Journal of Artificial Intelligence Research 59 (2017) 351–435 Submitted 1/17; published 7/17

AND/OR Branch-and-Bound on a Computational Grid

Lars Otten LOTTEN@UCI.EDU
Rina Dechter DECHTER@ICS.UCI.EDU
Department of Computer Science
University of California, Irvine
Irvine, CA 92697, U.S.A.

Abstract

We present a parallel AND/OR Branch-and-Bound scheme that uses the power of a computational grid to push the boundaries of feasibility for combinatorial optimization. Two variants of the scheme are described, one of which aims to use machine learning techniques for parallel load balancing. In-depth analysis identifies two inherent sources of parallel search space redundancies that, together with general parallel execution overhead, can impede parallelization and render the problem far from embarrassingly parallel. We conduct extensive empirical evaluation on hundreds of CPUs, the first of its kind, with overall positive results. In a significant number of cases parallel speedup is close to the theoretical maximum and we are able to solve many very complex problem instances orders of magnitude faster than before; yet analysis of certain results also serves to demonstrate the inherent limitations of the approach due to the aforementioned redundancies.

1. Introduction

Combinatorial optimization problems have many applications of practical significance, from computational biology and genetics to scheduling tasks and coding networks. One popular formalization is to express the problem as a Bayesian or Markov network, capturing dependencies and interactions between variables, and to pose the query for the most probable explanation (MPE). Solving this problem exactly is known to be NP-hard in general (Shimony, 1994). In practice, the limiting factor tends to be the induced width or tree width of a given problem instance, with many relevant problems rendered infeasible, and even harder ones introduced continually. Given 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 (see, for instance, Grama, Gupta, Karypis, & Kumar, 2003).

Here we put optimization problems over graphical models in this parallelization context, by describing a parallelized version of *AND/OR Branch-and-Bound* (AOBB), a state-of-the-art sequential algorithm for solving MPE or Weighted constraint problems (Dechter & Mateescu, 2007; Marinescu & Dechter, 2009b, 2009a) – AOBB placed first in the MPE category of the UAI 2012 Pascal Inference competition (Elidan, Globerson, & Heinemann, 2012) as well as second and first in the MPE and MMAP category, respectively, of the UAI 2014 Inference competition (Gogate, 2014).

In particular, we adapt and extend the established concept of parallel tree search (Kumar & Rao, 1987; Grama & Kumar, 1999; Grama et al., 2003), 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.

The distributed framework we target is a general grid computing environment, 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 2.2.2). This is admittedly restrictive compared to several other modern parallelism frameworks and inhibits a number of more advanced techniques developed in the field of distributed computing. Consequently the point of this paper is not to argue for the superiority of this framework. However, it facilitates maximum flexibility when it comes to deploying the proposed algorithm. Notably, we have integrated parallel AOBB into the Superlink-Online system, which is used by medical researchers worldwide to perform genetic linkage analysis. It incorporates volunteered, non-dedicated computing resources on many different networks that are geographically distributed, where no assumptions about speed, reliability, or even plain availability of communication channels can be made (Silberstein, Tzemach, Dovgolevsky, Fishelson, Schuster, & Geiger, 2006; Silberstein, 2011; Silberstein, Weissbrod, Otten, Tzemach, Anisenia, Shtark, Tuberg, Galfrin, Gannon, Shalata, Borochowitz, Dechter, Thompson, & Geiger, 2013).

The primary challenge in our work is therefore 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. In the context of optimization and AOBB, however, it is very hard to reliably predict and balance subproblem complexity: as this article will demonstrate, the usual structural bounds are wildly inadequate because of the algorithm's pruning power. In this work we apply a previously developed prediction scheme, in which offline regression analysis is applied to learn a complexity model from past problem instances (Otten & Dechter, 2012a), in order to detect and circumvent bottlenecks in subproblem runtime.

1.1 Contributions

We first give a brief overview of the landscape of parallel and distributed computing in Section 2 and put our approach in context. In Section 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 a complexity prediction scheme proposed by Otten and Dechter (2012a) in an attempt to balance subproblem complexity. To our knowledge, it is the first exploration and implementation of its kind, i.e. an exact optimization algorithm for general graphical models, running on a computational grid.

Section 3.5 provides in-depth algorithm analysis, including a number of examples. In particular, we illustrate different sources of overhead and repeated processing incurred as a consequence of the distributed execution. We also give a characterization of the parallelization frontier and investigate the question of its optimality, highlighting the importance of load balancing.

Related to that, in Section 4 we conduct a detailed investigation regarding the central concept of redundancies in the overall parallel search space explored by parallel AOBB. We identify and explain 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.

We then provide extensive empirical analysis of these theoretical results. Our comprehensive experimental evaluation in Section 5 is, to the best of our knowledge, the first of its kind and unparalleled in the context of general graphical models. 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 the latter in particular is shown to be far less pronounced than the theory in Section 4 suggests, in some cases almost irrelevant.

Accordingly, experimental results with respect to runtime and speedup are overall 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 some of the results with 500 CPUs hint at the limitations of the current implementation, at which point overhead and parallel search space redundancies, while still far from the theoretical worst case, become significant enough to meaningfully hinder performance. Overall, however, parallelization enables us to solve a number of hard problem instances within hours that were previously not practically feasible, taking many days or a few weeks to solve sequentially even when maxing out a single machine.

Section 6 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.

The full source code of our C++ implementation is accessible under an open-source GPL license at `http://github.com/lotten/daoopt`, which also has the problem instances used for evaluation available for download.

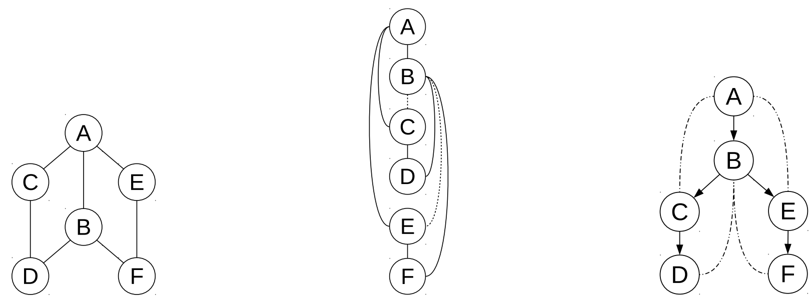
1.2 Article Outline

Section 2 gives an overview of distributed computing in general and parallel search implementations in particular. Section 3 then proposes our two specific implementations of parallel AND/OR Branch-and-Bound search and provides analysis. Section 4 specifically investigates and illustrates the issue of redundancies in the parallel search space. Section 5 begins by describing our experimental setup and benchmark problem instances in Sections 5.1 and 5.2. Sections 5.3 through 5.7 then perform extensive empirical evaluation and analysis, which is summarized in Section 5.8. Section 6 concludes.

2. Background & Related Work

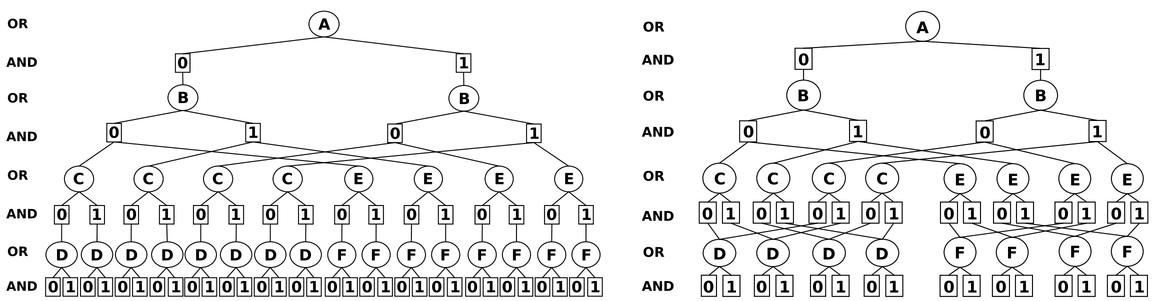
In this section we summarize relevant concepts and terminology, from AND/OR Branch-and-Bound to parallel and distributed computing, and put our work in context. We will also survey related work and delineate our approach against it.

We consider a MPE (most probable explanation, sometimes also called MAP, maximum *a posteriori* assignment) problem over a *graphical model*, defined by the tuple $(X, F, D, \max, \prod)$. $F = \{f_1, \ldots, f_r\}$ is a set of functions over variables $X = \{X_1, \ldots, X_n\}$ with discrete domains $D = \{D_1, \ldots, D_n\}$, we aim to compute $\max_X \prod_i f_i$, the probability of the most likely assignment. Another closely related combinatorial optimization problem is the *weighted constraint problem*, where we aim to minimize the sum of all costs, i.e. compute $\min_X \sum_i f_i$, though the problem specification can also include more explicit costraints like *alldifferent*(X_1, X_2, X_3), see for exam-



(a) Primal graph. (b) Induced graph. (c) Pseudo tree.

Figure 1: Example primal graph of a graphical model with six variables, its induced graph along ordering $d = A, B, C, D, E, F$, and a corresponding pseudo tree.



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

Figure 2: AND/OR search tree and context-minimal AND/OR search graph corresponding to the pseudo tree in Figure 1c.

ple the work of Wallace (1996). These tasks have many practical applications but are known to be NP-hard (Dechter, 2013).

The set of function scopes implies a primal graph and, given an ordering of the variables, an *induced graph* (where, from last to first, each node's earlier neighbors are connected) with a certain *induced width*, the maximum number of earlier neighbors over all nodes (Pearl, 1988; Kask, Dechter, Larrosa, & Dechter, 2005; Dechter, 2013).

EXAMPLE **1.** *Figure 1a depicts the (primal) graph of an example graphical model with six variables, A through F. The induced graph for the example problem along ordering $d = A, B, C, D,$ E, F is depicted in Figure 1b, with two new induced edges, (B, C) and (B, E). Its induced width is 2.*

Different orderings will vary in their induced width; finding an ordering of minimal induced width is known to be equally NP-hard. In practice heuristics like *min-fill* or *min-degree* have proven to produce reasonable approximations (Kjaerulff, 1990; Kask, Gelfand, Otten, & Dechter, 2011).

2.1 AND/OR Search Spaces and Branch-and-Bound

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 (Dechter & Mateescu, 2007). The search space is defined using a *pseudo tree* of the graph, which captures problem decomposition as follows:

DEFINITION **1.** *A* pseudo tree *of an undirected graph $G = (X, E)$ is a directed, rooted tree $\mathcal{T} = (X, E')$, such that every arc of G 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.*

EXAMPLE **2.** *A pseudo tree for the example in Figure 1a is shown in Figure 1c, corresponding to the induced graph in Figure 1b along ordering $d = A, B, C, D, E, F$. Note how B has two children, capturing the fact that the two subproblems over C, D and E, F, respectively, are independent once A and B have been instantiated.*

2.1.1 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 (Dechter & Mateescu, 2007). 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 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 **3.** *Figure 2a shows the AND/OR search tree resulting from the primal graph in Figure 1a when guided by the pseudo tree in Figure 1c. Note that the AND nodes for B have two children each, representing independent subtrees rooted at C and E, respectively, thereby capturing problem decomposition.*

In general, given a pseudo tree $\mathcal{T}$ of height h, the size of the AND/OR search tree based on $\mathcal{T}$ is $O(n \cdot k^h)$, where k bounds the domain size of variables (Dechter & Mateescu, 2007).

2.1.2 AND/OR SEARCH GRAPHS

Additional improvements in time complexity can be achieved by detecting and unifying redundant subproblems based on their *context*, the partial instantiation that separates the subproblem from the rest of the network.

DEFINITION **2** (OR context)**.** *(Dechter & Mateescu, 2007) 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.*

Identical subproblems, identified by their context (the partial instantiation that separates the subproblem from the rest of the network), can be merged, yielding an *AND/OR search graph* (Dechter & Mateescu, 2007). Merging all context-mergeable nodes yields the *context-minimal* AND/OR search graph. It was shown that the context-minimal AND/OR search graph has size $O(n \cdot k^{w^*})$, where w^* is the induced width of the problem graph along a depth-first traversal of $\mathcal{T}$ (Dechter & Mateescu, 2007).

EXAMPLE **4.** *Figure 2b displays the context-minimal AND/OR graph obtained when applying full caching to the AND/OR search tree in Figure 2a. In particular, 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 (namely, the assignment of A does not matter).*

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 $Sol_{S_{\mathcal{T}}}$.

2.1.3 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 . $v(n)$ can be computed recursively using the values of n's successors (Dechter & Mateescu, 2007).

2.1.4 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 (Marinescu & Dechter, 2009a, 2009b). 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 2.1.5). 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 (Marinescu & Dechter, 2009a).

Algorithm 1 shows pseudo code for AOBB on a high level: starting with just the root node $\langle X_0 \rangle$ on the stack, we iteratively take the top node n from the stack (line 3). 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 (details in Marinescu & Dechter, 2009b). 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). When the stack eventually becomes empty, the value of the root node $\langle X_0 \rangle$ is returned as the solution to the problem (18).

2.1.5 MINI-BUCKET HEURISTICS

The 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 variable elimination and computes approximations to reasoning problems over graphical models (Dechter & Rish, 2003). A control parameter i, referred to as i-bound, allows a trade-off between accuracy of the heuristic and its

Algorithm 1 AND/OR Branch-and-Bound (AOBB)

Given: Optimization problem $(X, D, F, \max, \prod)$ and pseudo tree $\mathcal{T}$ with root X_o, heuristic h.
Output: cost of optimal solution

OPEN $\leftarrow \{\langle X_0 \rangle\}$
while *OPEN* $\neq \emptyset$:
    $n \leftarrow$ top(*OPEN*) // top node from stack, depth-first
    if checkpruning(n, $h(n)$) $=$ true :
        prune(n) // perform pruning
    else if cachelookup(n) $\neq$ NULL :
        value(n) $\leftarrow$ cachelookup(n) // retrieve cached value
    else if $n = \langle X_i \rangle$ is OR node :
        for $x_j \in D_i$:
            create AND child $\langle X_i, x_j \rangle$
            add $\langle X_i, x_j \rangle$ to top of *OPEN*
    else if $n = \langle X_i, x_j \rangle$ is AND node :
        for $Y_r \in$ children$_{\mathcal{T}}(X_i)$:
            generate OR node $\langle Y_r \rangle$
            add $\langle Y_r \rangle$ to top of *OPEN*
    if children(n)$=\emptyset$: // n is leaf
        propagate(n) // upwards in search space
return value($\langle X_0 \rangle$) // root node has optimal solution

time and space requirements – higher values of i yield a more accurate heuristic but take more time and space to compute. It was shown that the intermediate functions generated by the mini-bucket algorithm MBE(i) can be used to derive a heuristic function that is *admissible*, namely in a maximization context it overestimates the optimal cost solution to a subproblem in the AND/OR search graph (Kask & Dechter, 2001). We also note recent work by Ihler, Flerova, Dechter, and Otten (2012) that demonstrated improvements to the MBE heuristic through join-graph cost shifting. This approach, however, has not been incorporated for the experimental evaluation in Section 5.

2.2 Parallel & Distributed Computing and Search

The notions of *parallel computing* and *distributed computing* have considerable overlap and there is no clear line to be drawn between them. 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* (Lynch, 1996; Grama et al., 2003; Ghosh, 2006).

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 (Foster & Kesselmann, 1998).

2.2.1 DATA & TASK PARALLELISM

Another angle of classification differentiates *data parallelism* versus *task* parallelism. Data parallelism describes an approach where concurrency is achieved by independently processing parts of a huge set of input data in parallel (often denoted "embarrassingly parallel"); examples include the SETI@home (Anderson, Cobb, Korpela, Lebofsky, & Werthimer, 2002) and Folding@home (Beberg, Ensign, Jayachandran, Khaliq, & Pande, 2009) projects as well as the general MapReduce framework (Dean & Ghemawat, 2004).

In task parallelism, on the other hand, the primary objective is to distribute a large amount of computations, typically stemming from an input problem whose specification size is small. A prime example of this challenge is the Superlink-Online system, which uses vast numbers of computers around the world to perform genetic linkage analysis on general pedigrees (Silberstein et al., 2006; Silberstein, 2011; Silberstein et al., 2013), which also serves as context and motivation for some of our own experiments in Section 5.

2.2.2 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 (Kumar & Rao, 1987; Grama et al., 2003).

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 (Grama et al., 2003). 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 (Lüling & Monien, 1992) and Traveling Salesman problems (Tschöke, Lüling, & Monie, 1995) to planning tasks in robotics (Challou, Gini, & Kumar, 1993). Adaptations have been proposed for parallelizing alpha-beta pruning and general game tree search (Ferguson & Korf, 1988), an area that has gained renewed prominence through IBM's massively parallel Deep Blue chess-playing system (Campbell, Hoane Jr., & Hsu, 2002) or more recent, very successful advances of parallel Monte-Carlo tree search in the game of Go (Cazenave & Jouandeau, 2008; Chaslot, Winands, & van den Herik, 2008).

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* (Powley, Ferguson, & Korf, 1993), or A*, resulting in parallel retracting A* or PRA* (Evett, Hendler, Mahanti, & Nau, 1995). 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 (Evett et al., 1995).

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 (Davis, Logemann, & Loveland, 1962), a depth-first backtrack search algorithm. Consequently, several SAT solvers based on parallel tree search have been proposed (see, e.g., Jurkowiak, Li, & Utard, 2005; Chu & Stuckey, 2008). However, the focus in recent years has shifted to parallelized portfolio solvers (Hamadi & Wintersteiger, 2012).

2.2.3 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 (Ferguson & Korf, 1988).

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 (Gendron & Crainic, 1994; Grama & Kumar, 1995, 1999).

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 3.5.

2.2.4 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 (that doesn't increase when parallelized, a nontrivial assumption) 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 2.3).

In shared-memory approaches to parallel computing, this problem is often tackled through *dynamic load balancing* (Kumar, Grama, & Vempaty, 1994; Grama et al., 2003), 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* (Chu, Schulte, & Stuckey, 2009). Using message passing schemes, dynamic load balancing can also be implemented for distributed-memory architectures (Lynch, 1996; Ghosh, 2006).

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 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.

2.3 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 (Grama et al., 2003). 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 3.5.

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 3.5 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.

2.4 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 (Amdahl, 1967). 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 **1** (Amdahl's Law)**.** *(Amdahl, 1967) 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. $1/(1-0.9+0.09) \approx 5.26$.

Note that Amdahl's Law doesn't strictly apply to parallel Branch-and-Bound algorithms since we are not dividing up a fixed amount of work. But it still illustrates the general challenge of parallelism and can provide a useful bound, in particular since we will find that in many cases the overall work does not change drastically as parallelism is increased (also cf. Section 4.2.4). We will put our results in this context when analyzing our parallel scheme in Section 3.5 and when conducting experimental evaluation in Section 5.

2.5 Other Related Work

We point out the work by Allouche, de Givry, and Schiex (2010), 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.

More closely related is the *Embarrassingly parallel search* (EPS) as presented by Régin, Rezgui, and Malapert (2013). EPS is different in that it explicitly targets constraint satisfaction and constraint optimization problems, but it uses a central set of conditioning variables, quite similar to the fixed-depth cutoff baseline approach we describe for AOBB in Section 3.2. The set is chosen as the first k variables along a given variable ordering such that the resulting number of subproblems is about 30 times the number of available worker nodes. Like in our scheme, subproblem threads/workers don't communicate, but in contrast to our approach, subproblem solutions are used to possibly derive better bounds for subsequent subproblems. With its focus on constraint programming problems, EPS also performs constraint propagation (e.g. alldifferent constraint) to filter out inconsistent subproblems before sending them to workers. This is not as applicable in our general MPE context and thus not a focus of AOBB, which instead uses the more universally applicable mini-bucket heuristic to avoid inconsistent and suboptimal subproblems. Régin et al. run experiments with up to 40 CPU cores on a single computer and reported parallel speedups on constraint optimization problems are 13x on average and up to 20x on some problems. Unfortunately the authors don't provide statistics about the runtime of individual subproblems; similarly they claim that the lack of propagation of bounds across workers does not have a practical impact but don't expand on that.

Régin, Rezgui, and Malapert (2014) adapt EPS for a "data center environment" with hundreds of workers. The authors identify the main bottleneck as the relative complexity of computing the required large number of conditioned subproblems and propose a multi-tiered parallelization of this step. Results on 512 workers yield an average parallel speedup of around $210x$, though the set of problems includes some non-optimization, "plain" constraint instances that are by definition more amenable to this parallel approach because there is no bounding information to propagate. Similar to the work by Régin et al. (2013), details about subproblem imbalance are omitted.

Lastly, we point out the work by Bergman, Ciré, Sabharwal, Samulowitz, Saraswat, and van Hoeve (2014) that recursively applies approximate Decision Trees to solve combinatorial optimization problems, in this case exemplified by the maximum independent set problem. Based on the X10 programming language (Ebcioglu, Saraswat, & Sarkar, 2004), the authors conduct experiments on a single shared-memory 32-core computer as well as a specialized message-passing network of up to eight such systems. Parallel speedups are not specified explicitly and hard to read exactly from the included logarithmic plots of parallel runtimes, but the authors claim "near-linear scaling" up to 64 cores and "still very good scaling", "satisfactory even if not linear" up to 256 cores. More generally, however, this approach involves extensive communication between worker processes which makes it hard to compare against the work in this article.

A fairly young field where parallel search is an active area of research is *distributed constraint reasoning* (Yeoh & Yokoo, 2012), 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, Durfee, Ishida, and Kuwabara (1998) and extensions by Zivan and Meisels (2005). For DCOPs there are, for instance, ADOPT (Asynchronous Distributed Optimization) by Modi, Shen, Tambe, and Yokoo (2005), which is based on parallel best-first search and BnB-ADOPT, an adaptation of ADOPT to depth-first search principles by Yeoh, Felner, and Koenig (2010).

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 work in this article is therefore not easily attainable.

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 2.2.2. To begin, Section 3.1 lays out the parallel environment we build upon, in line with the exposition of Section 2.2.

We then propose two variants of parallel AOBB that differ in how they determine the parallelization frontier. The first, in Section 3.2, chooses the subproblem root nodes at a fixed depth, in line with parallel tree search described in Section 2.2.2. In contrast, the second approach introduced in Section 3.3 uses estimates of subproblem runtime to determine a variable-depth frontier.

3.1 Parallel Setup

As indicated in Section 2.2, 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. This can be group of CPUs on a relatively fast local network, but more commonly it manifests itself in many inhomogeneous systems on different networks of varying speed and connectivity.

We impose 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 foregoing synchronization between workers, thus 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 (Silberstein et al., 2006), 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 (Silberstein, 2011; Silberstein et al., 2013).

Internally, Superlink-Online is built upon a specific grid software package, also called middleware, the *CondorHT* distributed workload management system for "high-throughput" computing (formerly just *Condor*, cf. Thain, Tannenbaum, & Livny, 2002, 2005), which we will also employ for our setup. CondorHT 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).

CondorHT 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-

Algorithm 2 Master process for fixed-depth parallelization.

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

1: $Stack \leftarrow \emptyset$ // last-in-first-out stack data structure
2: $Stack$.push($\langle X_0 \rangle$) // root node at depth 0
3: **while** $|Stack| > 0$:
4:     $n \leftarrow Stack$.pop()
5:     **if** depth$(n) == d_{cut}$:
6:         grid_submit(n)
7:     **else**
8:         **for** $n' \in$ children(n) :
9:             **if** checkpruning$(n', h(n'))$ = true :
10:                 prune(n')
11:             **else**
12:                 $Stack$.push(n')

come, first-served basis. Similarly, CondorHT can provide support for checkpointing and restarting, but it is not something we have implemented at this point.

We note that the extended CondorHT ecosystem includes “MW”, a generic master-worker programming interface. (Linderoth, Kulkarni, Goux, & Yoder, 2000) It does not pertain to Branch-and-Bound specifically, but it is closely related to our overall approach – had we found out about it earlier, it would have presented a worthwhile alternative that might have saved us some implementation effort for the underlying master-worker logic.

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 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 5) 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.

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. 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.

Pseudo code for this simple scheme is shown in Algorithm 2. 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 the branch-and-bound-style pruning logic in line 9 of Algorithm 2 which matches the one from Algorithm 1, using the mini-bucket heuristic and possibly a separately provided initial lower or upper bound on the problem's solution cost (obtained, for instance, through incomplete, local search or from a randomly sampled solution).

EXAMPLE **5.** *To illustrate, we apply Algorithm 2 to the problem from Example 1, for which the AND/OR search graph was shown in Figure 2b. The result of setting the cutoff depth at $d = 1$ is depicted in Figure 3a, the conditioning set $\{A\}$ yields two subproblems. Figure 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. In particular, the subproblems rooted at D and F are unified in Figure 2, yet in Figures 3a and 3b they are spread across different parallel subproblems rooted at C and E, 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 3.5.

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.

3.2.1 UNDERLYING VS. EXPLORED SEARCH SPACE

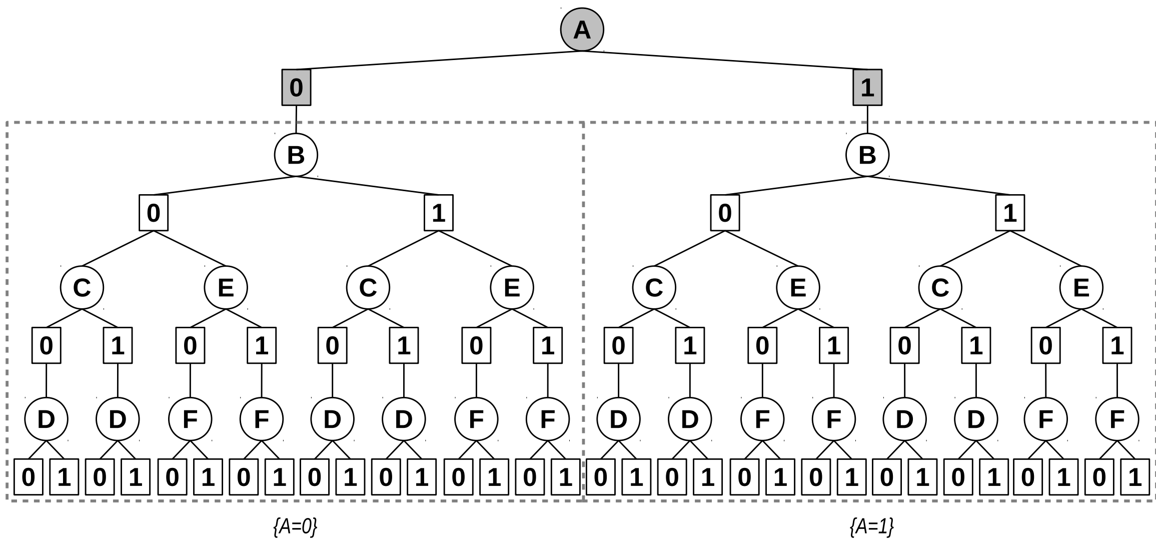
We note that Example 5 and Figures 2 and 3 can be somewhat misleading since they depict the full, underlying context-minimal AND/OR search graph. In practice, however, large parts of the underlying search space are ignored by AOBB, because of determinism or pruning based on the mini-bucket heuristic. This leads to a much smaller *explored search space* – a discrepancy that was also at the core of earlier work by Otten and Dechter (2012a), 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 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.

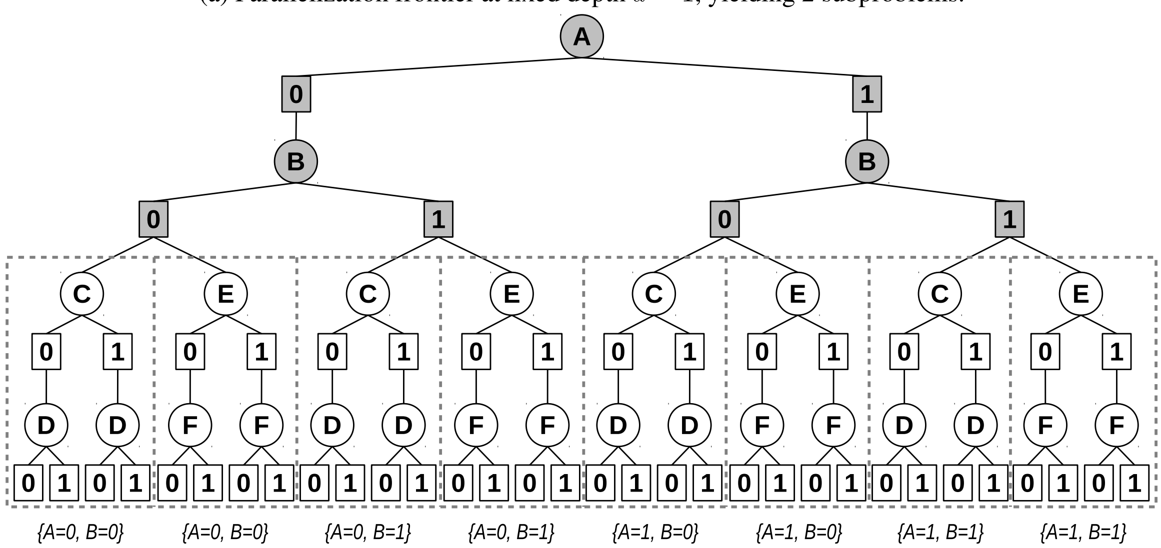
In the context of this section, we point out two things in particular. First, all subproblems originated at the same depth, thus the size of their underlying search space is in fact the same. As expected, however, we observe significant variance in the size of the explored search space. This is captured by the subproblem runtimes plotted in Figure 4, 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.

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



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



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

Figure 3: AND/OR search parallelization at fixed depth, applied to the example problem and search space from Figures 1 and 2, respectively. Conditioning nodes are shaded gray, the respective conditioning set is specified below each subproblem.

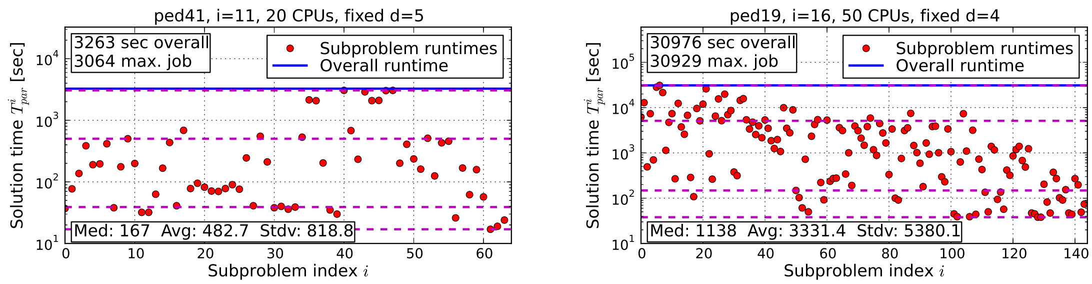


Figure 4: Subproblem statistics for two runs of fixed-depth parallel AOBB. Each dot represents a single subproblem, plotted in the order in which they were generated. Dashed horizontal lines mark the 0$^{\text{th}}$, 20$^{\text{th}}$, 80$^{\text{th}}$, and 100$^{\text{th}}$ percentile, the solid horizontal line is the overall parallel runtime using the number of CPUs specified in the plot title.

Algorithm 3 Master process for variable-depth parallelization.

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

1: $Frontier \leftarrow \{\langle X_0 \rangle\}$
2: **while** $|Frontier| < p$:
3: $\quad n' \leftarrow \arg\max_{n \in Frontier} \hat{N}(n)$
4: $\quad Frontier \leftarrow Frontier \setminus \{n'\}$
5: $\quad$ **for** $n'' \in \text{children}(n')$:
6: $\quad\quad$ **if** $\text{checkpruning}(n'', h(n'')) = \text{true}$:
7: $\quad\quad\quad \text{prune}(n'')$
8: $\quad\quad$ **else**
9: $\quad\quad\quad F \leftarrow Frontier \cup \{n''\}$
10: **while** $|Frontier| > 0$:
11: $\quad n' \leftarrow \arg\max_{n \in Frontier} \hat{N}(n)$
12: $\quad Frontier \leftarrow Frontier \setminus \{n'\}$
13: $\quad \text{grid\_submit}(n')$

depth. Namely, subproblems within one parallel run can be chosen at different depths in order 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 by Otten and Dechter (2012a), summarized in Section 3.4.

Algorithm 3 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 (lines 5-9, note again branch-and-bound-style pruning, cf. Algorithms 1 and 2). 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

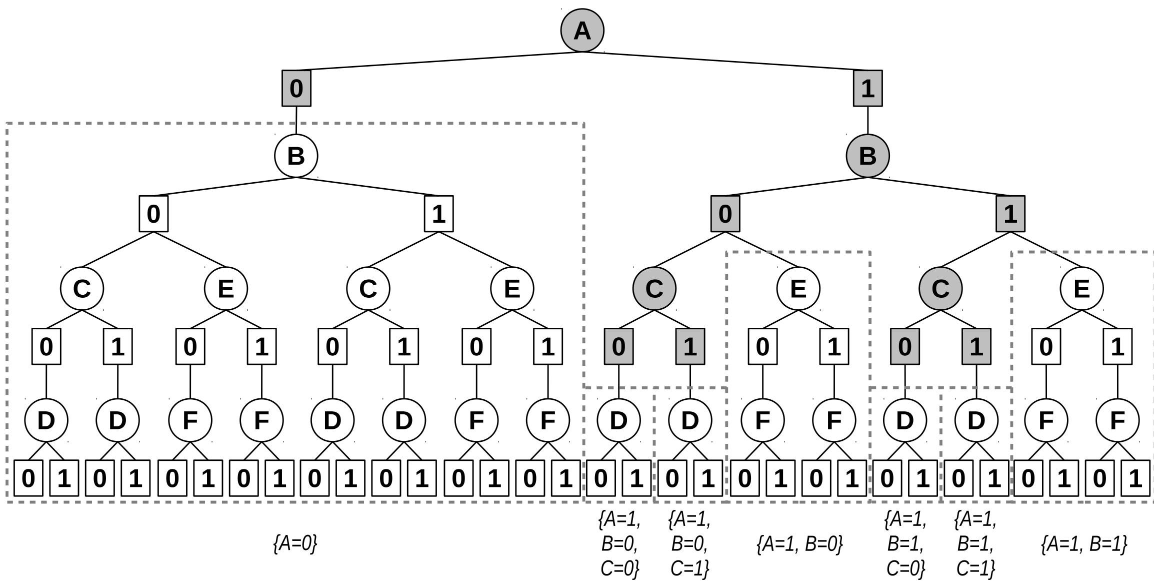


Figure 5: AND/OR search parallelization at variable depth, applied to the example search space from Figure 2, yielding seven subproblems. Conditioning nodes are shaded gray, the respective conditioning set is specified below each subproblem.

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 (Drozdowski, 2009), 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 (Garey, Johnson, & Sethi, 1976). 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 (Graham, 1969). 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.

EXAMPLE **6.** *Figure 5 shows an example of a variable-depth parallel cutoff applied to the same problem as in Example 5. 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 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.

3.4 Estimating Subproblem Complexity

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 2.1). This asymptotic complexity analysis, however, is by definition very loose. Hence a finer-grained *state space bound* can be defined, which takes into account each problem variable's specific domain size and context set. More formally, 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:

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

As shown by Otten and Dechter (2012a), however, this state space bound is generally still very loose since it doesn't account for determinism (early dead-ends due to zero probability entries in a Bayesian network, for instance) or, more significantly, the pruning power of AOBB and the accompanying mini-bucket heuristic. Otten and Dechter (2012a) therefore developed another method to estimate the complexity of a (sub)problem ahead of time, which we briefly summarize in the following.

3.4.1 COMPLEXITY PREDICTION AS REGRESSION LEARNING

As before, we identify a subproblem by its search space root node n and further measure the complexity of the subproblem rooted at n through the size of its explored search space, which is the number of node expansions required for its solution, denoted $N(n)$. We then aim to capture the exponential nature of the search space size by modeling $N(n)$ as an exponential function of various subproblem features $\phi_i(n)$ as follows:

$$N(n) = \exp(\sum_i \lambda_i \phi_i(n)) \tag{2}$$

Given a set of m sample subproblems and taking the logarithm of boths sides of Equation 2, finding suitable parameter values λ_j can thus be formulated as a well-known *linear regression* problem (cf., for instance, Draper & Smith, 1998; Seber & Lee, 2003; Bishop, 2006; Hastie, Tibshirani, & Friedman, 2009). The *mean squared error* (MSE) as the loss function $L(\lambda)$ we aim to minimize captures how well the learned regression model fits the sample data.

$$L(\lambda) = \frac{1}{m} \sum_{k=1}^{m} \Big(\sum_i \lambda_i \phi_i(n_k) - \log N(n_k) \Big)^2 \tag{3}$$

3.4.2 SUBPROBLEM SAMPLE FEATURES

Table 1 lists the set of subproblem features considered by Otten and Dechter (2012a). It was compiled mostly based on prior knowledge of what aspects can affect problem complexity. Features can be divided into two distinct classes: (1) static, which can be precompiled from the problem graph and pseudo tree and include things like number or variables, domain size, and induced width; (2)

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.
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 subproblem "constrainedness".
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 1: List of 35 features extracted from each subproblem as the basis for regression learning.

dynamic which are computed at runtime, as the parallelization frontier decision is made, including bound information and statistics from a limited search sample of the subproblem in question. Note that these features potentially need to be extracted for thousands of subproblems, hence it is important that they are relatively fast to compute.

3.4.3 Regression Algorithm and Subproblem Sample Data

Prior work has investigated a number of regression algorithms (Otten & Dechter, 2012a; Otten, 2013) but eventually settled on Lasso regression (Hastie et al., 2009), which imposes an L_1-penalty on the parameter vector by adding the term $\alpha \sum_i |\lambda_i|$ to Equation 3. We compiled one common set of 17,000 sample subproblems, drawn from previous fixed-depth parallel experiments on instances from all different problem domains considered. Namely, we sample subproblems from fixed-depth parallel runs of the four different problem classes (see Section 5). We select at most 250 subproblems from each run so that the resulting set is not overly skewed towards harder problem instances or larger problem classes. The end result is a sample set with approx. 40% linkage, 25% side-chain prediction, 25% haplotyping and 10% grid network subproblem instances.

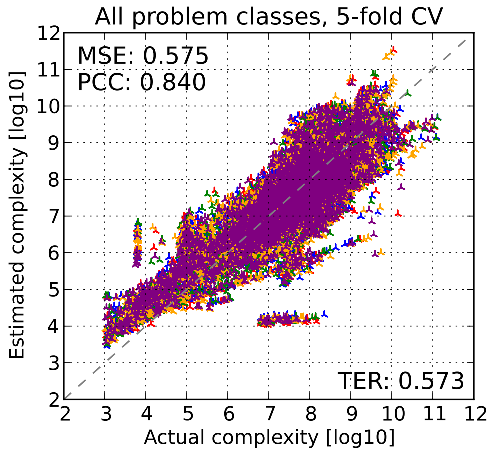


Figure 6: Comparison of actual vs. predicted subproblem complexity on subproblem sample data, using 5-fold cross validation.

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

Table 2: Features ϕ_i present in the model trained by lasso regression and their coefficients λ_i.

For the purpose of this article we learn one "global" prediction model which will be the basis for the experimental evaluation in Section 5. Details were presented by Otten and Dechter (2012a) and Otten (2013), here we simply note that we use $\alpha = 0.06$, found through cross-validation, and that Lasso regression "selected" nine features ϕ_i with $\lambda_i \neq 0$ as shown in Table 2 (with the caveat that Lasso regression tends to pick only one of several correlated features).

To give an idea of the resulting model at this point, Figure 6 compares the actual complexity (in terms of required AOBB node expansions) of all subproblem samples against the predicted complexity, using 5-fold cross validation. It also indicates the mean squared error (MSE) and training error (TER) as well as the Pearson correlation coefficient (PCC), which is the covariance between the vector of actual subproblem complexities and their estimates, normalized by the product of each vectors standard deviation. It is bounded by $[-1, 1]$, where 1 implies perfect linear correlation and -1 anticorrelation. Hence a value close to 1 is desirable in the parallelization context, as it signifies a model likely to correctly identify the hardest subproblems. It is evident that the prediction at this global level gives reasonably good results but is also far from perfect.

Conceptually it's worth pointing out that this implies that the parallel scheme is "aware" of the problem classes it operates on in Section 5. Prior work considered and performed comparisons among different scenarios of learning, from sample subproblems within or across various combinations of problem classes (Otten & Dechter, 2012a). We observed that making predictions on instances of a previously "unknown" problem class can lead to less accurate results. On the other hand, we found that predictions for a new problem instance from a known problem class still generally work well. Hence the setup for the experiments in Section 5 of learning a single global model from subproblem instances across all problem classes arguably provides a degree of simplification to an evaluation that is already significant in scope; but at the same time we believe that it is still suitable to illustrate the performance characteristics of the parallel scheme and its dependency on the accuracy of complexity prediction, which will be highlighted in a number of cases where results are less accurate and performance suffers.

3.5 Analysis of Parallel AOBB

This section provides analysis of the parallel algorithms' properties, also taking into account the performance measures introduced in Section 2.3.

3.5.1 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, as we analyze below.

Parallelization Decision. Algorithms for determining the parallel cutoff were described in Sections 3.2 and 3.3. Performed by a grid master host, this computation involves the following:

- General preprocessing, like problem parsing and evidence elimination, variable order computation, and mini-bucket heuristic compilation, is the same as for sequential AOBB but needs to be performed as part of the initial master process. Runtime for can vary greatly depending on the problem instance and, most centrally, the chosen i-bound of the mini-bucket heuristic.

- The master process gradually expands the conditioning set, until either a fixed depth has been reached (Algorithm 2) or a predetermined number of parallel subproblems has been generated (Algorithm 3). The following is well known and easy to show, but included here for completeness:

  THEOREM **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. 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 (relative to the full search space).

In the context of Amdahl's Law (cf. Section 2.3), the above steps can be seen as the non-parallelizable part of the computation.

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, which typically takes several seconds.

Subsequently the grid management software will match jobs to available worker hosts, transmit input files as necessary, and start remote execution of sequential AOBB. After a job finishes its output is transferred back to the master host – overall adding about 2-3 seconds to each job's runtime.

Repeated Preprocessing. As mentioned, the worker hosts invoke sequential AOBB on the conditioned subproblems, which entails some repeated preprocessing. Recomputing the pseudo tree is relatively easy if the worker receives the variable ordering from the master. The mini-bucket heuristic, however, is less straightforward as its recomputation (exponential in the i-bound) can take significant time for higher i-bounds. This can outweigh actual search time if subproblems are relatively simple and has the potential to significantly deteriorate parallel performance. On the other hand, the context instantiation can be taken into account when recomputing the mini-bucket heuristic for each subproblem, which will likely yield a stronger, tighter heuristic.

Alternatively we could transmit mini-bucket tables from the master process to the worker hosts, but their large size (generally hundreds of megabytes or more) makes that prohibitive in the presence of hundreds of worker hosts.

3.5.2 LARGEST SUBPROBLEM

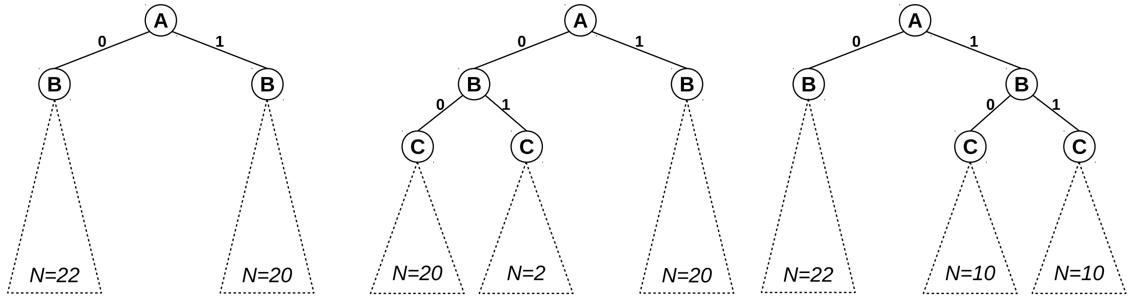
To consider the question of optimality of the parallel cutoff we focus on two aspects, the "balancedness" of the parallel subproblems (which impacts parallel resource utilization) in this section, and the size of the largest subproblem (which has the potential to dominate the overall parallel runtime) in the following Section 3.5.3. We note that, by design, the fixed-depth scheme is oblivious to these notions and will likely yield a cutoff that is suboptimal in both regards (cf. Section 3.2.1). Hence we focus on the variable-depth parallelization frontier.

The size of the largest parallel subproblem is important since it dominates the overall parallel runtime if the number of subproblems is equal or very close to the number of parallel CPUs.

THEOREM **3.** *If the subproblem complexity estimator $\hat{N}$ is exact, i.e., $\hat{N}(n) = N(n)$ for all nodes n, then the greedy Algorithm 3 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 3 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.



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

Figure 7: Example of subproblem splitting decision by the greedy, variable-depth parallelization scheme. Applying Algorithm 3 in state (a) will lead to (b), while (c) would be more balanced (N denotes each subproblem's complexity).

3.5.3 Subproblem Balancedness

To characterize the balancedness of the subproblems in the parallelization frontier we consider the variance over their runtimes. As the following counter example illustrates, however, we cannot make any claims regarding optimality of the parallel cutoff, even if an exact estimator $\hat{N}$ is available:

EXAMPLE **7.** *Assume we run Algorithm 3, 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 7a, 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 7b. The average subproblem size is then* $(20 + 2 + 20)/3 = 14$ *with variance* $(6^2 + 12^2 + 6^2)/3 = 72$*. Yet splitting the subproblem* $\{A = 1\}$ *on the right instead would yield two new subproblems, both of size 10, as depicted in Figure 7c. The average subproblem size is still* $(22 + 10 + 10)/3 = 14$*, but the variance is lower with* $(8^2 + 4^2 + 4^2)/3 = 32$*.*

Hence even with a perfect subproblem complexity estimator, optimality cannot be guaranteed with respect to subproblem balancedness, which also outlines some of the underlying intricacies. In particular, by the nature of branch-and-bound a given subproblem that is split further through conditioning can yield parts of vastly varying complexity (cf. Figure 7b), which is in direct contradiction to our objective of balancing the parallel workload.

4. Parallel Redundancies

Section 3.5.1 illustrated the overhead introduced in the parallel AOBB implementation by virtue of the grid paradigm. 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 becomes evident that the problem of parallelizing AND/OR search is far from embarrassingly parallel (cf. Section 2.2).

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

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

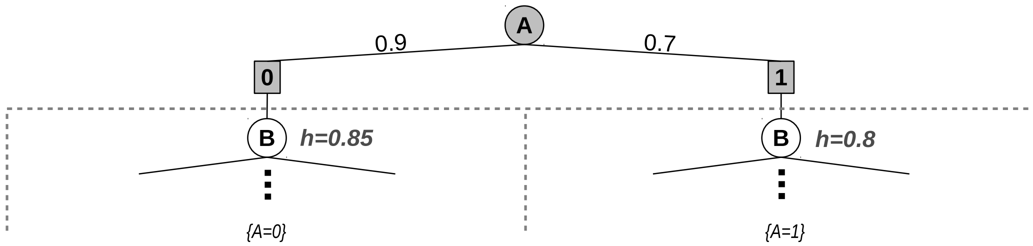


Figure 8: Example of impacted pruning across subproblems. Depending on the optimal solution cost to subproblem $\{A = 0\}$, subproblem $\{A = 1\}$ could be pruned. *Max-product* setting.

- **Impacted caching** of unifiable subproblems across workers.

We will investigate these aspects in the following and in particular 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.

4.1 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.

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 (Otten & Dechter, 2012b).

In the context of parallelizing AOBB on a computational grid, however, this property is compromised because 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 **8.** *Figure 8 shows the top part of the search space from Figure 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 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 8. 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, for instance stochastic local search (Hutter, Hoos, & Stützle, 2005) or limited discrepancy search (Harvey & Ginsberg, 1995; Prosser & Unsworth, 2011). This yields an overall solution that is possibly not optimal, i.e., 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.

Note that in theory an updated lower bound obtained from earlier subproblem solutions could be communicated to workers along with subsequent subproblems via the master (under the assumption that number of subproblems exceeds the number of workers). However, in our current implementation all subproblems are generated and enqueued in the CondorHT framework upfront.[1]

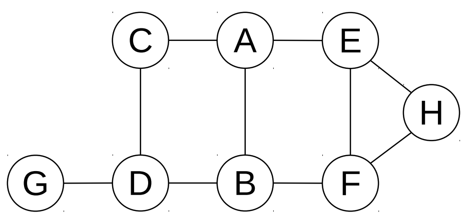
4.2 Impacted Caching across Parallel Subproblems

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 5 and 6, 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*.

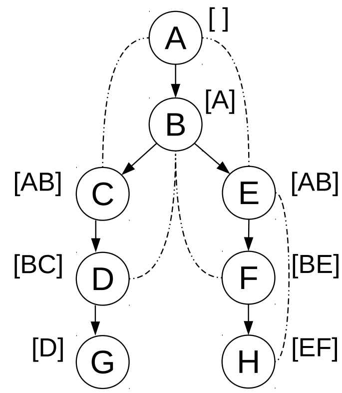
EXAMPLE **9.** *Figure 9a shows an example primal graph with eight problem variables A through H. A possible pseudo tree with induced width 2 is shown in Figure 9b, annotated with each variable's context. The corresponding context-minimal AND/OR search graph is shown in Figure 9c, having a total of 50 AND nodes. We note the following:*

- *Like in Example 3, 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.*
- *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.*

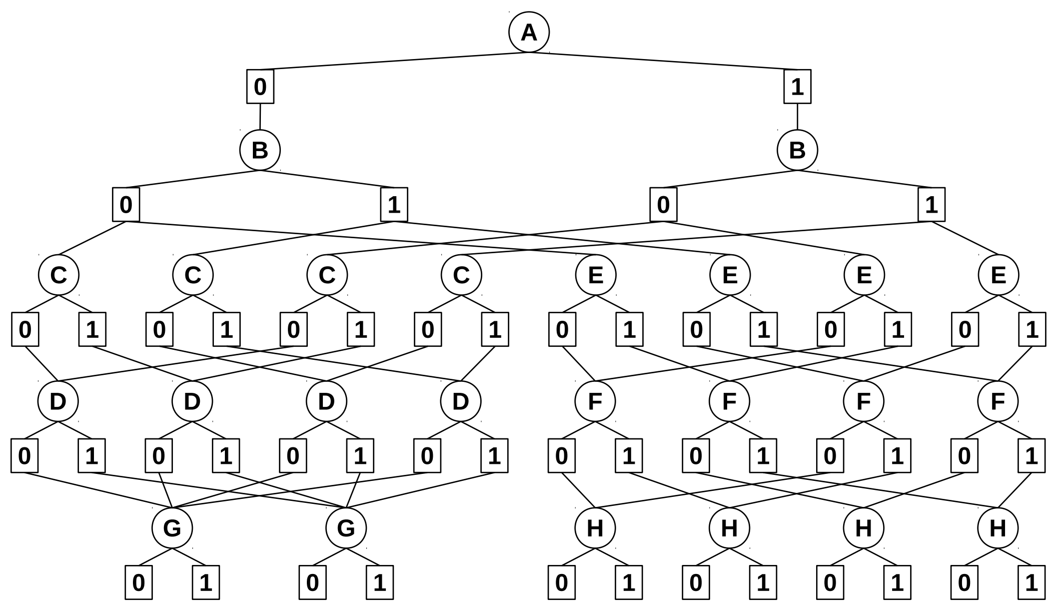
1. We spent some time exploring a more "interactive" approach where new subproblems are only generated when a worker becomes available. But we found that our current implementation, which relies on invoking the heavyweight `condor_submit` command-line tool for each set of subproblems, is not suited to scale this approach to hundreds of workers and thousands of subproblems in rapid succession. Instead, the core of the system would have to be re-implemented using something like the Condor MW interface (Linderoth et al., 2000).



(a) Primal graph with eight variables.



(b) Pseudo tree, induced width 2.



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

Figure 9: 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 H is $\{E, F\}$. With four possible context instantiations, we observe two incoming edges per subproblem rooted at H from the level above.*

As noted, caching can be 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. Our analysis will provide an upper bound on the overall *parallel search space*, i.e., the conditioning space and all parallel subproblem search spaces.

4.2.1 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). 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 $\mathcal{T}$. $L_j := \{X_i \in X \,|\, h(X_i) = j\}$ 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 $\mathrm{context}(X_i)$ we denote the set of variables in the context of X_i with respect to $\mathcal{T}$ (cf. Definition 2), and $w(X_i) := |\,\mathrm{context}(X_i)|$ is the width of X_i.

DEFINITION **3** (conditioned context, conditioned width)**.** *Given a node $X_i \in X$ and $j < h(X_i)$, $\mathrm{context}_j(X_i)$ denotes the* conditioned context *of X_i when placing the parallelization frontier at depth j, namely,*

$$\mathrm{context}_j(X_i) \;:=\; \{X' \in \mathrm{context}(X_i) \,|\, h(X') \geq j\} \;=\; \mathrm{context}(X_i) \setminus \Pi(\pi_j(X_i))\,. \qquad (4)$$

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

The following results extend the state space bound derived earlier for sequential AOBB. In particular, Section 3.4 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:

$$SS_{seq} \;=\; \sum_{i=1}^{n} k^{w(X_i)+1} \;=\; \sum_{j=0}^{h} \sum_{X' \in L_j} k^{w(X')+1}\,. \qquad (5)$$

Starting from Equation 5, 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** (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:*

$$SS_{par}(d) \;=\; \sum_{j=0}^{d} \sum_{X' \in L_j} k^{w(X')+1} \;+\; \sum_{j=d+1}^{h} \sum_{X' \in L_j} k^{w(\pi_d(X'))+w_d(X')+1} \qquad (6)$$

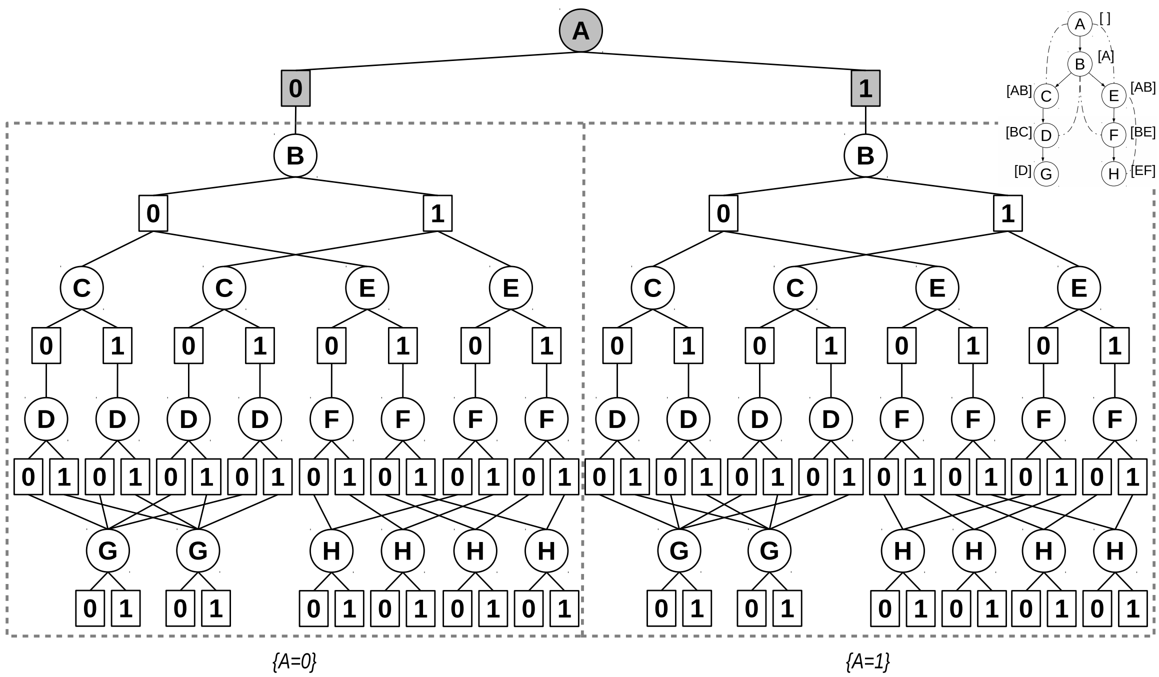


Figure 10: Illustration of parallelization impact on caching for AND/OR search graph from Figure 9c (page 377), with parallel cutoff at fixed depth $d = 1$. Guiding pseudo tree from Figure 9b included for reference.

Proof. The first part of the sum, over levels L_0 through L_d, remains unchanged from Equation 5, 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 6. $\square$

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.

4.2.2 PARALLEL SEARCH SPACE EXAMPLE

This section presents an example to better illustrate Theorem 4. Figures 10, 11, and 12 show examples of parallel search spaces resulting from introducing a fixed-depth parallelization frontier to the context-minimal AND/OR search graph from Example 9 / Figure 9c, with a state space bound of $SS_{seq} = 50$ AND nodes. We explain each figure in detail in the following:

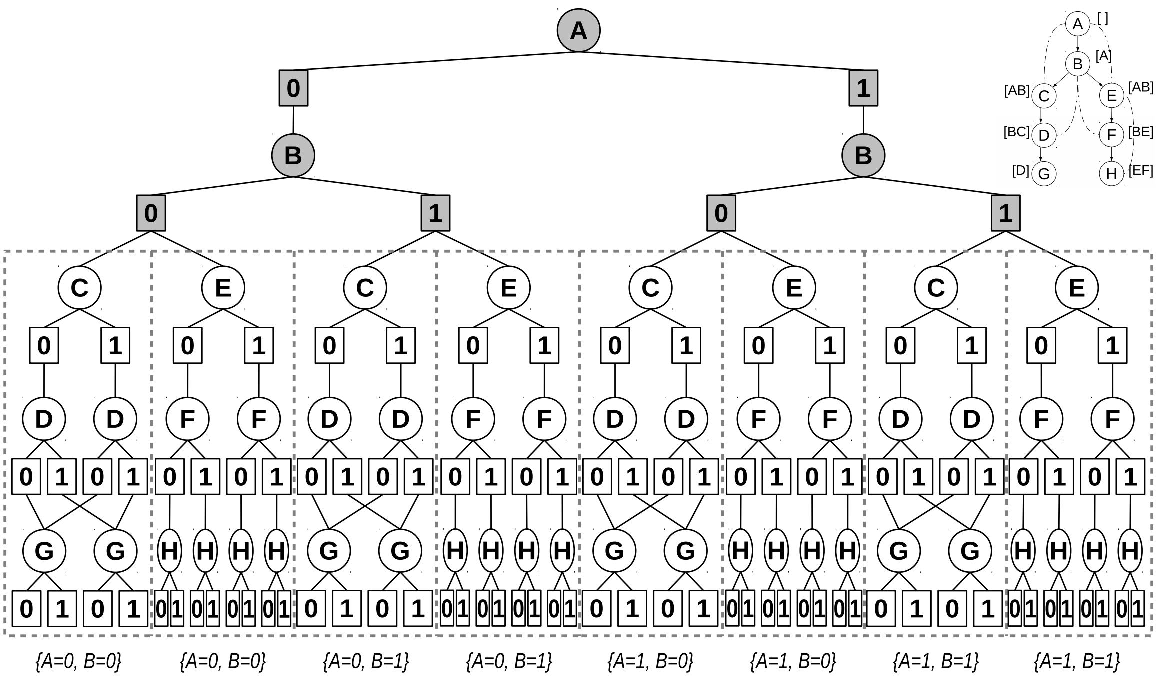


Figure 11: Illustration of parallelization impact on caching for AND/OR search graph from Figure 9c (page 377), with parallel cutoff at fixed depth $d = 2$. Guiding pseudo tree from Figure 9b included for reference.

EXAMPLE **10.** *Figure 10 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 10 is $SS_{par}(1) = 78$; the conditioning space has 2 and each parallel subproblem has 38 AND nodes.*

EXAMPLE **11.** *In Figure 11 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 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 11 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 **12.** *The parallel search space in Figure 12 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*

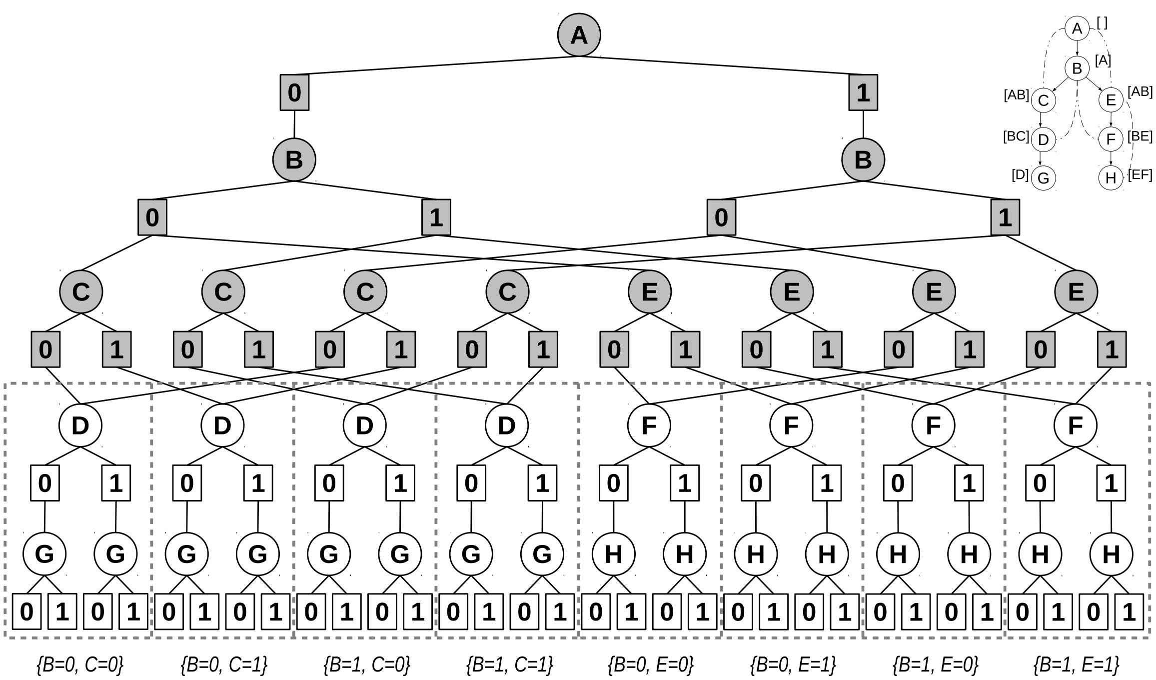


Figure 12: Illustration of parallelization impact on caching for AND/OR search graph from Figure 9c (page 377), with parallel cutoff at fixed depth $d = 3$. Guiding pseudo tree from Figure 9b included for reference.

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 12. In contrast to Figure 11 with cutoff $d = 2$, however, caching for nodes G and H is 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 12 is $SS_{par}(3) = 70$; the conditioning space and each parallel subproblem have 22 and 6 AND nodes, respectively.

4.2.3 ANALYSIS OF STRUCTURAL REDUNDANCY

Table 3 summarizes the properties of the parallel search space of the example problem and pseudo tree given in Figure 9 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.

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.

	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 3: Parallel search space statistics for example problem from Figure 9 with varying parallel cutoff depth d.

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 3.

We observe that the minimum is actually achieved at $d = 2$, where $SS_{par}(d)$ takes its maximum. This observation is also related to *Amdahl's Law* as discussed in Section 2.3. In spite of its limited applicability to Branch and Bound, 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 3.4 and 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 5.

4.2.4 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 5 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 ½ hours with sequential AOBB. For each of these two problems Figure 13 displays the following as a function of the cutoff depth d:

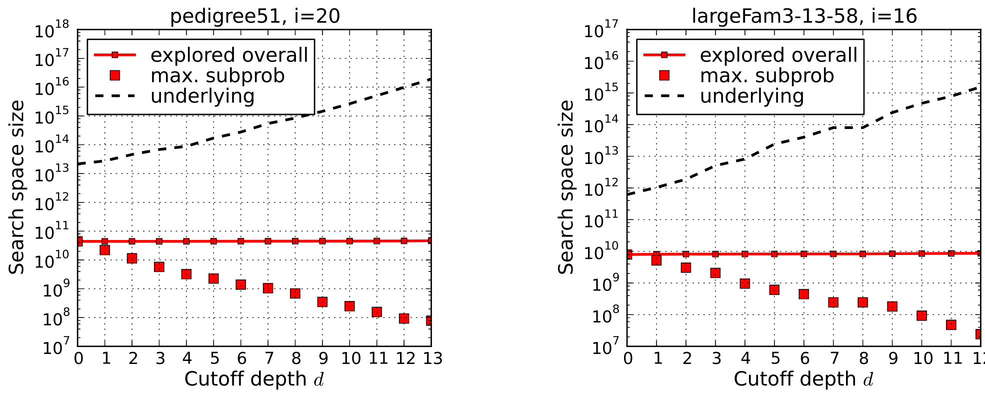


Figure 13: Comparison of underlying parallel search space size vs. size of explored search space (summed across subproblems) and largest subproblem, as a function of the cutoff depth d.

- "underlying": the size of *underlying* parallel search space $SS_{par}(d)$ (Equation 6).
- "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 5.6 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 13 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 5.6.

5. Empirical Evaluation

In this section we will evaluate and analyze the performance of parallel AOBB, as proposed in the previous Section 3. We first outline the experimental setup and grid environment in Section 5.1. Section 5.2 describes the problem instances on which we evaluate performance. In Section 5.3 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 2.3:

- Section 5.4 surveys the overall parallel performance results in terms of parallel runtime T_{par} and corresponding speedup S_{par}.
- In Section 5.5 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) and the resulting overhead O_{par} is investigated in Section 5.6.
- Related to the issue of speedup, Section 5.7 will specifically focus on the question of scaling of the parallel performance with increasing number of CPUs.

Finally, Section 5.8 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 sequential computation time. A comprehensive subset of results is presented and discussed in this article, with complete tables available in a report by Otten (2013).

5.1 Experimental Setup

Experimental evaluation was conducted on an in-house cluster of 27 computer systems, each with dual 2.67 GHz Intel Xeon X5650 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 CondorHT 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 CondorHT grid system.

The master host of our parallel scheme as well as the CondorHT grid scheduler was a separate, dedicated 2.83 GHZ Intel Xeon X3363 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 full source code of our C++ implementation is accessible under an open-source GPL license at `http://github.com/lotten/daoopt`, which also has the problem instances used for evaluation available for download.

5.2 Benchmark Problem Instances

This section introduces the set of benchmarks that form the basis of our experimental evaluation.

- **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.) (Fishelson & Geiger, 2002, 2004). 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 (Darwiche, Dechter, Choi, Gogate, & Otten, 2008; Elidan & Globerson, 2010). The number of problem variables ranges from several hundred to over 1200, with max. domain size between 3 and 7 and induced width in the 20s and 30s.

- **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 (Fishelson, Dovgolevsky, & Geiger, 2005; Allen & Darwiche, 2008). Problem sizes range from 2500 to almost 4000 variables with max. domain size of 3 or 4 and induced width in the 30s.

- **Protein side-chain prediction ("pdb"):** These networks correspond to side-chain conformation prediction tasks in the protein folding problem (Yanover, Schueler-Furman, & Weiss, 2008). The resulting instances have relatively few nodes, ranging from just over 100 to 240, but very a large domain size of 81, and induced width in the 10s. In fact, the largest feasible i-bound for these instances on a normal system with several GB of RAM is 3, 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 (Darwiche et al., 2008). Problems have 623 or 675 binary variables with induced width close to 40.

Parameters of individual problem instances will be more closely described throughout subsequent sections. In general, we will denote with the number of problem variables n, the maximum domain size k, and the induced width w and pseudo tree height h along a given min-fill variable ordering. In addition T_{seq} will denote the problem's solution time with sequential AOBB.

5.2.1 CUTOFF DEPTH AND SUBPROBLEM COUNT

Our general approach will be to run the fixed-depth parallel cutoff scheme (cf. Section 3.2) for varying cutoff depths d and record the number of subproblems generated by the master process; these subproblem counts will serve as input for subsequent variable-depth experiments, using the algorithm presented in Section 3.3). The variable-depth scheme relies on the estimator described in Section 3.4 for complexity estimates.

Note that we limit ourselves to reasonable values of fixed depth d. 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 2.4. 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,": the overhead from distributed communication, job setup time, and repeated preprocessing far exceeds the actual computation time, which generally destroys parallel performance.

5.2.2 CHOICE OF MINI-BUCKET i-BOUND

Recall that we have the workers recompute the mini-bucket heuristic for each subproblem. While this incurs additional, repeated preprocessing, it avoids transferring potentially large mini-bucket tables over the network and lets us use the subproblem's context instantiation to obtain a possibly stronger heuristic (cf. Section 3.5.1). The i-bound values for the heuristic in our experiments 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 (cf. Section 5.4.3).

- Complexity: In certain 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 and to give more depth to the experimental evaluation. Ped7 and ped9, for example, could be solved sequentially in minutes with higher i-bounds. Ped19 and ped51, on the other hand, are sufficiently complex and would take many days to solve sequentially even at their highest possible i-bound, as would all side-chain prediction instances.

In several 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.

5.2.3 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 – the implications of 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 – in practice this is 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.

5.3 Methodology and Three In-depth Case Studies

This section introduces 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 2, Section 3.2) on all benchmark instances for varying cutoff depths d. 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. We then run parallel

AOBB with variable-depth cutoff using these recorded subproblem counts as input (called p in Algorithm 3, Section 3.3) 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, 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.

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 comprehensive subset, with full result tables published separately (Otten, 2013).

5.3.1 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 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 T_{seq} using AOBB are 12 hours, over 26 hours, and almost 33 hours, respectively.

Table 4 presents overall parallel results for a subset of cutoff depths $d \in \{2, 4, 6, 8, 10, 12\}$. In particular, Table 4a 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 4b.

Each cell 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 p 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 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.

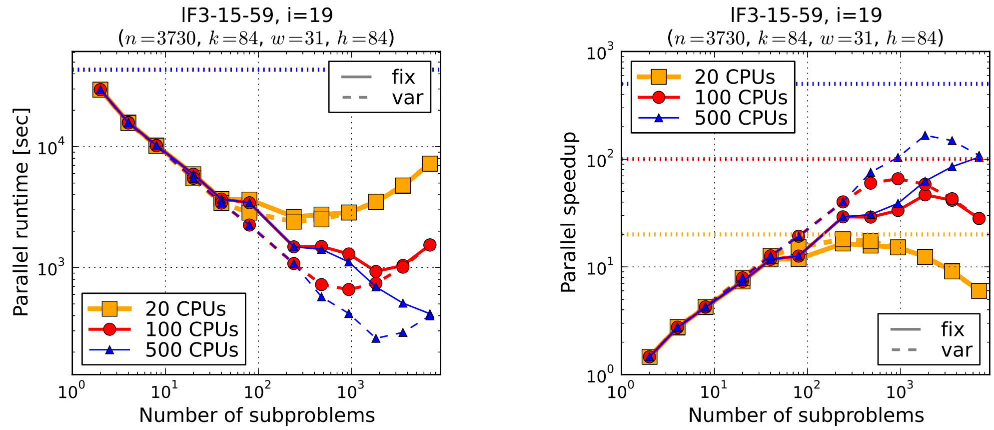
instance	i	T_{seq}	#cpu	Cutoff depth d 2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
lF3-15-59 (n=3730, k=3, w=31, h=84)	19	43307		*(p=4)*		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
			20	15858	15694	5909	5470	3649	2845	2744	2501	3482	3505	7222	7238
			100	15858	15694	5909	5470	3434	2247	1494	723	928	741	1540	1536
			500	15858	15694	5909	5470	3434	2247	1414	573	692	260	415	399
ped44 (n=811, k=4, w=25, h=65)	6	95830		*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
			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
ped7 (n=1068, k=4, w=32, h=90)	6	118383		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
			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.

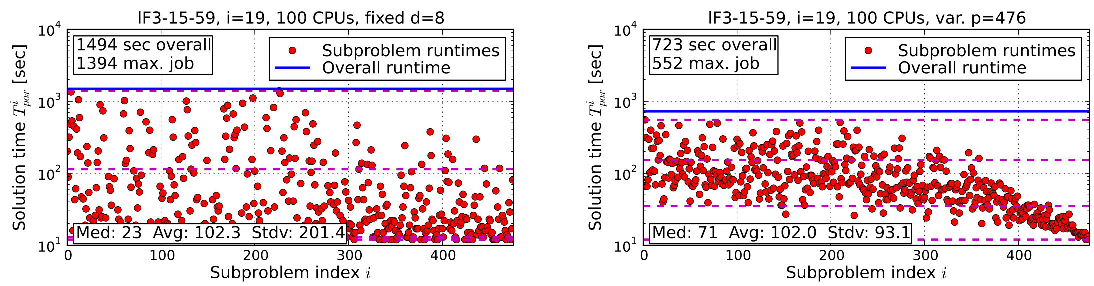
instance	i	T_{seq}	#cpu	Cutoff depth d 2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
lF3-15-59 (n=3730, k=3, w=31, h=84)	19	43307		*(p=4)*		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
			20	2.73	2.76	7.33	7.92	11.87	15.22	15.78	17.32	12.44	12.36	6.00	5.98
			100	2.73	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
ped44 (n=811, k=4, w=25, h=65)	6	95830		*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
			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
ped7 (n=1068, k=4, w=32, h=90)	6	118383		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
			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: 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. Also listed for each instance are the number of problem variables n, the max. domain size k, the induced width w, the height of the guiding pseudo tree h, the mini-bucket heuristic i-bound, and the associated sequential runtime T_{seq} .



(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 14: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on haplotyping instance largeFam3-15-59, $i = 19$.

5.3.2 CASE STUDY 1: LARGEFAM3-15-59

We consider first the largeFam haplotyping problem largeFam3-15-59 with 3730 variables and induced width 31, 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 are plotted in Figure 14a, 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. Note the log-log axes in both plots.

Subproblem Count. 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 3.5. More analysis of this will be provided in Sections 5.5 and 5.6.

For the variable-depth scheme in Figure 14a, 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). 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 one order magnitude larger (roughly 10 times) than the number of parallel CPUs. We will find this confirmed in subsequent parallel results and note that this relation actually matches a "rule of thumb" also reported, for instance, by the team of Superlink Online (Silberstein, 2011).

Fixed-Depth vs. Variable-Depth. It is evident that in the present case a variable-depth parallelization frontier, using regression-based complexity estimates, has a notable 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 4b 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 14b 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 in which they were generated and their position in the CondorHT job queue, which, in case of the variable-depth scheme, is thus sorted by decreasing complexity estimate. 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, with the overall runtime being 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 14b, 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. Section 5.5 and full result tables provided by Otten & Dechter, 2012b). 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.

5.3.3 CASE STUDY 2: PEDIGREE44

Secondly, we consider pedigree44 with $i = 6$ with 811 variables and induced width 25. Sequential runtime is $T_{seq} = 95830$ seconds or 26 hours and 37 minutes. Overall parallel runtime and corresponding speedup are listed in Tables 4a and 4b, respectively. As before, various aspects of parallel performance are plotted in Figure 15 for illustration.

Runtime and Speedup. Figure 15a 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 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) matches the 9 minutes runtime of the fixed-depth scheme.

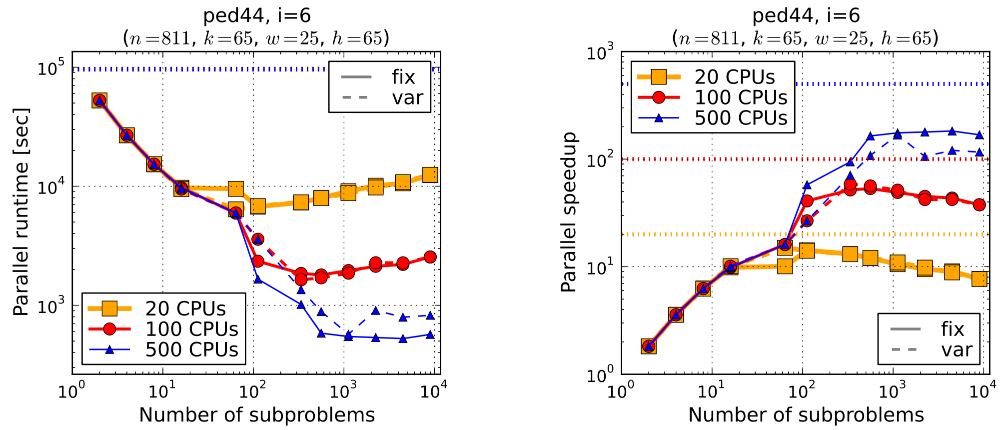
Performance Analysis through Subproblem Statistics. To analyze the inferior performance of the variable-depth scheme as observed above, Figure 15b illustrates the individual subproblem complexities for fixed-depth (left, with cutoff $d = 6$) and variable-depth (right, with corresponding $p = 112$ subproblems) 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 15b 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 15c – 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.

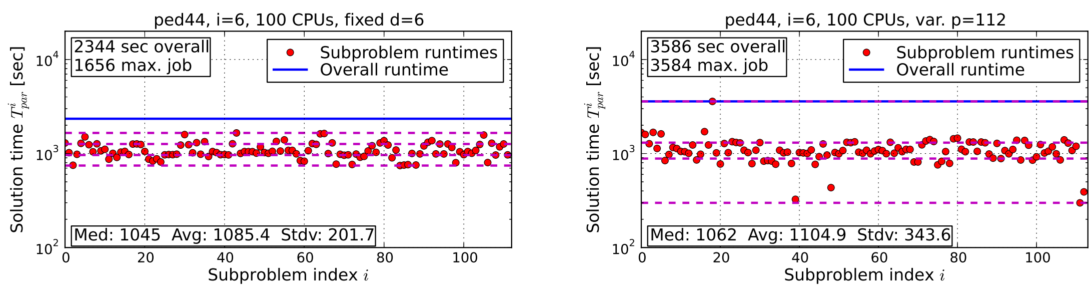
Subproblem Count. The behavior we observe regarding the choice of subproblem count is similar to the previous example instance largeFam3-15-59. In particular, for all three CPU counts parallel performance is identical for small number of subproblems and peaks when the number of subproblems is several times the number of CPUs – for 20, 100, and 500 CPUs at 112, 560, and 2240 subproblems, respectively.

5.3.4 CASE STUDY 3: PEDIGREE7

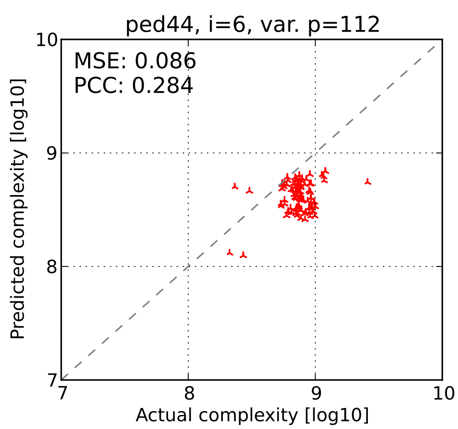
Lastly, we consider pedigree7 with i-bound $i = 6$ which has a sequential runtime of $T_{seq} = 118383$ seconds, or a little under 33 hours. The problem has 1068 variables and induced width 32. A subset of parallel runtimes using 20, 100, and 500 CPUs are shown in Table 4a, with corresponding speedups in Table 4b. As for the previous two examples we plot the progression of runtimes and speedups as the number of subproblem increases in Figure 16a.



(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.

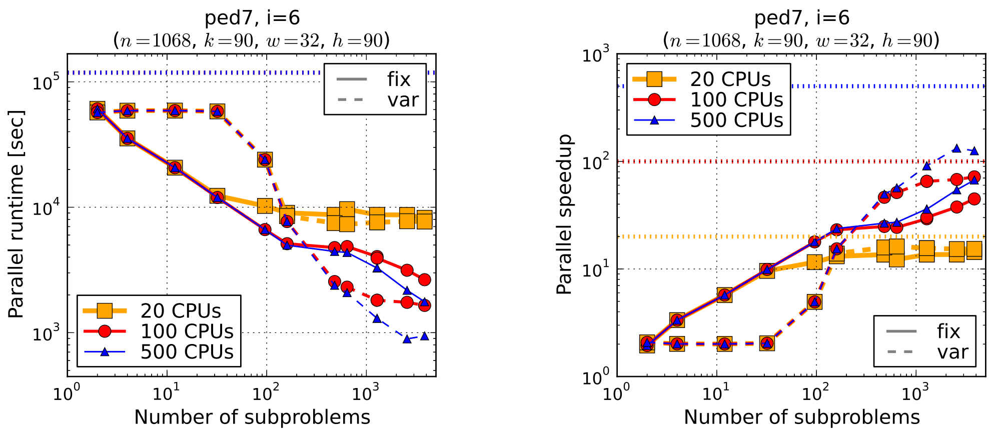


(b) *Left:* runtime statistics of individual subproblems for fixed-depth run with cutoff $d = 6$ using 100 CPUs. *Right:* corresponding variable-depth run with subproblem count $p = 112$.

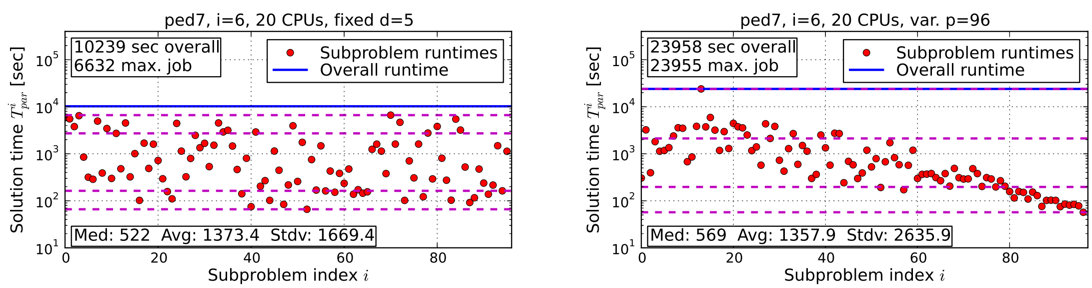


(c) Scatter plot of actual vs. predicted subproblem complexity (in node expansions) for variable-depth parallelization with $p = 112$ subproblems.

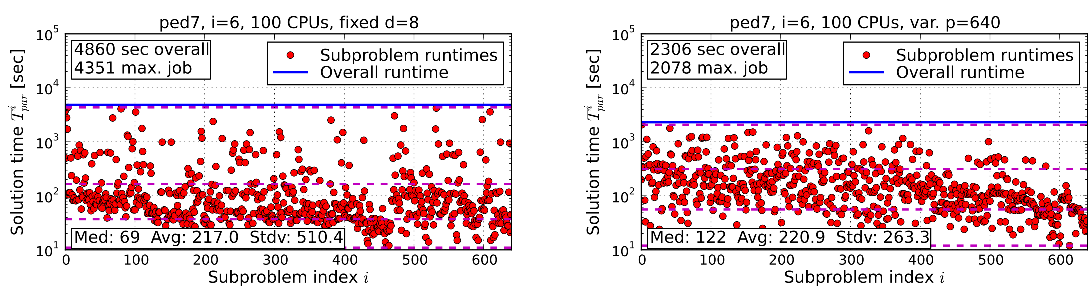
Figure 15: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on linkage instance pedigree44, $i = 6$.



(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 16: Parallel performance details of parallel AOBB with fixed-depth and variable-depth cutoff on linkage instance pedigree7, $i = 6$.

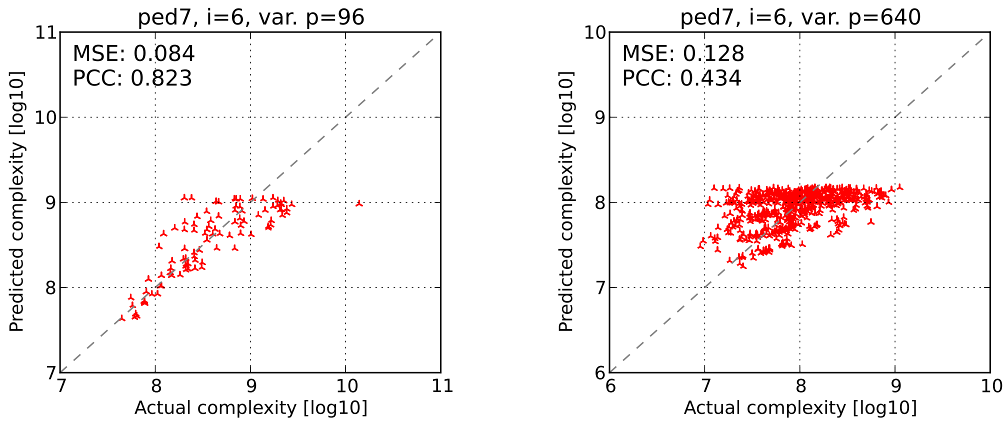


Figure 17: Scatter plot of actual vs. predicted subproblem complexity for variable-depth parallel runs with $p = 96$ (left) and $p = 640$ (right) subproblems on pedigree7, $i = 6$.

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 16b). 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 5.5). 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 17 (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 16c 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 17 (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 5.5).

5.4 Overall Parallel Performance

After highlighting and analyzing a variety of performance characteristics through three particular case studies in Section 5.3, this section will provide a more general overview and analysis on the results of using our parallel scheme.

To that end we provide a comprehensive subset of results for each problem class. For space reaons we focus on parallel speedup, a direct function of the runtime, and refer to full tabled provided by Otten and Dechter (2012b) for explicit runtime results. In addition we show additional performance plots of select problem instances, together with more detailed subproblem analysis in a number of cases.

5.4.1 Overall Analysis of Linkage Problems

Tables 5 and 6 show parallel speedup results for pedigree linkage analysis instances on a subset of cutoff depths. As for Table 4b, each cell gives the number of subproblems p as well as parallel speedup of fixed-depth (left) and variable-depth parallelization (right) with 20, 100, and 500 CPUs, from top to bottom. In addition, the last row lists, in italic font, the "best possible" parallel 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 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 speedups 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, Figure 18 shows parallel speedup plots, respectively, of six different linkage instances from Tables 5 and 6, 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 in Sections 5.3.2 through 5.3.4. 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 (from T_{seq} of 350 and 120 hours, respectively) Similarly, pedigree9 takes between 11 and 28 hours sequentially and 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 $i = 9$ (from $T_{seq} \approx 28$ hours). 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 T_{seq} under 2 hours and highest speedup of 31.

instance	i	T_{seq}	#cpu	d=2 fix	d=2 var	d=4 fix	d=4 var	d=6 fix	d=6 var	d=8 fix	d=8 var	d=10 fix	d=10 var	d=12 fix	d=12 var
ped13 (n=1077, k=3, w=32, h=102)	8	252654		(p=4)		(p=16)		(p=64)		(p=256)		(p=1024)		(p=4096)	
			20	**3.64**	1.85	**13.21**	3.56	**15.51**	10.22	16.59	**18.28**	17.95	18.20	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		(p=4)		(p=16)		(p=64)		(p=256)		(p=1024)		(p=4096)	
			20	3.31	3.23	**9.71**	8.54	**17.63**	10.55	**20.03**	17.99	19.60	18.37	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		(p=12)		(p=144)		(p=1440)		(p=5752)		(p=11254)			
			20	3.14	**5.07**	12.11	**13.38**	13.75	13.19	10.66	10.03	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		(p=6)		(p=32)		(p=160)		(p=800)		(p=6400)			
			20	3.77	3.69	11.31	12.14	13.00	11.86	9.99	10.17	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		(p=6)		(p=32)		(p=160)		(p=800)		(p=6400)			
			20	9.54	9.63	**26.98**	24.28	27.66	29.13	14.66	14.66	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		(p=4)		(p=16)		(p=64)		(p=256)		(p=1024)		(p=4096)	
			20	3.49	3.51	12.02	12.19	12.14	**14.08**	14.64	15.39	15.44	15.55	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		(p=4)		(p=16)		(p=64)		(p=256)		(p=1024)		(p=4096)	
			20	4.03	3.99	14.26	14.68	13.87	**16.23**	18.99	17.98	18.01	17.85	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		(p=4)		(p=16)		(p=64)		(p=256)		(p=1024)		(p=4096)	
			20	4.32	4.01	12.51	13.27	11.10	**14.42**	14.95	**17.48**	15.39	15.66	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		(p=3)		(p=6)		(p=24)		(p=96)		(p=384)		(p=1536)	
			20	1.96	1.86	3.65	3.95	9.26	**12.37**	12.57	12.87	12.87	13.85	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		(p=5)		(p=20)		(p=60)		(p=180)		(p=716)		(p=1896)	
			20	2.27	2.27	5.49	**6.67**	6.60	**7.84**	8.81	**10.30**	9.73	9.89	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		(p=5)		(p=20)		(p=60)		(p=180)		(p=720)		(p=1912)	
			20	1.62	1.61	**3.37**	2.25	3.66	3.67	4.35	4.71	4.70	4.58	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		(p=5)		(p=20)		(p=60)		(p=180)		(p=716)		(p=1896)	
			20	3.66	3.65	4.70	**6.29**	6.07	6.11	6.87	6.97	7.50	7.35	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		(p=4)		(p=16)		(p=128)		(p=768)		(p=2304)			
			20	2.47	2.43	4.44	4.41	9.52	9.11	9.35	**11.72**	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		(p=4)		(p=16)		(p=128)		(p=768)		(p=2304)			
			20	2.78	2.54	5.38	**6.13**	5.23	**7.54**	5.22	**7.20**	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 5: 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" 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 bolded. 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
ped41 n=1062 k=5 w=33 h=100	9	25607		*(p=8)*		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
			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
			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**
			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**
			∞	*1.58*	***1.79***	*5.61*	***10.51***	*20.13*	***32.29***	*39.40*	***75.31***	*115.35*	***148.88***	*63.86*	***173.02***
	10	46819		*(p=8)*		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
			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***
	11	27583		*(p=8)*		*(p=32)*		*(p=128)*		*(p=352)*		*(p=1408)*		*(p=4352)*	
			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***
ped44 n=811 k=4 w=25 h=65	5	207136		*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
			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**
			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
			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
			∞	*3.15*	*3.14*	***10.16***	*6.62*	***60.65***	*19.43*	***151.08***	*110.89*	***565.95***	*206.72*	***281.05***	*224.90*
	6	95830		*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
			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*
ped50 n=514 k=6 w=17 h=47	3	4135		*(p=4)*		*(p=144)*		*(p=2160)*		*(p=14401)*					
			20	2.78	2.80	9.78	**11.99**	8.89	9.17	2.38	2.39				
			100	2.78	2.80	11.99	11.99	20.88	**39.01**	10.85	10.94				
			500	2.78	2.80	11.99	11.99	26.01	**46.99**	32.56	**37.25**				
			∞	*2.78*	*2.80*	*11.99*	*11.99*	*27.03*	***46.99***	*46.46*	***78.02***				
	4	1780		*(p=4)*		*(p=144)*		*(p=2160)*		*(p=14400)*					
			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*				
ped51 n=1152 k=5 w=39 h=98	20	101788		*(p=4)*		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4064)*	
			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
			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
			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**
			∞	*3.73*	*3.73*	*12.32*	*13.22*	*29.44*	***38.29***	*57.44*	***87.52***	***149.47***	*70.69*	*147.73*	*150.80*
	21	164817		*(p=4)*		*(p=16)*		*(p=64)*		*(p=256)*		*(p=1024)*		*(p=4096)*	
			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*
ped7 n=1068 k=4 w=32 h=90	6	118383		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
			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**
			∞	***3.35***	*2.01*	***9.90***	*2.04*	***23.75***	*15.39*	*27.16*	***56.75***	*36.36*	***90.99***	*68.04*	***135.14***
	7	93380		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
			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***
	8	30717		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1276)*		*(p=3816)*	
			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***
ped9 n=1118 k=7 w=27 h=100	6	101172		*(p=4)*		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
			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
			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
			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
			∞	***3.66***	*1.94*	***9.76***	*5.68*	***18.60***	*14.42*	***71.96***	*37.78*	***256.13***	*108.67*	***595.13***	*315.18*
	7	58657		*(p=4)*		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
			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*
	8	41061		*(p=4)*		*(p=16)*		*(p=32)*		*(p=128)*		*(p=512)*		*(p=2048)*	
			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 6: 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" 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 bolded. The best value in each row is highlighted in gray.

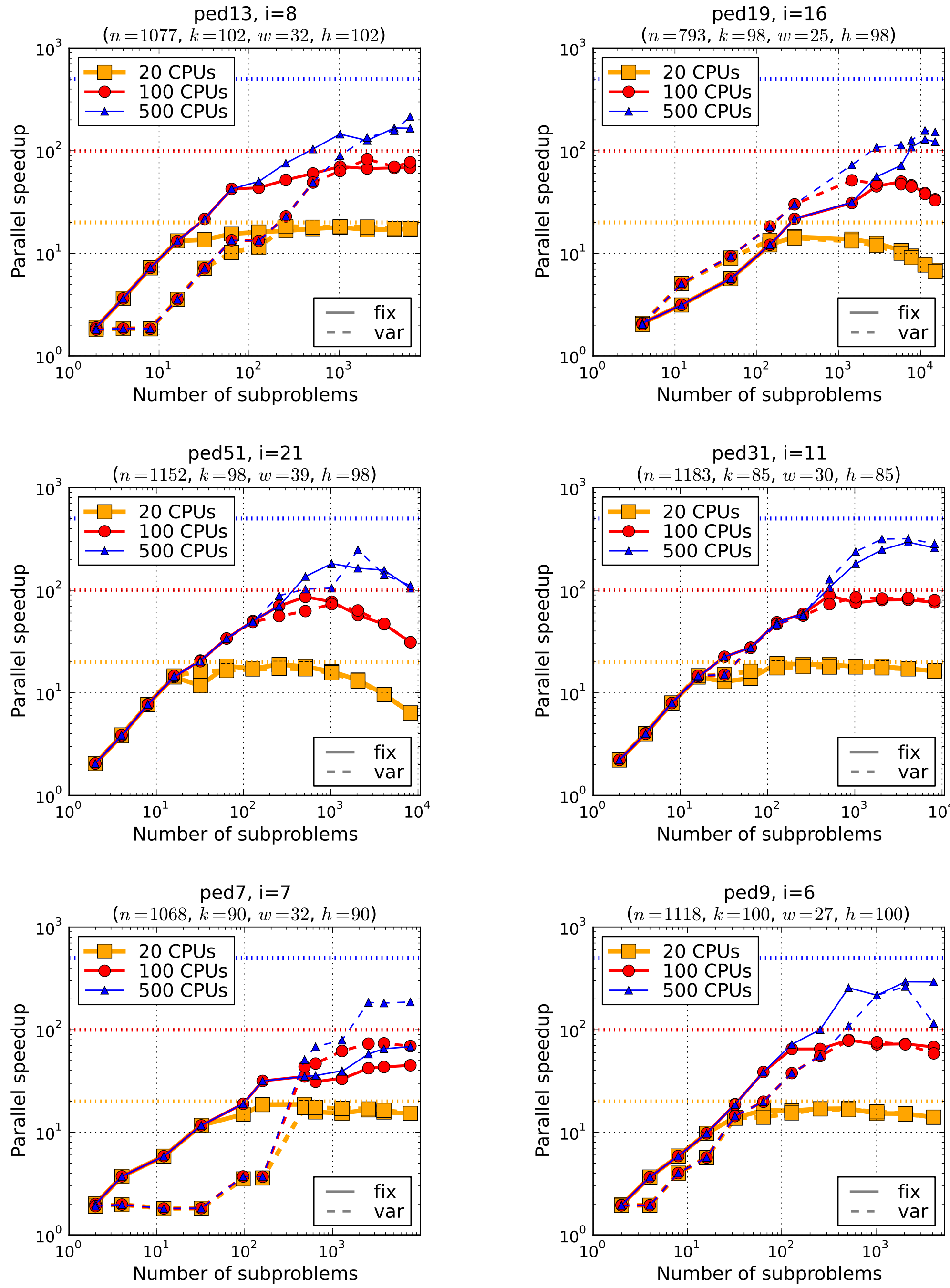
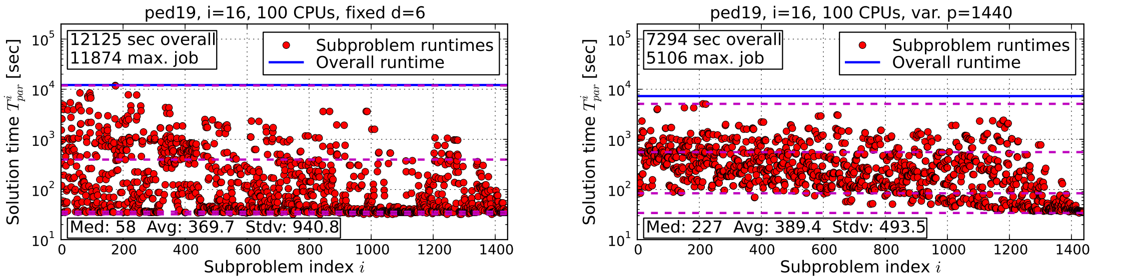
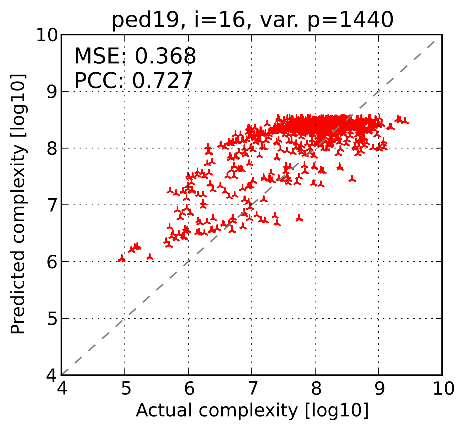


Figure 18: Parallel **speedup** plots for select **linkage** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.



(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 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.

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 $i = 8$ ($T_{seq} \approx 70$ hours), shown at the top left of Figure 18, and $i = 9$ (over 28 hours sequentially). Across all CPU counts, fixed-depth performance ceases earlier to be bound by the longest-running subproblem; for $i = 8$ and 100 CPUs at depth $d = 8$, for instance, the variable-depth cutoff speedup of 22.91 is still determined by the longest running subproblem (as implied by the last entry 5, in italic), while the fixed-depth scheme achieves a speedup of 51.91 in which case the possible optimal speedup, induced by the longest running subproblem, is 75.67 (in italic).

Pedigree19. Pedigree19, on the other hand, with $T_{seq} \approx 104$ hours even at the largest feasible i-bound value of $i = 16$, produces one of several promising experiments with respect to fixed-depth and variable-depth performance (top right in Figure 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 speedup 157 vs. speedup 129, with $p = 11254$ subproblems or depth

$d = 10$, respectively. Figure 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 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 a large 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 5.3.4). On the other hand, in the absence of such estimation outliers, we have shown the potential of the variable-depth scheme to improve 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 5 and 6 fixed-depth has the advantage in 40 cases and 27 cases by a 10% and 50% margin, respectively. Variable-depth is superior by 10% and 50% in just 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 wasn'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.

5.4.2 Overall Analysis of Haplotyping Problems

Tables 7 and 8 show the parallel speedup values on largeFam haplotyping instances. 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, Figure 20 shows plots of parallel speedup for six of the problems.

General Performance. As for pedigrees in Section 5.4.1 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 $i = 14, 16$ (sequential time of $\sim$13 and 5.5 hours), respectively, with $d = 7$ – or even above 19 (largeFam4-12-50 for $i = 14$, $d = 5$, $T_{seq} \approx 9$ hours). 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 $d = 13$, $T_{seq} \approx 96$ hours) or upper 70s (largeFam3-13-58 with $i = 14$, $d = 11$, $T_{seq} \approx 13$ hours). Finally, the best speedups with 500 CPUs are just over 200, for instance 207 for largeFam3-13-58 with $i = 14$ (13 hours sequentially) and $d = 13$, or 206 and 241 for largeFam4-12-50 at $d = 7$ with $i = 13, 14$, respectively (T_{seq} of 16 and $\sim$9 hours).

Overall, however, speedups are somewhat lower than what we saw for linkage instances in Section 5.4.1, 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 2.4). Secondly, in several other cases with weaker parallel results, we

				Cutoff depth d											
instance	i	T_{seq}	#cpu	3		5		7		9		11		13	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
				(p=6)		(p=30)		(p=60)		(p=120)		(p=360)		(p=1440)	
	15	121311	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
			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 n=2670 k=3 w=37 h=95	16	35820	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
			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)	
	17	18312	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
			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)	
	14	35457	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
			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 n=2711 k=3 w=32 h=73	15	8523	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
			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)	
	16	3023	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
			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)	
	14	46464	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
			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 n=3352 k=3 w=31 h=88	16	20270	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
			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)	
	18	7647	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
			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)	
	17	345544	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**
			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**
lF3-15-53 n=3384 k=3 w=32 h=108			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**
			∞	3.47	3.45	3.67	**5.80**	4.33	**10.49**	10.68	**28.16**	20.82	**47.18**	28.25	**73.10**
				(p=12)		(p=32)		(p=68)		(p=284)		(p=912)		(p=2496)	
	18	98346	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
			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 7: 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" 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 bolded. The best value in each row is highlighted in gray.

instance	i	T_{seq}	#cpu	3 fix	3 var	5 fix	5 var	7 fix	7 var	9 fix	9 var	11 fix	11 var	13 fix	13 var
lF3-15-59 n=3730 k=3 w=31 h=84	18	28613		(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
			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
			∞	**5.41**	4.25	13.92	13.84	33.98	**53.48**	44.99	**61.93**	114.45	**225.30**	113.09	**124.95**
	19	43307		(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
			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**
lF3-16-56 n=3930 k=3 w=38 h=77	15	1891710		(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
			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**
			∞	2.94	2.96	5.97	**10.13**	15.89	**44.49**	52.70	**120.08**	70.79	**185.01**	124.14	**443.85**
	16	489614		(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
			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**
lF4-12-50 n=2569 k=4 w=28 h=80	13	57842		(p=24)		(p=288)		(p=3456)							
			20	12.03	11.89	16.95	16.97	13.63	13.94						
			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**						
			∞	12.03	11.89	52.44	**64.34**	107.91	**245.09**						
	14	33676		(p=24)		(p=288)		(p=3456)							
			20	11.43	**13.90**	19.24	17.75	13.25	13.20						
			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**						
lF4-12-55 n=2926 k=4 w=28 h=78	13	104837		(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
			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
			∞	**6.44**	3.78	**28.09**	7.67	**62.07**	7.69	**156.01**	18.92	**173.00**	151.72	143.61	137.94
	14	25905		(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
			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**
lF4-17-51 n=3837 k=4 w=29 h=85	15	10607		(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
			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
			∞	3.76	3.81	8.02	8.24	13.85	13.72	31.85	31.57	33.46	31.95	**77.99**	27.13
	16	66103		(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
			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 8: 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" 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 bolded. The best value in each row is highlighted in gray.

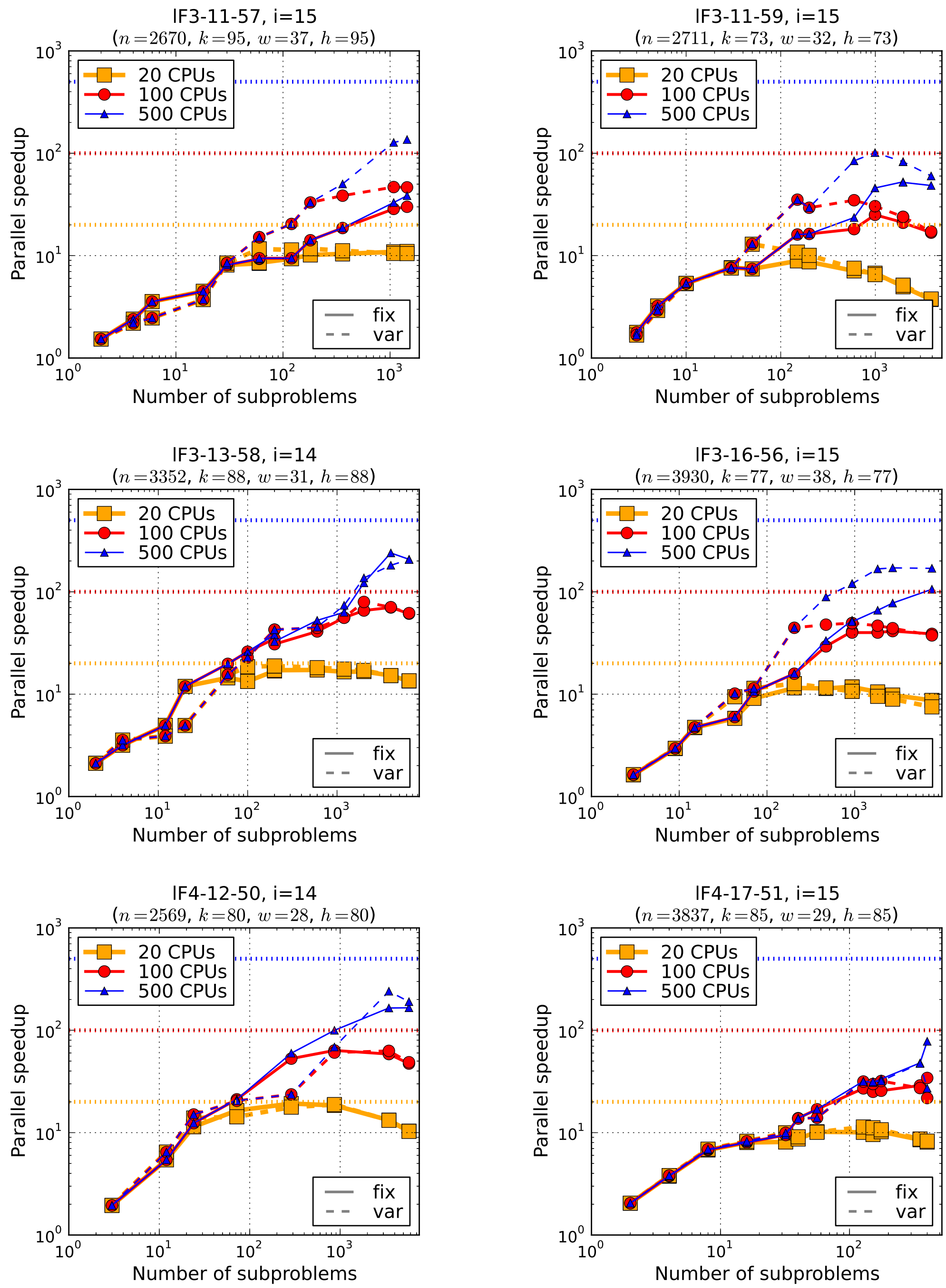


Figure 20: Parallel **speedup** plots for select **haplotyping** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.

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 7 and 8, 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. 5.3.4, e.g.) suggested a trade-off where p is about 10 times the CPU count, matching a "rule of thumb" reported by other researchers (Silberstein, 2011). As a reminder, the choice of p needs 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), which advocates for a lower cutoff depth.

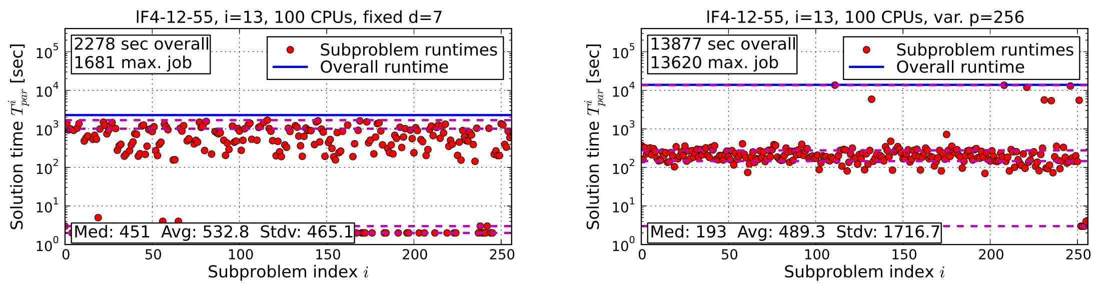
Figure 20 provides a convenient illustration of this trade-off. It is most notably for 20 and 100 CPUs, where performance begins to deteriorate once the number of subproblems grows too large, inducing disproportionate overhead. We can furthermore affirm this general behavior and, more specifically, the particular suggested rule of thumb by again consulting Tables 7 and 8. Consider, for instance, largeFam3-16-56 for both $i = 15, 16$. In Table 8 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 8 – cf. Otten, 2013).

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.

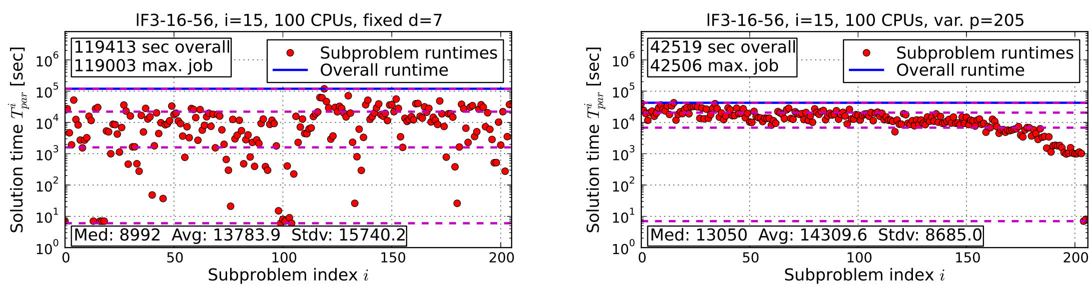
LargeFam4-12-55. Parallel speedup for instance largeFam4-12-55 ($n = 2926$ variables and induced width $w = 28$) is detailed in Table 8. For i-bound 13 (sequential runtime over 29 hours) the fixed-depth scheme is overall significantly faster than the variable-depth for all but the highest depth value 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 21: the scatter plot in Figure 21c clearly shows a handful of vastly underestimated outliers that dominate the parallel execution in Figure 21a (right). However, in Figure 21a we also see that, with 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 21a.

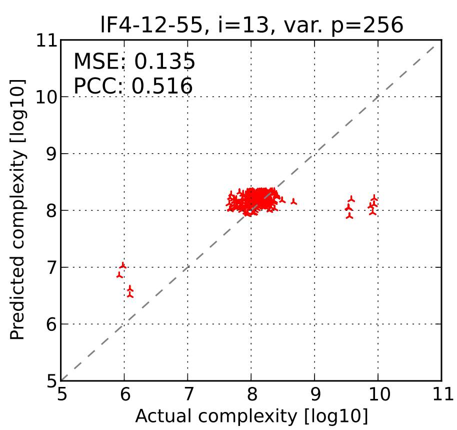
LargeFam3-15-56. One of several more positive examples is largeFam3-16-56 ($n = 3930$, $w = 38$), which has a sequential runtime of almost 22 days ($i = 15$). Figure 21b details the subproblem runtimes for fixed-depth parallelization with cutoff $d = 7$ (left) and the variable-depth scheme with the corresponding 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



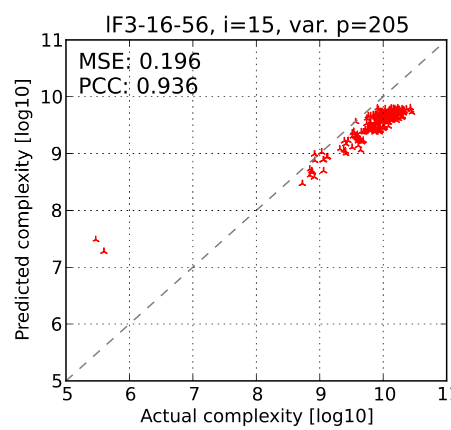
(a) Runtime statistics of individual subproblems *of largeFam4-12-55* using 100 CPUs; left: fixed-depth run with cutoff $d = 7$, right: corresponding variable-depth run with subproblem count $p = 256$.



(b) Runtime statistics of individual subproblems *of largeFam3-16-56* using 100 CPUs; left: fixed-depth run with cutoff $d = 7$, right: corresponding variable-depth run with subproblem count $p = 205$.



(c) Scatter plot of actual vs. predicted subproblem complexity for variable-depth parallel run on largeFam4-12-55 with $p = 256$ subproblems.



(d) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run on largeFam3-15-56 with $p = 205$ subproblems.

Figure 21: 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, as well as largeFam3-16-56 instance ($i = 15$) with $d = 7$ and corresponding $p = 205$ subproblems

hours. Figure 21d 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 outliers. This is also the reason why the subproblem runtimes on the right of Figure 21b 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 7 and 8. 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.

5.4.3 OVERALL ANALYSIS OF PROTEIN SIDE-CHAIN PREDICTION PROBLEMS

Table 9 shows parallel speedup results of running the two parallel schemes, as before with 20, 100, 500, and "unlimited" CPUs, on side-chain prediction instances. Also as before, Figure 22 contains the corresponding speedup plots 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 – chosing $i = 4$ 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 5.2). For instance, pdb1hd2 has 3777 subproblems with $d = 2$, but setting $d = 3$ would yield over 66 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 CondorHT 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 Table 9. For 20 CPUs we still see acceptable behavior, with speedups of 17 or 18 on a number of instances (e.g., pdb1duw at $d = 3$, sequential time 174 hours); 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 some 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.

				Cutoff depth d											
instance	i	T_{seq}	#cpu	1		2		3		4		5		6	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	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 9: 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" 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 bolded. The best value in each row is highlighted in gray.

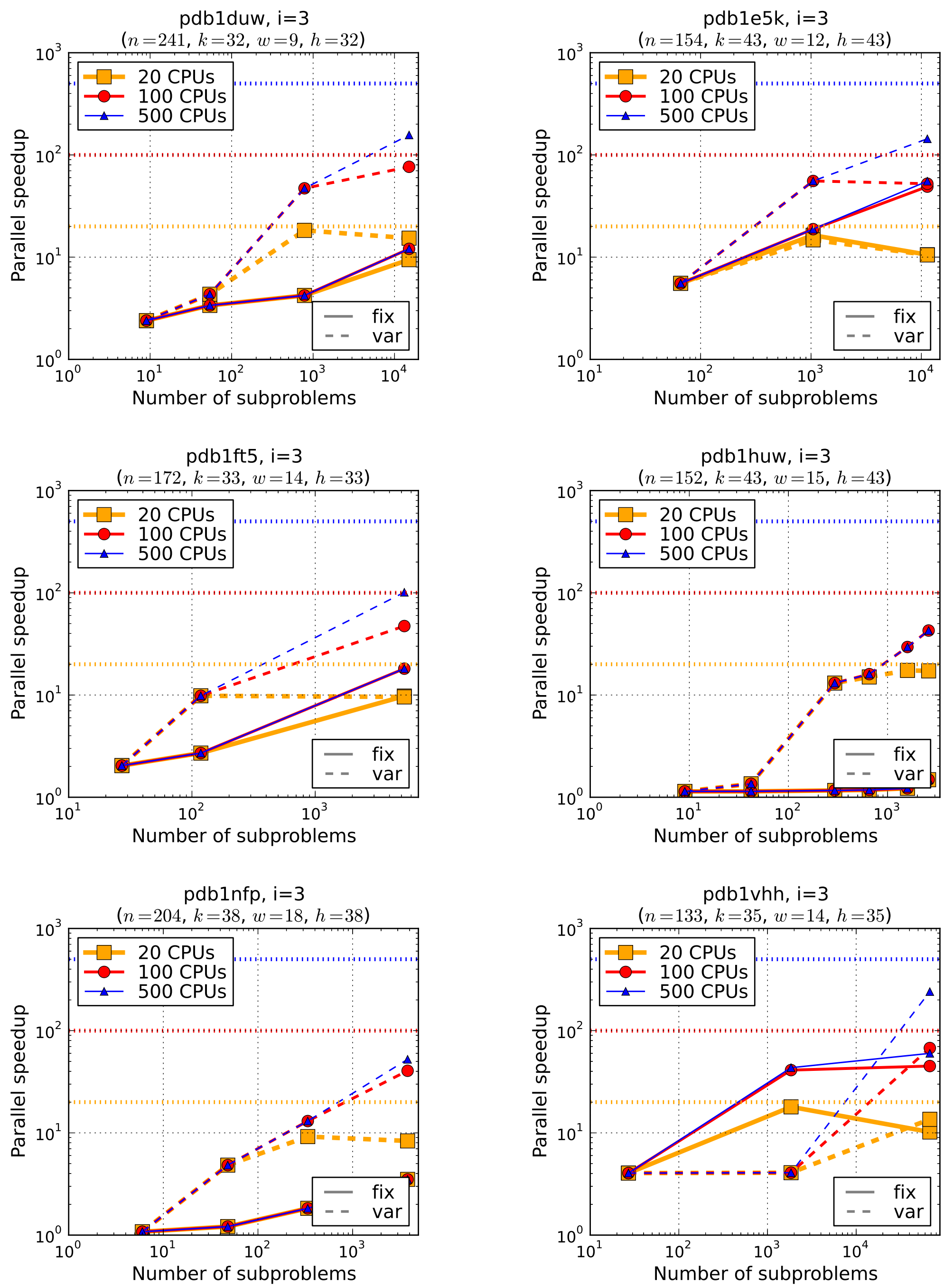
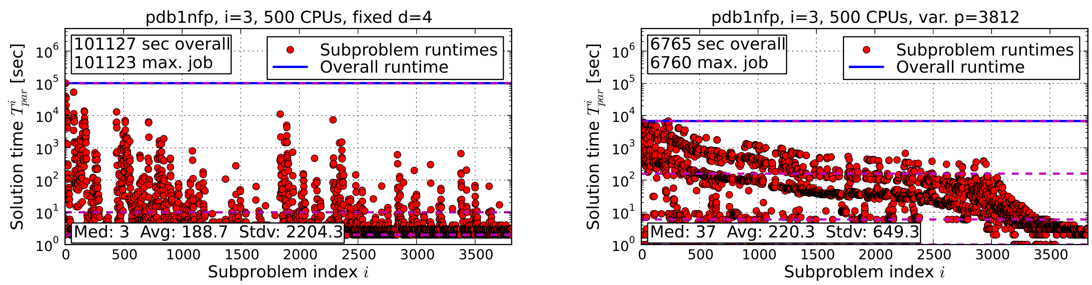
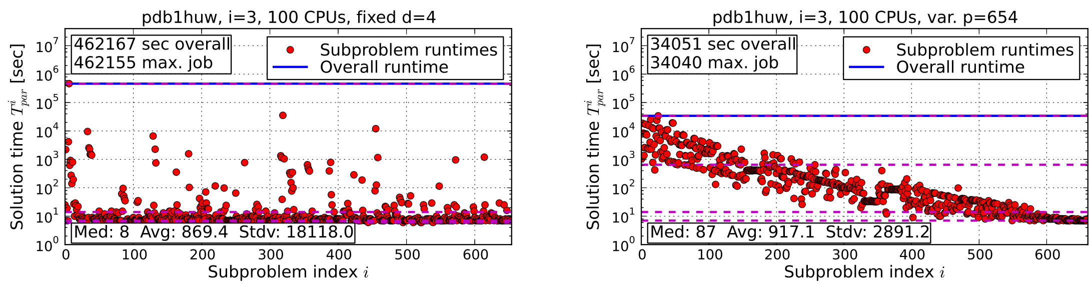


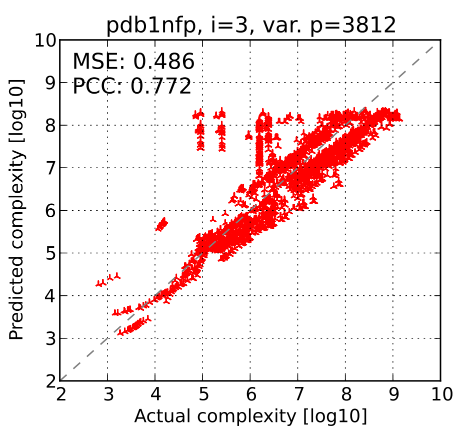
Figure 22: Parallel **speedup** plots for select **side-chain prediction** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.



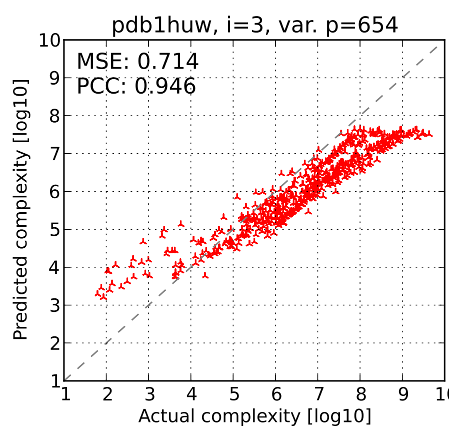
(a) Runtime statistics of individual subproblems *of pdb1nfp* using 500 CPUs; left: fixed-depth run with cutoff $d = 4$, right: corresponding variable-depth run with subproblem count $p = 3812$.



(b) Runtime statistics of individual subproblems *of pdb1huw* using 100 CPUs; left: fixed-depth run with cutoff $d = 4$, right: corresponding variable-depth run with subproblem count $p = 654$.



(c) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run on pdb1nfp with $p = 3812$ subproblems.



(d) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run on pdb1huw with $p = 654$ subproblems.

Figure 23: Performance details of fixed-depth and variable-depth parallel scheme on pdb1nfp instance ($i = 3$) with $d = 4$ and corresponding $p = 3812$ subproblems, respectively, as well as pdb1huw instance($i = 3$) with $d = 4$ and corresponding $p = 654$ subproblems.

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 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 23a 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 23c) – 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 9 and the exemplary plots in Figure 22, 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 23b 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. Figure 23d 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 9 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. 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).

5.4.4 OVERALL ANALYSIS OF GRID NETWORK PROBLEMS

Table 10 shows parallel speedup of parallel AOBB on grid network instances for a subset of fixed depths d and corresponding variable-depth subproblem count. 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 Table 10, Figure 24 plots the speedup results for a subset of instances.

				Cutoff depth d											
instance	i	T_{seq}	#cpu	5		7		9		11		13		15	
				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		(p=16)		(p=64)		(p=192)		(p=192)		(p=768)		(p=2112)	
			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***
	14	15402		(p=8)		(p=32)		(p=96)		(p=192)		(p=576)		(p=1584)	
			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***
75-25-3 n=624 k=2 w=37 h=115	12	104037		(p=6)		(p=24)		(p=48)		(p=144)		(p=576)		(p=1536)	
			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*
	15	33656		(p=6)		(p=24)		(p=48)		(p=144)		(p=576)		(p=1536)	
			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***
75-25-7 n=624 k=2 w=37 h=120	16	297377		(p=24)		(p=72)		(p=216)		(p=504)		(p=2016)		(p=3360)	
			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***
	18	21694		(p=24)		(p=72)		(p=216)		(p=504)		(p=2014)		(p=3325)	
			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***
75-26-10 n=675 k=2 w=39 h=124	16	46985		(p=16)		(p=32)		(p=128)		(p=384)		(p=768)		(p=1280)	
			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*
	18	26855		(p=16)		(p=48)		(p=160)		(p=480)		(p=960)		(p=1216)	
			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*
75-26-2 n=675 k=2 w=39 h=120	16	25274		(p=24)		(p=96)		(p=288)		(p=640)		(p=1280)		(p=3840)	
			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*
	20	8053		(p=24)		(p=96)		(p=288)		(p=640)		(p=1280)		(p=3840)	
			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*
75-26-6 n=675 k=2 w=39 h=133	10	199460		(p=32)		(p=128)		(p=128)		(p=384)		(p=1152)		(p=4608)	
			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*
	12	64758		(p=32)		(p=128)		(p=128)		(p=384)		(p=1152)		(p=4608)	
			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*
75-26-9 n=675 k=2 w=39 h=124	16	59609		(p=24)		(p=96)		(p=240)		(p=960)		(p=3840)		(p=7680)	
			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*
	18	66533		(p=24)		(p=96)		(p=240)		(p=960)		(p=3840)		(p=7680)	
			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***
	20	5708		(p=24)		(p=96)		(p=240)		(p=640)		(p=2560)		(p=5120)	
			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 10: Subset of parallel **speedup** results on **grid instances**. Each entry lists, from top to bottom, the speedup with 20, 100, (simulated) 500, and "unlimited" 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 bolded. The best value in each row is highlighted in gray.

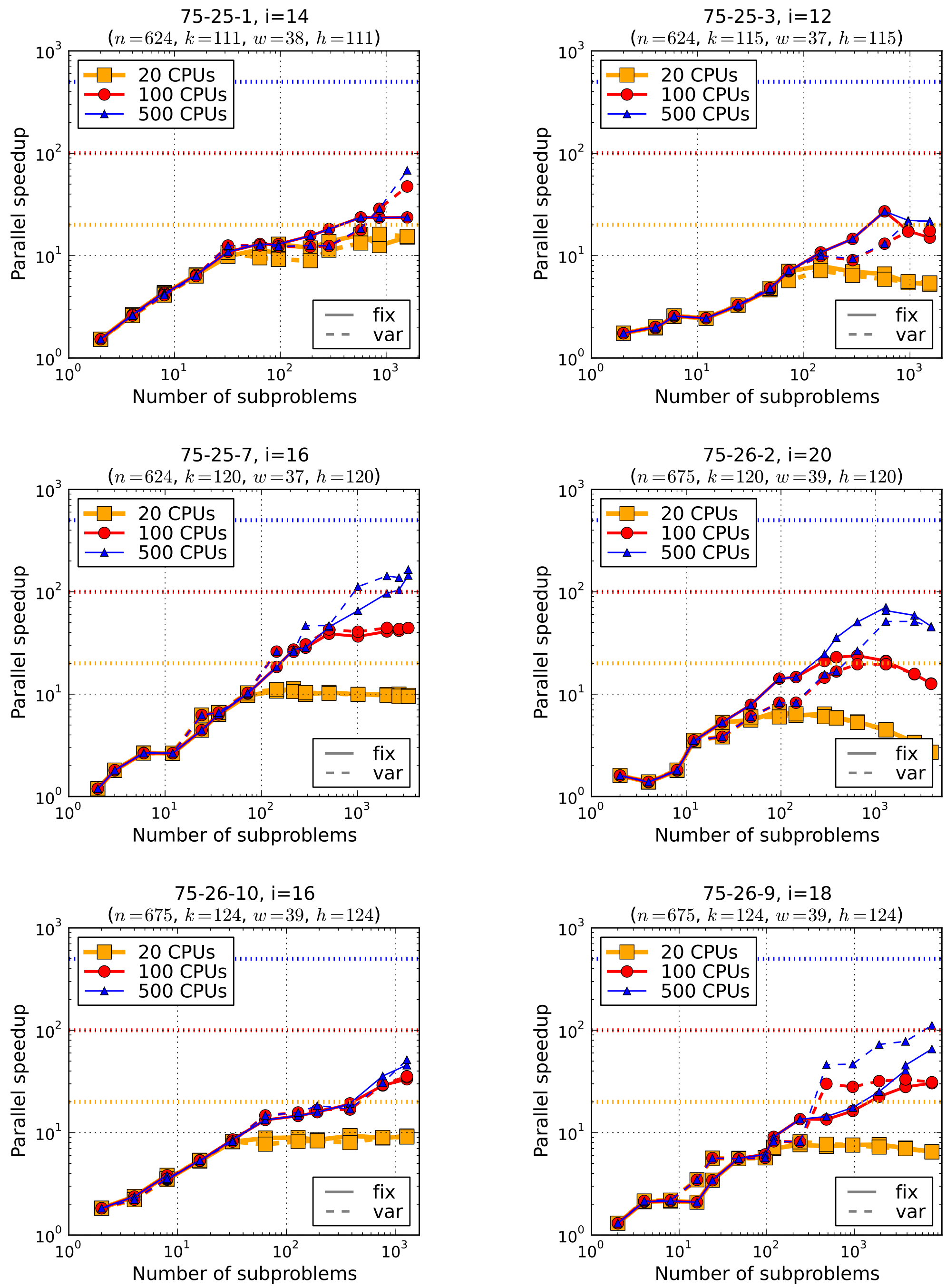


Figure 24: Parallel **speedup** plots for select **grid** problems. Shown is the speedup using 20, 100, and 500 CPUs as a function of subproblem count (corresponding to an increasing fixed-depth cutoff). Optimal speedups 20, 100, and 500 are marked by dashed horizontal lines.

General Performance. Results in Table 10 span a range of outcomes, although generally not as good as results observed, for instance, for grid and haplotyping problems in Sections 5.4.1 and 5.4.2. Using 20 CPUs, the best speedup we obtain is around 16 for 75-25-1 at depth $d = 13$ (sequential time over 21 and 4 hours, respectively, for the tested i-bounds). 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 with $i = 16$ at $d = 15$ ($T_{seq} \approx$ 82.5 hours) 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:

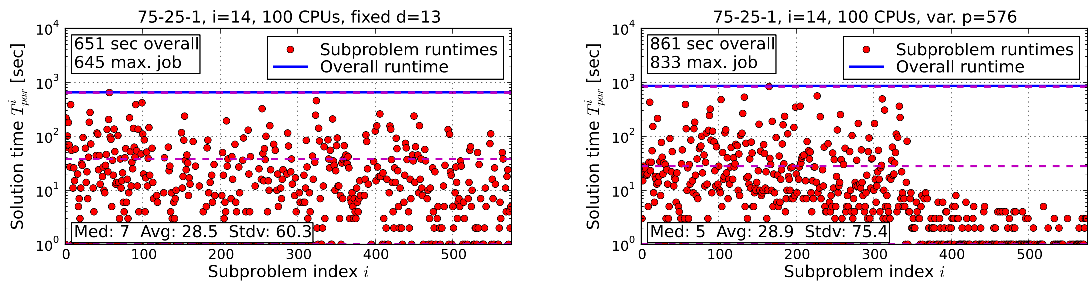
- First, in spite of running with higher cutoff depths, the subproblem count is still relatively low in some cases. In particular for higher CPU counts the number of subproblems does often not meet the "rule of thumb" we described earlier, according to which the number of subproblems should be about 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.

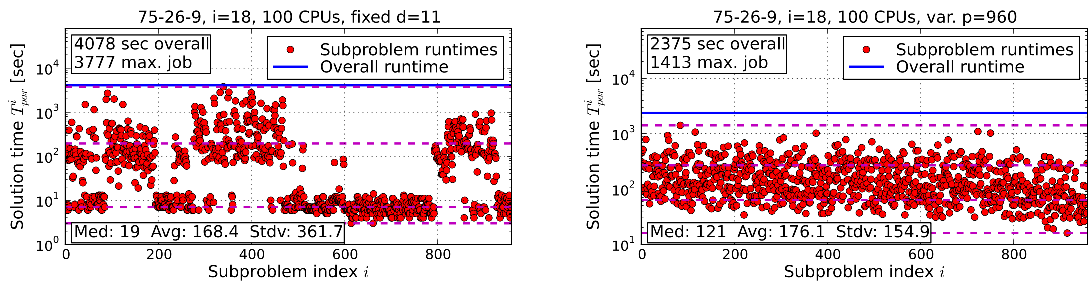
- Most importantly, and in contrast to previously discussed problem classes, experiments on grid networks exhibit a relative large degree of parallel redundancies, as defined in Section 4. This section will mention this only briefly, however, with full analysis to follow in Section 5.6.

75-25-1. As a first example, we consider problem 75-25-1 with $i = 14$ ($T_{seq} \approx$ 4.3 hours) – detailed subproblem statistics for fixed-depth $d = 13$ and corresponding variable-depth parallelization $p = 576$ are shown in Figure 25a. 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 25c 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.

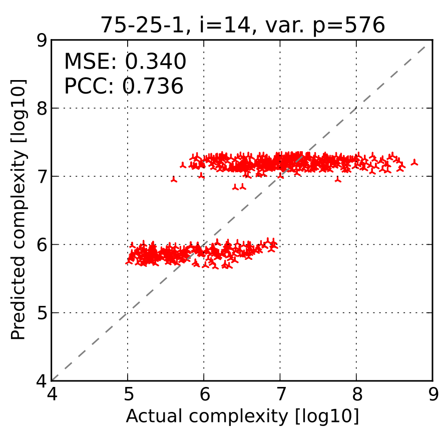
75-26-9. One of the few cases where the variable-depth scheme works better than the fixed-depth one is instance 75-26-9 ($i = 18$, sequential time 16.5 hours). Figure 25b shows subproblem statistics for fixed-depth (left) and variable-depth (right) parallelization. In this instance variable-depth performs 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 25d shows the corresponding scatter plot of



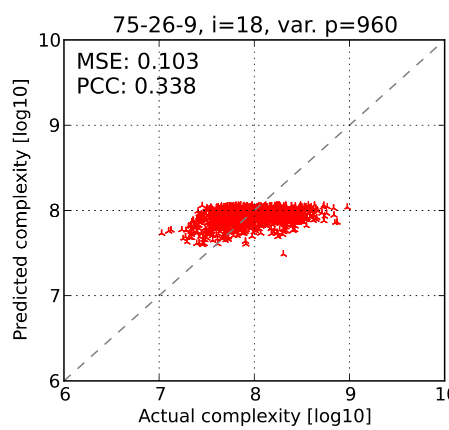
(a) Runtime statistics of individual subproblems *of 75-25-1* using 100 CPUs; left: fixed-depth run with cutoff $d = 13$, right: corresponding variable-depth run with subproblem count $p = 576$.



(b) Runtime statistics of individual subproblems *of 75-26-9* using 100 CPUs; left: fixed-depth run with cutoff $d = 11$, right: corresponding variable-depth run with subproblem count $p = 960$.



(c) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run on 75-25-1 with $p = 576$ subproblems.



(d) Scatter plot of actual vs. predicted subproblem complexities for variable-depth parallel run on 75-26-9 with $p = 960$ subproblems.

Figure 25: 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, as well 75-26-9 instance ($i = 18$) with $d = 11$ and corresponding $p = 960$ subproblems

				Cutoff depth d											
instance	i	T_{seq}	#cpu	2		4		6		8		10		12	
				fix	var	fix	var	fix	var	fix	var	fix	var	fix	var
75-25-1 (n=624, k=2, w=38, h=111)	14	15402		*(p=4)*		*(p=8)*		*(p=16)*		*(p=64)*		*(p=96)*		*(p=288)*	
			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
75-26-9 (n=675, k=2, w=39, h=124)	18	66533		*(p=4)*		*(p=16)*		*(p=48)*		*(p=120)*		*(p=480)*		*(p=1920)*	
			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**
lF3-15-59 (n=3730, k=3, w=31, h=84)	19	43307		*(p=4)*		*(p=20)*		*(p=80)*		*(p=476)*		*(p=1830)*		*(p=6964)*	
			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
lF3-16-56 (n=3930, k=3, w=38, h=77)	15	1891710		*(p=3)*		*(p=15)*		*(p=71)*		*(p=470)*		*(p=934)*		*(p=2707)*	
			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**
lF4-12-55 (n=2926, k=4, w=28, h=78)	13	104837		*(p=4)*		*(p=16)*		*(p=128)*		*(p=512)*		*(p=1024)*		*(p=1792)*	
			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
pdb1huw (n=152, k=81, w=15, h=43)	3	545249		*(p=42)*		*(p=654)*		*(p=2597)*							
			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**						
pdb1nfp (n=204, k=81, w=18, h=38)	3	354720		*(p=48)*		*(p=3812)*									
			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**								
ped19 (n=793, k=5, w=25, h=98)	16	375110		*(p=12)*		*(p=144)*		*(p=1440)*		*(p=5752)*		*(p=11254)*			
			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**		
ped44 (n=811, k=4, w=25, h=65)	6	95830		*(p=4)*		*(p=16)*		*(p=112)*		*(p=560)*		*(p=2240)*		*(p=8960)*	
			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
ped7 (n=1068, k=4, w=32, h=90)	6	118383		*(p=4)*		*(p=32)*		*(p=160)*		*(p=640)*		*(p=1280)*		*(p=3840)*	
			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 11: 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.

actual vs. predicted subproblem complexity, which has notably better prediction quality than what we observed for instance 75-25-1 in Figure 25c.

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.

5.5 Parallel Resource Utilization

This section will consider the parallel resource utilization, which we defined in Section 2.3 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 11 shows a subset of parallel resource utilization values for those problem instances that were used as examples above (complete set of results provided separately by Otten, 2013). 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.

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 CondorHT 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, quite 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 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 (cf. Otten, 2013). 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 5 and 6).

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 (Otten, 2013). 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 7 and 8).

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 (computed separately) by at least 500% in 9 cases. The corresponding case counts of the parallel speedup results in Table 9 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 10), 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.

5.6 Parallel Redundancies

In this section we investigate the issue of parallel redundancies, as discussed and analyzed in detail in Section 4. 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 4.1. Secondly, some degree of caching of context-unifiable subproblems is lost across subproblems – Section 4.2 provided detailed analysis and examples.

In Section 4.2 we also derived an expression $SS_{par}(d)$ (Equation 6) 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, 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.

5.6.1 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 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 by Otten and Dechter (2012a), respectively.

To that end Figures 26 through 29 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).

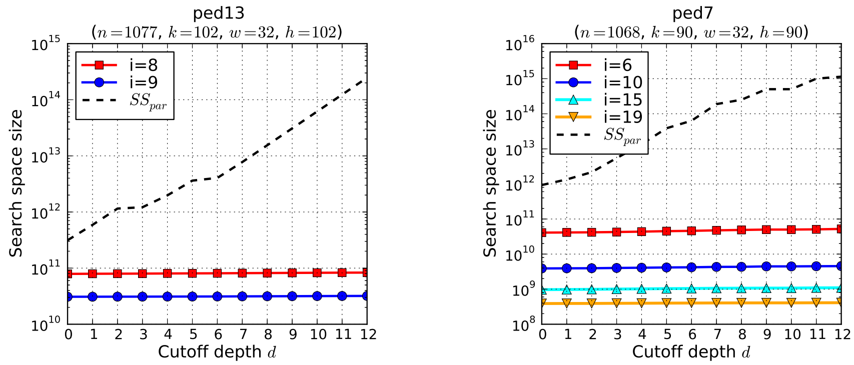


Figure 26: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **pedigree instances**.

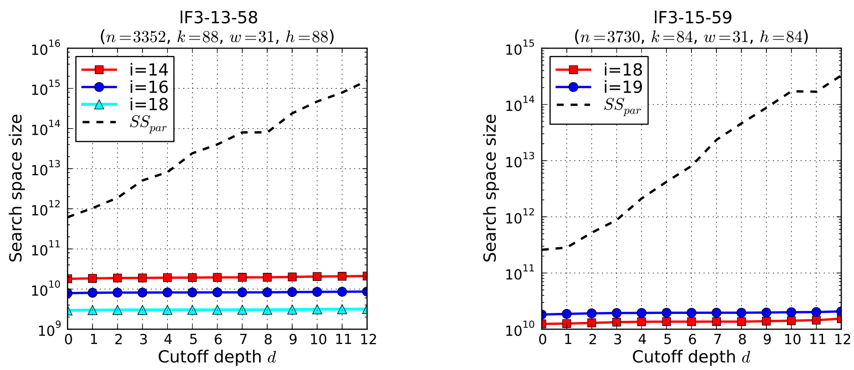


Figure 27: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **haplotyping instances**.

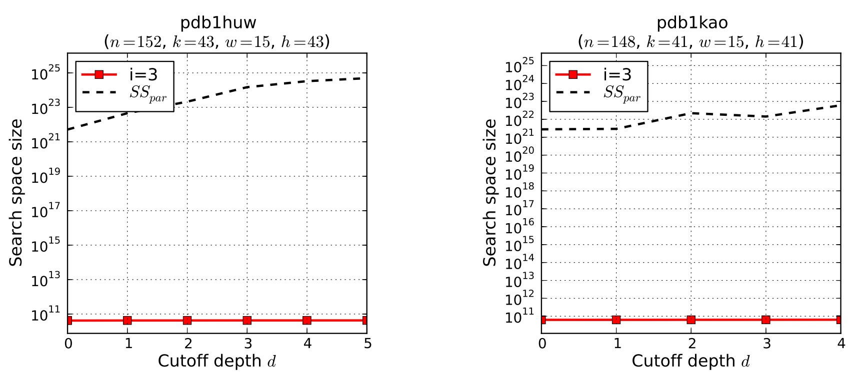


Figure 28: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **side-chain prediction instances**.

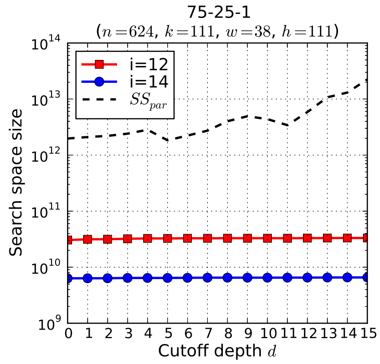


Figure 29: Comparison of the parallel state space upper bound $SS_{par}(d)$ against the actual number of node expansions $N_{par}(d)$ by parallel AOBB with various i-bounds, summed across subproblems, on **grid network instances**.

Upper Bound $SS_{par}(d)$**.** A number of observations can be made regarding $SS_{par}(d)$ across all instances in Figures 26 through 29. First, we can again confirm one of the central premises, namely that for sequential AOBB (corresponding to $d = 0$) the state space bound is loose by several orders of magnitude (cf. Otten & Dechter, 2012a) – this is most extreme for side-chain prediction instances in Figure 28, 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 4.2 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 29), where a growth of $N_{par}(d)$ is visible on the log scale, albeit still slower than the upper bound (Section 5.6.2 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 26 through 29 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 26, right), where the $i = 19$ expands approximately two orders of magnitude fewer nodes than the $i = 6$. Similarly, for the grid instance 75-26-10 in Figure 29 (right) the difference between $i = 15$ and $i = 20$ is almost one order of magnitude in expanded number of nodes.

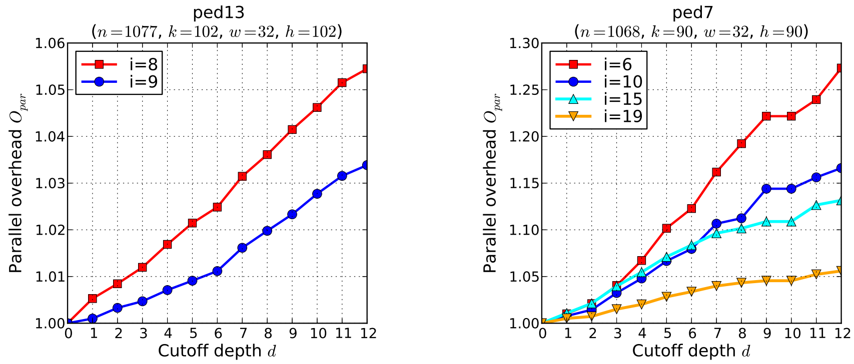


Figure 30: Parallel overhead $O_{par}(d)$ of the parallel scheme on **pedigree instances**, relative the number of node expansions of sequential AOBB.

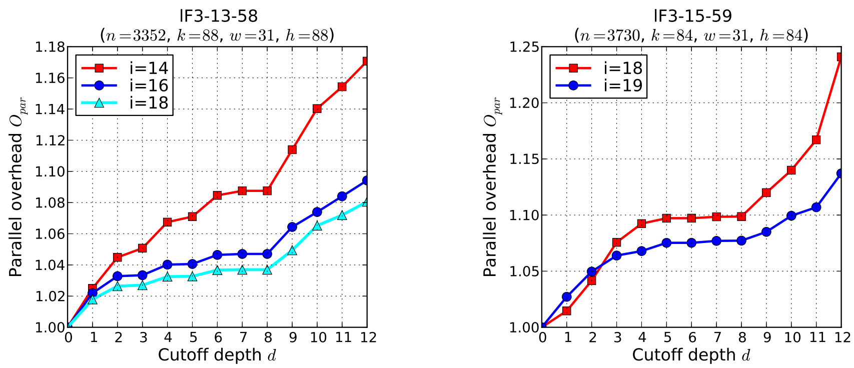


Figure 31: Parallel overhead $O_{par}(d)$ of the parallel scheme on **haplotyping instances**, relative the number of node expansions of sequential AOBB.

5.6.2 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 2.3 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 30 through 33 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 30, 31, and 32, respectively, we observe low overhead values close to 1.0 – pedigree7 with $i = 6$ here sees the highest 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 $SS_{par}(d)$.

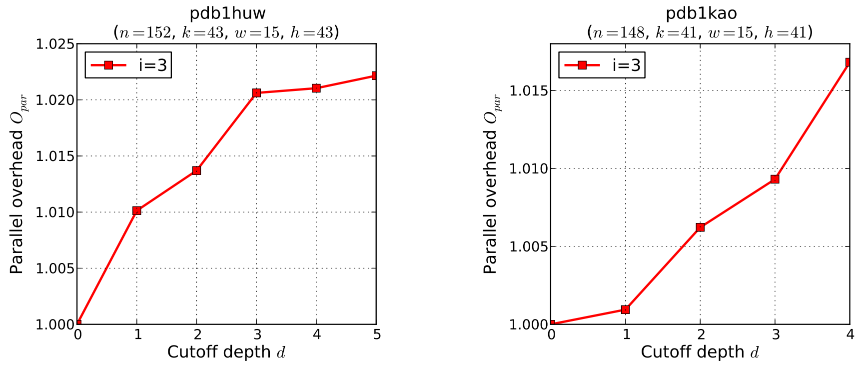


Figure 32: Parallel overhead $O_{par}(d)$ of the parallel scheme on **side-chain prediction instances**, relative the number of node expansions of sequential AOBB.

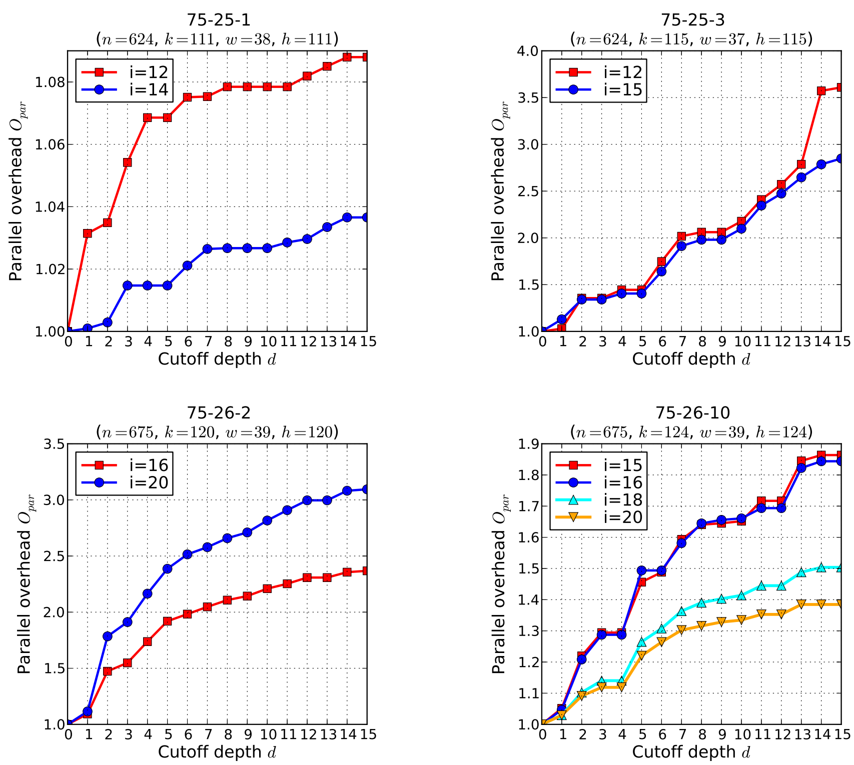


Figure 33: Parallel overhead $O_{par}(d)$ of the parallel scheme on **grid network instances**, relative the number of node expansions of sequential AOBB.

Parallel Overhead on Grid Networks. We observe slightly different results for grid networks in Figure 33. 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, for deeper cutoff depths d. To make the impact on parallel performance more explicit, we can formulate the following straightforward proposition.

PROPOSITION **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 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 10.

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.

5.7 Parallel Scaling

This section addresses the question how parallel performance scales with the number of CPUs. This information was already contained in Section 5.4. 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 34 through 37 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) for 50, 200, 300, and 400 CPUs.

Results in Section 5.4 and throughout this article 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 $10 \cdot c$ 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 35 (left) is taken to be 17.77 from the $d = 7$ column in Table 8, 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 8, cf. Otten, 2013) which in case of largeFam3-15-59 has $p = 476$ subproblems.

Pedigree Linkage Instances. Figure 34 show scaling results for three pedigree instances, reflecting what we pointed out in Section 5.4.1 already. Both pedigree7 and pedigree19 are instances

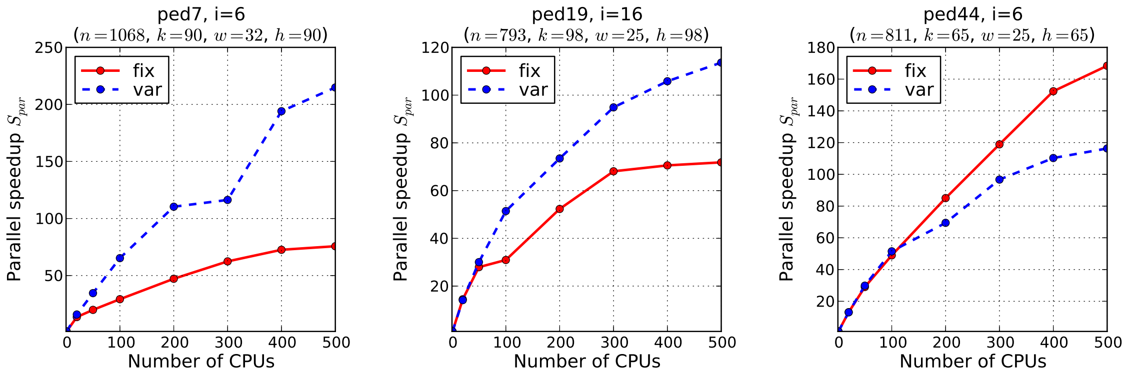


Figure 34: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **pedigree instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

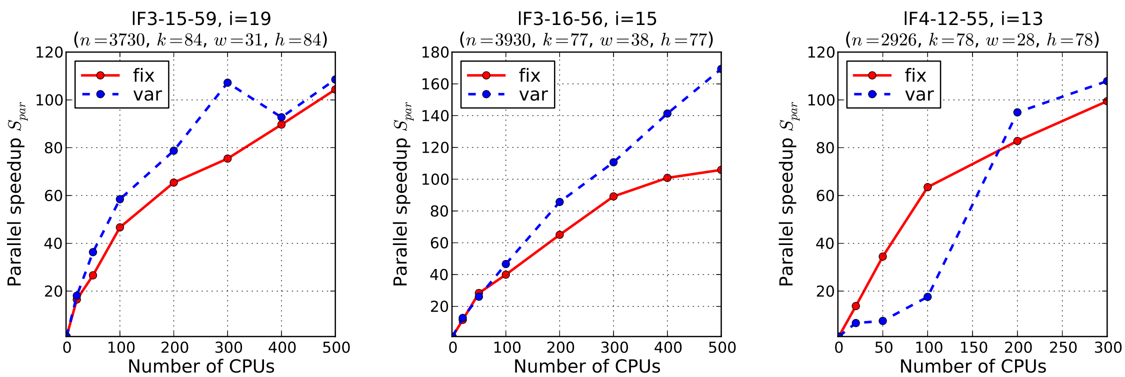


Figure 35: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **haplotyping instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

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 34.

LargeFam Haplotyping Instances. Scaling results for three haplotyping problems are depicted in Figure 35. 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 21, 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.)

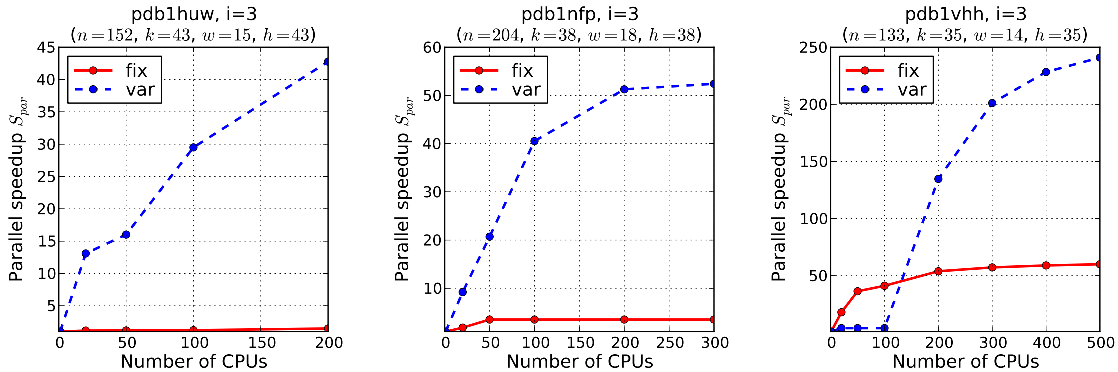


Figure 36: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **side-chain prediction instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

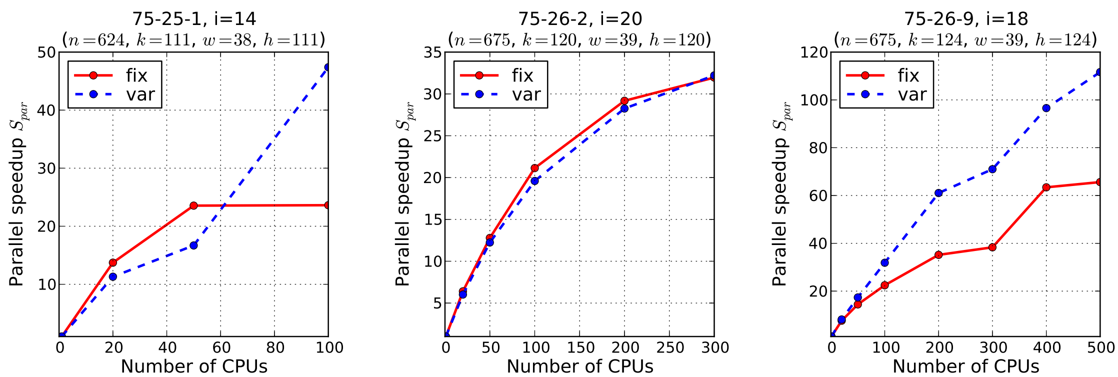


Figure 37: Scaling of parallel speedup with the number of CPUs, using fixed-depth or variable-depth parallelization on select **grid network instances**. Subproblem count chosen to be (approximately) 10 times the number of CPUs.

Pdb Protein Side-chain Prediction Instances. We plot scaling results for three side-chain prediction instances in Figure 36. As mentioned in Section 5.4.3, 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 36 (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.

Grid Network Instances. Finally, Figure 37 show scaling results for three grid network instances. Section 5.4.4 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. 37, 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.

5.8 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 by Otten and Dechter (2012a) in order to improve parallel load balancing. We have considered four different problem classes which we characterized in Section 5.2, with experiments conducted on a variety of problem configurations. We have furthermore investigated a number of metrics and performance measures in Sections 5.3 through 5.7, from parallel runtime and speedup to resource utilization and parallel overhead. In the following we put these results into a common, overall context.

5.8.1 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 parallelism to solve harder and harder problem instances, with more and more CPUs, requires 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 5.5, 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 **repeated preprocessing** and **distributed processing overhead** like communication delays becomes very noticeable. We have observed this in many cases of relatively simple problems in Section 5.4, 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 closely related

to **Amdahl's Law** (cf. Section 2.4); our experiments in Section 5.4 have yielded some indication of this, again mostly on simpler problem instances.

– Third, following the theoretical analysis of Section 4, Section 5.6 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. As outlined above, the results in Section 5.4 suggest that targetting a number of subproblems roughly ten times the number of workers achieves a good balance in this regard. For reference, Régin et al. (2013) found a factor of 30 to work best for Embarrassingly Parallel Search (EPS).

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.

5.8.2 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-depth 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 5.6 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.

6. Conclusion

This article 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 by Otten and Dechter (2012a) (cf. Section 3.4) 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, though not fully applicable here, and issues like grid communication delay become a detrimental factor at 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. Our current implementation does not generally support this due to technical limitations. But even with these constraints in mind, we still found variable-depth parallelization to greatly outperform the fixed-depth scheme.

Overall, we also reiterate that the performance of the parallel scheme depends to a large extend on the accuracy of the underlying subproblem estimates. And as noted by Otten and Dechter (2012a), prior knowledge of a problem class, in the form of sample subproblems for the complexity model's training set, is clearly advantageous in this regard, which limits our scheme's applicability to previously unseen problem classes.

Nevertheless 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, some practically infeasible before, orders of magnitude faster than the already very advanced and award-winning sequential AND/OR Branch-and-Bound.

6.1 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) (Fishelson et al., 2005; Silberstein, 2011). 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 CondorHT 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 by Silberstein et al. (2013).

6.2 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). All of this is in addition to expanding the work on the underlying complexity prediction methods, as a continuation of our prior work (Otten & Dechter, 2012a) as well as to incorporate and evaluate other recent advances (Lelis, Otten, & Dechter, 2013, 2014).

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 section 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.

Acknowledgments

Many thanks to Dan Geiger and Mark Silberstein for fruitful collaboration and helpful discussions over several years. We would also like to thank the anonymous reviewers for their constructive comments which improved this article.

This work was partially supported by NSF grant IIS-1526842 and NIH grant 5R01HG004175-03. Also supported in part by the Israeli Science Foundation.

References

Allen, D., & Darwiche, A. (2008). RC_Link: Genetic linkage analysis using Bayesian networks. *International Journal of Approximate Reasoning*, *48*(2), 499–525.

Allouche, D., de Givry, S., & Schiex, T. (2010). 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*, pp. 53–60.

Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, *30*, 483–485.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., & Werthimer, D. (2002). SETI@home: an experiment in public-resource computing. *Communications of the ACM*, *45*(11), 56–61.

Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., & Pande, V. S. (2009). Folding@home: Lessons from eight years of volunteer distributed computing. In *23rd IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8.

Bergman, D., Ciré, A. A., Sabharwal, A., Samulowitz, H., Saraswat, V. A., & van Hoeve, W. J. (2014). Parallel Combinatorial Optimization with Decision Diagrams. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming*, pp. 351–367.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Campbell, M., Hoane Jr., A. J., & Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, *134*(1-2), 57–83.

Cazenave, T., & Jouandeau, N. (2008). A Parallel Monte-Carlo Tree Search Algorithm. In *6th International Conference on Computers and Games*, pp. 72–80.

Challou, D. J., Gini, M. L., & Kumar, V. (1993). Parallel Search Algorithms for Robot Motion Planning. In *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 46–51.

Chaslot, G. M. J. B., Winands, M. H. M., & van den Herik, H. J. (2008). Parallel Monte-Carlo Tree Search. In *6th International Conference on Computers and Games*, pp. 60–71.

Chu, G., Schulte, C., & Stuckey, P. J. (2009). Confidence-Based Work Stealing in Parallel Constraint Programming. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pp. 226–241.

Chu, G., & Stuckey, P. J. (2008). PMiniSAT: A Parallelization of MiniSAT 2.0. Tech. rep., SAT-race 2008 solver descriptions.

Darwiche, A., Dechter, R., Choi, A., Gogate, V., & Otten, L. (2008). UAI 2008 Probabilistic Inference Evaluation. http://graphmod.ics.uci.edu/uai08/.

Davis, M., Logemann, G., & Loveland, D. W. (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394–397.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation*, pp. 137–150.

Dechter, R. (2013). *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms*. Morgan & Claypool Publishers.

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

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

Draper, N. R., & Smith, H. (1998). *Applied Regression Analysis* (3rd edition). Wiley-Interscience.

Drozdowski, M. (2009). *Scheduling for Parallel Processing*. Springer.

Ebcioglu, K., Saraswat, V., & Sarkar, V. (2004). X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In *International Workshop on Language Runtimes, OOPSLA*.

Elidan, G., & Globerson, A. (2010). UAI 2010 Approximate Inference Challenge. http://www.cs.huji.ac.il/project/UAI10/.

Elidan, G., Globerson, A., & Heinemann, U. (2012). PASCAL 2011 Probabilistic Inference Challenge. http://www.cs.huji.ac.il/project/PASCAL/.

Evett, M. P., Hendler, J. A., Mahanti, A., & Nau, D. S. (1995). PRA*: Massively Parallel Heuristic Search. *J. Parallel Distrib. Comput.*, *25*(2), 133–143.

Ferguson, C., & Korf, R. E. (1988). Distributed Tree Search and Its Application to Alpha-Beta Pruning. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pp. 128–132.

Fishelson, M., Dovgolevsky, N., & Geiger, D. (2005). Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, *59*, 41–60.

Fishelson, M., & Geiger, D. (2002). Exact genetic linkage computations for general pedigrees.. *Bioinformatics*, *18*(suppl 1), S189–S198.

Fishelson, M., & Geiger, D. (2004). Optimizing exact genetic linkage computations.. *Journal of Computational Biology*, *11*(2-3), 263–275.

Foster, I., & Kesselmann, C. (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc.

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, *1*(2), 117–129.

Gendron, B., & Crainic, T. G. (1994). Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research*, *42*(6), 1042–1066.

Ghosh, S. (2006). *Distributed Systems: An Algorithmic Approach*. Chapman and Hall/CRC.

Gogate, V. (2014). UAI 2014 Inference Competition. http://www.hlt.utdallas.edu/ vgogate/uai14-competition/.

Graham, R. L. (1969). Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, *17*(2), 416–429.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd edition). Addison-Wesley.

Grama, A., & Kumar, V. (1995). Parallel Search Algorithms for Discrete Optimization Problems. *INFORMS Journal on Computing*, *7*(4), 365–385.

Grama, A., & Kumar, V. (1999). State of the Art in Parallel Search Techniques for Discrete Optimization Problems.. *IEEE Trans. Knowl. Data Eng.*, *11*(1), 28–35.

Hamadi, Y., & Wintersteiger, C. M. (2012). Seven Challenges in Parallel SAT Solving. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*.

Harvey, W. D., & Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 607–613.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd edition). Springer.

Hutter, F., Hoos, H. H., & Stützle, T. (2005). Efficient Stochastic Local Search for MPE Solving. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 169–174.

Ihler, A., Flerova, N., Dechter, R., & Otten, L. (2012). Join-graph based cost-shifting schemes. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*.

Jurkowiak, B., Li, C. M., & Utard, G. (2005). A Parallelization Scheme Based on Work Stealing for a Class of SAT Solvers. *J. Autom. Reasoning*, *34*(1), 73–101.

Kask, K., & Dechter, R. (2001). A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, *129*(1-2), 91–131.

Kask, K., Dechter, R., Larrosa, J., & Dechter, A. (2005). Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, *166*(1-2), 165–193.

Kask, K., Gelfand, A., Otten, L., & Dechter, R. (2011). Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models.. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*.

Kjaerulff, U. (1990). Triangulation of Graphs – Algorithms Giving Small Total State Space. Tech. rep., Dept. of Mathematics and Computer Science, Aalborg University.

Kumar, V., Grama, A., & Vempaty, N. R. (1994). Scalable Load Balancing Techniques for Parallel Computers. *J. Parallel Distrib. Comput.*, *22*(1), 60–79.

Kumar, V., & Rao, V. N. (1987). Parallel depth first search. Part II. Analysis.. *International Journal of Parallel Programming*, *16*(6), 501–519.

Lelis, L. H. S., Otten, L., & Dechter, R. (2013). Predicting the Size of Depth-First Branch and Bound Search Trees. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pp. 594–600.

Lelis, L. H. S., Otten, L., & Dechter, R. (2014). Memory-Efficient Tree Size Prediction for Depth-First Search in Graphical Models. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, pp. 481–496.

Linderoth, J., Kulkarni, S., Goux, J.-P., & Yoder, M. (2000). An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pp. 43–50.

Lüling, R., & Monien, B. (1992). Load Balancing for Distributed Branch and Bound Algorithms. In *Proceedings of the 6th International Parallel Processing Symposium*, pp. 543–548.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc.

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

Marinescu, R., & Dechter, R. (2009b). Memory intensive AND/OR search for combinatorial optimization in graphical models. *Artificial Intelligence*, *173*(16-17), 1492–1524.

Modi, P. J., Shen, W.-M., Tambe, M., & Yokoo, M. (2005). Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, *161*(1-2), 149–180.

Otten, L. (2013). *Extending the Reach of AND/OR Search for Optimization in Graphical Models*. Ph.D. thesis, University of California, Irvine.

Otten, L., & Dechter, R. (2012a). A Case Study in Complexity Estimation: Towards Parallel Branch-and-Bound over Graphical Models. In *Proceedings of the 28th Conference in Uncertainty in Artificial Intelligence*.

Otten, L., & Dechter, R. (2012b). Anytime AND/OR Depth-first Search for Combinatorial Optimization. *AI Communications*, *25*(3).

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems*. Morgan Kaufmann Publishers Inc.

Powley, C., Ferguson, C., & Korf, R. E. (1993). Depth-First Heuristic Search on a SIMD Machine. *Artificial Intelligence*, *60*(2), 199–242.

Prosser, P., & Unsworth, C. (2011). Limited discrepancy search revisited. *J. Exp. Algorithmics*, *16*.

Régin, J., Rezgui, M., & Malapert, A. (2013). Embarrassingly Parallel Search. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, pp. 596–610.

Régin, J., Rezgui, M., & Malapert, A. (2014). Improvement of the Embarrassingly Parallel Search for Data Centers. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, pp. 622–635.

Seber, G. A. F., & Lee, A. J. (2003). *Linear Regression Analysis* (2nd edition). John Wiley & Sons, Inc.

Shimony, S. E. (1994). Finding MAPs for Belief Networks is NP-Hard. *Artificial Intelligence*, *68*(2), 399–410.

Silberstein, M. (2011). 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*, pp. 174–183.

Silberstein, M., Tzemach, A., Dovgolevsky, N., Fishelson, M., Schuster, A., & Geiger, D. (2006). 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.

Silberstein, M., Weissbrod, O., Otten, L., Tzemach, A., Anisenia, A., Shtark, O., Tuberg, D., Galfrin, E., Gannon, I., Shalata, A., Borochowitz, Z. U., Dechter, R., Thompson, E., & Geiger, D. (2013). A system for exact and approximate genetic linkage analysis of SNP data in large pedigrees. *Bioinformatics*, *29*(2), 197–205.

Thain, D., Tannenbaum, T., & Livny, M. (2002). Condor and the Grid. In Berman, F., Fox, G., & Hey, T. (Eds.), *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc.

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

Tschöke, S., Lüling, R., & Monie, B. (1995). 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*, pp. 182–189.

Wallace, R. J. (1996). Analysis of heuristic methods for partial constraint satisfaction problems. In *Proc. of the 2nd International Conference on Principles and Practice of Constraint Programming (CP 1996)*, pp. 482–496. Springer, Berlin.

Yanover, C., Schueler-Furman, O., & Weiss, Y. (2008). Minimizing and learning energy functions for side-chain prediction.. *Journal of Computational Biology*, *15*(7), 899–911.

Yeoh, W., Felner, A., & Koenig, S. (2010). BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *J. Artif. Intell. Res. (JAIR)*, *38*, 85–133.

Yeoh, W., & Yokoo, M. (2012). Distributed Problem Solving. *AI Magazine*, *33*(3), 53–65.

Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowl. Data Eng.*, *10*(5), 673–385.

Zivan, R., & Meisels, A. (2005). Dynamic Ordering for Asynchronous Backtracking on DisCSPs. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, pp. 32–46.