Tree Shaped Computations as a Model for Parallel Applications

Peter Sanders
Max-Planck-Institute for Computer Science
66123 Saarbrücken, Germany
E-mail: sanders@mpi-sb.mpg.de

Abstract

It is shown how a large class of applications can be parallelized by modeling them as *tree shaped computations*. In particular this class contains many highly irregular and completely unpredictable computations as they occur in heuristic search. We explain why the model even remains useful in the presence of some frequently observed subproblem dependencies.

1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking or depth first principle for traversing large implicitly defined trees (e.g. [ABF93, FMM94, FM87, KC94, KGGK94, Kor85, MT90]). In addition, some adaptive numerical algorithms for numerical integration by [PTVF92], for finding eigenvalues of tridiagonal matrices [CRY94] or for nonlinear optimization [NM96] have a similar structure. We will see, that even modeling the seemingly unrelated problem of loop scheduling in this way can be advantageous.

We introduce a model for the parallelization and load balancing aspects of these applications named *tree shaped computations* which makes the common properties visible while hiding unnecessary detail. Essentially, the model is applicable whenever we have applications which can be subdivided into independent subtasks. The model makes almost no assumptions about shape or predictability of the computations and can therefore handle highly irregular problems. Nevertheless, using *random polling*, a well known and simple receiver initiated load balancing algorithm, all tree shaped computations can be parallelized almost optimally.

Tree shaped computations, random polling and some other algorithms and a spectrum of applications have been investigated in the PhD thesis [San97]. The present paper can be considered a summary of the modeling aspects of this work. There is a large body of related work. We can only give a cross section and refer to the references in [San97] for a more detailed discussion. Early work on random polling and an application independent library is described in [FM87]. Random polling and other receiver initiated load balancing methods are also of central importance for parallel functional and logical programming languages (e.g., [ABF93, KC94]). Tree shaped computations can be considered a generalization of the α-splitting model used in [KGGK94]. A related model based on a subclass

of multithreaded computations is used in the Cilk project [BL94, BFJ+96]. The ZRAM library [BMFN98] is another recent implementation effort.

We neither want to focus on the theoretical model and its analysis alone nor on a particular application. Rather, we try to help bridging the gap between theory and application going from the abstract to the concrete. In Section 2 we therefore first introduce the abstract model and complement this by the practically most interesting algorithmic results in Section 3. Then we explain how the the simple model can be adapted to situations like depth first tree traversal, branch-and-bound or loop scheduling. Only then, in Section 5 we give some more detailed examples.

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* $\text{split}(I)$ 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 $\text{work}(I,t)$ 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

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

regardless when and where I_1 and I_2 are worked on. In Section 4.5 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* $\text{gen}(I)$ recursively defined by $\text{gen}(I_{\text{root}}) := 0$ and $\text{split}(I) = (I_1, I_2) \implies \text{gen}(I_1) = \text{gen}(I_2) = \text{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* h which guarantees that the size of subproblems with $\text{gen}(I) \geq h$ 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 [San97] a number of load balancing algorithms are investigated using a detailed model for message passing parallel computers with P PEs coupled via various interconnection

[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.

networks. Here, we contend ourselves with an outline of the practically most important algorithm using a simplified version of the the LogP model [CKP+93] as the machine model. The communication costs are expressed in terms of $T_{\text{rout}} := L + o + g$, 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 [FM87] 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*

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

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

Even in its simplest form, this algorithm also performs very well in practice. For small instances a few fine points can have a noticable influence however.

- In the basic algorithm we have a flurry of acitvity before all PEs get something to do. Although it can be shown that this period is very short, we can save some time by broadcasting the root problem initially and splitting it into P disjoint parts. This can be done without communication in time $\lceil \log P \rceil \, T_{\text{split}}$.
- We should use a termination detection protocol which works in $\mathrm{O}(T_{\text{rout}} \log P)$ time. In [San97] it is explained how this can be done using an asynchronous reduce-add operation.
- For some network interfaces it makes sense to try overlapping communication and computation. For this purpose we propose to maintain two subproblems on each PE. When one becomes empty start a request and switch to the other subproblem while waiting for a reply. More precisely, each PE should work on the local subproblem with the largest generation and split the subproblem with the smallest generation when answering a request.

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. We can even tolerate a complete 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 Making the Model Concrete

In Section 4.1 we start by explaining how the main application area of tree shaped computations – parallel depth first heuristic search in implicitly defined trees – can be modeled by tree shaped computations. Then Section 4.2 complements this with other seemingly unrelated applications where tree shaped computations can nevertheless be useful. Section 4.3 then gives a number of ways how an application can be interfaced with a load balancer based on tree shaped computations. Some simple aspects of many applications which are *not* modeled by tree shaped computations turn out to be implementation details in Section 4.4. If the independence assumption (1) is violated, we have a more serious problem which is discussed in detail in Section 4.5.

4.1 Depth First Traversal of Trees

A sequential algorithm for depth first traversal represents its state using a stack of nodes. (This may or may not be the recursion stack.) It pops nodes from the stack, evaluates them and pushes successors building a new level of the stack for each new level of the tree. Such a stack can also be used to represent a subproblem for the corresponding tree shaped computation and the operation "work" is the same as the sequential traversal algorithm. T_{atomic} is a bound on the time needed to evaluate a single node. In order to split a subproblems we produce two stacks which represent disjoint collections of subtrees. In the simplest case, we split off of a single subtree, preferably as deep as possible in the stack. If the the maximum branching factor of the tree is large, it is better to consider all the nodes in the deepest level of the stack together and to divide them between the two child problems in an alternating fashion. By this simple measure we can guarantee that $h \leq d \lceil \log B \rceil$ if d is the depth of the tree and B its maximum branching factor. Sometimes we can do even better by dividing nodes on all levels of the stack together. These different approaches to tree splitting have been described in [RK87].

One of our motivations for introducing tree shaped computations was to abstract from these implementation details without making the model less accurate.

4.2 Scheduling Intervals

Parallel programming languages and even preprocessor directives for Fortran of C/C++ often allow the automatic parallelization of for-loops whose individual iterations represent independent computations [Ope97]. If the individual iterations have strong variations in their execution time, the common practice is to use a centralized scheduler giving out chunks of iteration indices to the worker PEs. Even if heuristics are used which start with large chunks and later polish load imbalance by fading the chunk size, for large P this can become a delicate tradeoff between insufficient load balancing and communication overload of the master. A more robust and scalable solution is to use tree shaped computations by representing a subproblem as an interval of iteration indices. Splitting simply halves the interval of indices not yet iterated.

Very irregular instances of the loop scheduling problem are for example observed if the individual iterations are in fact tree search problem. For example, in airline crew pairing generation [GHL97] a heuristic enumeration of round trips for anonymous crew members is done. Each loop iteration is a backtrack search starting with a particular connection. In [AKRS90] each loop iterations represents a possible fault in a VLSI circuit

and a backtracking search is used to find a test pattern covering this fault. This application is so irregular that random polling is used to parallelize both the outer loop and the backtrack search.

Many numerical and geometrical divide-and-conquer applications can be modeled by computations working on a multidimensional interval which is adaptively subdivided (e.g., [PTVF92]). Again, tree shaped computations are a good model for this if the subproblems can be treated largely independently and if the amount of work is irregularly distributed over the subintervals. Note that there can be splitting functions which use application specific knowledge to select the dimension in which to cut or to cut not in equal halves in order to achieve a smaller h. A hybrid between interval based and tree based schemes are applications working with quad-trees or oct-trees.

4.3 Implementation Interface

Tree shaped computations can directly be used as an implementation interface. In the PIGSeL library [San96], the slimmest interface requires the application to supply functions for initializing, splitting, packing and unpacking subproblems. More abstract interfaces can automate some of this. In the PIGSeL library there is an interface based on specifying node expansions. We have already seen that a compiler can parallelize loops and in parallel functional or logical programming languages the run time system can manipulate the stack to split recursive computations (e.g., [ABF93, KC94]).

4.4 Initialization, Completion and Subproblem Encoding

Tree shaped computations omit some details from the model which do not influence the load balancing problem much but are nevertheless of some practical importance. First, on distributed memory machines most applications require some replicated information to be broadcast to all PEs. In particular, this often encompasses the entire description of the problem instance. This is the reason why we can later assume that a subproblem representation is so compact that it fits into a single message. For example, the state of a Prolog computation can be compactly represented as a bit string encoding the decisions made by the program [KC94].

Tree shaped computations should not be mixed up with divide-and-conquer applications like quicksort however, where we start with very large subproblem descriptions which become shorter quickly. On the one hand, these problems are often more predictable than difficult tree shaped computations because there is a strong correlation between the size of a subproblem representation and the work left to be done. On the other hand, we dearly need this information in order to minimize communication. Considering representation lengths to be short and uniform is too crude an approximation here. (At least if we cannot assume a shared memory.)

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).

4.5 Speculativity

Perhaps the most severe limitation of tree shaped computation is that the independence assumption (1) is not always true. We now explain why the model can nevertheless be useful even in the presence of subproblem dependencies.

Finding the first solution: Often tree search applications stop when they find the first node which represents a solution. Translated into our language this means that all subproblems suddenly become empty. This leads to the well known phenomenon of *speedup anomalies*, i.e., speedups $S \ll P$ or $S \gg P$ because the parallel algorithm happens to find a solution very late or very early. But often these algorithms are used for verifying that no solution exist, e.g., in order to prove the unsatisfiaility of a logical formula [BS96]. In this case no anomalies occur. 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., [SMV87, RK93]). 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 run into dead ends. We give an example in Section 5.2. More extreme cases of superlinar speedup are analyzed in [Ert92].

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.

Other pruning heuristics: Note that pruning heuristics in backtrack search only invalidate the independence assumption if they depend on the evaluation of subtrees whereas many simple heuristics are only a function of the path leading from the root to the present node.

A path in a backtrack 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* generate dependencies it is usually most effective on small subtrees whereas the load balancing mostly involves large subtrees which are usually independent.

In contrast, game-tree-search with $\alpha\beta$-pruning produces dependencies throughout the tree and consequently parallelization is rather difficult. Nevertheless, random polling turned out to be a good load balancing algorithm for game-tree-search [Fel93, FMM94]. 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.

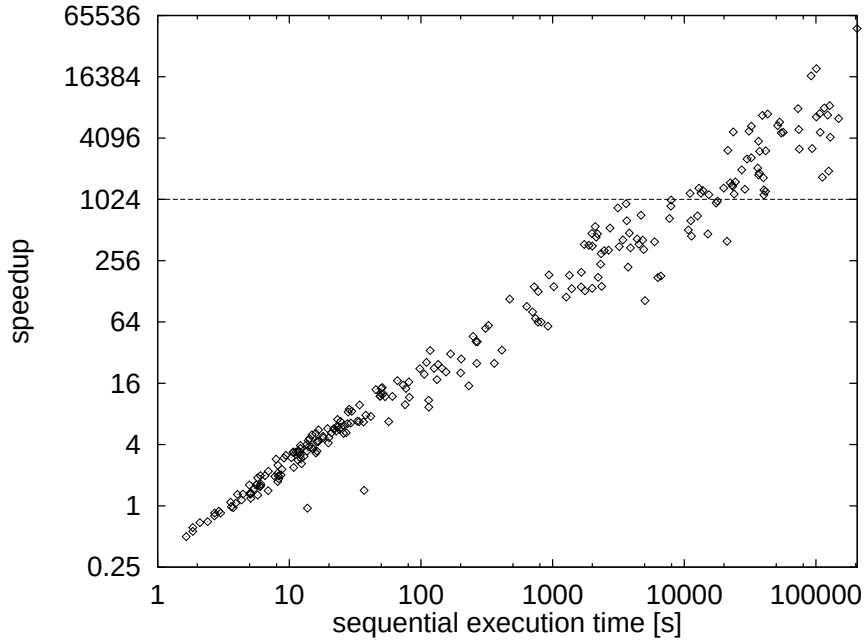

Figure 1: Speedup for 256 instances of the knapsack problem on 1024 PEs.

5 Implementation experiences

A number of example applications have been implemented most of them using the library PIGSeL [San96]. In the following we do not reiterate that whenever tree shaped computations are a good approximation to the real situation we get good parallel efficiency whenever the parallel execution time exceeds a few seconds even on many PEs. Rather, we want to focus on the experiences regarding how well the model fits.

5.1 Golomb rulers

stem from discrete mathematics [BG77] but have applications in radio astronomy and coding theory. A ruler of length m with k marks is defined by integer positions $0 = m_1$, $m_2, \ldots, m_k = m$ with the property $|\{m_j - m_i : 1 \leq i < j \leq k\}| = k(k-1)/2$, i.e., the ruler can be used to measure a maximum number of distances. For a given k we want to find a ruler with minimal m. Most systematic search algorithms for this problem can be viewed as a variant of depth first branch-and-bound. There are many additional heuristics but they do not introduce additional dependencies. Speedup anomalies are quite small because finding the optimal solution usually takes less time than verifying that this is the optimal solution. Also, good heuristics are available which sometimes yield the optimal solution immediately. In this case, a pure verification search is done which is perfectly modeled by tree shaped computations. This is not unusual for branch-and-bound applications.

5.2 The 0-1 knapsack problem

is one of the most intensively studied problems in combinatorial optimization [MT90]. An instance 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 from items in the knapsack without exceeding its capacity. For large m and arbitrary w_i, the best known algorithms are based

on a very fine-grained depth first branch-and-bound search [MT90]. An experiment was done with random instances generated using a statistics which yields large, difficult to solve yet tractable instances with very deep irregular search trees. The double-logarithmic plot in Figure 1 shows the relation between speedup and sequential execution time for 256 random instances with $m = 2000$, random $w_i \in [0.01, 1.01]$, random $p_i \in [w_i + 0.1, w_i + 0.125]$ and $M = \sum w_i/2$ on 1024 PEs of a Parsytec GCel.[2]

From a load balancing point of view it is interesting that the strategy to split on all levels of the tree at once is quite successful here. But more astonishing is that the average speedup over the 256 instances generated was 1410 on 1024 processors. This indicates that simple sequential depth first search is not robust enough for difficult instances because it sometimes bogs down in huge subtrees which do not contain the optimum.

5.3 The 15-puzzle

[Kor85] is a well known toy widely used as a benchmark in AI. You have to shift 15 scrambled squares in a 4×4 frame into the right order. In [Kor85] it is solved using iterative deepening search, i.e., a sequence of depth first searchs where the number of moves from the starting position is limited. All but the last iterations are perfectly modeled by tree shaped computations. The last iteration stops when the first solution is found. Speedup anomalies are present but not overwhelming. The extremly fine-grained search used in [Kor85] is an interesting challenge for the implementation. Using tree shaped computations as an implementation interface directly it was possible to use a single search algorithm for all levels of the tree and to get responsiveness to splitting requests with little overhead although less than 100 machine instructions are needed per node expansion.

5.4 The Firing Squad Synchronization Problem

is a classical problem in cellular automata theory asking for (time-optimal) algorithms for synchronizing a one-dimensional cellular automaton (using a minimal number of states). Using a heuristic backtrack search in the space of transition tables it was possible to show that there is no time-optimal solution with four states [San94]. Even a massively parallel implementation of this algorithm on 16384 PEs searches only a few percent more nodes although the dead-end heuristics mentioned above is crucial for its sequential performance.

6 Conclusions

Tree shaped computations are a good model for a wide range of applications. With random polling we have an algorithm which parallelizes them 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. Even if dependencies between subproblems are present, the predictions made by the simple model are often correct and the load balancer works well. Nevertheless, more accurate models are an important area for future

[2] We thank the Paderborn Center for Parallel Computing (PC^2) for making this machine available.

work. The subroutine call semantics modeled by the *fully strict multithreaded computations* used in the Cilk system [BL94] are one step into this direction. However, they do not model the unpredictable dependencies observed in the examples considered here.

References

[ABF93] G. Aharoni, Amnon Barak, and Yaron Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, 9:163–174, 1993.

[AKRS90] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh. Automatic test pattern generator on parallel processors. Technical Report TR 90-20, University of Minnesota, 1990.

[BFJ+96] 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.

[BG77] G. S. Bloom and S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.

[BL94] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, pages 356–368, Santa Fe, 1994.

[BMFN98] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 1998. to appear.

[BS96] M. B"ohm and E. Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subromonian, 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.

[CRY94] 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.

[Ert92] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*. Dissertation, TU München, 1992.

[Fel93] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. Dissertation, Universität Paderborn, August 1993.

[FM87] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, April 1987.

[FMM94] 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.

[GHL97] C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, number 1332 in LNCS, Cracow, Poland, 1997.

[KC94] J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[Kor85] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[MT90] S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, 1990.

[NM96] 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.

[Ope97] OpenMP. *OpenMP Fortran Application Interface*, 1.0 edition, 1997.

[PTVF92] W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.

[RK87] V. N. Rao and V. Kumar. Parallel depth first search. Part I. *International Journal of Parallel Programming*, 16(6):470–499, 1987.

[RK93] 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.

[San94] 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.

[San96] 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.

[San97] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.

[SMV87] 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*, volume 297 of *LNCS*, pages 985–993, Athens, Greece, June 1987. Springer.