10.02	**11.94**	11.19	**15.49**				
k=81			100	0.97	0.97	3.41	**6.62**	10.05	**14.73**	11.23	**15.60**				
w=12			500	0.97	0.97	3.41	**6.62**	10.06	**14.73**	11.24	**15.60**				
h=35			∞	*0.97*	*0.97*	*3.41*	***6.62***	*10.06*	***14.73***	*11.24*	***15.60***				
pdb1vhh				*(p=27)*		*(p=1842)*		*(p=67760)*							
n=133	3	944633	20	4.04	4.04	**17.97**	4.08	10.20	**13.57**						
k=81			100	4.04	4.04	**41.13**	4.08	45.06	**67.62**						
w=14			500	4.04	4.04	**43.43**	4.08	59.99	**240.92**						
h=35			∞	*4.04*	*4.04*	***43.43***	*4.08*	*63.49*	***301.51***						
Better by 10%				0x	0x	5x	39x	0x	33x	0x	20x	0x	4x	0x	4x
Better by 50%				0x	0x	4x	30x	0x	28x	0x	16x	0x	4x	0x	4x

Table B.15: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **side-chain prediction instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d																					
				1		2		3		4		5		6		7		8		9		10		11		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	(p=2) 1.62	1.62	(p=4) 2.92	2.94	(p=8) 3.72	3.83	(p=16) 6.88	6.74	(p=16) 6.69	6.74	(p=32) 9.58	9.78	(p=64) 13.15	13.49	(p=128) **13.95**	11.76	(p=192) **14.37**	12.07	(p=192) **14.97**	12.07	(p=192) **14.05**	12.07
			100	1.62	1.62	2.92	2.94	3.72	3.83	6.88	6.74	6.69	6.74	9.58	9.78	15.76	**19.78**	20.33	19.91	20.34	20.09	19.59	20.09	20.14	20.09
			500	1.62	1.62	2.92	2.94	3.72	3.83	6.88	6.74	6.69	6.74	9.58	9.78	15.76	**19.78**	20.33	20.27	20.34	21.42	19.59	21.42	20.14	21.42
			∞	*1.62*	*1.62*	*2.92*	*2.94*	*3.72*	*3.83*	*6.88*	*6.74*	*6.69*	*6.74*	*9.58*	*9.78*	*15.76*	***19.78***	*20.33*	*20.27*	*20.34*	*21.42*	*19.59*	*21.42*	*20.14*	*21.42*
	14	15402	20	(p=2) 1.53	1.53	(p=4) 2.65	2.60	(p=8) 4.37	4.12	(p=8) 4.34	4.12	(p=8) 4.24	4.12	(p=16) 6.48	6.32	(p=32) 9.85	10.75	(p=64) **11.52**	9.53	(p=96) **10.41**	9.22	(p=96) **12.94**	9.22	(p=192) **11.88**	8.92
			100	1.53	1.53	2.65	2.60	4.37	4.12	4.34	4.12	4.24	4.12	6.48	6.32	10.90	**12.55**	13.01	12.59	12.72	12.29	12.98	12.29	**15.60**	12.19
			500	1.53	1.53	2.65	2.60	4.37	4.12	4.34	4.12	4.24	4.12	6.48	6.32	10.90	**12.55**	13.01	12.59	12.72	12.29	12.98	12.29	**15.60**	12.55
			∞	*1.53*	*1.53*	*2.65*	*2.60*	*4.37*	*4.12*	*4.34*	*4.12*	*4.24*	*4.12*	*6.48*	*6.32*	*10.90*	***12.55***	*13.01*	*12.59*	*12.72*	*12.29*	*12.98*	*12.29*	***15.60***	*12.55*
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	(p=2) 1.75	1.75	(p=4) 2.00	2.01	(p=4) 1.96	2.01	(p=6) 2.55	2.57	(p=6) 2.57	2.57	(p=12) 2.45	2.45	(p=24) 3.27	3.29	(p=48) 4.64	4.81	(p=48) 4.77	4.81	(p=72) **7.03**	5.70	(p=144) **7.93**	7.13
			100	1.75	1.75	2.00	2.01	1.96	2.01	2.55	2.57	2.57	2.57	2.45	2.45	3.27	3.29	4.64	4.81	4.77	4.81	7.03	7.16	10.76	9.93
			500	1.75	1.75	2.00	2.01	1.96	2.01	2.55	2.57	2.57	2.57	2.45	2.45	3.27	3.29	4.64	4.81	4.77	4.81	7.03	7.16	10.76	9.93
			∞	*1.75*	*1.75*	*2.00*	*2.01*	*1.96*	*2.01*	*2.55*	*2.57*	*2.57*	*2.57*	*2.45*	*2.45*	*3.27*	*3.29*	*4.64*	*4.81*	*4.77*	*4.81*	*7.03*	*7.16*	*10.76*	*9.93*
	15	33656	20	(p=2) 1.33	1.33	(p=4) 2.06	2.08	(p=4) 2.05	2.08	(p=6) 2.54	2.56	(p=6) 2.58	2.56	(p=12) 4.03	3.96	(p=24) 6.18	6.13	(p=48) 6.85	**7.64**	(p=48) 7.00	7.64	(p=72) **7.87**	7.46	(p=144) 7.53	7.22
			100	1.33	1.33	2.06	2.08	2.05	2.08	2.54	2.56	2.58	2.56	4.03	3.96	6.18	6.13	9.45	**10.85**	9.60	**10.85**	**14.55**	12.39	**23.24**	19.57
			500	1.33	1.33	2.06	2.08	2.05	2.08	2.54	2.56	2.58	2.56	4.03	3.96	6.18	6.13	9.45	**10.85**	9.60	**10.85**	**14.55**	12.39	**23.24**	19.57
			∞	*1.33*	*1.33*	*2.06*	*2.08*	*2.05*	*2.08*	*2.54*	*2.56*	*2.58*	*2.56*	*4.03*	*3.96*	*6.18*	*6.13*	*9.45*	***10.85***	*9.60*	***10.85***	***14.55***	*12.39*	***23.24***	*19.57*
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	(p=2) 1.19	1.19	(p=3) 1.80	1.81	(p=6) 2.67	2.65	(p=12) 2.64	2.65	(p=24) 4.45	**6.25**	(p=36) 6.34	6.62	(p=72) 9.71	10.29	(p=144) 10.71	11.09	(p=216) 11.35	10.61	(p=288) 9.94	10.34	(p=504) 10.39	10.15
			100	1.19	1.19	1.80	1.81	2.67	2.65	2.64	2.65	4.45	**6.25**	6.34	6.62	10.14	10.29	18.43	**26.03**	27.26	25.92	28.44	30.65	38.90	**42.83**
			500	1.19	1.19	1.80	1.81	2.67	2.65	2.64	2.65	4.45	**6.25**	6.34	6.62	10.14	10.29	18.43	**26.03**	27.26	25.92	28.44	**46.65**	45.28	46.82
			∞	*1.19*	*1.19*	*1.80*	*1.81*	*2.67*	*2.65*	*2.64*	*2.65*	*4.45*	***6.25***	*6.34*	*6.62*	*10.14*	*10.29*	*18.43*	***26.03***	*27.26*	*25.92*	*28.44*	***46.65***	*45.28*	*46.82*
	18	21694	20	(p=2) 1.01	1.01	(p=3) 1.30	1.33	(p=6) 2.25	2.06	(p=12) **2.25**	2.03	(p=24) **3.35**	2.06	(p=36) **4.31**	1.98	(p=72) **7.51**	1.98	(p=144) **8.07**	1.97	(p=216) **8.29**	3.35	(p=288) **8.16**	3.12	(p=504) **7.98**	3.33
			100	1.01	1.01	1.30	1.33	2.25	2.06	**2.25**	2.03	**3.35**	2.06	**4.31**	2.04	**7.51**	2.05	**11.82**	2.04	**17.55**	3.40	**17.77**	3.49	**28.85**	3.52
			500	1.01	1.01	1.30	1.33	2.25	2.06	**2.25**	2.03	**3.35**	2.06	**4.31**	2.04	**7.51**	2.05	**11.82**	2.05	**17.55**	3.40	**17.77**	3.49	**28.85**	3.52
			∞	*1.01*	*1.01*	*1.30*	*1.33*	*2.25*	*2.06*	***2.25***	*2.03*	***3.35***	*2.06*	***4.31***	*2.04*	***7.51***	*2.05*	***11.82***	*2.05*	***17.55***	*3.40*	***17.77***	*3.49*	***28.85***	*3.52*
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	(p=2) 1.83	1.83	(p=4) 2.38	2.21	(p=8) 3.83	3.54	(p=8) 3.48	3.54	(p=16) 5.38	5.35	(p=16) 5.28	5.35	(p=32) 8.14	8.19	(p=64) **8.86**	7.72	(p=128) 8.96	8.21	(p=192) 8.49	8.31	(p=384) **9.29**	8.00
			100	1.83	1.83	2.38	2.21	3.83	3.54	3.48	3.54	5.38	5.35	5.28	5.35	8.57	8.19	13.31	**14.74**	14.58	15.74	15.86	16.76	**19.06**	16.86
			500	1.83	1.83	2.38	2.21	3.83	3.54	3.48	3.54	5.38	5.35	5.28	5.35	8.57	8.19	13.31	**14.74**	14.58	15.74	15.86	**18.34**	**19.06**	17.01
			∞	*1.83*	*1.83*	*2.38*	*2.21*	*3.83*	*3.54*	*3.48*	*3.54*	*5.38*	*5.35*	*5.28*	*5.35*	*8.57*	*8.19*	*13.31*	***14.74***	*14.58*	*15.74*	*15.86*	***18.34***	***19.06***	*17.01*
	18	26855	20	(p=2) 1.50	1.50	(p=4) 2.20	2.38	(p=8) 3.62	3.58	(p=8) 3.61	3.58	(p=16) **5.74**	5.07	(p=24) **6.90**	5.61	(p=48) 11.27	11.51	(p=80) 10.61	10.47	(p=160) **11.46**	10.22	(p=240) 11.19	**12.44**	(p=480) 10.99	11.42
			100	1.50	1.50	2.20	2.38	3.62	3.58	3.61	3.58	**5.74**	5.07	**6.90**	5.61	13.08	12.50	21.11	22.62	25.85	26.30	29.51	29.84	32.05	32.47
			500	1.50	1.50	2.20	2.38	3.62	3.58	3.61	3.58	**5.74**	5.07	**6.90**	5.61	13.08	12.50	21.11	22.62	25.85	**29.81**	30.04	29.84	38.09	38.25
			∞	*1.50*	*1.50*	*2.20*	*2.38*	*3.62*	*3.58*	*3.61*	*3.58*	***5.74***	*5.07*	***6.90***	*5.61*	*13.08*	*12.50*	*21.11*	*22.62*	*25.85*	***29.81***	*30.04*	*29.84*	*38.09*	*38.25*
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	(p=2) 1.58	1.58	(p=4) **2.25**	2.01	(p=8) **2.83**	2.29	(p=12) **3.74**	2.34	(p=24) **5.73**	3.80	(p=48) **7.27**	6.16	(p=96) **8.17**	6.64	(p=144) 7.93	8.36	(p=288) **8.78**	7.87	(p=384) 7.91	7.65	(p=640) 8.17	7.48
			100	1.58	1.58	**2.25**	2.01	**2.83**	2.29	**3.74**	2.34	**5.73**	3.80	**9.88**	6.16	**17.59**	10.91	**17.55**	11.22	23.45	21.92	**29.84**	24.80	**36.79**	26.03
			500	1.58	1.58	**2.25**	2.01	**2.83**	2.29	**3.74**	2.34	**5.73**	3.80	**9.88**	6.16	**17.59**	10.91	**17.55**	11.22	25.35	25.35	**44.65**	25.20	**64.47**	40.05
			∞	*1.58*	*1.58*	***2.25***	*2.01*	***2.83***	*2.29*	***3.74***	*2.34*	***5.73***	*3.80*	***9.88***	*6.16*	***17.59***	*10.91*	***17.55***	*11.22*	*25.35*	*25.35*	***44.65***	*25.20*	***64.47***	*40.05*
	20	8053	20	(p=2) 1.61	1.61	(p=4) 1.38	1.39	(p=8) 1.79	1.81	(p=12) 3.50	3.54	(p=24) **5.30**	3.82	(p=48) 5.58	5.99	(p=96) **6.81**	5.99	(p=144) 6.20	6.43	(p=288) 6.40	6.02	(p=384) 5.90	5.85	(p=640) 5.36	5.29
			100	1.61	1.61	1.38	1.39	1.79	1.81	3.50	3.54	**5.30**	3.82	**7.86**	5.99	**14.18**	8.24	**14.67**	8.25	**21.08**	14.51	**22.94**	16.64	**23.76**	19.59
			500	1.61	1.61	1.38	1.39	1.79	1.81	3.50	3.54	**5.30**	3.82	**7.86**	5.99	**14.18**	8.24	**14.67**	8.25	**24.70**	15.58	**35.63**	16.64	**50.65**	26.49
			∞	*1.61*	*1.61*	*1.38*	*1.39*	*1.79*	*1.81*	*3.50*	*3.54*	***5.30***	*3.82*	***7.86***	*5.99*	***14.18***	*8.24*	***14.67***	*8.25*	***24.70***	*15.58*	***35.63***	*16.64*	***50.65***	*26.49*
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	(p=2) 0.71	0.71	(p=4) 1.21	1.21	(p=8) 2.35	2.14	(p=16) 2.66	2.67	(p=32) 3.62	**4.15**	(p=64) 3.78	**4.59**	(p=128) 4.56	4.87	(p=128) 4.71	4.87	(p=128) 4.74	4.87	(p=256) 5.07	5.41	(p=384) 5.69	5.23
			100	0.71	0.71	1.21	1.21	2.35	2.14	2.66	2.67	4.18	4.15	4.86	4.79	7.99	7.92	8.07	7.92	8.03	7.92	12.37	13.14	15.09	16.02
			500	0.71	0.71	1.21	1.21	2.35	2.14	2.66	2.67	4.18	4.15	4.86	4.79	7.99	7.92	8.07	7.92	8.03	7.92	12.94	13.14	16.70	16.65
			∞	*0.71*	*0.71*	*1.21*	*1.21*	*2.35*	*2.14*	*2.66*	*2.67*	*4.18*	*4.15*	*4.86*	*4.79*	*7.99*	*7.92*	*8.07*	*7.92*	*8.03*	*7.92*	*12.94*	*13.14*	*16.70*	*16.65*
	12	64758	20	(p=2) 0.74	0.74	(p=4) 0.98	0.98	(p=8) 1.78	1.80	(p=16) 2.13	2.13	(p=32) **3.36**	2.07	(p=64) **3.31**	2.20	(p=128) 3.86	**4.26**	(p=128) 3.67	**4.26**	(p=128) 3.94	4.26	(p=256) **4.12**	3.69	(p=384) **4.76**	3.97
			100	0.74	0.74	0.98	0.98	1.78	1.80	2.13	2.13	**3.36**	2.21	**3.78**	2.20	6.09	6.15	5.88	6.15	6.14	6.15	**8.19**	6.11	**9.62**	6.18
			500	0.74	0.74	0.98	0.98	1.78	1.80	2.13	2.13	**3.36**	2.21	**3.78**	2.20	6.09	6.15	5.88	6.15	6.14	6.15	**8.40**	6.11	**10.28**	6.18
			∞	*0.74*	*0.74*	*0.98*	*0.98*	*1.78*	*1.80*	*2.13*	*2.13*	***3.36***	*2.21*	***3.78***	*2.20*	*6.09*	*6.15*	*5.88*	*6.15*	*6.14*	*6.15*	***8.40***	*6.11*	***10.28***	*6.18*
75-26-9 n=675 k=2 w=39 h=124	16	59609	20	(p=2) 1.38	1.38	(p=4) 2.35	2.33	(p=8) 2.44	2.50	(p=16) 2.49	**4.27**	(p=24) 4.37	4.30	(p=48) **6.67**	4.37	(p=96) 6.57	**7.33**	(p=120) 7.38	7.55	(p=240) 8.47	8.50	(p=480) 7.66	7.71	(p=960) 7.47	7.01
			100	1.38	1.38	2.35	2.33	2.44	2.50	2.49	**4.27**	4.37	4.30	**6.67**	4.37	7.61	7.66	**10.01**	8.97	**14.21**	10.27	14.87	15.54	16.95	**21.63**
			500	1.38	1.38	2.35	2.33	2.44	2.50	2.49	**4.27**	4.37	4.30	**6.67**	4.37	7.61	7.66	**10.01**	8.97	**14.21**	10.27	15.72	15.54	18.45	**25.57**
			∞	*1.38*	*1.38*	*2.35*	*2.33*	*2.44*	*2.50*	*2.49*	***4.27***	*4.37*	*4.30*	***6.67***	*4.37*	*7.61*	*7.66*	***10.01***	*8.97*	***14.21***	*10.27*	*15.72*	*15.54*	*18.45*	***25.57***
	18	66533	20	(p=2) 1.30	1.30	(p=4) 2.11	2.15	(p=8) 2.15	2.19	(p=16) 2.09	**3.46**	(p=24) 3.43	**5.64**	(p=48) 5.62	5.57	(p=96) 5.72	5.67	(p=120) 7.01	7.21	(p=240) 7.62	8.13	(p=480) 7.35	7.71	(p=960) 7.57	7.56
			100	1.30	1.30	2.11	2.15	2.15	2.19	2.09	**3.46**	3.43	**5.64**	5.62	5.57	6.13	5.67	**9.09**	8.23	**13.54**	8.13	13.45	**30.04**	16.32	**28.01**
			500	1.30	1.30	2.11	2.15	2.15	2.19	2.09	**3.46**	3.43	**5.64**	5.62	5.57	6.13	5.67	**9.09**	8.23	**13.54**	8.13	14.38	**46.11**	17.57	**46.72**
			∞	*1.30*	*1.30*	*2.11*	*2.15*	*2.15*	*2.19*	*2.09*	***3.46***	*3.43*	***5.64***	*5.62*	*5.57*	*6.13*	*5.67*	***9.09***	*8.23*	***13.54***	*8.13*	*14.38*	***46.11***	*17.57*	***46.72***
	20	5708	20	(p=2) 1.64	1.64	(p=4) 2.64	2.64	(p=8) 2.82	2.82	(p=16) 1.99	**2.44**	(p=24) 2.41	**3.57**	(p=48) 3.54	3.55	(p=96) 3.88	**4.76**	(p=120) 4.84	**7.04**	(p=240) 6.45	6.28	(p=320) 5.56	5.96	(p=640) 4.90	4.97
			100	1.64	1.64	2.64	2.64	2.82	2.82	1.99	**2.44**	2.41	**3.57**	3.54	3.55	4.34	4.76	5.83	**10.91**	9.71	**17.19**	15.26	**17.24**	16.03	**20.53**
			500	1.64	1.64	2.64	2.64	2.82	2.82	1.99	**2.44**	2.41	**3.57**	3.54	3.55	4.34	4.76	5.83	**10.91**	9.71	**17.19**	16.26	17.24	25.37	**34.39**
			∞	*1.64*	*1.64*	*2.64*	*2.64*	*2.82*	*2.82*	*1.99*	***2.44***	*2.41*	***3.57***	*3.54*	*3.55*	*4.34*	*4.76*	*5.83*	***10.91***	*9.71*	***17.19***	*16.26*	*17.24*	*25.37*	***34.39***
Better by 10%				0x	0x	4x	0x	4x	0x	8x	12x	20x	13x	23x	1x	12x	9x	19x	15x	17x	8x	20x	9x	27x	10x
Better by 50%				0x	0x	0x	0x	0x	0x	4x	8x	12x	4x	15x	0x	10x	0x	10x	3x	9x	3x	8x	5x	11x	3x

instance	i	T_{seq}	#cpu	Cutoff depth d																					
				13		14		15																	
				fix	var	fix	var	fix	var																
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	(p=768) 15.75	14.67	(p=1152) 15.08	16.19	(p=2112) 15.70	15.68																
			100	**24.13**	19.59	**39.60**	27.74	**45.45**	40.87	47.44	**58.03**														
			500	**24.77**	20.17	**43.52**	27.74	**48.20**	42.99	47.44	**58.03**														
			∞	***24.77***	*20.17*	***43.52***	*27.74*	***48.20***	*42.99*	*47.44*	***58.03***														
	14	15402	20	(p=576) **16.01**	13.28	12.55	**16.16**	15.19	15.42																
			100	**18.18**	12.44	**23.66**	17.89	23.51	**28.74**	23.62	**47.39**														
			500	**18.18**	12.44	**23.66**	18.36	23.51	**28.74**	23.73	**68.15**														
			∞	***18.18***	*12.44*	***23.66***	*18.36*	*23.51*	***28.74***	*23.73*	***70.33***														
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	(p=576) **6.61**	5.88	5.44	5.59	5.27	5.42																
			100	**27.09**	13.08	17.41	17.18	15.00	**17.44**																
			500	**27.09**	13.08	22.11	21.77	21.72	21.71																
			∞	***27.09***	*13.08*	*22.11*	*22.38*	*22.62*	*22.48*																
	15	33656	20	(p=576) 7.19	6.76	6.86	6.79	6.40	6.58																
			100	30.16	27.59	31.25	31.37	21.31	**29.68**																
			500	**33.62**	25.27	55.91	59.36	69.97	66.78	39.60	**56.00**														
			∞	***33.62***	*25.27*	*55.91*	*59.36*	*69.97*	*70.85*	*46.87*	***56.00***														
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	(p=2016) 9.90	9.76	10.00	9.57	9.72	9.49																
			100	40.99	44.39	41.68	43.27	44.21	44.24																
			500	96.65	**142.97**	104.05	**137.48**	144.22	**164.75**																
			∞	*96.65*	***202.99***	*113.81*	***248.64***	*148.54*	***239.43***																
	18	21694	20	(p=2014) 6.77	6.69	6.59	6.38	6.28	6.21																
			100	20.68	19.20	**27.60**	19.16	23.89	25.89																
			500	**30.43**	26.98	**36.10**	30.60	30.90	**70.21**																
			∞	***31.35***	*26.98*	*36.10*	*34.82*	*30.90*	***98.61***																
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	(p=768) 8.87	8.95	9.25	9.08	9.28	9.08																
			100	28.93	29.33	33.25	35.49	34.75	35.49																
			500	**35.76**	30.63	45.75	**51.46**	45.66	**51.46**																
			∞	***35.76***	*30.63*	*45.75*	***51.46***	*45.66*	***51.46***																
	18	26855	20	(p=960) 9.63	10.52	**10.70**	9.21	**10.65**	9.21																
			100	37.82	35.90	36.05	34.43	**55.14**	34.43																
			500	38.25	38.25	37.82	35.90	36.05	34.43	**55.14**	34.43														
			∞	*38.25*	*38.25*	*38.58*	*35.90*	*37.61*	*36.74*	***55.14***	*36.74*														
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	(p=1280) 7.82	7.82	7.49	7.47	7.10	7.05																
			100	27.83	27.00	32.61	33.56	33.12	32.36																
			500	51.58	49.46	48.05	**77.53**	**117.01**	95.02																
			∞	*58.24*	*56.80*	*51.79*	***77.53***	***163.06***	*127.01*																
	20	8053	20	(p=1280) 4.47	4.47	3.37	3.38	2.71	2.71																
			100	20.86	19.59	15.64	15.76	12.70	12.60																
			500	**65.47**	51.29	**58.78**	51.29	46.02	45.24																
			∞	***78.95***	*57.11*	***105.96***	*84.77*	***91.51***	*75.97*																
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	(p=1152) 5.68	5.51	5.56	5.24	5.55	5.52																
			100	17.93	19.37	**23.52**	20.11	**26.42**	23.12																
			500	26.95	27.39	33.86	33.41	56.17	53.08																
			∞	*27.33*	*27.39*	*34.72*	*35.58*	***61.56***	*55.21*																
	12	64758	20	(p=1152) 4.74	4.39	4.57	4.23	4.52	4.49																
			100	**11.85**	9.83	**11.82**	9.52	**19.46**	13.60																
			500	**15.81**	10.71	**13.23**	10.62	**33.64**	15.85																
			∞	***15.81***	*10.71*	***13.26***	*10.67*	***33.64***	*15.85*																
75-26-9 n=675 k=2 w=39 h=124	16	59609	20	(p=3840) 6.62	6.60	6.68	6.60	6.32	6.19																
			100	28.21	26.13	27.49	26.13	29.92	28.47																
			500	31.00	**57.10**	42.92	**57.10**	61.33	67.28																
			∞	*31.00*	***59.61***	*44.68*	***59.61***	*68.67*	*67.81*																
	18	66533	20	(p=3840) 7.07	6.97	7.01	6.97	6.54	6.47																
			100	28.07	**33.15**	27.98	**33.15**	30.44	31.09																
			500	40.74	**78.00**	45.63	**78.00**	65.61	**111.63**																
			∞	*42.84*	***85.30***	*46.08*	***85.30***	*71.16*	***218.86***																
	20	5708	20	(p=2560) 2.66	2.64	2.67	2.64	1.67	1.66																
			100	12.30	12.41	12.49	12.41	7.89	7.81																
			500	35.90	**44.94**	37.06	**44.94**	31.02	29.88																
			∞	*50.07*	***78.19***	*47.17*	***78.19***	*54.88*	***74.13***																
Better by 10%				20x	9x	13x	17x	12x	19x																
Better by 50%				5x	6x	0x	5x	5x	8x																

Table B.16: Parallel speedup with 20, 100, 500, and "unlimited" CPUs, on **grid instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
ped13 n=1077 k=3 w=32 h=102	8	252654		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
			20	0.10	0.10	**0.19**	0.10	**0.38**	0.10	**0.71**	0.19	**0.73**	0.38	**0.83**	0.55	**0.87**	0.62	0.90	0.99	0.95	0.97	0.98	1.00	0.96	1.00	0.97	1.00	0.98	0.99
			100	0.02	0.02	**0.04**	0.02	**0.08**	0.02	**0.14**	0.04	**0.23**	0.08	**0.45**	0.14	**0.47**	0.14	**0.56**	0.25	**0.66**	0.53	0.77	0.70	0.76	**0.92**	0.78	0.81	0.78	**0.89**
			500	0.00	0.00	**0.01**	0.00	**0.02**	0.00	**0.03**	0.01	**0.05**	0.02	**0.09**	0.03	**0.11**	0.03	**0.16**	0.05	**0.23**	0.11	**0.32**	0.20	0.29	0.30	0.39	0.37	0.39	**0.51**
	9	102385		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=6144)*	
			20	0.09	0.09	0.16	0.16	**0.33**	0.21	**0.49**	0.42	**0.55**	0.42	**0.85**	0.52	**0.96**	0.74	**0.99**	0.89	0.94	0.98	0.97	0.94	0.93	1.00	0.96	1.00	0.97	1.00
			100	0.02	0.02	0.03	0.03	**0.07**	0.04	**0.10**	0.08	**0.15**	0.08	**0.28**	0.10	**0.30**	0.15	**0.52**	0.31	0.55	0.53	**0.76**	0.51	0.63	**0.75**	0.77	0.73	0.77	0.81
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.02**	0.02	**0.03**	0.02	**0.06**	0.02	**0.06**	0.03	**0.12**	0.06	0.13	0.13	**0.25**	0.12	0.25	0.25	0.28	0.28	0.29	0.29
ped19 n=793 k=5 w=25 h=98	16	375110		*(p=4)*		*(p=12)*		*(p=48)*		*(p=144)*		*(p=288)*		*(p=1440)*		*(p=2880)*		*(p=5752)*		*(p=7672)*		*(p=11254)*		*(p=14968)*					
			20	0.11	0.11	0.18	**0.29**	0.33	**0.55**	0.78	**0.89**	0.93	0.95	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00				
			100	0.02	0.02	0.04	**0.06**	0.07	**0.12**	0.16	**0.24**	0.28	**0.41**	0.44	**0.78**	0.72	**0.81**	0.95	0.95	0.99	1.00	1.00	1.00	1.00	**1.00**				
			500	0.00	0.00	0.01	**0.01**	0.01	**0.02**	0.03	**0.05**	0.06	**0.08**	0.09	**0.22**	0.18	**0.37**	0.27	**0.46**	0.48	**0.57**	0.69	**0.87**	0.76	**0.98**				
ped20 n=437 k=5 w=22 h=60	3	5136		*(p=2)*		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
			20	0.10	0.10	0.20	0.20	0.38	0.40	0.71	0.77	0.82	0.84	**0.93**	0.85	0.94	0.98	0.99	0.99	1.00	0.99	1.00	1.00						
			100	0.02	0.02	0.04	0.04	0.08	0.08	0.17	0.17	0.30	0.30	**0.67**	0.31	**0.67**	0.55	0.82	0.76	0.98	0.96	0.99	0.98						
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	0.06	0.06	**0.16**	0.06	**0.25**	0.13	**0.41**	0.15	**0.82**	0.71	0.93	0.85						
	4	2185		*(p=2)*		*(p=6)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=800)*		*(p=3200)*		*(p=6400)*							
			20	0.09	0.09	0.24	0.24	**0.47**	0.29	**0.74**	0.67	0.82	0.88	0.97	0.99	0.96	0.99	0.98	0.99	1.00	1.00	1.00	1.00						
			100	0.02	0.02	0.05	0.05	**0.09**	0.06	**0.24**	0.13	0.45	0.44	**0.75**	0.62	0.81	**0.96**	0.90	0.97	0.98	0.98	0.99	0.99						
			500	0.00	0.00	0.01	0.01	**0.02**	0.01	**0.05**	0.03	0.09	0.09	**0.22**	0.12	0.40	0.40	0.58	**0.65**	0.86	0.92	0.95	0.95						
ped31 n=1183 k=5 w=30 h=85	10	1258519		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	0.10	0.10	0.20	0.20	0.38	0.39	0.72	0.72	0.70	**0.82**	0.74	**0.85**	0.84	**0.93**	0.90	0.96	0.91	0.98	0.97	0.98	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.08	0.08	0.14	0.14	**0.25**	0.18	0.31	0.31	**0.47**	0.42	0.51	**0.59**	0.63	**0.85**	0.81	**0.90**	0.92	0.97	0.97	**0.97**	0.96	0.97
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	**0.05**	0.04	0.06	0.06	**0.11**	0.08	0.13	**0.18**	0.25	**0.32**	0.39	**0.56**	0.56	**0.79**	0.72	**0.84**	0.80	0.87
	11	433029		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	0.10	0.10	0.20	0.20	0.38	0.40	0.74	0.75	0.67	**0.78**	0.72	**0.86**	0.98	0.93	0.99	0.98	1.00	0.99	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.08	0.08	0.15	0.15	**0.24**	0.16	0.29	0.29	0.50	0.50	0.62	0.61	**0.94**	0.81	0.82	**0.96**	0.89	0.95	0.94	0.99	0.94	0.99
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	**0.05**	0.03	0.06	0.06	0.10	0.10	0.12	0.12	0.23	**0.28**	0.40	**0.53**	0.55	**0.73**	0.71	0.77	0.66	0.73
	12	16238		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	0.10	0.10	0.20	0.20	0.36	0.39	0.65	0.68	**0.64**	0.57	0.60	**0.77**	0.65	**0.86**	0.83	**0.95**	0.85	**0.99**	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.07	0.08	0.13	0.14	**0.21**	0.11	0.15	**0.21**	0.23	**0.38**	0.41	0.45	0.53	**0.72**	0.74	**0.91**	0.89	0.97	0.97	1.00	0.99	1.00
			500	0.00	0.00	0.01	0.01	0.01	0.02	0.03	0.03	**0.04**	0.02	0.03	**0.04**	0.05	**0.08**	0.09	0.09	0.17	0.17	0.31	0.34	0.52	**0.75**	0.70	**0.96**	0.81	**0.97**
ped33 n=798 k=4 w=28 h=98	4	6010		*(p=2)*		*(p=3)*		*(p=6)*		*(p=6)*		*(p=12)*		*(p=24)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=1536)*		*(p=1536)*	
			20	0.07	0.07	0.09	0.09	0.17	0.19	0.17	0.19	0.35	**0.41**	0.47	**0.63**	0.59	**0.79**	0.74	0.80	0.80	**0.95**	0.91	0.98	0.89	**0.99**	0.99	0.99	0.96	0.99
			100	0.01	0.01	0.02	0.02	0.03	0.04	0.03	0.04	0.07	**0.08**	0.12	**0.17**	0.17	**0.21**	0.28	**0.43**	0.32	**0.63**	0.49	**0.77**	0.55	**0.84**	0.77	**0.90**	0.71	**0.90**
			500	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	**0.03**	0.03	**0.04**	0.06	**0.09**	0.07	**0.13**	0.14	**0.19**	0.18	**0.29**	0.29	**0.47**	0.30	**0.47**
ped34 n=1160 k=5 w=31 h=102	10	962006		*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=952)*		*(p=1896)*		*(p=3752)*	
			20	0.11	0.11	0.14	0.14	0.22	0.22	0.37	**0.48**	0.49	0.50	0.54	**0.67**	0.82	0.85	0.79	**0.95**	0.91	0.97	0.93	0.98	0.93	0.98	0.98	0.99	0.99	1.00
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.07	**0.10**	0.10	0.10	0.14	**0.22**	0.23	0.23	0.41	0.45	0.55	**0.77**	0.68	**0.90**	0.71	**0.91**	0.89	0.91	0.94	0.99
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	0.02	0.03	**0.04**	0.05	0.05	0.08	0.09	0.15	**0.18**	0.27	**0.34**	0.33	0.36	0.58	0.63	0.74	0.79
	11	350574		*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=720)*		*(p=956)*		*(p=1912)*		*(p=3808)*	
			20	0.07	0.07	0.11	0.11	**0.19**	0.14	**0.38**	0.26	0.47	0.43	0.60	0.61	**0.76**	0.66	0.81	0.88	0.93	0.98	0.98	0.98	0.96	0.99	0.99	1.00	0.99	1.00
			100	0.01	0.01	0.02	0.02	**0.04**	0.03	**0.08**	0.05	0.09	0.10	0.14	0.15	0.25	0.25	0.43	0.44	0.58	0.60	0.76	0.80	0.80	0.81	0.91	0.95	0.94	0.98
			500	0.00	0.00	0.00	0.00	**0.01**	0.01	**0.02**	0.01	0.02	0.02	0.03	0.03	0.05	0.05	0.09	0.09	0.17	0.17	0.28	**0.32**	0.33	**0.36**	0.58	0.54	0.68	**0.80**
	12	96122		*(p=3)*		*(p=5)*		*(p=10)*		*(p=20)*		*(p=30)*		*(p=60)*		*(p=90)*		*(p=180)*		*(p=360)*		*(p=716)*		*(p=948)*		*(p=1896)*		*(p=3728)*	
			20	0.10	0.10	0.18	0.19	0.25	0.27	0.36	**0.52**	0.54	0.57	0.63	0.69	0.69	**0.83**	0.78	0.84	0.94	0.96	0.95	0.98	0.95	0.99	0.99	0.99	0.99	1.00
			100	0.02	0.02	0.04	0.04	0.05	0.05	0.07	**0.10**	0.11	0.11	0.15	**0.24**	0.24	0.24	0.40	0.41	0.56	**0.66**	0.70	**0.87**	0.74	**0.90**	0.90	0.90	0.92	0.95
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	0.02	0.03	**0.05**	0.05	0.05	0.08	0.08	0.14	**0.17**	0.23	**0.40**	0.30	**0.48**	0.52	0.55	**0.63**	0.51
ped39 n=1272 k=5 w=21 h=76	4	6632		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
			20	0.10	0.10	0.12	0.12	0.13	0.14	0.24	0.23	0.52	**0.62**	0.54	0.54	0.51	**0.59**	0.66	**0.85**	0.79	**0.95**	0.98	0.99	1.00	1.00				
			100	0.02	0.02	0.02	0.02	0.03	0.03	0.05	0.05	0.11	0.12	0.13	0.14	0.14	**0.17**	0.18	**0.30**	0.30	**0.46**	0.55	**0.76**	0.79	**0.92**				
			500	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.03	0.03	**0.04**	0.04	**0.06**	0.07	**0.11**	0.16	**0.24**	0.29	**0.38**				
	5	2202		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=384)*		*(p=768)*		*(p=1152)*		*(p=2304)*		*(p=4608)*					
			20	0.10	0.10	0.14	0.14	0.18	0.18	0.34	**0.39**	0.37	**0.42**	0.39	**0.56**	0.49	**0.76**	0.59	**0.85**	0.74	**0.95**	0.99	0.99	0.99	1.00				
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.07	**0.08**	0.09	0.09	0.10	0.11	0.13	**0.25**	0.18	**0.38**	0.31	**0.59**	0.61	**0.83**	0.81	**0.98**				
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	0.02	0.02	0.02	0.03	**0.05**	0.04	**0.10**	0.08	**0.12**	0.20	**0.33**	0.37	**0.69**				
Better by 10%				0x	0x	3x	3x	12x	3x	12x	12x	13x	9x	11x	16x	11x	17x	6x	14x	4x	19x	3x	17x	0x	17x	0x	4x	1x	6x
Better by 50%				0x	0x	3x	3x	9x	3x	5x	2x	9x	0x	9x	6x	5x	6x	5x	7x	1x	6x	3x	4x	0x	4x	0x	1x	0x	1x

Table B.17: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var
ped41 n=1062 k=5 w=33 h=100	9	25607		*(p=3)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8556)*	
			20	0.07	0.07	0.10	0.10	0.19	**0.36**	0.37	**0.61**	0.61	**0.84**	0.85	0.87	0.97	0.94	0.86	**0.99**	0.99	0.99	0.99	1.00	1.00	0.98	1.00	1.00	1.00	NA
			100	0.01	0.01	0.02	0.02	0.04	**0.07**	0.08	**0.14**	0.14	**0.29**	0.28	**0.48**	0.46	0.49	0.51	**0.73**	0.89	0.90	0.90	0.96	**0.90**	0.61	0.76	**0.93**	0.99	NA
			500	0.00	0.00	0.00	0.00	0.01	**0.01**	0.02	**0.03**	0.03	**0.06**	0.06	**0.10**	0.10	**0.13**	0.13	**0.25**	0.26	**0.35**	0.40	**0.49**	**0.32**	0.18	0.28	**0.62**	0.70	NA
	10	46819		*(p=3)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8576)*	
			20	0.06	0.06	0.07	**0.14**	0.14	**0.27**	0.27	**0.54**	0.51	**0.74**	0.72	**0.82**	0.83	0.83	0.83	0.89	0.98	0.98	0.98	0.98	0.98	1.00	1.00	1.00	1.00	NA
			100	0.01	0.01	0.01	**0.03**	0.03	**0.05**	0.05	**0.11**	0.11	**0.18**	0.19	**0.32**	0.33	**0.37**	0.37	**0.49**	0.59	0.60	0.67	0.62	0.69	**0.84**	**0.99**	0.84	0.83	NA
			500	0.00	0.00	0.00	**0.01**	0.01	**0.01**	0.01	**0.02**	0.02	**0.04**	0.04	**0.06**	0.07	**0.08**	0.09	**0.10**	0.14	0.15	**0.17**	0.13	0.22	0.23	0.30	0.30	0.28	NA
	11	27583		*(p=3)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=176)*		*(p=352)*		*(p=704)*		*(p=1408)*		*(p=2176)*		*(p=4352)*		*(p=8460)*	
			20	0.06	0.06	0.07	**0.14**	0.15	**0.25**	0.25	**0.40**	0.48	**0.77**	0.74	**0.87**	0.81	**0.93**	0.82	**0.95**	0.90	**0.99**	0.94	0.98	0.93	0.99	1.00	1.00	1.00	NA
			100	0.01	0.01	0.01	**0.03**	0.03	**0.05**	0.05	**0.08**	0.10	**0.20**	0.20	**0.30**	0.28	**0.40**	0.34	**0.61**	0.42	**0.78**	**0.69**	0.49	0.46	**0.63**	0.87	**0.98**	0.94	NA
			500	0.00	0.00	0.00	**0.01**	0.01	**0.01**	0.01	**0.02**	0.02	**0.04**	0.04	**0.06**	0.06	**0.08**	0.08	**0.14**	0.10	**0.25**	**0.26**	0.13	0.13	0.13	0.40	**0.63**	0.51	NA
ped44 n=811 k=4 w=25 h=65	5	207136		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
			20	0.10	0.10	0.17	0.17	**0.32**	0.17	**0.55**	0.35	**0.69**	0.63	0.94	0.89	0.96	0.99	0.97	0.99	0.99	0.99	0.99	0.99	1.00	0.99	1.00	0.99	1.00	1.00
			100	0.02	0.02	0.03	0.03	**0.06**	0.03	**0.11**	0.07	**0.20**	0.13	**0.70**	0.22	0.77	0.82	0.83	0.89	0.94	0.92	0.96	0.89	**0.98**	0.84	0.99	0.93	1.00	0.97
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.04**	0.03	**0.14**	0.04	**0.26**	0.22	**0.43**	0.34	**0.72**	0.57	**0.83**	0.58	**0.86**	0.43	**0.95**	0.66	**0.98**	0.78
	6	95830		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=112)*		*(p=336)*		*(p=560)*		*(p=1120)*		*(p=2240)*		*(p=4480)*		*(p=8960)*		*(p=17920)*	
			20	0.09	0.09	0.18	0.18	0.34	0.34	0.57	0.58	0.63	**0.93**	0.90	0.92	0.95	0.98	0.97	0.99	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.07	0.07	0.11	0.12	0.20	0.20	**0.52**	0.35	0.75	**0.88**	0.86	0.92	0.92	0.94	**0.95**	0.86	0.98	0.96	0.99	1.00	1.00	0.99
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.04	0.04	**0.15**	0.07	**0.27**	0.21	**0.54**	0.36	0.67	0.61	**0.78**	0.43	**0.88**	0.56	**0.95**	0.64	0.97	0.97
ped50 n=514 k=6 w=17 h=47	3	4135		*(p=2)*		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14401)*											
			20	0.09	0.09	0.14	0.14	0.44	0.41	0.53	**0.66**	0.80	0.84	0.95	0.99	1.00	1.00	1.00	1.00										
			100	0.02	0.02	0.03	0.03	0.09	0.08	0.13	0.13	0.23	**0.44**	0.45	**0.88**	0.87	**0.97**	1.00	0.99										
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	0.05	**0.13**	0.11	**0.21**	0.29	**0.78**	0.79	**0.91**										
	4	1780		*(p=2)*		*(p=4)*		*(p=24)*		*(p=144)*		*(p=720)*		*(p=2160)*		*(p=5760)*		*(p=14400)*											
			20	0.10	0.10	0.18	0.19	0.67	0.67	0.85	**0.94**	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00										
			100	0.02	0.02	0.04	0.04	0.13	0.13	0.31	0.31	0.66	**0.88**	0.94	0.96	0.99	0.99	1.00	1.00										
			500	0.00	0.00	0.01	0.01	0.03	0.03	0.06	0.06	0.19	**0.27**	0.49	**0.77**	0.91	0.94	0.97	0.97										
ped51 n=1152 k=5 w=39 h=98	20	101788		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4064)*		*(p=7968)*	
			20	0.10	0.10	0.19	0.19	0.39	0.38	0.66	0.69	0.65	**0.82**	0.79	**0.88**	0.88	0.95	0.88	0.97	0.94	0.98	0.98	1.00	0.99	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.08	0.08	0.13	0.14	0.20	0.20	0.32	**0.41**	0.41	**0.46**	0.49	**0.68**	0.74	0.70	**0.83**	0.60	0.84	**0.99**	0.92	0.99	0.96	1.00
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.03	0.04	0.04	0.06	**0.08**	0.08	**0.09**	0.13	**0.20**	0.27	0.27	**0.38**	0.17	0.43	0.44	0.49	**0.56**	**0.82**	0.65
	21	164817		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*		*(p=8192)*	
			20	0.10	0.10	0.19	0.19	0.38	0.37	0.72	0.73	0.60	**0.83**	0.93	0.85	0.87	0.91	0.99	0.94	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.08	0.07	0.14	0.15	0.21	0.21	0.35	0.35	0.51	0.53	**0.75**	0.61	**0.97**	0.73	0.98	0.95	0.87	**0.99**	0.98	0.99	0.99	0.99
			500	0.00	0.00	0.01	0.01	0.02	0.01	0.03	0.03	0.04	0.04	0.07	0.07	0.10	0.11	0.15	**0.20**	**0.31**	0.24	**0.48**	0.27	0.52	**0.83**	**0.71**	0.62	0.70	0.75
ped7 n=1068 k=4 w=32 h=90	6	118383		*(p=2)*		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*	
			20	0.10	0.10	**0.17**	0.10	**0.29**	0.11	**0.52**	0.11	**0.64**	0.28	0.74	0.79	0.79	**0.95**	0.73	**0.98**	0.84	**0.97**	0.84	**0.97**	0.87	**1.00**	0.94	1.00	0.97	1.00
			100	0.02	0.02	**0.03**	0.02	**0.06**	0.02	**0.11**	0.02	**0.20**	0.06	**0.26**	0.17	0.29	**0.55**	0.29	**0.62**	0.36	**0.81**	0.37	**0.81**	0.49	**0.89**	0.59	**0.95**	0.63	**0.98**
			500	0.00	0.00	**0.01**	0.00	**0.01**	0.00	**0.02**	0.00	**0.04**	0.01	**0.05**	0.03	0.06	**0.12**	0.06	**0.14**	0.09	**0.23**	0.09	**0.23**	0.14	**0.35**	0.18	**0.34**	0.22	**0.67**
	7	93380		*(p=2)*		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1280)*		*(p=1280)*		*(p=2560)*		*(p=3840)*		*(p=7680)*	
			20	0.09	0.09	**0.17**	0.09	**0.30**	0.09	**0.61**	0.09	**0.74**	0.18	**0.94**	0.19	0.99	0.96	0.83	**0.97**	0.84	**0.98**	0.85	**0.98**	0.94	1.00	0.94	0.98	0.98	1.00
			100	0.02	0.02	**0.03**	0.02	**0.06**	0.02	**0.12**	0.02	**0.19**	0.04	**0.32**	0.04	0.37	**0.48**	0.33	**0.52**	0.37	**0.71**	0.37	**0.71**	0.48	**0.88**	0.52	**0.91**	0.60	**0.95**
			500	0.00	0.00	**0.01**	0.00	**0.01**	0.00	**0.02**	0.00	**0.04**	0.01	**0.06**	0.01	0.07	**0.11**	0.07	**0.15**	0.09	**0.18**	0.09	**0.18**	0.13	**0.46**	0.16	**0.47**	0.19	**0.58**
	8	30717		*(p=2)*		*(p=4)*		*(p=12)*		*(p=32)*		*(p=96)*		*(p=160)*		*(p=480)*		*(p=640)*		*(p=1276)*		*(p=1276)*		*(p=2552)*		*(p=3816)*		*(p=7588)*	
			20	0.08	0.08	**0.15**	0.08	**0.26**	0.08	**0.51**	0.08	**0.67**	0.15	**0.67**	0.15	0.72	0.78	0.76	**0.87**	0.72	**1.00**	0.83	**1.00**	0.91	1.00	0.97	1.00	0.97	1.00
			100	0.02	0.02	**0.03**	0.02	**0.05**	0.02	**0.10**	0.02	**0.14**	0.03	**0.24**	0.03	0.27	**0.33**	0.29	**0.34**	0.29	**0.38**	0.34	**0.38**	0.41	**0.52**	0.46	**0.72**	0.53	**0.96**
			500	0.00	0.00	**0.01**	0.00	**0.01**	0.00	**0.02**	0.00	**0.03**	0.01	**0.05**	0.01	0.06	**0.07**	0.06	**0.08**	0.07	**0.09**	0.07	**0.09**	0.11	**0.15**	0.13	**0.26**	0.15	**0.50**
ped9 n=1118 k=7 w=27 h=100	6	101172		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
			20	0.10	0.10	**0.19**	0.10	**0.30**	0.21	**0.53**	0.30	0.77	0.75	0.77	0.75	**0.90**	0.78	0.93	0.87	0.98	0.99	0.99	0.99	0.98	0.99	1.00	0.99	1.00	1.00
			100	0.02	0.02	**0.04**	0.02	**0.06**	0.04	**0.11**	0.06	**0.21**	0.16	**0.21**	0.16	**0.43**	0.22	**0.74**	0.43	**0.75**	0.65	0.93	0.94	0.93	0.95	0.96	0.96	**0.98**	0.85
			500	0.00	0.00	**0.01**	0.00	**0.01**	0.01	**0.02**	0.01	**0.04**	0.03	**0.04**	0.03	**0.09**	0.04	**0.17**	0.09	**0.23**	0.13	**0.60**	0.26	0.57	0.55	**0.81**	0.73	**0.92**	0.34
	7	58657		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
			20	0.10	0.10	0.19	0.19	0.31	0.30	**0.58**	0.51	0.66	0.70	0.65	0.70	0.85	0.85	0.93	0.93	0.96	0.96	0.97	0.99	0.98	0.99	1.00	0.99	1.00	0.99
			100	0.02	0.02	0.04	0.04	0.06	0.06	**0.12**	0.10	0.23	0.23	0.23	0.23	**0.46**	0.31	0.65	0.62	0.78	0.74	0.86	0.93	**0.90**	0.72	0.96	0.95	0.98	0.96
			500	0.00	0.00	0.01	0.01	0.01	0.01	**0.02**	0.02	0.05	0.05	0.05	0.05	**0.09**	0.06	**0.18**	0.12	**0.30**	0.25	**0.60**	0.40	**0.61**	0.21	**0.78**	0.39	**0.90**	0.80
	8	41061		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=32)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=2048)*		*(p=4096)*	
			20	0.10	0.10	0.19	0.19	0.30	0.30	0.45	0.49	0.64	**0.70**	0.64	0.70	**0.84**	0.66	**0.94**	0.72	0.97	0.99	0.99	0.99	0.98	0.99	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	0.06	0.06	0.09	0.10	**0.21**	0.14	**0.21**	0.14	**0.36**	0.25	**0.64**	0.26	**0.75**	0.49	0.90	0.84	0.90	0.94	0.97	0.98	0.98	0.98
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	**0.04**	0.03	**0.04**	0.03	**0.07**	0.05	**0.16**	0.05	**0.22**	0.10	**0.38**	0.21	**0.56**	0.37	0.78	0.81	0.89	0.89
Better by 10%				0x	0x	12x	6x	15x	9x	18x	11x	16x	17x	16x	14x	10x	18x	9x	21x	8x	13x	12x	10x	8x	12x	6x	11x	5x	6x
Better by 50%				0x	0x	12x	6x	12x	9x	15x	9x	11x	9x	10x	7x	2x	4x	5x	8x	3x	6x	6x	4x	5x	5x	1x	8x	1x	6x

Table B.18: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **linkage instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1		2		3		4		5		6		7		8		9		10		11		12		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
IF3-11-57 n=2670 k=3 w=37 h=95	15	121311	20	*(p=2)* 0.08	0.08	*(p=4)* 0.13	0.13	*(p=6)* **0.21**	0.15	*(p=18)* **0.30**	0.25	*(p=30)* 0.57	0.59	*(p=30)* 0.56	0.59	*(p=60)* 0.60	**0.85**	*(p=60)* 0.60	**0.85**	*(p=120)* 0.70	**0.91**	*(p=180)* 0.79	**0.97**	*(p=360)* 0.86	**0.97**	*(p=1080)* 0.97	0.98	*(p=1440)* 0.99	0.98
			100	0.02	0.02	0.03	0.03	**0.04**	0.03	**0.06**	0.05	0.11	0.12	0.11	0.12	0.13	**0.22**	0.13	**0.22**	0.14	**0.33**	0.22	**0.55**	0.31	**0.67**	0.51	**0.87**	0.54	**0.87**
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.01**	0.01	0.02	0.02	0.02	0.02	0.03	**0.04**	0.03	**0.04**	0.03	**0.07**	0.04	**0.11**	0.06	**0.17**	0.12	**0.48**	0.14	**0.52**
	16	35820	20	*(p=2)* 0.08	0.08	*(p=4)* 0.12	0.12	*(p=6)* **0.22**	0.15	*(p=18)* 0.28	**0.40**	*(p=30)* 0.49	0.49	*(p=30)* 0.47	0.49	*(p=60)* 0.66	0.63	*(p=60)* 0.68	0.63	*(p=120)* 0.73	**0.88**	*(p=180)* 0.74	**0.92**	*(p=360)* 0.95	0.99	*(p=1080)* 0.88	**1.00**	*(p=1440)* 0.97	1.00
			100	0.02	0.02	0.02	0.02	**0.04**	0.03	0.06	**0.08**	**0.11**	0.10	0.11	0.10	0.14	0.14	0.14	0.14	0.15	**0.23**	0.23	**0.34**	0.33	**0.57**	0.53	**0.75**	0.60	**0.90**
			500	0.00	0.00	0.00	0.00	**0.01**	0.01	0.01	**0.02**	**0.02**	0.02	0.02	0.02	0.03	0.03	0.03	0.03	0.03	**0.05**	0.05	**0.07**	0.07	**0.12**	0.12	**0.23**	0.15	**0.38**
	17	18312	20	*(p=2)* 0.08	0.08	*(p=4)* 0.12	**0.16**	*(p=6)* **0.21**	0.16	*(p=18)* 0.29	**0.37**	*(p=30)* 0.53	0.51	*(p=30)* 0.51	0.51	*(p=60)* 0.66	**0.73**	*(p=60)* 0.65	**0.73**	*(p=120)* 0.67	**0.96**	*(p=180)* 0.87	**0.98**	*(p=360)* 0.85	**0.99**	*(p=1080)* 0.99	1.00	*(p=1440)* 0.95	1.00
			100	0.02	0.02	0.02	**0.03**	**0.04**	0.03	0.06	**0.07**	0.11	0.10	0.10	0.10	0.13	**0.15**	0.14	0.15	0.15	**0.31**	0.25	**0.44**	0.36	**0.68**	0.63	**0.94**	0.70	**0.98**
			500	0.00	0.00	0.00	**0.01**	**0.01**	0.01	0.01	**0.01**	0.02	0.02	0.02	0.02	0.03	**0.03**	0.03	0.03	0.03	**0.06**	0.05	**0.09**	0.07	**0.14**	0.18	**0.28**	0.20	**0.64**
IF3-11-59 n=2711 k=3 w=32 h=73	14	35457	20	*(p=3)* 0.08	0.08	*(p=5)* 0.18	0.16	*(p=10)* 0.32	0.34	*(p=10)* 0.33	0.34	*(p=30)* 0.48	**0.60**	*(p=50)* 0.62	**0.87**	*(p=150)* 0.82	**0.97**	*(p=200)* 0.84	**0.98**	*(p=600)* 0.91	1.00	*(p=1000)* 0.89	**1.00**	*(p=2000)* 0.99	1.00	*(p=2000)* 0.99	1.00	*(p=4000)* 1.00	1.00
			100	0.02	0.02	0.04	0.03	0.06	0.07	0.07	0.07	0.10	**0.12**	0.12	**0.19**	0.25	**0.45**	0.36	**0.69**	0.56	**0.95**	0.59	**0.96**	0.66	**0.98**	0.87	**0.98**	0.76	**0.99**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.02	**0.02**	0.02	**0.04**	0.05	**0.09**	0.07	**0.15**	0.11	**0.41**	0.23	**0.68**	0.24	**0.60**	0.26	**0.60**	0.29	**0.92**
	15	8523	20	*(p=3)* 0.08	0.08	*(p=5)* 0.16	0.16	*(p=10)* 0.29	0.30	*(p=10)* 0.29	0.30	*(p=30)* 0.49	0.53	*(p=50)* 0.52	**0.91**	*(p=150)* 0.75	**0.97**	*(p=200)* 0.78	**0.98**	*(p=596)* 0.83	**0.99**	*(p=992)* 0.99	1.00	*(p=1962)* 0.99	1.00	*(p=1962)* 0.99	1.00	*(p=3886)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	0.06	0.06	0.06	0.06	0.10	0.11	0.10	**0.18**	0.27	**0.64**	0.29	**0.58**	0.43	**0.95**	0.76	**0.97**	0.87	**0.99**	0.89	**0.99**	0.98	0.99
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	**0.04**	0.05	**0.13**	0.06	**0.12**	0.11	**0.49**	0.29	**0.75**	0.48	**0.86**	0.51	**0.86**	0.72	**0.95**
	16	3023	20	*(p=3)* 0.08	0.08	*(p=5)* 0.18	0.19	*(p=10)* 0.27	**0.37**	*(p=10)* 0.37	0.37	*(p=30)* 0.51	**0.62**	*(p=50)* 0.42	**0.76**	*(p=150)* 0.85	**0.97**	*(p=200)* 0.86	**0.98**	*(p=600)* 0.99	0.98	*(p=1000)* 0.99	0.99	*(p=1999)* 1.00	1.00	*(p=1999)* 1.00	1.00	*(p=3992)* 1.00	1.00
			100	0.02	0.02	0.04	0.04	0.05	**0.07**	0.07	0.07	0.12	0.12	0.11	**0.22**	0.48	**0.54**	0.49	**0.63**	0.79	**0.95**	0.90	0.96	0.98	0.98	0.97	0.98	0.99	0.99
			500	0.00	0.00	0.01	0.01	0.01	**0.01**	0.01	0.01	0.02	0.02	0.02	**0.04**	0.10	0.11	0.12	0.13	0.24	**0.60**	0.53	**0.76**	0.72	**0.88**	0.75	**0.88**	0.91	0.92
IF3-13-58 n=3352 k=3 w=31 h=88	14	46464	20	*(p=2)* 0.10	0.10	*(p=4)* 0.15	0.16	*(p=12)* **0.23**	0.19	*(p=20)* **0.59**	0.24	*(p=60)* 0.71	0.76	*(p=100)* 0.67	**0.93**	*(p=200)* 0.86	**0.95**	*(p=200)* 0.87	0.95	*(p=600)* 0.93	0.99	*(p=1200)* 0.98	1.00	*(p=2000)* 0.98	1.00	*(p=4000)* 1.00	1.00	*(p=6400)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	**0.05**	0.04	**0.12**	0.05	**0.20**	0.15	**0.26**	0.23	0.35	**0.43**	0.31	**0.43**	0.44	0.49	0.67	0.64	0.80	**0.96**	0.98	0.99	0.99	1.00
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.04**	0.03	**0.05**	0.05	0.08	**0.09**	0.07	**0.09**	**0.11**	0.10	0.15	**0.17**	0.31	**0.34**	**0.81**	0.58	0.93	0.90
	16	20270	20	*(p=2)* 0.08	0.08	*(p=4)* 0.13	0.13	*(p=12)* 0.19	0.19	*(p=20)* 0.42	0.43	*(p=60)* 0.62	**0.72**	*(p=100)* 0.73	**0.83**	*(p=200)* 0.83	0.90	*(p=200)* 0.85	0.90	*(p=600)* 0.86	0.86	*(p=1200)* 0.99	1.00	*(p=1998)* 0.99	1.00	*(p=3990)* 1.00	1.00	*(p=6390)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.08	0.09	0.14	0.14	0.18	**0.25**	0.25	**0.41**	0.25	**0.41**	0.29	**0.40**	0.75	0.81	0.77	**0.98**	0.98	0.99	0.99	0.99
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.03	0.04	**0.05**	0.05	**0.08**	0.05	**0.08**	0.06	**0.10**	0.22	0.23	0.22	**0.40**	0.83	0.90	0.86	0.95
	18	7647	20	*(p=2)* 0.10	0.10	*(p=4)* 0.16	0.16	*(p=12)* 0.22	**0.24**	*(p=20)* **0.43**	0.36	*(p=60)* 0.56	**0.92**	*(p=100)* 0.77	**0.95**	*(p=200)* 0.84	**0.98**	*(p=200)* 0.88	**0.98**	*(p=591)* 0.97	0.99	*(p=1181)* 0.99	0.99	*(p=1958)* 0.99	1.00	*(p=3858)* 1.00	1.00	*(p=6121)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	0.04	**0.05**	**0.09**	0.07	0.15	**0.18**	0.31	0.30	0.42	**0.61**	0.42	**0.61**	0.62	**0.95**	0.91	0.97	0.93	0.98	0.98	0.99	0.99	0.99
			500	0.00	0.00	0.01	0.01	0.01	**0.01**	**0.02**	0.01	0.03	**0.04**	0.06	0.06	0.10	**0.12**	0.10	**0.12**	0.14	**0.45**	0.48	**0.75**	0.73	**0.91**	0.88	0.95	0.93	0.95
IF3-15-53 n=3384 k=3 w=32 h=108	17	345544	20	*(p=2)* 0.08	0.08	*(p=4)* 0.16	0.16	*(p=12)* 0.16	0.17	*(p=16)* 0.17	0.17	*(p=34)* 0.18	**0.28**	*(p=46)* 0.21	**0.48**	*(p=78)* 0.21	**0.55**	*(p=201)* 0.46	**0.78**	*(p=358)* 0.53	**0.78**	*(p=632)* 0.61	**0.79**	*(p=1093)* 0.68	**0.83**	*(p=1927)* 0.76	**1.00**	*(p=2831)* 0.77	**1.00**
			100	0.02	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.04	**0.06**	0.04	**0.11**	0.04	**0.11**	0.09	**0.24**	0.11	**0.31**	0.17	**0.31**	0.24	**0.45**	0.32	**0.76**	0.33	**0.79**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.02**	0.02	**0.05**	0.02	**0.06**	0.03	**0.06**	0.05	**0.11**	0.06	**0.17**	0.07	**0.18**
	18	98346	20	*(p=2)* 0.08	0.08	*(p=4)* 0.16	0.16	*(p=12)* 0.17	0.16	*(p=16)* 0.16	0.17	*(p=32)* 0.18	**0.20**	*(p=44)* 0.20	**0.52**	*(p=68)* 0.20	**0.55**	*(p=165)* 0.45	**0.86**	*(p=284)* 0.51	**0.93**	*(p=526)* 0.59	**0.90**	*(p=912)* 0.67	**1.00**	*(p=1572)* 0.80	**1.00**	*(p=2496)* 0.81	**1.00**
			100	0.02	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.04	**0.04**	0.04	**0.10**	0.04	**0.11**	0.09	**0.31**	0.11	**0.31**	0.18	**0.37**	0.25	**0.54**	0.34	**0.80**	0.35	**0.71**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.02**	0.02	**0.06**	0.02	**0.06**	0.04	**0.08**	0.05	**0.11**	0.07	**0.18**	0.08	**0.20**
Better by 10%				0x	0x	0x	3x	12x	6x	9x	6x	4x	14x	2x	20x	0x	28x	0x	25x	1x	26x	0x	23x	0x	24x	1x	18x	0x	15x
Better by 50%				0x	0x	0x	0x	0x	0x	3x	0x	0x	4x	0x	14x	0x	14x	0x	14x	0x	19x	0x	13x	0x	13x	0x	10x	0x	10x

Table B.19: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 1 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1		2		3		4		5		6		7		8		9		10		11		12		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
IF3-15-59 n=3730 k=3 w=31 h=84	18	28613		*(p=2)*		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=942)*		*(p=1855)*		*(p=3633)*		*(p=7098)*		*(p=13781)*	
			20	0.09	0.09	0.18	0.18	**0.27**	0.21	**0.50**	0.31	0.52	**0.72**	0.59	**0.93**	0.89	**0.99**	0.87	**1.00**	0.95	1.00	0.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.02	0.02	0.04	0.04	**0.05**	0.04	**0.10**	0.06	0.14	0.14	0.18	**0.23**	0.36	**0.55**	0.37	**0.73**	0.48	**0.87**	0.63	**0.96**	0.94	0.99	0.98	1.00	1.00	1.00
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.02**	0.01	0.03	0.03	0.04	**0.05**	0.08	**0.12**	0.08	**0.18**	0.13	**0.17**	0.24	**0.65**	0.48	**0.82**	0.74	**0.96**	0.94	0.99
	19	43307		*(p=2)*		*(p=4)*		*(p=8)*		*(p=20)*		*(p=40)*		*(p=80)*		*(p=240)*		*(p=476)*		*(p=936)*		*(p=1830)*		*(p=3571)*		*(p=6964)*		*(p=13482)*	
			20	0.07	0.07	0.13	0.13	0.20	0.21	0.36	0.39	0.60	0.63	0.59	**0.77**	0.88	**0.97**	0.91	1.00	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			100	0.01	0.01	0.03	0.03	0.04	0.04	0.07	0.08	0.12	0.13	0.13	**0.20**	0.31	**0.44**	0.34	**0.70**	0.42	**0.89**	0.77	**0.98**	0.95	0.99	0.99	0.99	1.00	1.00
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.03	**0.04**	0.06	**0.09**	0.07	**0.18**	0.10	**0.29**	0.21	**0.62**	0.41	**0.81**	0.90	0.96	0.98	0.98
IF3-16-56 n=3930 k=3 w=38 h=77	15	1891710				*(p=3)*		*(p=9)*		*(p=15)*		*(p=43)*		*(p=71)*		*(p=205)*		*(p=470)*		*(p=934)*		*(p=934)*		*(p=1827)*		*(p=2707)*		*(p=7582)*	
			20			0.08	0.08	0.16	0.16	0.26	0.27	0.36	**0.59**	0.61	**0.78**	0.86	**0.99**	0.91	0.97	0.98	0.99	0.98	0.99	0.98	1.00	0.99	1.00	0.98	1.00
			100			0.02	0.02	0.03	0.03	0.05	0.05	0.07	**0.13**	0.14	0.16	0.24	**0.69**	0.47	**0.81**	0.68	**0.92**	0.67	**0.92**	0.74	**0.98**	0.83	**0.99**	0.88	**1.00**
			500			0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.03**	0.03	0.03	0.05	**0.14**	0.11	**0.30**	0.17	**0.45**	0.17	**0.45**	0.25	**0.70**	0.31	**0.77**	0.48	**0.91**
	16	489614		*(p=2)*		*(p=3)*		*(p=9)*		*(p=15)*		*(p=42)*		*(p=70)*		*(p=201)*		*(p=455)*		*(p=900)*		*(p=900)*		*(p=1766)*		*(p=2629)*		*(p=7122)*	
			20	0.06	0.06	0.08	0.08	0.16	**0.23**	0.24	**0.35**	0.37	**0.61**	0.65	0.70	0.77	**0.96**	0.86	**0.98**	0.93	1.00	0.97	1.00	0.98	1.00	0.99	1.00	0.99	1.00
			100	0.01	0.01	0.02	0.02	0.03	**0.05**	0.05	**0.07**	0.08	**0.12**	0.16	**0.21**	0.24	**0.35**	0.47	**0.78**	0.55	**0.91**	0.64	**0.91**	0.78	**0.99**	0.75	**1.00**	0.80	**1.00**
			500	0.00	0.00	0.00	0.00	0.01	**0.01**	0.01	**0.01**	0.02	**0.02**	0.03	**0.04**	0.05	**0.07**	0.11	**0.25**	0.16	**0.36**	0.16	**0.36**	0.24	**0.51**	0.28	**0.71**	0.39	**0.91**
IF4-12-50 n=2569 k=4 w=28 h=80	13	57842		*(p=3)*		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
			20	0.08	0.08	0.29	0.30	0.54	0.54	0.75	**0.88**	0.89	0.88	0.98	0.99	1.00	1.00	0.99	1.00										
			100	0.02	0.02	0.06	0.06	0.11	0.11	0.19	0.20	0.51	**0.57**	**0.78**	0.68	0.95	0.99	0.97	0.99										
			500	0.00	0.00	0.01	0.01	0.02	0.02	0.04	0.04	0.11	**0.13**	0.23	0.25	0.32	**0.65**	**0.79**	0.29										
	14	33676		*(p=3)*		*(p=12)*		*(p=24)*		*(p=72)*		*(p=288)*		*(p=864)*		*(p=3456)*		*(p=5760)*											
			20	0.09	0.09	0.28	0.31	0.48	**0.57**	**0.67**	0.59	**0.94**	0.82	0.94	1.00	1.00	1.00	0.99	1.00										
			100	0.02	0.02	0.06	0.06	0.10	**0.12**	0.17	0.17	**0.52**	0.22	0.66	0.65	0.91	0.99	0.96	0.99										
			500	0.00	0.00	0.01	0.01	0.02	**0.02**	0.03	0.03	**0.12**	0.04	**0.21**	0.15	0.56	**0.89**	0.82	**0.96**										
IF4-12-55 n=2926 k=4 w=28 h=78	13	104837		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=128)*		*(p=256)*		*(p=512)*		*(p=1024)*		*(p=1024)*		*(p=1792)*		*(p=1792)*		*(p=3072)*	
			20	0.10	0.10	0.19	0.19	**0.35**	0.20	**0.54**	0.37	**0.77**	0.39	**0.83**	0.39	**0.90**	0.40	**0.96**	0.42	**0.98**	0.73	**0.98**	0.73	0.98	1.00	0.99	1.00	0.98	1.00
			100	0.02	0.02	0.04	0.04	**0.07**	0.04	**0.11**	0.07	**0.34**	0.09	**0.44**	0.09	**0.60**	0.09	**0.76**	0.10	**0.87**	0.24	**0.86**	0.24	0.79	**0.87**	0.86	0.87	0.81	**0.94**
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.07**	0.02	**0.09**	0.02	**0.16**	0.02	**0.27**	0.02	**0.43**	0.05	**0.44**	0.05	0.39	**0.44**	0.41	0.44	0.38	0.39
	14	25905		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=48)*		*(p=96)*		*(p=192)*		*(p=384)*		*(p=768)*		*(p=768)*		*(p=1536)*		*(p=1536)*		*(p=3072)*	
			20	0.09	0.09	0.18	0.18	**0.37**	0.19	**0.68**	0.20	**0.72**	0.40	0.72	0.77	0.82	**0.91**	0.85	**0.95**	0.93	1.00	0.94	1.00	0.96	1.00	0.99	1.00	0.99	1.00
			100	0.02	0.02	0.04	0.04	**0.07**	0.04	**0.14**	0.04	**0.26**	0.08	**0.27**	0.15	0.46	0.43	0.50	**0.67**	0.66	**0.84**	0.68	**0.84**	0.80	**0.97**	0.83	**0.97**	0.87	**0.99**
			500	0.00	0.00	0.01	0.01	**0.01**	0.01	**0.03**	0.01	**0.05**	0.02	**0.05**	0.03	**0.12**	0.09	0.16	**0.21**	0.28	**0.39**	0.23	**0.39**	**0.44**	0.29	**0.45**	0.29	0.36	**0.67**
IF4-17-51 n=3837 k=4 w=29 h=85	15	10607		*(p=2)*		*(p=4)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=40)*		*(p=56)*		*(p=128)*		*(p=152)*		*(p=176)*		*(p=352)*		*(p=400)*	
			20	0.10	0.10	0.20	0.20	0.19	0.20	0.38	0.39	0.47	0.48	0.52	**0.66**	0.60	0.63	0.74	0.71	0.80	**0.91**	0.80	**0.94**	0.92	0.93	0.97	0.98	0.97	0.98
			100	0.02	0.02	0.04	0.04	0.04	0.04	0.08	0.08	0.09	0.10	0.12	0.13	0.19	0.19	**0.25**	0.20	0.44	**0.51**	0.43	**0.51**	0.47	**0.57**	0.68	0.63	**0.86**	0.53
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.03	0.04	0.04	**0.05**	0.04	0.10	0.10	0.11	0.11	0.12	0.11	0.23	0.23	**0.42**	0.13
	16	66103		*(p=2)*		*(p=4)*		*(p=8)*		*(p=16)*		*(p=32)*		*(p=64)*		*(p=80)*		*(p=112)*		*(p=256)*		*(p=304)*		*(p=352)*		*(p=704)*		*(p=800)*	
			20	0.09	0.09	**0.18**	0.11	**0.19**	0.11	**0.36**	0.11	**0.44**	0.11	**0.49**	0.22	**0.68**	0.22	0.66	**0.74**	0.83	0.90	0.85	0.87	0.91	0.86	0.98	0.95	0.98	0.97
			100	0.02	0.02	**0.04**	0.02	**0.04**	0.02	**0.07**	0.02	**0.09**	0.02	**0.10**	0.04	**0.18**	0.04	0.21	0.20	0.41	0.38	0.40	0.40	0.39	0.36	0.68	0.67	0.72	0.76
			500	0.00	0.00	**0.01**	0.00	**0.01**	0.00	**0.01**	0.00	**0.02**	0.00	**0.02**	0.01	**0.04**	0.01	0.04	0.04	0.08	0.09	**0.09**	0.08	0.09	0.09	0.19	0.18	**0.27**	0.17
Better by 10%				0x	0x	3x	0x	12x	6x	13x	4x	12x	9x	10x	10x	7x	15x	6x	15x	3x	12x	4x	12x	1x	10x	1x	6x	3x	7x
Better by 50%				0x	0x	3x	0x	9x	0x	9x	0x	11x	6x	8x	3x	6x	6x	4x	8x	2x	6x	2x	6x	1x	4x	1x	2x	3x	3x

Table B.20: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **haplotyping instances, part 2 of 2**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var
pdb1a6m				*(p=9)*		*(p=81)*		*(p=511)*							
n=124	3	198326	20	0.09	0.09	0.11	**0.25**	0.18	**0.38**						
k=81			100	0.02	0.02	0.02	**0.05**	0.04	**0.08**						
w=15			500	0.00	0.00	0.00	**0.01**	0.01	**0.02**						
h=34															
pdb1duw				*(p=9)*		*(p=54)*		*(p=784)*		*(p=15081)*					
n=241	3	627106	20	0.13	0.13	0.18	**0.23**	0.22	**1.00**	0.57	**1.00**				
k=81			100	0.03	0.03	0.04	**0.05**	0.04	**0.52**	0.14	**1.00**				
w=9			500	0.01	0.01	0.01	**0.01**	0.01	**0.10**	0.03	**0.41**				
h=32															
pdb1e5k				*(p=66)*		*(p=1046)*		*(p=11321)*							
n=154	3	112654	20	0.28	0.28	1.00	1.00	1.00	1.00						
k=81			100	0.06	0.06	0.23	**0.75**	0.93	1.00						
w=12			500	0.01	0.01	0.05	**0.15**	0.21	**0.55**						
h=43															
pdb1f9i				*(p=81)*		*(p=6534)*									
n=103	3	68804	20	0.45	0.45	1.00	1.00								
k=81			100	0.09	0.09	0.54	**1.00**								
w=10			500	0.02	0.02	0.11	**0.33**								
h=24															
pdb1ft5				*(p=27)*		*(p=118)*		*(p=5281)*							
n=172	3	81118	20	0.11	0.11	0.14	**0.54**	1.00	1.00						
k=81			100	0.02	0.02	0.03	**0.11**	0.37	**1.00**						
w=14			500	0.00	0.00	0.01	**0.02**	0.07	**0.43**						
h=33															
pdb1hd2				*(p=79)*		*(p=3777)*									
n=126	3	101550	20	0.09	0.09	0.41	**1.00**								
k=81			100	0.02	0.02	0.08	**0.57**								
w=12			500	0.00	0.00	0.02	**0.11**								
h=27															
pdb1huw				*(p=9)*		*(p=42)*		*(p=293)*		*(p=654)*		*(p=1588)*		*(p=2597)*	
n=152	3	545249	20	0.06	0.06	0.06	0.06	0.06	**0.72**	0.06	**0.84**	0.06	**1.00**	0.07	**1.00**
k=81			100	0.01	0.01	0.01	0.01	0.01	**0.14**	0.01	**0.18**	0.01	**0.34**	0.01	**0.50**
w=15			500	0.00	0.00	0.00	0.00	0.00	**0.03**	0.00	**0.04**	0.00	**0.07**	0.00	**0.10**
h=43															
pdb1kao				*(p=27)*		*(p=215)*		*(p=752)*		*(p=3241)*					
n=148	3	716795	20	0.15	0.15	0.18	**0.58**	0.26	**0.99**	0.61	**1.00**				
k=81			100	0.03	0.03	0.04	**0.13**	0.05	**0.25**	0.12	**0.59**				
w=15			500	0.01	0.01	0.01	**0.03**	0.01	**0.05**	0.02	**0.12**				
h=41															
pdb1nfp				*(p=6)*		*(p=48)*		*(p=336)*		*(p=3812)*					
n=204	3	354720	20	0.06	0.06	0.07	**0.43**	0.13	**0.99**	0.36	**1.00**				
k=81			100	0.01	0.01	0.01	**0.09**	0.03	**0.28**	0.07	**0.97**				
w=18			500	0.00	0.00	0.00	**0.02**	0.01	**0.06**	0.01	**0.25**				
h=38															
pdb1rss				*(p=8)*		*(p=109)*		*(p=908)*		*(p=1336)*					
n=115	3	378579	20	0.05	0.05	0.19	**0.37**	0.59	**0.72**	0.68	**0.95**				
k=81			100	0.01	0.01	0.04	**0.07**	0.12	**0.18**	0.14	**0.19**				
w=12			500	0.00	0.00	0.01	**0.01**	0.02	**0.04**	0.03	**0.04**				
h=35															
pdb1vhh				*(p=27)*		*(p=1842)*		*(p=67760)*							
n=133	3	944633	20	0.21	0.21	**1.00**	0.23	1.00	1.00						
k=81			100	0.04	0.04	**0.46**	0.05	0.89	**1.00**						
w=14			500	0.01	0.01	**0.10**	0.01	0.24	**0.72**						
h=35															
Better by 10%				0x	0x	3x	25x	0x	23x	0x	15x	0x	3x	0x	3x
Better by 50%				0x	0x	3x	22x	0x	19x	0x	12x	0x	3x	0x	3x

Table B.21: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **side-chain prediction instances**.

instance	i	T_{seq}	#cpu	Cutoff depth d = 1 fix	1 var	2 fix	2 var	3 fix	3 var	4 fix	4 var	5 fix	5 var	6 fix	6 var	7 fix	7 var	8 fix	8 var	9 fix	9 var	10 fix	10 var	11 fix	11 var	12 fix	12 var	13 fix	13 var	14 fix	14 var	15 fix	15 var
75-25-1 n=624 k=2 w=38 h=111	12	77941	20	*(p=2)* 0.09	0.09	*(p=4)* 0.15	0.14	*(p=8)* 0.21	0.21	*(p=16)* 0.39	0.37	*(p=16)* 0.37	0.37	*(p=32)* 0.56	0.57	*(p=64)* 0.72	0.76	*(p=128)* **0.78**	0.67	*(p=192)* **0.82**	0.69	*(p=192)* **0.85**	0.69	*(p=192)* **0.81**	0.69	*(p=384)* **0.86**	0.75	*(p=768)* 0.90	0.84	*(p=1152)* 0.88	0.94	*(p=2112)* 0.94	0.96
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.08	0.07	0.07	0.07	0.11	0.11	0.17	**0.22**	0.23	0.23	0.23	0.23	0.22	0.23	0.23	0.23	**0.27**	0.22	**0.45**	0.32	**0.53**	0.48	0.57	**0.72**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.02	0.02	0.03	**0.04**	0.05	0.05	0.05	0.05	0.04	0.05	0.05	0.05	**0.06**	0.05	**0.10**	0.06	**0.11**	0.10	0.11	**0.14**
	14	15402	20	*(p=2)* 0.08	0.08	*(p=4)* 0.14	0.14	*(p=8)* 0.23	0.22	*(p=8)* 0.23	0.22	*(p=8)* 0.23	0.22	*(p=16)* 0.35	0.34	*(p=32)* 0.52	**0.58**	*(p=64)* **0.63**	0.52	*(p=96)* **0.57**	0.50	*(p=96)* **0.71**	0.50	*(p=192)* **0.65**	0.49	*(p=288)* **0.76**	0.63	*(p=576)* **0.92**	0.77	*(p=864)* 0.74	**0.97**	*(p=1584)* 0.98	1.00
			100	0.02	0.02	0.03	0.03	0.05	0.04	0.05	0.04	0.05	0.04	0.07	0.07	0.12	**0.14**	0.14	0.14	0.14	0.13	0.14	0.13	**0.17**	0.13	**0.20**	0.14	**0.27**	0.21	0.28	**0.35**	0.31	**0.63**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	**0.03**	0.03	0.03	0.03	0.03	0.03	0.03	**0.03**	0.03	**0.04**	0.03	**0.05**	0.04	0.06	**0.07**	0.06	**0.18**
75-25-3 n=624 k=2 w=37 h=115	12	104037	20	*(p=2)* 0.09	0.09	*(p=4)* 0.14	0.14	*(p=4)* 0.13	0.14	*(p=6)* 0.19	0.19	*(p=6)* 0.19	0.19	*(p=12)* 0.22	0.22	*(p=24)* 0.35	0.35	*(p=48)* 0.50	0.53	*(p=48)* 0.51	0.53	*(p=72)* **0.81**	0.66	*(p=144)* **0.98**	0.89	*(p=288)* 0.93	0.85	*(p=576)* 0.95	0.95	*(p=960)* 0.99	0.99	*(p=1536)* 0.97	0.98
			100	0.02	0.02	0.03	0.03	0.03	0.03	0.04	0.04	0.04	0.04	0.04	0.04	0.07	0.07	0.10	0.11	0.10	0.11	0.16	0.17	0.27	0.25	**0.39**	0.24	**0.78**	0.42	0.64	0.61	0.55	**0.63**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.03	0.05	0.05	**0.08**	0.05	**0.16**	0.08	0.16	0.15	0.16	0.16
	15	33656	20	*(p=2)* 0.08	0.08	*(p=4)* 0.14	0.14	*(p=4)* 0.14	0.14	*(p=6)* 0.19	0.19	*(p=6)* 0.19	0.19	*(p=12)* 0.34	0.34	*(p=24)* 0.62	0.61	*(p=48)* 0.70	**0.80**	*(p=48)* 0.72	**0.80**	*(p=72)* 0.86	0.86	*(p=144)* 0.92	0.91	*(p=288)* 0.97	0.95	*(p=576)* 0.98	0.97	*(p=960)* 0.99	1.00	*(p=1536)* 0.97	0.99
			100	0.02	0.02	0.03	0.03	0.03	0.03	0.04	0.04	0.04	0.04	0.07	0.07	0.12	0.12	0.19	**0.23**	0.20	**0.23**	**0.32**	0.28	**0.57**	0.50	**0.82**	0.69	0.82	0.79	0.91	0.92	0.65	**0.90**
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.04	**0.05**	0.04	**0.05**	**0.06**	0.06	**0.11**	0.10	**0.17**	0.14	0.31	**0.34**	0.41	0.40	0.24	**0.34**
75-25-7 n=624 k=2 w=37 h=120	16	297377	20	*(p=2)* 0.07	0.07	*(p=3)* 0.12	0.12	*(p=6)* 0.18	0.18	*(p=12)* 0.18	0.18	*(p=24)* 0.32	**0.46**	*(p=36)* 0.47	0.50	*(p=72)* 0.75	**0.83**	*(p=144)* 0.85	**0.96**	*(p=216)* 0.96	0.97	*(p=288)* 0.86	**0.96**	*(p=504)* 0.96	0.96	*(p=1008)* 0.94	0.96	*(p=2016)* 0.96	0.98	*(p=2688)* 0.98	0.99	*(p=3360)* 0.99	0.99
			100	0.01	0.01	0.02	0.02	0.04	0.04	0.04	0.04	0.06	**0.09**	0.09	0.10	0.16	0.17	0.29	**0.45**	0.46	0.47	0.49	**0.57**	0.72	**0.81**	0.70	**0.79**	0.80	**0.90**	0.82	0.90	0.90	0.93
			500	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	0.02	0.03	0.03	0.06	**0.09**	0.09	0.09	0.10	**0.17**	0.17	0.18	0.25	**0.44**	0.38	**0.58**	0.41	**0.58**	0.59	**0.70**
	18	21694	20	*(p=2)* 0.07	0.07	*(p=3)* 0.09	0.09	*(p=6)* 0.16	0.16	*(p=12)* 0.16	0.17	*(p=24)* **0.29**	0.17	*(p=36)* **0.40**	0.17	*(p=72)* **0.67**	0.18	*(p=144)* **0.77**	0.18	*(p=216)* **0.85**	0.31	*(p=288)* **0.88**	0.32	*(p=504)* **0.94**	0.38	*(p=1008)* 0.94	0.97	*(p=2014)* 0.97	0.99	*(p=2661)* 0.99	0.98	*(p=3325)* 0.98	0.99
			100	0.01	0.01	0.02	0.02	0.03	0.03	0.03	0.03	**0.06**	0.03	**0.08**	0.04	**0.13**	0.04	**0.22**	0.04	**0.36**	0.06	**0.38**	0.07	**0.68**	0.08	**0.73**	0.35	0.60	0.57	**0.85**	0.60	0.76	**0.85**
			500	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.03**	0.01	**0.04**	0.01	**0.07**	0.01	**0.08**	0.01	**0.14**	0.02	**0.19**	0.07	0.18	0.16	**0.22**	0.19	0.20	**0.50**
75-26-10 n=675 k=2 w=39 h=124	16	46985	20	*(p=2)* 0.09	0.09	*(p=4)* 0.14	0.14	*(p=8)* 0.26	0.24	*(p=8)* 0.24	0.24	*(p=16)* 0.44	0.44	*(p=16)* 0.44	0.44	*(p=32)* 0.72	0.73	*(p=64)* **0.81**	0.71	*(p=128)* 0.83	0.77	*(p=192)* 0.78	0.78	*(p=384)* **0.88**	0.77	*(p=384)* **0.88**	0.77	*(p=768)* 0.92	0.93	*(p=1280)* 0.97	0.95	*(p=1280)* 0.97	0.95
			100	0.02	0.02	0.03	0.03	0.05	0.05	0.05	0.05	0.09	0.09	0.09	0.09	0.15	0.15	0.24	**0.27**	0.27	0.29	0.29	0.32	**0.36**	0.32	**0.36**	0.32	0.60	0.61	0.70	0.75	0.73	0.75
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.03	0.05	**0.05**	0.05	0.06	0.06	**0.07**	**0.07**	0.07	**0.07**	0.07	**0.15**	0.13	0.19	**0.22**	0.19	**0.22**
	18	26855	20	*(p=2)* 0.08	0.08	*(p=4)* 0.13	0.14	*(p=8)* 0.22	0.22	*(p=8)* 0.22	0.22	*(p=16)* **0.39**	0.33	*(p=24)* **0.49**	0.39	*(p=48)* 0.82	0.84	*(p=80)* 0.80	0.78	*(p=160)* **0.88**	0.78	*(p=240)* 0.86	**0.97**	*(p=480)* 0.88	0.92	*(p=480)* 0.92	0.92	*(p=960)* 0.85	**0.97**	*(p=1216)* **0.97**	0.83	*(p=1216)* **0.96**	0.83
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.04	0.04	**0.08**	0.07	**0.10**	0.08	0.19	0.18	0.32	0.34	0.40	0.40	0.46	0.47	0.51	0.52	**0.62**	0.52	0.51	0.50	**0.48**	0.43	**0.79**	0.43
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	**0.02**	0.01	**0.02**	0.02	0.04	0.04	0.06	0.07	0.08	**0.09**	0.09	0.09	0.12	0.12	0.12	0.12	0.14	0.13	0.13	0.12	**0.20**	0.12
75-26-2 n=675 k=2 w=39 h=120	16	25274	20	*(p=2)* 0.09	0.09	*(p=4)* **0.17**	0.14	*(p=8)* **0.23**	0.20	*(p=12)* **0.34**	0.22	*(p=24)* **0.58**	0.37	*(p=48)* **0.75**	0.64	*(p=96)* **0.87**	0.73	*(p=144)* 0.88	0.93	*(p=288)* 0.98	0.90	*(p=384)* 0.94	0.91	*(p=640)* 0.99	0.91	*(p=1280)* 0.99	0.98	*(p=1280)* 0.99	0.98	*(p=2560)* 0.99	0.99	*(p=3840)* 1.00	0.99
			100	0.02	0.02	**0.03**	0.03	**0.05**	0.04	**0.07**	0.04	**0.12**	0.07	**0.20**	0.13	**0.38**	0.24	**0.39**	0.25	0.52	0.50	**0.71**	0.59	**0.89**	0.63	**0.95**	0.68	0.71	0.68	0.89	0.92	0.97	0.95
			500	0.00	0.00	**0.01**	0.01	**0.01**	0.01	**0.01**	0.01	**0.02**	0.01	**0.04**	0.03	**0.08**	0.05	**0.08**	0.05	0.11	0.12	**0.21**	0.12	**0.32**	0.20	**0.55**	0.25	0.27	0.25	0.27	**0.44**	**0.78**	0.62
	20	8053	20	*(p=2)* 0.09	0.09	*(p=4)* **0.12**	0.11	*(p=8)* 0.17	0.17	*(p=12)* 0.39	0.39	*(p=24)* **0.65**	0.46	*(p=48)* 0.73	0.78	*(p=96)* **0.93**	0.84	*(p=144)* 0.88	0.92	*(p=288)* 0.98	0.96	*(p=384)* 0.97	0.98	*(p=640)* 0.99	0.98	*(p=1280)* 1.00	0.99	*(p=1280)* 0.99	0.99	*(p=2560)* 1.00	1.00	*(p=3840)* 1.00	1.00
			100	0.02	0.02	**0.03**	0.02	0.03	0.03	0.08	0.08	**0.13**	0.09	**0.21**	0.16	**0.39**	0.23	**0.42**	0.24	**0.66**	0.47	**0.77**	0.56	**0.90**	0.74	0.97	0.90	0.97	0.98	0.97	0.98	0.99	0.98
			500	0.00	0.00	**0.01**	0.00	0.01	0.01	0.02	0.02	**0.03**	0.02	**0.04**	0.03	**0.08**	0.05	**0.08**	0.05	**0.15**	0.10	**0.24**	0.11	**0.40**	0.20	**0.72**	0.51	**0.69**	0.51	**0.87**	0.75	0.87	0.87
75-26-6 n=675 k=2 w=39 h=133	10	199460	20	*(p=2)* 0.07	0.07	*(p=4)* 0.16	0.16	*(p=8)* 0.30	0.30	*(p=16)* 0.36	0.37	*(p=32)* 0.53	**0.62**	*(p=64)* 0.59	**0.73**	*(p=128)* 0.74	0.80	*(p=128)* 0.77	0.80	*(p=128)* 0.77	0.80	*(p=256)* 0.85	0.91	*(p=384)* 0.96	0.89	*(p=576)* 0.96	0.94	*(p=1152)* 0.98	0.95	*(p=2304)* 0.99	0.93	*(p=4608)* 1.00	0.99
			100	0.01	0.01	0.03	0.03	0.06	0.06	0.07	0.07	0.12	0.12	0.15	0.15	0.26	0.26	0.26	0.26	0.26	0.26	0.41	0.44	0.51	0.55	0.50	0.52	0.62	0.67	**0.84**	0.71	**0.95**	0.84
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.03	0.05	0.05	0.05	0.05	0.05	0.05	0.09	0.09	0.11	0.11	0.11	0.11	0.19	0.19	0.24	0.24	0.41	0.39
	12	64758	20	*(p=2)* 0.08	0.08	*(p=4)* 0.14	0.14	*(p=8)* 0.27	0.26	*(p=16)* **0.34**	0.31	*(p=32)* **0.58**	0.33	*(p=64)* **0.60**	0.36	*(p=128)* 0.74	0.80	*(p=128)* 0.71	**0.80**	*(p=128)* 0.76	0.80	*(p=256)* **0.80**	0.72	*(p=384)* **0.95**	0.77	*(p=576)* **0.89**	0.74	*(p=1152)* **0.97**	0.87	*(p=2304)* **0.99**	0.87	*(p=4608)* 0.99	0.96
			100	0.02	0.02	0.03	0.03	0.05	0.05	**0.07**	0.06	**0.12**	0.07	**0.14**	0.07	0.23	0.23	0.23	0.23	0.24	0.23	**0.32**	0.24	**0.38**	0.24	**0.39**	0.32	**0.49**	0.39	**0.51**	0.39	**0.86**	0.59
			500	0.00	0.00	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.01	**0.03**	0.01	0.05	0.05	0.05	0.05	0.05	0.05	**0.07**	0.05	**0.08**	0.05	**0.08**	0.07	**0.13**	0.09	**0.12**	0.09	**0.30**	0.14
75-26-9 n=675 k=2 w=39 h=124	16	59609	20	*(p=2)* 0.08	0.08	*(p=4)* 0.15	0.15	*(p=8)* 0.15	**0.18**	*(p=16)* 0.20	**0.38**	*(p=24)* 0.37	0.40	*(p=48)* **0.62**	0.43	*(p=96)* 0.64	**0.77**	*(p=120)* 0.76	0.81	*(p=240)* 0.93	0.97	*(p=480)* 0.92	0.93	*(p=960)* 0.97	0.94	*(p=1920)* 0.96	0.96	*(p=3840)* 1.00	1.00	*(p=3840)* 1.00	1.00	*(p=7680)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	0.03	**0.04**	0.04	**0.08**	0.07	0.08	**0.12**	0.09	0.15	0.16	0.21	0.19	**0.31**	0.23	0.36	0.38	0.44	**0.58**	0.62	**0.78**	0.86	0.80	0.83	0.80	0.97	0.94
			500	0.00	0.00	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.01	0.02	**0.02**	0.02	0.03	0.03	0.04	0.04	**0.06**	0.05	0.08	0.08	0.10	**0.14**	0.14	**0.26**	0.19	**0.35**	0.26	**0.35**	0.41	**0.46**
	18	66533	20	*(p=2)* 0.08	0.08	*(p=4)* 0.13	0.13	*(p=8)* 0.13	0.13	*(p=16)* 0.17	**0.31**	*(p=24)* 0.31	**0.54**	*(p=48)* 0.54	0.57	*(p=96)* 0.55	**0.61**	*(p=120)* 0.72	0.79	*(p=240)* 0.82	**0.94**	*(p=480)* 0.86	**0.96**	*(p=960)* 0.93	0.97	*(p=1920)* 0.99	0.98	*(p=3840)* 1.00	0.99	*(p=3840)* 0.99	0.99	*(p=7680)* 0.99	1.00
			100	0.02	0.02	0.03	0.03	0.03	0.03	0.03	**0.06**	0.06	**0.11**	0.11	0.11	0.12	0.12	0.19	0.18	**0.29**	0.19	0.32	**0.75**	0.40	**0.72**	0.58	**0.87**	0.80	**0.96**	0.80	**0.96**	0.94	0.98
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.02	0.02	0.02	0.02	0.04	0.04	**0.06**	0.04	0.07	**0.23**	0.09	**0.24**	0.13	**0.40**	0.23	**0.46**	0.26	**0.46**	0.42	**0.77**
	20	5708	20	*(p=2)* 0.10	0.10	*(p=4)* 0.17	0.17	*(p=8)* 0.19	0.19	*(p=16)* 0.19	**0.24**	*(p=24)* 0.24	**0.38**	*(p=48)* 0.39	0.40	*(p=96)* 0.46	**0.60**	*(p=120)* 0.62	**0.93**	*(p=240)* 0.91	0.98	*(p=320)* 0.85	**0.97**	*(p=640)* 0.94	0.99	*(p=1280)* 1.00	0.99	*(p=2560)* 0.99	1.00	*(p=2560)* 1.00	1.00	*(p=5120)* 1.00	1.00
			100	0.02	0.02	0.03	0.03	0.04	0.04	0.04	**0.05**	0.05	**0.08**	0.08	0.08	0.10	**0.12**	0.15	**0.29**	0.28	**0.55**	0.48	**0.57**	0.63	**0.84**	0.87	0.93	0.95	0.99	0.98	0.99	0.99	0.99
			500	0.00	0.00	0.01	0.01	0.01	0.01	0.01	**0.01**	0.01	**0.02**	0.02	0.02	0.02	**0.02**	0.03	**0.06**	0.06	**0.11**	0.10	**0.11**	0.20	**0.29**	0.35	**0.70**	0.61	**0.86**	0.67	**0.86**	0.96	0.94
Better by 10%				0x	0x	6x	0x	3x	3x	6x	9x	15x	10x	17x	1x	9x	11x	10x	12x	12x	7x	15x	11x	20x	7x	22x	7x	12x	8x	11x	10x	7x	13x
Better by 50%				0x	0x	0x	0x	0x	0x	3x	6x	9x	6x	8x	0x	7x	0x	7x	5x	6x	2x	5x	3x	7x	2x	5x	4x	4x	3x	0x	2x	3x	4x

Table B.22: Average resource utilization with 20, 100, 500, and "unlimited" CPUs, on **grid instances**.