Parallelizing $\mathcal{NP}$-Complete Problems Using Tree Shaped Computations

Peter Sanders
Max-Planck-Institut für Informatik,
Im Stadtwald, 66123 Saarbrücken, Germany.
E-mail: sanders@mpi-sb.mpg.de

May 5, 1999

Abstract

We explain how the parallelization aspects of a large class of applications can be modeled as *tree shaped computations*. This model is particularly suited for $\mathcal{NP}$-complete problems. One reason for this is that any computation on a nondeterministic machine can be emulated on a deterministic machine using a tree shaped computation. We then proceed to a particular example, the knapsack problem It turns out that a parallel depth first branch-and-bound algorithm based on tree shaped computations yields superlinear average speed-up using 1024 processors. This even holds for large relatively easy problems which produce a very irregular search tree and only a moderate amount of work.

1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking or depth first traversal principle for traversing large implicitly defined trees (e.g. [1, 11, 12, 14, 16, 15, 18]). In addition, some adaptive numerical algorithms for integration [21], for finding eigenvalues of tridiagonal matrices [7] or for nonlinear optimization [20] have a similar structure. Even modeling the seemingly unrelated problem of loop scheduling in this way can be advantageous [30]. In Section 2, we first review the model of tree shaped computations which exposes the common properties of the above mentioned application. Section 3 gives an overview of the quite strong algorithmic techniques available for tree shaped computations. Having established that tree shaped computations are both simple and and efficiently parallelizable we go into further detail regarding relations between tree shaped computations on the one hand and the particular case of $\mathcal{NP}$-complete problems on the other hand. Perhaps the most fundamental relation is that every problem in $\mathcal{NP}$ can easily be translated into a tree shaped computatatoin of polynomial depth. Section 4 explains the details. An even more trivial relation is that $\mathcal{NP}$-complete problems can be time

consuming to solve, so that parallelization might help boost performance. The main weakness of this argumentation is that even hundreds of processors are of little help if the execution time for real world instances actually grows exponentially with a basis significantly larger to one. Therefore, in Section 5 we discuss a number of heuristics which can be used to prune the search space. We conclude our discussion in Section 6 by looking at a particular example, the knapsack problem, for which pruning heuristics can be very successful. This makes parallelization difficult yet interesting.

Related Work

Most of the results presented here stem from the PhD thesis [28]. Modelling aspects beyond $\mathcal{NP}$-complete problems are also discussed in [30]. Algorithmic aspects are covered in [24, 23, 26]. It should also be noted that tree shaped computations are also a useful abstraction for a portable and reusable load balancing library [27].

There is a large body of other related work. We can only give a cross section and refer to the references in [28] for a more detailed discussion. Early work on random polling and an application independent library is described in [12]. Random polling and other receiver initiated load balancing methods are also of central importance for parallel functional and logical programming languages (e.g., [1, 14]). Tree shaped computations can be considered a generalization of the α-splitting model used in [16]. A related model based on multithreaded computations is used in the Cilk project [4, 3, 2]. The ZRAM library [6] is another recent implementation effort.

2 The Abstract Model

All the work to be done by a tree shaped computation is initially subsumed in a single *root problem* I_{root} located on a processing element (PE) numbered 0. All other PEs start idle, i.e., they only have an *empty problem* $I_\emptyset$.

What makes parallelization attractive, is the property that problem instances can be subdivided into *subproblems* which can be solved independently by different PEs. We model this property by a *splitting operation* split(I) which splits a given (sub)problem into two new subproblems subsuming the parent problem. Let T_{split} denote a bound on the time required for the split operation.

The operation work(I, t) transforms a given subproblem I by performing sequential work on it for t time units. The operation also returns when the subproblem is exhausted.

What makes parallelization difficult, is that the *size*, i.e., the execution time $T(I) := \min\{t \mid \text{work}(I, t) = I_\emptyset\}$, of a subproblem cannot be predicted. In addition, the splitting operation will rarely produce subproblems of equal size. For the analysis we assume however that subproblems are independent in the sense that

$$\forall I : \text{split}(I) = (I_1, I_2) \Rightarrow T(P) = T(I_1) + T(I_2) \tag{1}$$

regardless when and where I_1 and I_2 are worked on. In Section 5.4 we discuss what happens if this assumption is violated.

Next we quantify some guaranteed "progress" made by splitting subproblems. Every subproblem I belongs to a *generation* $\mathrm{gen}(I)$ recursively defined by $\mathrm{gen}(I_{\mathrm{root}}) := 0$ and $\mathrm{split}(I) = (I_1, I_2) \Rightarrow \mathrm{gen}(I_1) = \mathrm{gen}(I_2) = \mathrm{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* h which guarantees that the size of subproblems with $\mathrm{gen}(I) \geq h$ cannot exceed some *atomic grain size* T_{atomic}. Since h is the only factor which constrains the shape of the emerging "subproblem splitting tree", it can be viewed as a measure for the irregularity of a problem instance.[1]

Finally, subproblems can be moved to other PEs by sending a message.

3 Load Balancing Algorithms

In [28] a number of load balancing algorithms are investigated using a detailed model for message passing parallel computers with P PEs coupled via various interconnection networks. Here, we contend ourselves with an outline of the practically most important algorithm using a simplified version of the the LogP model [8] as the machine model. The communication costs are expressed in terms of $T_{\mathrm{rout}} := L + o + g$, i.e., the sum of communication latency, sending overhead and gap between messages. We assume that the characteristic message length is defined in such a way that a subproblem can be specified using a single message.

The *random polling* algorithm is very simple: Each PE handles exactly one (possibly empty) subproblem at any point in time. If a PE runs out of work it sends requests to randomly chosen PEs until a busy one is found which splits its piece of work and transmits it to the requester. This algorithm was discovered independently multiple times. Refer to [12] for an early reference. Despite of its simplicity and the unpredictablility of tree shaped computations, random polling is very efficient:

Theorem 1 *The expected parallel execution time for solving a tree shaped computation using random polling is*

$$\mathbf{E}T_{\mathrm{par}} \leq (1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + \mathrm{O}\left(T_{\mathrm{atomic}} + h\left(\frac{1}{\epsilon} + T_{\mathrm{rout}} + T_{\mathrm{split}}\right)\right) \quad \textit{for any } \epsilon > 0 \ .$$

In particular, $\mathrm{O}(h)$ consecutive splitting and routing operations and an overall message traffic in $\mathrm{O}(Ph)$ are sufficient. This is optimal in the sense that that there are tree shaped computations which require at least as many splits [29].

The algorithm also works well if the speed of the PEs in a network of workstation varies dynamically due to external load since the additional irregularity introduced by this is comparably small [29, 2]. We can even tolerate a complete

[1] Obviously, very regular instances with large h are possible. But in applications where this is frequently the case, one would look for a splitting function exploiting these regularities to decrease h.

deactivation of a worker process as long as it still answers load requests. Such a mode is desirable for "guest" jobs on interactively used workstations. Even the time for splitting subproblems can be saved if we introduce the additional rule that a deactivated worker process sends its entire subproblem when it gets a request.

4 Relation to $\mathcal{NP}$-complete Problems

By definition of the complexity class $\mathcal{NP}$, a class of decision problems $\mathcal{L}$ is in $\mathcal{NP}$ if and only if there is nondeterministic Turing machine M accepting the language defined by $\mathcal{L}$ using polynomially many steps $t(n) \in \text{Poly}(n)$ (with respect to the problem size n [13]).

How can such a nondeterministic computation be emulated on a sequential deterministic machine? Perhaps the most natural approach is outlined in the following pseudo-code.

```
Function accept(t)
    if t = 0 then return fail
    emulate any deterministic computations
    if accepting state reached then return true
    if nondeterministic transition
        for each possible transition do
            make this transition
            if accept(t-1) then return true
    (* Rejecting state or failed subtree *)
    undo the computations just performed
    return false
```

This algorithm enumerates all possible computations of the nondeterministic machine with up to t nondeterministic decisions. By *iterative deepening*, i.e., calling 'accept' with ever increasing t, an answer will eventually be found. From now on we look at a single iteration of this process since in many application a good estimate for the maximal number of nondeterministic decisions is known and since otherwise the last iteration is likely to dominate execution time. The routine 'accept' could be efficiently implemented on a multi-tape Turing machine (with an extra tape for recording undo-information). When moving towards a practical program, we use the RAM (random access machine) model instead and we implement the undo in an efficient problem-specific way.

The memory content of such a program can be viewed as a subproblem in the sense of tree shaped computations. We only have to specify, how a subproblem I should be split: First make two copies I_1 and I_2. Then, consider the first entry in the recursion stack (deepest inside) which still has $k \geq 1$ untried possible state transitions. Manipulate this stack entry and the state of the computations in both copies in such a way that the computation represented by I_1 tries $\lfloor k/2 \rfloor$ of the open alternatives and the currently considered alternative while I_2 tries the remaining $\lceil k/2 \rceil$ alternatives.

Now, a load balancing algorithm like random polling can be used to execute the computation in parallel. Let $s(n)$ define a bound on the memory consumption for a problem of size n. Let $d(n)$ bound the number of steps spent in consecutive deterministic computations, let $i(n)$ denote the maximum number of nondeterministic steps needed and let b denote the maximimum number of alternatives in a nondeterministic step. There is a simple connection between the above parameters and the parameters of tree shaped computations: $T_{\text{atomic}} = \mathrm{O}(d(n))$, $T_{\text{split}} = \mathrm{O}(s(n) + d(n))$ (note that we may have to do $d(n)$ deterministic computations before the next splitting opportunity is found) and $h \leq \lceil \log b \rceil \, i(n)$. The only serious problem is that the sequential execution time depends on the order in which alternatives are tried. For now let us assume that all alternatives must be tried. For example, because most problem instances produce the answer 'false'. Then parallelization is simple. The computations are done in parallel and each processor remembers whether it ever found a 'true'. At the very end, the global or of these values can be computed in time $\mathrm{O}(T_{\text{rout}} \log P)$ to yield the overall result. Theorem 1 therefore bounds the parallel execution time of our emulation program. Section 5 discusses many refinements of this basic approach.

5 Refinements

Whereas the purpose of Section 4 was mainly a theoretical one, we now investigate how tree shaped computations can be useful for practically parallelizing $\mathcal{NP}$-complete problems efficiently. One way to look at this issue is that the less we know about a problem, the more likely is it that the sequential solution strategy will resemble the basic algorithm from Section 4. The more we learn, the more will the solution strategy deviate and the more will parallelization take different approaches. Eventually, all similarity to backtrack search may be lost, for example, if dynamic programming is used or if we even give up the strive for an optimal solution.

The main purpose of this paper is to collect basic generally useful techniques which can be used when relatively few things are known which help to guide the search for an optimal solution.

Section 5.1 gives some more detail on how to split a problem description into two parts and then cover a few details beyond the model of tree shaped computations in sections 5.2 and 5.3. Then we explain the impact of the fact that many search algorithms stop as soon as a solution is found in Seciton 5.4. Finally, we discuss a number of frequently used heuristics for reducing the search space in Section 5.5.

5.1 Splitting Strategies

Figure 1 outlines two useful strategies for splitting the search stack into two independent subproblem. At the top, we see the original state with a solid arrow for each untried choice. At the bottom, we see the two resulting subproblems.

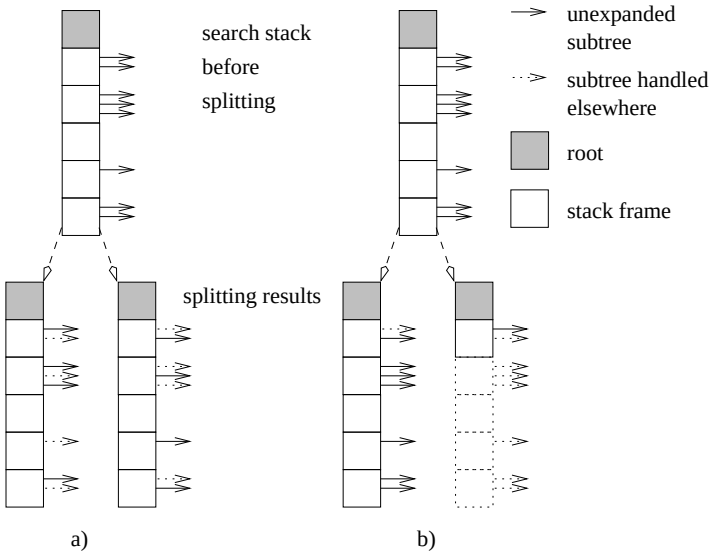

Figure 1: Two stack splitting strategies; a): split on all levels. b): split as close to the root as possible.

The strategy in b) is perhaps the most common one and has already been described in Section 4. It is simple and guarantees good worst case bounds on the maximum splitting depth h.

Strategy a) has similar worst case behavior and is slightly more compolicated but it performs better in some practical cases. It distributes the open alternatives on all levels and is more robust in applications where most subtrees—even those deep in the stack—represent only small subproblems,

5.2 Initialization, Completion and Subproblem Encoding

Our assumption that the entire memory content of the sequential computation has to be copied and transmitted was very conservative. For example, the subproblem description and precomputed tables can be broadcast once and for all in the beginning. Many backtracking algorithms only require a few words of variable memory for each level of recursion. If the communication bandwidth of the system is low, it pays to compress the variable state of the computation before subproblem transmission. For example, the state of a Prolog computation can be compactly represented as a bit string encoding the decisions made by the program [14].

5.3 Result Handling

In practice, optimization problems are much more frequent than plain decision problems. Tree shaped computations are only concerned with how the work is distributed and not how the results are collected. This is usually much easier than the distribution however, because many subproblems will turn out not to

contribute to the result. At worst, we can always retrace the distribution to combine subproblem solutions to an overall solution. This is particularly simple if the function combining the results is associative and commutative because we can then simply use a global reduction operation (e.g., counting solutions, finding one best solution, finding all solutions).

5.4 Early stopping

The sequential algorithm from Section 4 stops as soon as a 'true' is found. Likewise, optimization problems stop as soon as a good solution is found or at least they prune parts of the search tree which are guaranteed to contain only solutions inferior to previously found solutions. Translated into the language of tree shaped computations, finding a solution (e.g., a 'true' in a decision problem) means that all subproblems suddenly become empty.

Before we explain the impact of early stopping on the behavior of the parallel search, we note that search algorithms are often used for verifying that no solution exist, e.g., in order to prove the unsatisfiability of a logical formula [5] or to prove that that a solution found heuristically has no significantly better solution. In this case no anomalies occur.

When solutions exist, the well known phenomenon of *speedup anomalies* can occur, i.e., speedups $S \ll P$ or $S \gg P$ because the parallel algorithm happens to find a solution very late or very early. As long as there are no heuristics ordering the successors of a node by their likelihood to lead to a solution, we can reasonably expect that negative anomalies do not outweigh positive anomalies and the only measure we have to take is to stop all PEs quickly when a solution is found (e.g., [31, 22]). Even with node ordering heuristics, many practical applications work surprisingly well. We can add splitting heuristics which try to produce subproblems which have an about equal chance of leading to a solution. Furthermore, parallel search can even achieve superlinear speedup on the average relative to sequential depth first search since it is less likely to run into dead ends. We give an example in Section 6. More extreme cases of superlinar speedup are analyzed in [9].

5.5 Tree Pruning

Search programs often mainly consist of numerous heuristics[2] to reduce the number and degree of nonderterministic decisions. The effect is some subtrees of the search space are not considered at all – the are *pruned*. The assumptions of tree shaped computations are not violated as long as the decision which paths to take only depend on the path leading from the root to the present node. Only if information obtained by searching a subtree is used to prune siblings, can the speedup be affected.

[2] In this context, "heuristic" means that these rules are not guaranteed to reduce the search space. Nevertheless, the solutions will often remain optimal.

Evading Dead Ends: A path in a backtrack search tree usually represents a sequence of decisions leading to a solution or a dead end. A heuristics that is sometimes useful tries to prove that backtracking a particular decision cannot lead out of the dead end. In this case all alternatives of this decision can be pruned and backtracking proceeds further up the tree. Although this heuristics *can* affect the speedup, it is usually most effective on small subtrees whereas the load balancing mostly involves large subtrees which are not pruned by this heuristics. For an example refer to [25].

Depth-first branch-and-bound behaves similar to applications where we are looking for the first solution. Here, whenever an *improved* solution is found, all other subproblems should learn about the new quality bounds and are thereby reduced in size. We should not simply broadcast new bounds since this can lead to severe contention if many new bound are found concurrently. Rather, the bounds should first be send along a reduction tree to PE 0. Then suboptimal bounds can be thrown away early and we still need only $O(T_{\text{rout}} \log P)$ time to distribute a new globally best bound.

Game-Tree-Search uses th $\alpha\beta$-pruning strategy which heavily depends on the evaluation of sibling trees and is crucial for performance the throughout the searched. Consequently, parallelization is rather difficult. Nevertheless, random polling turned out to be a good load balancing algorithm for game-tree-search [10, 11]. In this case tree shaped computations can still be considered a good model for the load balancing aspect of the application although additional more application-specific models for the "speculativity" aspect are needed.

6 Example: The Knapsack problem

An instance of the *0-1 knapsack problem* is defined by m items with weight w_i and profit p_i and a knapsack of capacity M. We are looking for $x_i \in \{0,1\}$ such that $\sum p_i x_i$ is maximized subject to the constraint $\sum w_i x_i \leq M$, i.e., we want to achieve a maximal profit with items in the knapsack without exceeding its capacity. Next to the traveling salesman problem, the knapsack problem might be one of the most extensively studied discrete optimization problems [18].

There are two basic approaches to exact solutions of the knapsack problem. Dynamic programming is good if m is not too large and the w_i lie within a small discrete range. In other cases, dynamic programming fails due to its exponential memory requirements. For these cases, variants of depth-first branch-and-bound are better. The items are first sorted by their profit-density (from now on, let w_i, p_i refer to the i-th best item); then depth-first branch-and-bound traverses a binary search tree where x_i is determined at level i of the tree. Lower bounds for use in the branch-and-bound heuristics are based on relaxing the integrality constraints on the x_i. The bounds can be computed quickly (in $O(\log m)$ time) using binary search and some precomputation. Due to this fine granularity, best first search is not competitive here. The costs for managing the required

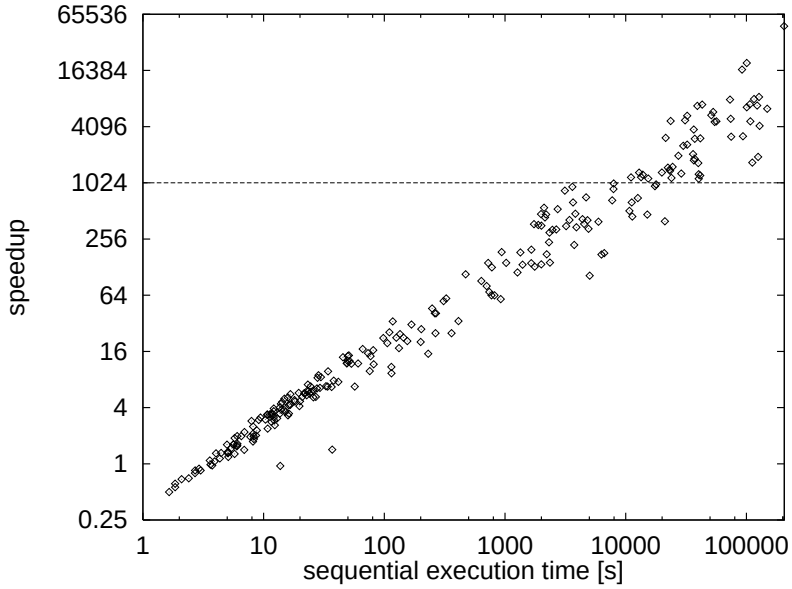

Figure 2: Speedup for 256 instances of the knapsack problem on 1024 PEs.

data structures would be too high. Parallelizing the best first approach is also difficult. In [19] the speedup on 16 PEs remains below 6.

The search space is split by evenly dividing open subproblems on all tree levels between the subproblems. The simpler (and often sufficient) approach of only splitting the top open problem would generate very unequal splits for the knapsack problem.

The standard way for testing the performance of algorithms for the knapsack problem is to generate random instances by choosing w_i uniformly at random from an interval $[w_{\max}, w_{\min}]$. p_i is either choosen independently from an interval $[p_{\min}, p_{\max}]$ or it is correlated to w_i by choosing $p_i \in [w_i + p_{\min}, w_i + p_{\max}]$. The heuristics turns out to be so effective for the uncorrelated instances that the average number of node expansions is close to m – the search tree is almost a linear list. Clearly, no speedup for parallel tree search is possible here.[3] For correlated instances, the shape of the search tree varies widely with the choice of the parameters. There are very simple classes of instances but also difficult ones where $m = 100$ already means intractable problems. We expect hard problems to be easily parallelizable. We focus on sequentially tractable problems with large m which still contain parallelism. The thin, irregular shape of the search tree and the high subproblem transmission cost make this a challenge to the load balancer.

We have generated 256 random instances with $m = 2000$, $w_i \in [0.01, 1.01]$, $p_i \in [w_i + 0.1, w_i + 0.125]$, $M = \sum w_i/2$ using the (32-bit) random number generator of INMOS-C. The double-logarithmic plot in Figure 2 shows the rela-

[3] Speedups reported in [17] for this class of instances are an artifact of a very inefficient node evaluation function.

tion between speedup and sequential execution time. There is a large number of very small problems for which we cannot expect any significant speedup. Beginning at per PE loads of about 10s we start to observe good performance. Very large problems show a considerable superlinear speedup. For these instances the sequential algorithm appears to have run into some kind of "dead end". The parallel algorithm is more robust because it follows multiple search paths at once. The overall parallel execution time for 1024 PEs is 1410 times smaller than the sequential time. This indicates that the traditional pure depth-first strategy is not the best choice for a sequential algorithm.

7 Conclusions

Search problems related to $\mathcal{NP}$-complete problems and many other applications can be modelled by tree shaped computations. With random polling we have an algorithm which parallelizes tree shaped computations very efficiently although very irregular and completely unpredictable computations are allowed. At the same time, the model is the basis for an efficient and very slim interface between the load balancer and a reusable and portable load balancing library [27, 29]. Even if dependencies between subproblems are present, the predictions made by the simple model are often correct and the load balancer works well.

References

[1] G. Aharoni, A. Barak, and Y. Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, 9:163–174, 1993.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[3] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foudations of Computer Science*, pages 356–368, Santa Fe, 1994.

[5] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

[6] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 1998. to appear.

[7] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, 1994.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, 1993.

[9] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.* Dissertation, TU München, 1992.

[10] R. Feldmann. *Game Tree Search on Massively Parallel Systems.* Dissertation, Universität Paderborn, August 1993.

[11] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 94–103, 1994.

[12] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, Apr. 1987.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[15] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[16] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[17] W. Loots and T. H. C. Smith. A parallel algorithm for the 0-1 knapsack problem. *International Journal of Parallel Programming*, 21(5):349–362, 1992.

[18] S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations.* Wiley, 1990.

[19] G. P. McKeown, V. J. Rayward-Smith, and S. A. Rush. Parallel branch-and-bound. In *Advances in Parallel Algorithms*, pages 349–362. Blackwell, 1992.

[20] A. Nonnenmacher and D. A. Mlynski. Liquid crystal simulation using automatic differentiation and interval arithmetic. In G. Alefeld and A. Frommer, editors, *Scientific Computing and Validated Numerics.* Akademie Verlag, 1996.

[21] W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 2. edition, 1992.

[22] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[23] P. Sanders. A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 382–389, Kanazawa, Japan, 1994.

[24] P. Sanders. Massively parallel search for transition-tables of polyautomata. In *Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, pages 99–108, Potsdam, 1994.

[25] P. Sanders. Portable parallele Baumsuchverfahren: Entwurf einer effizienten Bibliothek. In *Transputer Anwender Treffen*, pages 168–177, Aachen, 1994. IOS Press.

[26] P. Sanders. Better algorithms for parallel backtracking. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 333–347, Lyon, 1995. Springer.

[27] P. Sanders. A scalable parallel tree search library. In S. Ranka, editor, *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, Honolulu, Hawaii, 1996.

[28] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche.* Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.

[29] P. Sanders. Randomized priority queues for fast parallel access. Technical Report IB 7/97, Universität Karlsruhe, Fakultät für Informatik, 1997.

[30] P. Sanders. Tree shaped computations as a model for parallel applications. In *ALV'98 Workshop on application based load balancing.* SFB 342, TU München, Germany, March 1998. `http://www.mpi-sb.mpg.de/~sanders/papers/alv.ps.gz`.

[31] E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In C. D. Houstis, E. N.; Papatheodorou, T. S.; Polychronopoulos, editor, *Proceedings of the 1st International Conference on Supercomputing*, number 297 in LNCS, pages 985–993, Athens, Greece, June 1987. Springer.