Towards Parallel Search for Optimization in Graphical Models

Lars Otten and Rina Dechter
Bren School of Information and Computer Sciences
University of California, Irvine
{lotten,dechter}@ics.uci.edu

Abstract

We introduce a strategy for parallelizing a state-of-the-art sequential search algorithm for optimization on a grid of computers. Based on the AND/OR graph search framework, the procedure exploits the structure of the underlying problem graph. Worker nodes concurrently solve subproblems that are generated by a single master process. Subproblem generation is itself embedded into an AND/OR Branch and Bound algorithm and dynamically takes previous subproblem solutions into account. Drawing upon the underlying graph structure, we provide some theoretical analysis of the parallelization parameters. A prototype has been implemented and we present promising initial experimental results on genetic haplotyping and Mastermind problem instances, at the same time outlining several open questions.

1 Introduction

The practical relevance of graphical models like constraint satisfaction problems or Bayesian networks has evolved greatly in the past and is still increasing. The concepts have been successfully applied to areas as diverse as digital circuit design and verification, earth observation satellite scheduling, medical diagnosis, and human genetic analysis. This has driven research in the area and various algorithmic improvements have been made over the years.

Because these problems are typically NP-hard, however, many relevant problem instances remain infeasible, and even harder ones are continually introduced. With today's 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.

This paper puts optimization for general graphical models into this parallelization context. We will specifically focus on AND/OR Branch and Bound depth-first search, a state-of-the-art algorithm which exploits independencies within the search space and caches identical subproblems (Marinescu & Dechter 2009). While such schemes were demonstrated to solve harder and larger problems than ever before, they may have reached their limits within a single-processor framework due to their memory restrictions and the inherent exponential complexity of these tasks.

The idea of parallelized Branch and Bound is not new (see for instance (Gendron & Crainic 1994)). The novelty in our work is in investigating parallelism for Bayesian networks and weighted constraint problems, thus taking into consideration the structural properties of the underlying graph.

In this paper, we show how problem decomposition specific to graphical models, using the concept of AND/OR search spaces, can be exploited for parallel processing; we argue how the process of generating subproblems for parallelization can itself be accomplished through an AND/OR Branch and Bound procedure, where subproblem solutions are used as bounds for pruning. On the other hand, we demonstrate how the notion of caching (to avoid redundant computations) is compromised, since the exchange of information across subproblems is not possible in the assumed grid computation framework. We develop a metric to quantify these redundancies, based on the underlying graph structure of a given problem.

Subsequently, we provide results of our initial empirical evaluation on genetic haplotyping and Mastermind problems, which are promising and show that our parallel scheme, although still in its early stages, can greatly improve solution times when applied to complex problems. In conjunction with our theoretic analysis, however, the results also point to a number of issues and open problems which will be the subject of future work.

The paper is organized as follows: Section 2 provides necessary definitions and concepts, while in Section 3 we detail our parallelized AND/OR Branch and Bound scheme. Section 4 analytically investigates redundancy issues and provides metrics for the parallel search space as a function of the parallelization frontier. Section 5 presents empirical results and analysis, before Section 6 concludes.

2 Background

We assume the following basic definitions:

DEFINITION **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, a *weighted constraint satisfaction* problem is typically expressed through a set of cost functions over the variables, with the goal of finding the minimum of the sum over these costs. In the area of probabilistic reasoning, the *most probable explanation* task over a Bayesian network is defined as maximizing the product of the probabilities.

DEFINITION **2 (primal graph)** *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.*

Figure 1(a) depicts the primal graph of an example problem with six variables, A, B, C, D, E, F.

DEFINITION **3 (induced graph, induced width)** *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 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.*

The induced graph for the example problem along ordering $d = A, B, C, D, E, F$ is depicted in Figure 1(b), with two new induced edges, (B, C) and (B, E). 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* are used to obtain approximations.

2.1 AND/OR Search Spaces

The concept of AND/OR search spaces has recently been introduced as a unifying framework for advanced algorithmic schemes for graphical models to better capture the structure of the underlying graphical model (Dechter & Mateescu 2007). 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 **4 (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.*

See (Dechter & Mateescu 2007; Marinescu & Dechter 2009) for details on how to generate pseudo trees.

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 of the graphical model. 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 (Dechter & Mateescu 2007).

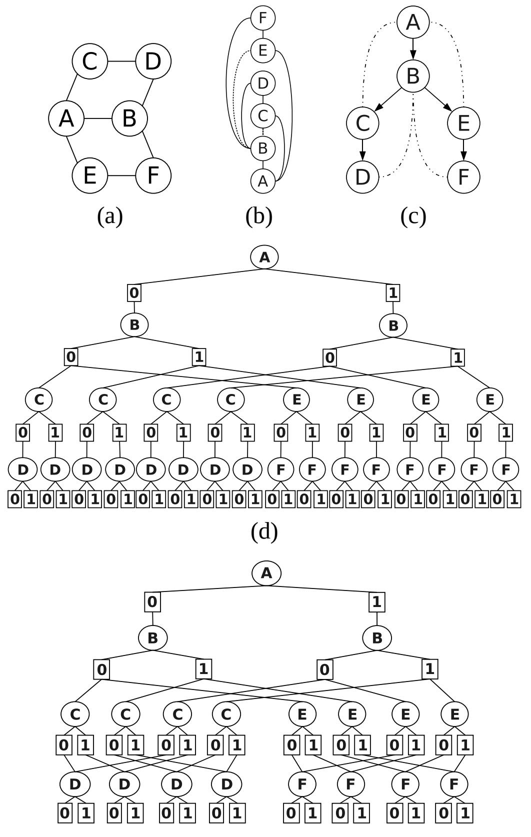

Figure 1: Example primal graph with six variables, its induced graph along ordering $d = A, B, C, D, E, F$, a corresponding pseudo tree, the resulting AND/OR search tree, and the context-minimal AND/OR search graph.

AND/OR Search Graphs. In an AND/OR search tree, different nodes may root identical subproblems. These nodes can be merged, yielding an *AND/OR search graph* of smaller size, at the expense of using additional memory during search. Some mergeable nodes can be identified by their *contexts*:

DEFINITION **5 (context)** *Given the pseudo tree $\mathcal{T}$ of an AND/OR search space, the context of an OR node X_i is the set of ancestors of X_i in $\mathcal{T}$, that are connected in the primal graph to X_i or its descendants (in $\mathcal{T}$).*

The context of X_i separates the subproblem below X_i from the rest of the network. Merging all context-mergeable nodes in the AND/OR search tree yields the *context minimal* AND/OR search graph (Dechter & Mateescu 2007).

PROPOSITION **6** *(Dechter & Mateescu 2007) Given a graphical model instance, 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 again bounds the domain size.*

EXAMPLE **7** *Figure 1(c) depicts a pseudo-tree extracted from the induced graph in Figure 1(b) and Figure 1(d) shows the corresponding AND/OR search tree. Merging nodes based on their context yields the context-minimal AND/OR search graph in Figure 1(e). 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 (Dechter & Mateescu 2007).

2.2 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.

Each node n in the search graph has an associated heuristic value $h(n)$ that underestimates $v(n)$. The algorithm can also compute an upper bound on $v(n)$ based on the portion of the search space below n that has already been solved: the upper bound $ub(n)$ on $v(n)$ is the current minimal cost of the solution subtree rooted at n. Upon expanding a node n in the search space, if for any of its ancestors m along the path to the root we have $h(m) > ub(m)$, the algorithm can safely prune n .

After expanding a node n, AND/OR Branch and Bound recursively moves upwards in the search space, from n towards the root, updating node values along the way: OR nodes revise their value by minimization over the values of their AND children, while AND nodes employ the combination operator of the graphical model – in a belief network, for instance, the child OR node values correspond to conditional probabilities and are multiplied.

Note that the heuristic estimate $h(n)$ can also guide the exploration process and, for instance, suggest node orderings for the children of an OR node, thereby expanding the more "promising" nodes first.

Mini-Bucket Heuristics. The primary heuristic $h(n)$ that we use in our experiments is the Mini-Bucket heuristic. It is based on Mini-Bucket elimination, which is an approximate variant of a variable elimination scheme and computes approximations to reasoning problems over graphical models (Dechter & Rish 2003). It was shown that the intermediate functions generated by the Mini-Bucket algorithm MBE(i) can be used to derive a heuristic function that underestimates the minimal cost solution to a subproblem in the AND/OR search graph (Marinescu & Dechter 2009).

3 Parallelizing AND/OR Search

We will now describe our scheme for parallelizing AND/OR branch and bound search to optimally solve a graphical model reasoning problem. The parallel architecture we assume is very simple and rather restrictive. We operate on a *computational grid*, a group of independent computer systems that are connected over some network. The computing nodes are assumed to be organized in a *master-worker* hierarchy, where the *master* node runs a central process which coordinates the *workers*. It is further assumed that the worker nodes cannot communicate with each other, which is often the case in practical grid environments.

DEFINITION **8 (start pseudo tree, parallelization frontier)** *Given an undirected graph $G = (X, E)$, a directed rooted tree $\mathcal{T}_c = (X_c, E_c)$, where $X_c \subset X$, is a* start pseudo tree *if it has the same root as, and is a subgraph of some pseudo tree of G. Given a start pseudo tree T_c, we refer to the set of variables corresponding to the leaf nodes of T_c as the* parallelization frontier.

Given a reasoning problem over a graphical model instance and a start pseudo tree $\mathcal{T}_c$, a straightforward approach to parallelize the search process is as follows:

- The master process begins exploring its master search space through AND/OR graph search guided by the start pseudo tree $\mathcal{T}_c$. It stops at AND nodes that correspond to assignments to variables in the parallelization frontier.
- Each OR child of these terminal AND nodes represents a conditioned subproblem, which the master process submits to a worker for concurrent solving.
- Worker nodes solve their assigned subproblems in parallel by sequential AND/OR Branch and Bound graph search and transmit the optimal subproblem solution back to the master process.
- The master collects the solutions from the worker nodes and combines them to obtain the overall solution.

EXAMPLE **9** *Consider again the AND/OR search graph in Figure 1(e). If we pick the simple start pseudo tree where B is the child of A, we can illustrate the parallel scheme through Figure 2: The search space of the master process is marked in gray and comprises the OR and AND nodes*

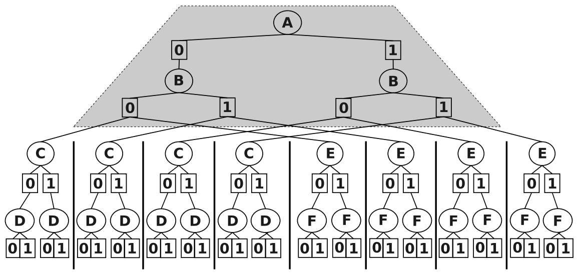

Figure 2: Schematic application of the parallelization scheme to the example problem from Figure 1. We pick a start pseudo tree where B is the child of A, which yields the master search space marked in gray and eight independent subproblems.

for variables A and B. Each of the OR nodes for variables C and E represents one of eight independent subproblems that will be solved by worker nodes. Note that the parallelization introduces some redundancy in the subproblems that was eliminated by caching in the unconditioned search space from Figure 1(e) – namely, the number of nodes for D and F has doubled.

It is worth pointing out that this approach is closely related to AND/OR cutset conditioning (Mateescu & Dechter 2005), which improves upon classical cutset conditioning (Pearl 1988; Dechter 1990) by exploiting the same notion of a start pseudo tree.

For our implementation we assume that the number of workers on the grid is a constant p and pass this number as a parameter to the master process, which will initially generate just the first p subproblems. Further ones are only generated when workers finish computing and become available again. As noted, the worker nodes execute AOBB(i), which stands for sequential AND/OR Branch and Bound graph search with a Mini-Bucket heuristic, augmented to allow conditioning on specific variables (to define a subproblem).

3.1 Master Process Details

The master process performs AND/OR Branch and Bound on the master search space along the start pseudo tree T_c. In particular, every time the master resumes its search procedure to generate the next subproblem, previously received subproblem solutions will be used as bounds in order to prune parts of the master search space. Furthermore, the heuristic value of nodes can be used to define a hierarchy over the respective subproblems and process the more "promising" ones first.

An outline of the master process is given in the pseudo code of Algorithms 1 and 2. As a Branch and Bound algorithm it implements node expansion and propagation, which have been laid out as two concurrent subprocedures ("threads").

Exploration Thread. The main thread in Algorithm 1 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 function $h(n)$ to make pruning decisions (line 9), where the computation of the upper bound $ub(m)$ takes into account subproblem solutions received from worker nodes so far. Exploration is halted when an OR node corresponding to a leaf of T_c is expanded. The master then submits the respective subproblem, given by the subproblem root variable and its context instantiation, to be solved by a worker node.

Propagation Thread. An auxiliary, concurrent thread inside the master process collects and processes subproblem solutions from the worker nodes (Algorithm 2). Upon receipt of a solved subproblem, in lines 3-5 the subproblem solution is assigned as the value of the respective node of the master search space (it will thereby be used for future pruning decisions by the exploration thread). Then the information is recursively propagated upwards in the search space towards the root (lines 6-10), updating node values as de-

Algorithm 1 Main exploration thread in master process

```
Parameters: Reasoning problem $\mathcal{P}$, start pseudo tree $\mathcal{T}_c$ with root
    $X_0$, max. number of workers $p$
Output: optimal solution to $\mathcal{P}$
 1: stack $\Leftarrow \{X_0\}$ // root OR node
 2: workers $\Leftarrow 0$ // shared counter
 3: fork auxiliary propagation thread // see Algorithm 2
 4: while ( stack $\neq \emptyset$ )
 5:   if ( workers $== p$ )
 6:     wait(workers$< p$)
 7:   $n \Leftarrow$ stack.pop()
 8:   for ( $m$ in ancestors($n$) ) // go from $n$ to root
 9:     if ( $h(m) > ub(m)$ )
10:       $v(n) \Leftarrow \infty$ // dead end, prune
11:       goto line 4
12:   if ( $n$ is AND node, denote $n = \langle X_i, a \rangle$ )
13:     for ( $X_j$ in $\mathrm{children}_{T_c}(X_i)$ ) // pseudo tree children
14:       stack.push($X_j$) // child OR nodes
15:   else // $n$ is OR node, denote $n = X_i$
16:     if ( $X_i$ is leaf node in $T_c$ )
17:       submit subproblem under $X_i$ to grid
18:       workers $\Leftarrow$ workers$+1$ // increase shared counter
19:     else // $X_i$ is not leaf node in $T_c$
20:       for ( $x_i$ in $domain(X_i)$ )
21:         stack.push($\langle X_i, x_i \rangle$) // child AND nodes
22: wait(workers$== 0$)
23: terminate auxiliary propagation thread
24: return $v(X_0)$ // final value of root node is solution
```

Algorithm 2 Propagation thread in master process

```
 1: while ( true ) // will be terminated externally
 2:   wait for next subproblem solution from a worker node
 3:   $c' \Leftarrow$ optimal solution received from worker
 4:   $n \Leftarrow$ root node for subproblem in master search space
 5:   $v(n) \Leftarrow c'$ // store optimal solution
 6:   for ( $m$ in ancestors($n$) ) // go from $n$ to root
 7:     if ( $m$ is OR node $X_i$ )
 8:       $v(m) \Leftarrow \min\{v(x_i) + l(X_i, x_i) | x_i \in \mathrm{children}(m)\}$
 9:     else // $m$ is AND node $\langle X_i, x_i \rangle$
10:       $v(m) \Leftarrow combine\{v(X_j) \mid X_j \in \mathrm{children}(m)\}$
11:   workers $\Leftarrow$ workers$-1$ // decrease shared counter
```

scribed previously, identical to sequential AND/OR Branch and Bound: OR nodes minimize over their AND childrens' values, while AND nodes combine the values of their OR children by the graphical model combination operator (for instance, multiplication in Bayesian networks).

Since these two threads concurrently access the master search space, proper synchronization and locking mechanisms have to be put in place. Furthermore, access to the shared variable *workers* needs to enclosed in atomic blocks of operations. Details have been omitted from the pseudo code for reasons of clarity.

4 Choosing the Parallelization Frontier

A central issue is how to pick the start pseudo tree T_c (and thereby the parallelization frontier), which in Algorithm 1 is provided as input. This will determine the search space within the master as well as the number, size, and complexity of subproblems assigned to the workers. If one picks the start pseudo tree too small, it will result in only few subproblems, which might still be very complex and yield low parallel granularity. On the other hand, if the start pseudo tree is too large, many small and easy subproblems will be generated, which will entail a significant overhead in terms of relatively expensive grid communication.

We experimented with using the induced width of each subproblem as a parameter to characterize and predict the subproblem complexity: nodes are added to the start pseudo tree as long as the subproblem induced width is above a set bound. In practice, however, the asymptotic nature of the induced width as a bound turned out to be not fine-grained enough: varying the induced width limit by one often implied a significant change in the depth of the parallelization frontier and in the practical subproblem complexity.

4.1 Quantifying Redundancy

To facilitate a more analytical approach, we observe that the choice of parallelization frontier can introduce varying degrees of redundancy, which we will try to capture by using graphical model properties. Specifically, the issue lies in identical subproblems, which would have been solved only once in non-parallelized AND/OR graph search (because of caching, cf. Examples 7 and 9).

We next develop expressions that characterize the size of the search space resulting from placing the parallelization frontier at a fixed depth d. Even though a more flexible choice of a frontier with varying depth could be beneficial in practice, our hope is that we can shed light on various aspects impacting the effectiveness of parallelization.

In the following we assume a pseudo tree $\mathcal{T}$ for a graphical model with n nodes corresponding to variables $X = \{X_1, \dots, X_n\}$, each of which has 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 $\mathcal{T}$. $L_j := \{X_i \in X \,|\, h(X_i) = j\}$ is the set of variables at depth j in $\mathcal{T}$. For every variable X_i, 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 $\mathcal{T}$.

By $\mathrm{context}(X_i)$ we denote the set of variables in the context of X_i with respect to $\mathcal{T}$ (see Definition 5), and as before $w(X_i) := |\mathrm{context}(X_i)|$ is the *width* of X_i.

DEFINITION **10 (conditioned context, conditioned width)** *Given a node X_i and $j < h(X_i)$, $\mathrm{context}_j(X_i)$ denotes the* conditioned context of X_i *when placing the parallelization frontier at level j, namely,* $\mathrm{context}_j(X_i) := \mathrm{context}(X_i) \setminus \Pi(\pi_j(X_i)) = \{X' \in \mathrm{context}(X_i) \,|\, h(X') \geq j\}$. *The* conditioned width *of variable X_i is then defined as:*

$$w_j(X_i) := |\mathrm{context}_j(X_i)| \quad \textit{for } j < h(X_i)$$

The number of AND nodes in the AND/OR context minimal graph is then bounded by:

$$S_{AO} = \sum_{i=1}^{n} k^{w(X_i)+1} = \sum_{j=0}^{h} \sum_{X' \in L_j} k^{w(X')+1} \qquad (1)$$

This is because, by virtue of subproblem caching, each variable x cannot contribute more AND nodes to the context-minimal search space than the number of assignments to its context (times its own domain size), which is $k^{w(x)+1}$.

Again we assume that fix the parallelization frontier at depth d, i.e., condition up to depth d. Since caching is not possible across subproblems that are solved independently, we develop an expression that quantifies the redundancy introduced and gives the overall size of the conditioned search space.

THEOREM **11** *With the parallelization frontier at depth d, the overall number of AND nodes in the master search space and all conditioned subproblems is bounded by:*

$$S_{AO}(d) = \sum_{j=0}^{d} \sum_{X' \in L_j} k^{w(X')+1} + \sum_{j=d+1}^{h} \sum_{X' \in L_j} k^{w(\pi_d(X')) + w_d(X') + 1} \qquad (2)$$

Proof. The elements of the first sum over $j = 0, \dots, d$ remain unchanged from Expression 1, since levels 0 through d are still subject to full caching. We then note that the variables in L_d are those rooting the subproblems that will be solved by the worker nodes. 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 subproblems rooted at $\hat{X}$ will be generated. For a variable X' that is a child of $\hat{X}$ in 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 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 Expression 2. □

Observe that $S_{AO}(0) = S_{AO}(h) = S_{AO}$. For $d = 0$ the entire problem is given to a single worker node, while $d = h$ implies we solve the problem centrally in the master node.

EXAMPLE **12** *Going back to Example 9, consider variable D. We have $h(D) = 3$ and $\text{context}(D) = \{B, C\}$ with $w(D) = 2$. Since the cutoff level is $d = 2$, we need to consider $\pi_2(D) = C$ and $\text{context}(C) = \{A, B\}$, which yields $\text{context}_2(D) = \{C\}$ with conditioned width $w_2(D) = 1$. Given domain size $k = 2$, the overall contribution of AND nodes from variable D in the conditioned search space is therefore $k^{w(\pi_2(D))+w_2(D)+1} = 2^{2+1+1} = 16$. Iterating over all variables yields $S_{AO}(2) = 54$, matching Figure 2. Contrasting this with $S_{AO} = 38$ (cf. Figure 1(e)) highlights the redundancy introduced in this case.*

Parallel Search Space. $S_{AO}(d)$ does not account for any parallelism in solving the subproblems; rather, we need a notion of *parallel search space size*, where nodes that are processed in parallel are only counted once. An approximation of this can be obtained by simply dividing the second summand of $S_{AO}(d)$ by the number of worker nodes p :

COROLLARY **13** *The number of AND nodes in the parallel search space explored by p workers after parallelizing at depth d is:*

$$PS^{p}_{AO}(d) = \sum_{j=0}^{d} \sum_{X' \in L_j} k^{w(X')+1} + \frac{1}{p} \sum_{j=d+1}^{h} \sum_{X' \in L_j} k^{w(\pi_d(X'))+w_d(X')+1} \qquad (3)$$

As a function of d, $PS^{p}_{AO}(d)$ can guide the parallelization strategy. This approach, however, turns out to be not powerful enough, as evident from the results in the next section.

5 Implementation & Results

We implemented the parallel scheme in C++ on top of the *Condor* grid workload distribution software package (Thain, Tannenbaum, & Livny 2005), which we use on a dedicated grid via the "vanilla" universe, i.e., without checkpointing, rescheduling and other advanced Condor features. One submits a *job* to the grid by specifying an executable file and its input. Condor then selects a suitable worker, transfers the required files, and initiates execution. The Condor system also provides fault tolerance in case of node failures, for instance. Upon completion the results are automatically transmitted back to the submitter. Note that communication between workers or any concept of shared memory is not possible. For our experiments we used a dedicated Condor pool with up to 20 nodes (of which one was used by the master process). Each node has at least 3 GB of memory and varies in CPU speed from 2.33 to 3.0 GHz.

5.1 Empirical Evaluation

We ran experiments on a set of so-called pedigree networks from the area of human genetic analysis, specifically haplotyping problems[1]. These can be translated into a most probable explanation task (MPE) over a Bayesian network (cf. (Fishelson, Dovgolevsky, & Geiger 2005)) or, by moving to the log domain, a weighted CSP.

[1]available for download at http://graphmod.ics.uci.edu/

Table 1: Results on relatively simple pedigree instances. Sequential performance is compared against the parallel scheme with $p = 5$ workers and varying cutoff depth d.

							d=0	d=1	d=2	d=3	d=4	d=5	d=6	d=7	d=8
inst.	n	k	w	h	d^*	T_s	T_p	T_p	T_p	T_p	T_p	T_p	T_p	T_p	T_p
ped1	298	4	15	48	0	**1**	34	10	30	9	9	9	43	29	30
ped20	387	5	22	60	2	124	134	74	**54**	64	137	354	575	1,640	2,717
ped23	309	5	25	51	0	**3**	29	48	79	95	111	154	148	211	372
ped30	1015	5	21	108	0	209	219	220	179	**119**	132	189	272	530	1,039
ped33	581	4	28	98	2	73	82	89	74	**51**	71	73	94	233	415
ped37	726	5	21	56	0	**11**	41	39	98	85	69	151	149	436	1,297
ped38	581	5	17	69	26	681	702	709	657	627	456	**372**	394	554	768
ped39	953	5	21	76	0	121	134	109	**77**	108	108	325	551	1,586	3,160
ped50	478	6	17	47	1	**7**	39	54	95	115	115	177	339	643	1,089

Table 2: Results for parallelized AND/OR Branch and Bound on very hard pedigree instances. The parallel scheme was run with $p = 15$ workers and varying cutoff depth d. (Timeout 24 hours, not all combinations tested.)

							d=5	d=6	d=7	d=8	d=9	d=10	d=11
inst.	n	k	w	h	d^*	T_s	T_p	T_p	T_p	T_p	T_p	T_p	T_p
ped7	1068	4	32	90	0	19,114	8,281	4,827	**3,236**	3,868	5,437	4,883	7,800
ped13	1077	3	32	102	0	5,961	892	**743**	881	1,544	2,622		
ped19	793	5	25	98	1	*time*	34,900	**32,372**	33,280	45,720	53,955		
ped31	1183	5	30	85	1	34,247			28,799	19,919	10,917	**7,778**	13,268
ped41	1062	5	33	100	1	78,040	18,528	11,438	**8,693**	10,486	12,466	14,798	
ped51	1152	5	39	98	2	*time*					*time*	84,816	**60,879**

For each problem instance, we record the number of variables n, the maximum variable domain size k, and the induced width w. For the latter we computed 100 randomized minfill orderings and chose the one giving lowest induced width; h specifies the height of the corresponding pseudo tree. Solution time (in seconds) when solving the problem by sequential AND/OR Branch and Bound graph search on a single 3.0 GHz processor is denoted T_s . The value of d that minimizes the expression $PS^{p}_{AO}(d)$, as derived in Section 4, is given by d^* .

Easy Haplotyping Instances. In a first round of tests we looked at relatively easy instances (solvable in at most 10 minutes on a single processor), results are listed in Table 1. We allocated $p = 5$ workers and enforced a global cutoff depth ranging from $d = 0$ to $d = 8$. In each case we recorded the overall running time T_p .

As expected, for easy problems that take less than 60 seconds on a single computer (ped1, 23, 37, and 50), the overall solution time increases for the parallel scheme because delays and overhead originating in the grid environment easily dominate the actual computation in these cases.

On the other hand, for somewhat more complex problems like ped20, 30, and 38, we find that the improvement through parallelization can be noticeable. The solution time for ped38, for instance, keeps improving as we increase the cutoff depth from $d = 2$ to $d = 5$, and only then starts to suffer more heavily from the grid overhead (as subproblems become to small) and redundancies. With $d = 5$ the overall solution time for ped38 is roughly half that of the non-parallel algorithm, for ped20 and ped30 this is the case for $d = 2$ and $d = 3$, respectively.

Table 3: Results on Mastermind problem instances from the UAI'08 competition. The parallel scheme was run with $p = 10$ workers and varying cutoff depth d.

							d=5	d=6	d=7	d=8	d=9	d=10
instance	n	k	w	h	d^*	T_s	T_p	T_p	T_p	T_p	T_p	T_p
mm_03_08_05-0011	3612	2	37	89	23	9,715	1,753	**1,463**	1,592	1,668	2,692	4,706
mm_03_08_05-0012	3612	2	37	81	23	7,568	2,534	2,164	**1,487**	1,556	1,860	3,012
mm_04_08_04-0000	2616	2	37	79	3	10,620	1,575	1,362	**1,330**	1,384	1,607	2,332
mm_06_08_03-0000	1814	2	31	72	29	12,595	1,822	1,809	**1,789**	1,823	1,933	2,264
mm_10_08_03-0000	2606	2	47	99	4	26,102	4,909	4,449	4,581	**3,895**	4,317	4,373
mm_10_08_03-0011	2558	2	47	101	3	84,029	13,823	11,028	11,509	11,386	**10,935**	10,778
mm_10_08_03-0012	2558	2	47	82	4	5,630	2,536	1,357	1,358	**1,351**	1,880	1,489
mm_10_08_03-0013	2558	2	46	99	3	10,385	4,231	4,216	2,525	**2,466**	2,531	2,651

Complex Haplotyping Instances. Results on some very complex pedigree instances are shown in Table 2. For the parallel scheme we used $p = 15$ worker nodes. It is immediately obvious that these harder problems benefit more from the parallel approach, solution times improve for all cases that we tested: ped13 can be solved is less than 13 minutes for $d = 6$, whereas the sequential algorithm takes almost 1 hour and 40 minutes. ped41 takes close to 22 hours on a single computer, while the parallel scheme with $d = 7$ solves the problem in 2 hours and 23 minutes. Moreover, the parallel scheme was able to solve two instances, ped19 and ped51, on which the sequential algorithm timed out after 24 hours.

Complex Mastermind Instances. Lastly, we experimented on some hard Mastermind instances from the UAI'08 evaluation[2], which take from a few hours to almost a day on a single processor, denoted by T_s in Table 3. The distributed scheme was ran with $p = 10$ workers and varying cutoff d, the respective solution times are reported as T_p . Similar to the results on complex pedigree instances, the parallel scheme enabled significant improvements in overall running time. For example, with $d = 7$, mm_04_08_04-0000 went from three hours to little over 22 minutes (a factor of 8), while mm_10_08_03-0011 improved from more than 23 hours to about three hours (a factor of 7.6) for $d = 9$.

Parallelization Frontier. Tables 1, 2, and 3 also record d^*, the depth of the parallelization frontier for which the expression $PS^p_{AO}(d)$ is minimized given a particular problem instance. Unfortunately, this parameter estimate exhibits little correlation with the actual performance of the parallel scheme, it is often much too cautious and suggests a very low cutoff. We believe this is because a number of practical factors are not reflected in the metrics developed in Section 4, as we point out next.

Load Balancing. In order to obtain deeper insight into the performance of the parallel scheme, we recorded the complexity of the generated subproblems, measured via the number of nodes expanded by the worker node that processed it. Exemplary results for three Mastermind instances are shown in Figure 3, where subproblems are indexed by the time of their generation in the master process (only the first 50 are plotted). It should be noted that within each of

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

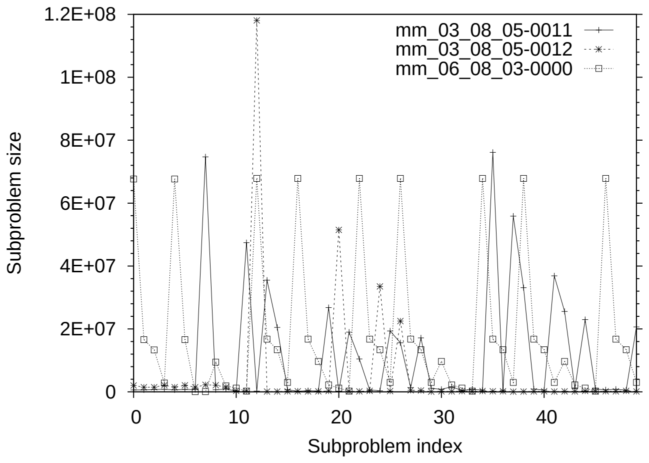

Figure 3: Subproblem size (in number of expanded nodes) for the first 50 subproblems, indexed by order of generation, of three select Mastermind problems ($p = 10, d = 7$).

the three problem instances, with the parallelization frontier at depth $d = 7$, the a priori bound on subproblem complexity implied by Theorem 11 is the same across all subproblems. Yet the three plots exhibit significant variance, going anywhere from a few thousand nodes (i.e., solvable in seconds) to many millions, with the worker taking 10 minutes or more. These findings hold for other Mastermind and haplotyping instances as well – in fact, in some cases the solution time of the most complex subproblem is almost identical to the overall running time of the parallel scheme.

The reason for this behaviour is twofold, we believe: Both in the haplotyping and Mastermind domain, problems possess a considerable degree of determinism (inconsistent tuples, zeros in conditional probability tables), which leads to early dead ends during search space exploration. Furthermore, the AND/OR Branch-and-Branch procedure will typically be able to prune significant portions of the search space by virtue of the heuristic function it is equipped with. The impact of both these phenomena is hard to predict in practice, and neither is reflected in our current metrics.

In the end, these results show that finding a good parallelization strategy automatically is still a challenging, yet interesting open question for future research.

6 Conclusion & Future Work

This paper presents a framework that allows for parallelization of AND/OR brand and bound graph search, a state-of-the-art optimization algorithm in the area of graphical models. The parallelization is performed on a computational grid of computers, where each system is connected to a common network but has its own CPU and memory.

Central to this new line of work in the context of graphical models parallelization is the idea that generating independent subproblems can itself be done through an AND/OR Branch and Bound procedure, where previous subproblem solutions are dynamically used as bounds for pruning.

Our preliminary results with a prototype system on genetic haplotyping and Mastermind problems are promising: on complex problem instances we have demonstrated greatly improved overall solution times, while the scheme's effectiveness for simple instances is not guaranteed and depends on the correct choice of parameters.

Indeed, the choice of parameters like cutoff depth, or more generally the parallelization frontier, remains a central issue. We have developed expressions that try to a priori quantify the level of redundancy introduced by the conditioning process versus potential gains from parallelization. But as we have found empirically, at this stage it does not seem to capture the underlying complexity of the problem, probably because it does not account for various other relevant aspects like determinism and the pruning power of AND/OR Branch and Bound.

Future work will therefore, besides performing a broader empirical evaluation, focus first and foremost on the development of more powerful parallelization strategies that can automatically determine good parameter values. We plan to extend and improve the above estimation scheme, for instance by accounting for determinism as in (Otten & Dechter 2008). We further intend to incorporate more of an online learning approach, where the master process monitors the actual complexity of subproblems (through the number of nodes expanded, for instance) and adjusts the size of subsequent subproblems accordingly. This will also facilitate deployment in many real-world, large-scale grid environments, where resources like running time are often limited in the grid nodes.

Acknowledgements

We thank the reviewers for their helpful feedback. This work was partially supported by NSF grant IIS-0713118 and NIH grant 5R01HG004175-03.

References

[1] Dechter, R., and Mateescu, R. 2007. AND/OR search spaces for graphical models. *Artificial Intelligence* 171(2-3):73–106.

[2] Dechter, R., and Rish, I. 2003. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM* 50(2):107–153.

[3] Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence* 41(3):273–312.

[4] Fishelson, M.; Dovgolevsky, N.; and Geiger, D. 2005. Maximum likelihood haplotyping for general pedigrees. *Human Heredity* 59:41–60.

[5] Gendron, B., and Crainic, T. G. 1994. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research* 42(6):1042–1066.

[6] Marinescu, R., and Dechter, R. 2009. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artificial Intelligence* 173(16-17):1457–1491.

[7] Mateescu, R., and Dechter, R. 2005. AND/OR cutset conditioning. In *19th International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK*, 230–235.

[8] Otten, L., and Dechter, R. 2008. Refined bounds for instance-based search complexity of counting and other #P problems. In *CP*, 576–581.

[9] Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

[10] Thain, D.; Tannenbaum, T.; and Livny, M. 2005. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience* 17(2-4):323–356.