FINDING MOST LIKELY HAPLOTYPES IN GENERAL PEDIGREES THROUGH PARALLEL SEARCH WITH DYNAMIC LOAD BALANCING

LARS OTTEN and RINA DECHTER

Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697, U.S.A.
{lotten,dechter}@ics.uci.edu

General pedigrees can be encoded as Bayesian networks, where the common MPE query corresponds to finding the most likely haplotype configuration. Based on this, a strategy for grid parallelization of a state-of-the-art Branch and Bound algorithm for MPE is introduced: independent worker nodes concurrently solve subproblems, managed by a Branch and Bound master node. The likelihood functions are used to predict subproblem complexity, enabling efficient automation of the parallelization process. Experimental evaluation on up to 20 parallel nodes yields very promising results and suggest the effectiveness of the scheme, solving several very hard problem instances. The system runs on loosely coupled commodity hardware, simplifying deployment on a larger scale in the future.

1. Introduction

Given a general pedigree expressing ancestral relations over a set of individuals, the haplotyping problem is to infer the most likely ordered haplotypes for each individual from measured unordered genotypes. This has previously been cast as solving an optimization problem over a appropriately constructed Bayesian network,[6] for which powerful inference algorithms can be exploited. Yet practical problems remain infeasible as more data becomes available, for example through SNP sequencing, suggesting a shift to parallel or distributed computation.

This paper therefore explores parallelization of combinatorial optimization tasks over such Bayesian networks, which are typically generalized through the framework of graphical models. Specifically, we consider one of the best exact search algorithms for solving the MPE/MAP task over graphical models, AND/OR Branch and Bound (AOBB). AOBB, which exploits independencies and unifiable subproblems, has demonstrated superior performance for these tasks compared with other state-of the art exact solvers (e.g., it was ranked first or second in several competitions[13]).

To parallelize AOBB we use the established concept of parallel tree search[8] where the search space is explored centrally up to a certain depth and the remaining subtrees are solved in parallel. For graphical models this can be implemented straightforwardly by exploring the search space of partial instantiations up to a certain depth and solving the remaining conditioned subproblems in parallel. This approach has already proven successful for likelihood computation in Superlink-Online, which parallelizes cutset conditioning for linkage analysis tasks.[16] Our work differs in focusing on optimization (e.g., MPE/MAP) and in exploiting the AND/OR paradigm, leveraging additional subproblem independence for parallelism. Moreover, we use the power of Branch and Bound in a central search space that manages (and prunes) the set of conditioned subproblems.

The main difference however is that, compared to likelihood computation, optimization presents far greater challenges with respect to load balancing. Hence the primary challenge in search tree parallelization is to determine the “cutoff”, the *parallelization frontier*. Namely, we need a mechanism to decide when to terminate a branch in the central search space and send the corresponding subproblem to a machine on the network. There are two primary issues: *(1)* Avoid *redundancies*: caching of unifiable subproblems is lost across the independently solved subproblems, hence some work might be duplicated; *(2)* Maintain *load balancing* among the grid resources, dividing the total work equally and without major idling periods. While introducing redundancy into the search space can be counterproductive for both tasks, load balancing is a far greater challenge for optimization, since the cost function is exploited in pruning the search space. Capturing this aspect is essential in predicting the size of a subproblem and thus the focus of this paper.

The contribution of this work is thus as follows: We suggest a parallel BaB scheme in a graphical model context and analyze some of its design trade-offs. We devise an estimation scheme that predicts the size of future subproblems based on cost functions and learns from previous subproblems to predict the extent of BaB pruning within future subproblems. We show that these complexity estimates enable effective load distribution (which was not possible via redundancy analysis only), and yield very good performance on several very hard practical problem instances, some of which were never solved before. Our approach assumes the most general master-worker scenario with minimal communication and can hence be deployed on a multitude of grid setups spanning hundreds, if not thousands of computers worldwide. While our current empirical work is tested on up to 20 machines so far, its potential for scaling up are very promising.

Related work: The idea of parallelized Branch and Bound in general is not new, but existing work often assumes a shared-memory architecture or extensive inter-process communication,[3,7,8] or specific grid hierarchies.[1] Earlier results on estimating the performance of search predict the size of general backtrack tress through random probing.[10,12] Similar schemes have been devised for Branch and Bound algorithms, where the algorithm is ran for a limited time and the partially explored tree is extrapolated.[4] Our method, on the other hand, is not sampling-based but only uses parameters available a priori and information learned from past subproblems which is facilitated through the use of depth-first branch and bound to explore the master search space.

2. Background

Our approach is based on the general framework of graphical model reasoning:

Definition 2.1 (graphical model). *A* graphical model *is given as a set of variables $X = \{X_1, \ldots, X_n\}$, their respective finite domains $D = \{D_1, \ldots, D_n\}$, a set of cost functions $F = \{f_1, \ldots, f_m\}$, each defined over a subset of X (the function's* scope*), and a combination operator (typically sum, product, or join) over functions. Together with a marginalization operator such as* $\min_X$ *and* $\max_X$ *we obtain a* reasoning problem.

For instance, the *MPE* problem (most probable explanation) is typically posed over a Bayesian Network structure, representing the factorization of a joint distribution into conditional probabilities, with the goal of finding an assignment with maximum probability. In the area of constraint reasoning, a *weighted CSP* is defined as minimizing the sum of a set of cost functions over the variables.

Definition 2.2 (primal graph, induced graph, induced width). *The* primal graph *of a graphical model is an undirected graph, $G = (X, E)$. It has the variables as its vertices and an edge connecting any two variables that appear in the scope of the same function. Given an undirected graph G and an ordering $d = X_1, \ldots, X_n$ of its nodes, the width of a node is the number of neighbors that precede it*

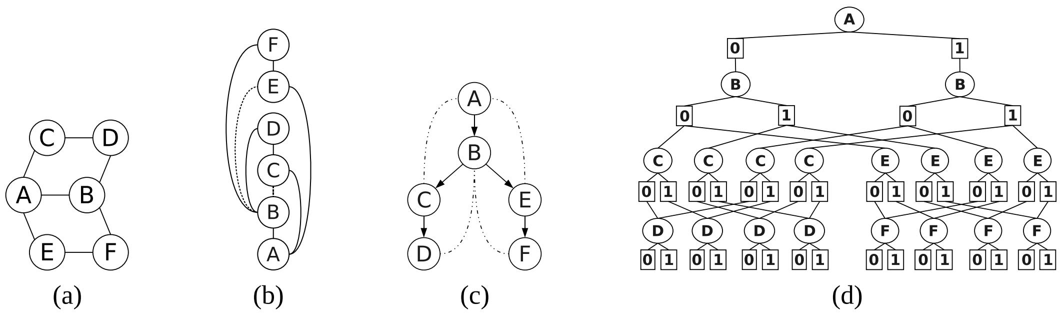


Fig. 1: (a) Example primal graph with six variables, (b) its induced graph along ordering $d = A, B, C, D, E, F$, (c) a corresponding pseudo tree, and (d) the resulting context-minimal AND/OR search graph.

in d*. The* induced graph G' *of* G *is obtained as follows: from last to first in* d*, each node's preceding neighbors are connected to form a clique (where new edges are taken into account when processing the remaining nodes). The* induced width w^* *is the maximum width over all nodes in the induced graph along ordering* d*.*

Figure 1(a) depicts the primal graph of an example problem with six variables. The induced graph for the example problem along ordering $d = A, B, C, D, E, F$ is depicted in Figure 1(b), its induced width is 2. Note that different orderings will vary in their implied induced width; finding an ordering of minimal induced width is known to be NP-hard, in practice heuristics like *minfill*[11] are used to obtain approximations.

2.1. *Encoding Pedigrees as Bayesian Networks*

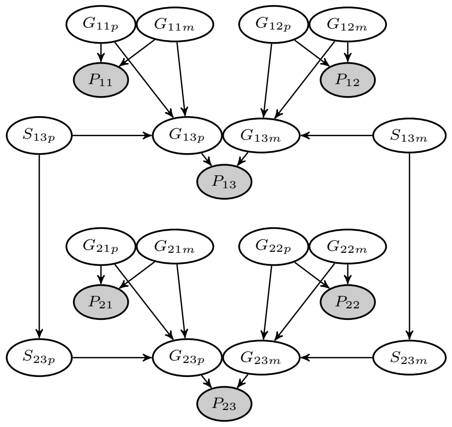


Fig. 2: Example fragment of a Bayesian network encoding of a general pedigree.

Expressing a particular pedigree as a Bayesian Network utilizes three building blocks: (1) For each individual and each locus, the two haplotypes are represented by two variables, with the possible alleles as their domain and a probability distribution conditioned on the variables representing the parents' haplotypes at this locus. (2) The measured, unordered genotypes are captured as phenotype variables, which are conditioned on the corresponding pair of haplotypes. (3) Auxiliary binary selector variables are linked across loci, to capture recombination events.

Figure 2 shows a simple example of such a Bayesian network, the displayed fragment includes three individuals (two parents and their child) and two loci. For instance, G_{13p} is the paternal haplotype of individual 3 (the child) at locus 1. It depends on the father's haplotypes G_{11p} and G_{11m}, where the inheritance is determined by the selector variable S_{13p} i.e., $G_{13p} = G_{11p}$ if $S_{13p} = 0$ and $G_{13p} = G_{11m}$ if $S_{13p} = 1$. Together with the maternal haplotype G_{13m}, G_{13p} determines the genotype in P_{13}. The value of the inheritance selector S_{23p} for the paternal haplotype of individual 3 at locus 2 is dependent on the selector S_{13p} for locus 1, where the actual probabilities are recombination fractions between these two loci, provided as input.

With this construction, the joint distribution of the Bayesian network captures the probability over all haplotype configurations. Given a set of evidence (i.e., measurements for some or all of the unordered genotypes), the solution to the common problem of finding the most probable explanation (MPE) will yield the most likely haplotypes.[6]

2.2. *AND/OR Search Spaces*

The concept of AND/OR search spaces has been introduced as a unifying framework for advanced algorithmic schemes for graphical models to better capture the structure of the underlying graph.[5] Its main virtue consists in exploiting conditional independencies between variables, which can lead to exponential speedups. The search space is defined using a *pseudo tree*, which captures problem decomposition:

Definition 2.3 (pseudo tree). *Given an undirected graph $G = (X, E)$, a* pseudo tree *of G is a directed, rooted tree $\mathcal{T} = (X, E')$ with the same set of nodes X, such that every arc of G that is not included in E' is a back-arc in $\mathcal{T}$, namely it connects a node in $\mathcal{T}$ to an ancestor in $\mathcal{T}$. The arcs in E' may not all be included in E.*

AND/OR Search Trees : Given a graphical model instance with variables X and functions F, its primal graph (X, E), and a pseudo tree $\mathcal{T}$, the associated *AND/OR search tree* consists of alternating levels of OR and AND nodes. OR nodes are labeled X_i and correspond to the variables in X. AND nodes are labeled $\langle X_i, x_i \rangle$, or just x_i and correspond to the values of the OR parent's variable. The structure of the AND/OR search tree 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 X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$ that are consistent with the assignments along the path from the root; the children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of X_i in $\mathcal{T}$, representing conditionally independent subproblems. It was shown that, given a pseudo tree $\mathcal{T}$ of height h, the size of the AND/OR search tree based on $\mathcal{T}$ is $\mathcal{O}(n \cdot k^h)$, where k bounds the domain size of variables.[5]

AND/OR Search Graphs : Different nodes may root identical and can be merged through *caching*, yielding an *AND/OR search graph* of smaller size, at the expense of using additional memory during search. A mergeable node X_i can be identified by its *context*, the partial assignment of the ancestors of X_i which separates the subproblem below X_i from the rest of the network. Merging all context-mergeable nodes yields the *context minimal* AND/OR search graph.[5]

Proposition 2.1. *Given a graphical model, its primal graph G, and a pseudo tree $\mathcal{T}$, the size of the context-minimal AND/OR search graph is $\mathcal{O}(n \cdot k^{w^*})$, where w^* is the induced width of G over a depth-first traversal of $\mathcal{T}$ and k bounds the domain size.*

Example 2.1. Figure 1(c) depicts a pseudo tree extracted from the induced graph in Figure 1(b) and Figure 1(d) shows the corresponding context-minimal AND/OR search graph. Note that the AND nodes for B have two children each, representing independent subproblems and thus demonstrating problem decomposition. Furthermore, the OR nodes for D (with context $\{B, C\}$) and F (context $\{B, E\}$) have two edges converging from the AND level above them, signifying caching.

Weighted AND/OR Search Graphs : 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 sum 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 in the AND/OR search graph can be associated with a *value*: the value $v(n)$ of a node n is the minimal solution cost to the subproblem rooted at n, subject to the current variable instantiation along the path from the root to n. $v(n)$ can be computed recursively using the values of n's successors.[5]

2.3. *AND/OR Branch and Bound*

AND/OR Branch and Bound is a state-of-the-art algorithm for solving optimization problems over graphical models. Assuming a minimization task, it traverses the context-minimal AND/OR graph in a depth-first manner while keeping track of a current upper bound on the optimal solution cost. It 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.[5]

3. Setup and Parallel Scheme

We assume a very general parallel framework in which autonomous hosts are loosely connected over some network – in our case we use ten dual-core desktop computers, with CPU speeds between 2.33 and 3.0 GHz, on a local Ethernet, thus allowing experiments with up to 20 parallel nodes. We impose a *master-worker* hierarchy on the computers in the network, where a special *master* node runs a central process to coordinate the *workers*, which cannot communicate with each other. This general model is chosen to accommodate a wide range of parallel resources, where direct node communication is often either prohibitively slow or entirely impossible; it also facilitates flexible deployment on geographically dispersed, heterogeneous resources in the future.

The setup is similar to Superlink-Online,[16] which has been very successful in using large-scale parallelism in likelihood algorithms for genetic linkage analysis, or SETI@home,[2] which uses Internet-connected PCs around the world to search through enormous amounts of radio data. Like Superlink-Online, our system is implemented on top of the *Condor* grid middleware.[17]

3.1. *Parallel AND/OR Branch and Bound*

We include here only a brief outline of the master process and refer to Ref. 15 for details and pseudo code. As a Branch and Bound scheme, exploration and propagation alternate as follows:

Master Exploration. The master process explores the AND/OR graph in a depth-first manner guided by the start pseudo tree T_c. Upon expansion of a node n it consults a heuristic lower bound $lb(n)$ to make pruning decisions, where the computation of the upper bound $ub(n)$ can take into account previous subproblem solutions. If $lb(n) \geq ub(n)$, the current subtree can be pruned. Exploration is halted when the parallelization frontier is reached. The master then sends the respective subproblem, given by the subproblem root variable and its context instantiation, to a worker node.

Master Propagation. The master process also collects and processes subproblem solutions from the worker nodes. Upon receipt of a solved subproblem, its solution is assigned as the value of the respective node in the master search space and recursively propagated upwards towards the root, updating node values identical to sequential AOBB.

With a fixed number of workers p, the master initially generates only the first p subproblems; worker nodes solve subproblems using sequential AOBB[13] and send the solution back to the master, where it is propagated; the central exploration is then resumed to generate the next subproblem.

Example 3.1. Consider again the AND/OR search graph in Figure 1(d). Given a start pseudo tree having A and B, we can illustrate the parallelization scheme through Figure 3: the search space of the master process is marked in gray, and each of the eight independent subproblems rooted at C or E can be solved in parallel.

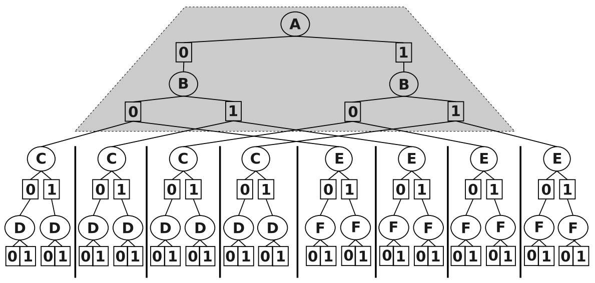


Fig. 3: Parallelization scheme applied to the example problem: master search space (gray) and eight independent subproblems.

The central decision is obviously where to place the *parallelization frontier*, i.e., at which point to cut off the master search space. Preliminary experiments, conducted with globally enforced fixed-depth cutoff, have shown that the parallel scheme carries great potential.[15] It also became evident, however, that the issue of load balancing is crucial for the overall performance (while structural redundancy, for instance, does not seem to have a major impact). In particular, the scheme needs to ensure that the workload is evenly distributed over all processing units, each of which should be utilized equally. Secondly, it is critical to minimize overhead resulting from network communication and resource management.

In the fixed cutoff experiments we observed great variance in subproblem complexity with relative differences of up to three orders of magnitude. In the following section we will therefore focus on estimating subproblem complexity ahead of time[15] . With this the master can dynamically decide at which point a given subproblem is “simple enough” for parallelization (to avoid excessively hard tasks) and also avoid very easy subproblems, whose solution time will be dominated by the distributed system overhead.

4. Predicting Subproblem Size Using the Cost Function

In this section we derive a scheme for estimating the size of the explored search space of a conditioned subproblem using parameters associated with the problem’s cost function, allowing us to enforce an upper bound on the complexity of subproblems.

When considering a particular subproblem rooted at node n, we propose to estimate its complexity $N(n)$ (i.e., the number of node AOBB explores to solve it) as a function of the heuristic lower bound $L(n)$ as well as the upper bound $U(n)$, which can be computed based on earlier parts of the search space or through an approximation algorithm like local search; we will also use the height $h(n)$ of the subproblem pseudo tree.

4.1. *Main Assumptions*

We consider a node n that roots the subproblem $P(n)$. If the search space below n was a perfectly balanced tree of height D, with every node having exactly b successors, clearly the total number of nodes is $N = (b^{D+1} - 1)/(b - 1) \approx b^D$.

However, even if the underlying search space is balanced, the portion expanded by BaB, guided by some heuristic evaluation function, is not: the more accurate the heuristic, the more focused around the optimal solution paths the search space will be. In state-based search spaces it is therefore common to measure effectiveness in post-solution analysis via the *effective branching factor* defined as $b = \sqrt[D]{N}$ where D is the length of the optimal solution path and N is the actual number of nodes generated.[14]

Inspired by this approach, for a subproblem rooted at n we adopt the idea of approximating the explored search space by a balanced tree and express its size through $N(n) = b(n)^{D(n)}$. However, in place of the optimal solution path length (which corresponds to the pseudo tree height in our case), we propose to interpret $D(n)$ as the average leaf node depth $\bar{D}(n)$ defined as follows:

Definition 4.1 (Average leaf node depth). *Let $l_1, \ldots, l_j$ denote the leaf nodes generated when solving subproblem $P(n)$. We define the* average leaf node depth *of $P(n)$ to be $\bar{D}(n) := \frac{1}{j} \sum_{k=1}^{j} d_n(l_k)$, where $d_n(l_i)$ denotes the depth of leaf node i relative to the subproblem root n.*

We next aim to express $b(n)$ and $\bar{D}(n)$ as functions of the subproblem parameters $L(n)$, $U(N)$, and $h(n)$ (using other parameters is subject to future research).

4.2. *Estimating the Effective Branching Factor*

For the sake of simplicity we assume an underlying, "true" effective branching factor b that is constant for all possible subproblems. We feel this is a reasonable assumption since all subproblems are conditioned within the same graphical model. We thus model $b(n)$ as a normally distributed random variable and take its mean as the constant b, which we found to be confirmed in experiments. An obvious way to learn this parameter is then to average over the effective branching factors of previous subproblems, which is known to be the right statistic for estimating the true average of a population.

Estimating b for new Subproblem $P(n)$: Given a set of already solved subproblems $P(n_1), \ldots, P(n_r)$, we can compute $\bar{D}(n_i)$ and derive effective branching degrees $b(n_i) = \sqrt[\bar{D}(n_i)]{N(n_i)}$ for all i. We then estimate b through $b^* = \frac{1}{r} \sum_{i=1}^{r} b(n_i)$.

4.3. *Deriving and Predicting Average Leaf Depth*

With each subproblem $P(n)$ rooted at a node n we associate a lower bound $L(n)$ based on the heuristic estimate and an upper bound $U(n)$ derived from the best solution from previous subproblems[a]. Both $L(n)$ and $U(n)$ are known before we start solving $P(n)$. We can assume $L(n) < U(n)$, since otherwise n itself could be pruned and $P(n)$ was trivially solved. We denote with $lb(n')$ and $ub(n')$ the lower and upper bounds of nodes n' within the subproblem $P(n)$ at the time of their expansion and similarly assert that $lb(n') < ub(n')$ for any expanded node n'.

[a]We assume a graphical model with addition as the combination operator. Adaption to multiplication is straightforward.

Since the upper bound is derived from the best solution found so far it can only improve throughout the search process. Furthermore, assuming a monotonic heuristic function (that provides for any node n' a lower bound on the cost of the best solution path going through n'), the lower bounds along any path in the search space are non-decreasing and we can state that any node n' expanded within $P(n)$ satisfies:

$$L(n) \leq lb(n') < ub(n') \leq U(n)$$

Consider now a single path within $P(n)$, from n down to leaf node l_k, and denote it by $\pi_k = (n'_o, \ldots, n'_{d_n(l_k)})$, where $n'_0 = n$ and $d_n(l_k)$ is again the depth of l_k with respect to n (and hence $n'_{d_n(l_k)} = l_k$). We will write lb_i for $lb(n'_i)$ and ub_i for $ub(n'_i)$, respectively, and can state that $lb_i \geq lb_{i-1}$ and $ub_i \leq ub_{i-1}$ for all $1 \leq i \leq d_n(l_k)$ (note that $lb_0 = L(n)$ and $ub_0 = U(n)$). An internal node n' is pruned iff $lb(n') \geq ub(n')$ or equivalently $ub(n') - lb(n') \leq 0$, hence we consider the (non-increasing) sequence of values $(ub_i - lb_i)$ along the path π_k; in particular we are interested in the average change in value from one node to the next, which we capture as follows:

Definition 4.2 (Average path increment). *The* average path increment of π_k *within* $P(n)$ *is defined by the expression:*

$$inc(\pi_k) = \frac{1}{d_n(l_k)} \sum_{i=1}^{d_n(l_k)} \left((ub_i - lb_i) - (ub_{i-1} - lb_{i-1})\right) \tag{1}$$

If we assume $(ub_{d_n(l_k)} - lb_{d_n(l_k)}) = 0$, the sum reduces to $(U(n) - L(n))$. Thus rewriting Expression 1 for $d_n(l_k)$ and averaging to get $\bar{D}(n)$ as in Definition 4.1 yields:

$$\bar{D}(n) = (U(n) - L(n)) \, \frac{1}{j} \sum_{k=1}^{j} \frac{1}{inc(\pi_k)} \tag{2}$$

We now define $inc(n)$ of $P(n)$ through $inc(n)^{-1} = \frac{1}{j}\sum_{k=1}^{j} \frac{1}{inc(\pi_k)}$, with which Expression 2 becomes $\bar{D}(n) = (U(n) - L(n)) \cdot inc(n)^{-1}$, namely an expression for $\bar{D}(n)$ as a ratio of the distance between the initial upper and lower bounds and $inc(n)$. Note that in post-solution analysis $\bar{D}(n)$ is known and $inc(n)$ can be computed directly, without considering each π_j.

One more aspect that has been ignored in the analysis so far, but which is likely to have an impact, is the actual height $h(n)$ of the subproblem pseudo tree. We therefore propose to scale $\bar{D}(n)$ by a factor of the form $h(n)^\alpha$; in our experiments we found $\alpha = 0.5$ to yield good results[b]. The general expression we obtain is thus:

$$\frac{\bar{D}(n)}{h(n)^\alpha} = \frac{U(n) - L(n)}{inc(n)} \tag{3}$$

Predicting $\bar{D}(n)$ for New Subproblem $P(n)$: Given previously solved subproblems $P(n_1), \ldots, P(n_r)$, we need to estimate $inc(n)$ in order to predict $\bar{D}(n)$. Namely, we compute $inc(n_i) = (U(n_i) - L(n_i)) \cdot h(n_i)^\alpha \cdot \bar{D}(n_i)^{-1}$ for $1 \leq i \leq r$. Assuming again that $inc(n)$ is a random variable distributed normally we take the sample average to estimate $inc^* = \frac{1}{r}\sum_{i=1}^{r} inc(n_i)$.

[b]Eventually α could be subject to learning as well.

Using Equation 3, our prediction for $\bar{D}(n)$ is:

$$\bar{D}^*(n) = \frac{(U(n) - L(n)) \cdot h(n)^\alpha}{inc^*} \tag{4}$$

Predicting $N(n)$ for a New Subproblem $P(n)$: Given the estimates b^* and inc^* as derived above, we will predict the number of nodes $N(n)$ generated within $P(n)$ as:

$$N^*(n) = b^{*\,\bar{D}^*(n)} \tag{5}$$

The assumption that inc and b are constant across subproblems is clearly too strict, more complex dependencies will be investigated in the future. For now, however, even this basic approach has proven to yield good results, as we will demonstrate in Section 5.

4.4. *Parameter Initialization*

To find an initial estimate of both the effective branching factor as well as the average increment, the master process performs 15 seconds of sequential search. It keeps track of the largest subproblem $P(n_0)$ solved within that time limit and extracts $b(n_o)$ as well as $inc(n_0)$, which will then be used as initial estimates for the first set of cutoff decisions. Additionally, we perform a 60 second run of stochastic local search,[9] which returns a solution that is not necessarily optimal, but in practice usually close to it. This provides an initial lower bound for subproblem estimation and pruning.

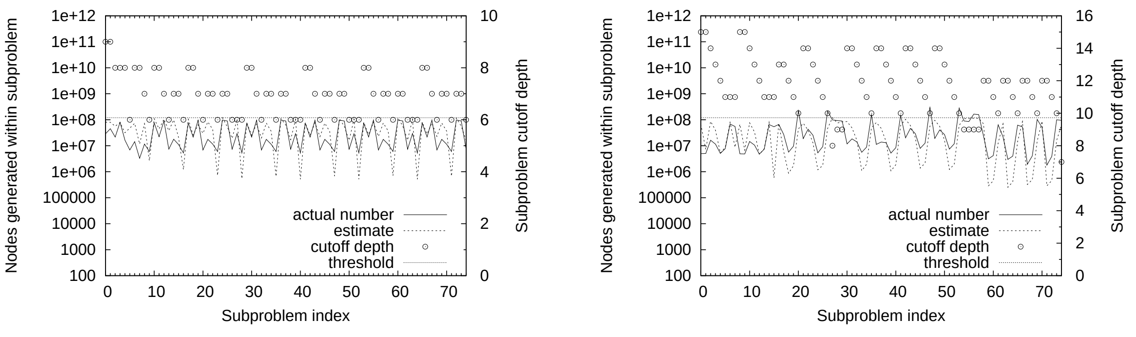
5. Experiments

We conducted experiments with our parallel AOBB scheme using the above prediction scheme to make the cutoff decision fully automatically. The cutoff threshold was set to $T = 12 \cdot 10^8$, which corresponds to roughly 20 minutes of processing time and was deemed to be a good compromise between subproblem granularity and parallelization overhead.

Overall solution times are given in Table 1. n, k, and w denote the number of variables, max. domain size, and induced width of the problem's Bayesian network. For reference we include the sequential solution time seq and the time par_{fix} of the best-performing parallel run with fixed cutoff depth from previous work.[15] seq/sls is then the time of the sequential scheme prefaced by 60 seconds

Table 1: Results of the automated parallel scheme (ped: 15 workers, mm: 10 workers).

instance	n	k	w	seq	par_{fix}	seq/sls	par^*/sls
ped7 (25/20)	1068	4	32	19,114	3,352	19,369	**2,843**
ped13 (20/20)	1077	3	32	2,752	**379**	2,856	419
ped19 (15/20)	793	5	25	*time*	27,372	*time*	**10,671**
ped31 (25/20)	1183	5	30	77,580	15,230	37,904	**3,970**
ped41 (25/20)	1062	5	33	14,643	**2,173**	14,059	2,311
ped51 (25/20)	1152	5	39	*time*	65,818	*time*	**59,975**
mm3.8.5-11	3612	2	37	9,715	1,443	3,003	**1,145**
mm3.8.5-12	3612	2	37	7,568	**1,430**	2,090	1,644
mm6.8.3-00	1814	2	31	12,595	1,797	319	**288**
mm10.8.3-11	2558	2	47	84,920	10,044	39,821	**6,906**
mm10.8.3-12	2558	2	47	5,630	1,357	2,549	**814**
mm10.8.3-13	2558	2	46	10,385	2,413	5,397	**2,208**



(a) ped31 (b) ped51

Fig. 4: Subproblem statistics for the first 75 subproblem of ped31 and ped51.

of stochastic local search providing an initial lower bound. Finally, column par^*/sls contains the overall solution time of the automated parallel scheme (similary including SLS preprocessing).

Pedigree Networks : The first set of problems consists of some very hard pedigree networks, encoded as Bayesian networks as described in Section 2.1, with the number of individuals and loci, respectively, given after the instance name in Table 1. We can see that in all cases the automatic scheme does at least as good as the best fixed cutoff, in some cases even better. Again it is important to realize that par_{fix} in Table 1 is the result of trying various fixed cutoff depths and selecting the best one, whereas par^*/sls requires no such "trial and error". In case of pedigree31 the SLS initialization is quite effective for the sequential algorithm, cutting computation from 21 to approx. 10 hours – yet the automated scheme improved upon this by a factor of almost 10, to just above one hour. Furthermore, for ped51 and in particular ped19, both of which could not be solved sequentially, par^*/sls marks a good improvement over par_{fix} .

Mastermind Networks : While not as practically relevant, these hard problems encoding board game states can provide further insight into the parallel performance. Here we find that for most problems the automated scheme performs at least as well as the best fixed cutoff (determined after trying various depths); in general, however, we believe that the overall problem complexity is too close to the subproblem threshold, inhibiting better parallel performance.

5.1. *Subproblem Statistics*

Figures 4(a) and (b) contain detailed subproblem statistics for the first 75 subproblems generated by the automated parallelization scheme on ped31 and ped51, respectively. Each plot shows actual and predicted number of nodes as well as the (constant) threshold that was used in the parallelization decision. The cutoff depth of the subproblem root is depicted against a separate scale to the right.

As expected, the scheme does not give perfect predictions, but it reliably captures the trend. Furthermore, the actual subproblem complexities are all contained within an interval of roughly one order of magnitude, which is significantly more balanced than the results for fixed cutoff depth.[15] We also note that "perfect" load balancing is impossible to obtain in practice, because subproblem complexity can vary greatly from one depth level to the next along a single path. In particular, if a subproblem at depth d is deemed too complex, most of this complexity might stem from only one of its child subproblems at depth $d{+}1$, with the remaining ones relatively simple – yet solved separately. In light of this, we consider the above results very promising.

5.2. *Performance Scaling*

At this time we only have a limited set of computational resources at our disposal, yet we wanted to perform a preliminary evaluation of how the system scales with p, the number of workers. We hence ran the automated parallel scheme with $p \in \{5, 10, 15, 20\}$ workers and recorded the overall solution time in each case.

Figure 5 plots the relative overall speedup in relation to $p = 5$ workers. For nearly all instances the behavior is as expected, at times improving linearly with the number of workers, although not always at a 1:1 ratio. It is evident that relatively complex problem instances profit more from more resources; in particular ped51 sees a two-, three-, and fourfold improvement going to twice, thrice, and four times the number of workers, respectively. For simpler instances, we think the subproblem threshold of approx. 20 minutes is too close to the overall problem complexity, thereby inhibiting better scaling.

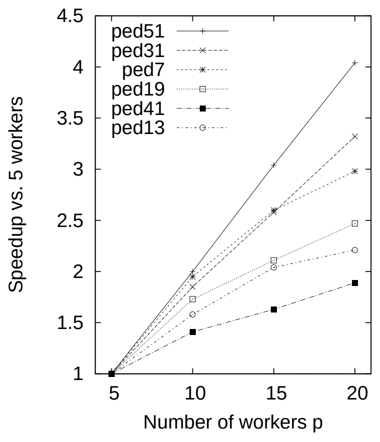


Fig. 5: Performance relative to $p = 5$ workers.

6. Conclusion & Future Work

This paper presents a new framework for parallelization of AND/OR Branch and Bound (AOBB), a state-of-the-art optimization algorithm over graphical models, with applications to haplotyping for general pedigrees. In extending the known idea of parallel tree search to AOBB, we show that generating independent subproblems can itself be done through an AOBB procedure, where previous subproblem solutions are dynamically used as bounds for pruning new subproblems.

The underlying parallel framework is very general and makes minimal assumptions about the available parallel infrastructure, making this approach viable on many different parallel and distributed resource pools (e.g., a set of networked desktop computers in our case).

Experiments have shown that the central requirement for good performance lies in effective load balancing. We have therefore derived an expression that captures subproblem complexity using an exponential functional form using three subproblem parameters, including the cost function. We then proposed a scheme for learning this function's free parameters from previously solved subproblems. We have demonstrated empirically the effectiveness of the estimates, leading to far better workload balancing and improved solution times when computing the most likely haplotypes on a number of hard pedigree instances.

We acknowledge that this initial estimation scheme, while justified and effective, still includes some ad hoc aspects. We aim to advance the scheme by taking into account additional parameters and by providing firm theoretical grounds for our approach. Besides extending the scheme itself, future work will also more thoroughly investigate the issue of parallel scaling, using larger grid setups than what we had access to so far (or performing simulations to that effect).

Furthermore, we plan to conduct more experiments on larger and harder problems from the haplotyping domain. In that context we are currently also working on a more in-depth analysis relating the size and structure of the pedigree and the number of loci in the problem to our scheme's performance. And while some problems may remain out of reach due to their inherent complexity, we do believe that our scheme will scale to many instances of interest; our confidence is in part based on the results obtained with the Superlink Online system,[16] which exploits a very similar strategy in the context of linkage analysis tasks and has proven very successful.

Finally, we note that in practice a small loss in accuracy can often be tolerated if it leads to significant time savings or better scaling. To that end, we intend to extend our current exact inference scheme to approximate reasoning; in particular, our parallel implementation should adapt very well to the concept of anytime search.

Acknowledgements

This work was partially supported by NSF grant IIS-0713118 and NIH grant 5R01HG004175-03.

References

1. Kento Aida, Wataru Natsume, and Yoshiaki Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *CCGRID*, pages 156–163, 2003.
2. David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.
3. Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241, Lisbon, Portugal, September 2009. Springer-Verlag.
4. Gérard Cornuéjols, Miroslav Karamanov, and Yanjun Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.
5. Rina Dechter and Robert Mateescu. AND/OR search spaces for graphical models. *Artif. Intell.*, 171(2-3):73–106, 2007.
6. Maáyan Fishelson, Nickolay Dovgolevsky, and Dan Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 59:41–60, 2005.
7. Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.
8. Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. Knowl. Data Eng.*, 11(1):28–35, 1999.
9. Frank Hutter, Holger H. Hoos, and Thomas Stützle. Efficient stochastic local search for MPE solving. In *IJCAI*, pages 169–174, 2005.
10. Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI*, pages 1014–1019. AAAI Press, 2006.
11. Uffe Kjaerulff. Triangulation of graphs – algorithms giving small total state space. Technical report, Aalborg University, 1990.
12. Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.
13. Radu Marinescu and Rina Dechter. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1457–1491, 2009.
14. Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.
15. Lars Otten and Rina Dechter. Towards parallel search for optimization in graphical models. In *ISAIM*, 2010.
16. Mark Silberstein, Anna Tzemach, Nickolay Dovgolevsky, Maáyan Fishelson, Assaf Schuster, and Dan Geiger. Online system for faster multipoint linkage analysis via parallel execution on thousands of personal computers. *American Journal of Human Genetics*, 78(6):922–935, 2006.
17. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.