A Scalable Parallel Tree Search Library

Peter Sanders
Department of Computer Science, University of Karlsruhe
76128 Karlsruhe, Germany
Email: `sanders@ira.uka.de`

Abstract

This paper reports design and implementation experiences with the portable and reusable library PIGSeL for parallel tree search. It is discussed how efficiency, flexibility and usability of the library can be reconciled. Two sample applications demonstrate its effectiveness for the case of parallel depth-first search. On a mesh of 1024 Transputers near optimal speedup even for small instances of the *Golomb ruler* problem is achieved. The *0/1 knapsack problem* is more challenging but the library achieves good speedups for quite irregular problem instances. From the algorithmic point of view, this is due to the random polling load balancing algorithm which turns out to perform well even on high-diameter networks, and also due to a fast initialization scheme, a bottleneck free implementation of the branch-and-bound heuristics and an adaption of the tree based double-counting termination detection algorithm.

1 Introduction

Many applications are based on the traversal of large implicitly defined trees, e.g., backtraching or best first branch-and-bound. Examples can be found in NP-hard problems from operations research, AI or VLSI-design and also in discrete mathematics, logic/functional programming or automatic theorem proving. Tree search is also quite interesting from the point of view of parallel processing research. On the one hand, highly parallel execution is possible in principle. On the other hand, tree search is a challenge to the load balancing algorithm. The computations are data dependent and often highly irregular. The load balancer must distribute the tree without a priori knowledge about the shape of the search tree. This must be achieved without undue communication overhead, even if the computations are very fine-grained. (For some concrete examples refer to Section 3.6.)

For the very same reasons, effective parallel tree search applications are not very widespread yet. The current status of parallel programming tools and the background of application programmers make parallelization difficult. This problem can be (partially) solved by encapsulating the parallelization into an application independent library. Considering the fast turnaround in new parallel machines, such a library must also be portable over a wide spectrum of architectures.

This paper is structured as follows: In Section 2 we discuss the lessons learned in designing the library PIGSeL (Parallel Implicit Graph SEarch Library). It combines the flexibility of an object oriented design with the portability and efficiency of plain ANSI-C. In Section 3 we describe the implementation. Three aspects are of particular interest there. The load balancer and other algorithmic aspects are critical for an efficient parallelization. Interfacing between the load balancer and the application decides about the usability of the library. Finally, experiences with applications make it possible to judge the bottom line performance. Section 4 summarizes the results.

2 Library design

The library PIGSeL is subdivided into several layers as depicted in Figure 1. After Section 2.1 discusses some basic principles for designing a layer and in particular its interfaces to other layers, the subsequent sections introduce the individual layers in a bottom up manner.

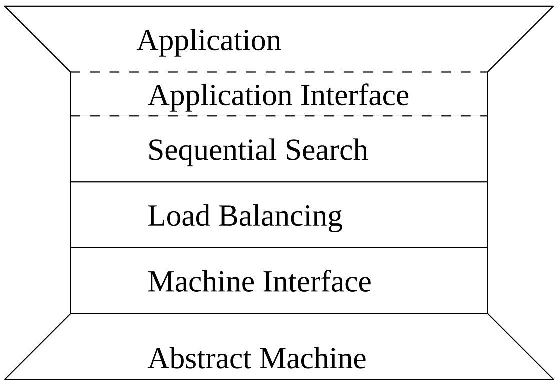

Figure 1: Basic layering of PIGSeL.

2.1 Basic Principles

The layers of the library are not intended as single monolithic modules implementing a fixed functionality. In this case the particular layering would only be interesting for the implementors as a means of organizing work. Rather, the layers form the abstract root classes of a hierarchy of possible implementations. Layers can also have a sub-layering or may be collapsed into a single module for reasons of efficiency and simplicity. What *is* important is that there should be simple and well defined interfaces between layers, making it possible to mix and match different implementations. For example, the machine interface layer should have a uniform interface to the load balancer making it possible to freely combine a range of machines and load balancing strategies. All in all, PIGSeL is not a clearly delineated system but an expandable collection of re-combinable classes.

Considering this object oriented design, an object oriented implementation language appears to be appropriate. Together with the aim of portability, C++ would currently be the prime candidate. Unfortunately, even C++ is not available on all parallel machines. At least if certain requirements regarding efficiency of the generated code and conformance to the language standard are made. We therefore choose plain ANSI-C as an implementation language. Besides Fortran, C is the only programming language fully supported on virtually all parallel machines. Interestingly, C is powerful enough to express the most important class relations directly. One reason for this is that although there can be many implementations of an abstract class (a layer) there is only a single implementation present in a particular program. Code inheritance is no problem as long as it is clear whether the abstract class or the implementation class implements a function. When this is insufficient, the C-preprocessor can be used to override default implementations. Still, the decision for C is an additional reason to keep the library simple and the class hierarchy flat.

2.2 Machine layer

The Hardware, the operating system, the compiler and its parallel libraries constitute an abstract machine the library has to adapt to. We expect that it should be possible to port our library to all MIMD-architectures currently in use.[1]

2.3 Machine interface layer

The machine interface layer is the only layer of PIGSeL which is allowed to be machine specific. The dilemma is that on the one hand, the load balancer should be able to use the strengths of a particular machine. On the other hand, the interface to the load balancer should be uniform in order to make it portable. This problem cannot be solved entirely. Aspects like multi-threading or properties of the interconnection network can never be fully unified. But the following compromise works quite well: On the one hand, we fix a minimum common functionality which has to be implementable on every machine. This is asynchronous[2] message passing between arbitrary PEs (Processing Elements) with a single FIFO receive buffer on each PE. On every PE, there is only a single identical process available for executing the layers above the machine interface layer. Quite complex interactions can be elegantly implemented by adding two important details: Message tags are assigned by a central instance avoiding clashes between different modules. Messages are “active” in the sense that a handler routine is called upon reception.

In addition, there are extensions which offer higher level services. Currently there is a sub-

[1]Even SIMD machines can relatively efficiently emulate MIMD-behavior [20]. But automatically compiling an entire library written for MIMD architectures for efficient SIMD execution is currently unrealistic.

[2]Asynchronous in the sense that the sender does not need to block until the receiving process has accepted the message.

layer for collective communications (broadcast, reduce, ...) and a rudimentary I/O support. If the underlying machine offers efficient primitives for this functionality these can be used. Otherwise, the services can be transparently implemented in terms of the basic functionality.

2.4 Load balancing layer

This layer contains all the components required for parallelizing the search. This is mainly the load balancing algorithm but there are also components for handling solutions and tree pruning heuristics. In our experience, there is no single load balancing principle equally well suited for all tree search problems. It is not even feasible to use a single interface to the layers above. However, there are interfaces which can span a quite wide range of applications.

For example, for depth-first search *tree shaped computations* are a useful abstraction. The basic data type is a *subproblem*. A subproblem can be worked on sequentially and it can be split into two new subproblems which can be worked on on different PEs. The only thing the load balancer knows about a subproblem is whether the subproblem is exhausted or not. In Section 3.3 it turns out that this is sufficient to design an efficient load balancing algorithm. The basic model can be enhanced by additional functionality, e.g., for the branch-and-bound heuristics or game tree search.

For best first search, a distributed priority queue is an appropriate abstraction, i.e., a sequential best first search algorithm runs on each PE but its priority queue accesses are serviced by the load balancer. For example, the library PPBB [22] is based on local priority queues which periodically exchange nodes. For more coarse grained applications, even a global priority queue can be used with the same interface. In [21] a bottleneck free implementation of a global priority queue is described.

2.5 Sequential search layer

Ideally, the library should completely hide the fact that the search is parallelized from the user. But the search strategy has to interact with the load balancer. One solution is to use a generic search algorithm. This concept has previously been used for didactic purposes [2] or for the classification of algorithms [14], also it is standard procedure in object oriented design. It has also been used in the library DIB [4].

We have introduced abstract data types for search tree `Nodes`, `Diff`erences between nodes (i.e. edges of the search tree) and `Solutions`. The differences leading to the successors of a node can be determined by an iterator over the successors of a node, i.e., there is a set of functions `initSucc`, `nextSucc`, `currSucc` and `moreSucc` which can generate the successors of a node one after the other. Moving up and down the tree is possible using `apply(Node, Diff)` and `unApply(Node, Diff)`. Together with some additional protocol for handling solutions and packing data structures, this functionality is sufficient to implement depth-first search, best first search, hybrid search strategies, etc. So, an additional benefit is that the user does not have to fix the choice of the sequential search strategy.

2.6 Application interface layer

In principle, the application can directly be implemented in terms of the interface defined by the generic search algorithm. But the author of a generic search algorithm wants a very general interface in order to be able to serve a large number of applications. On the other hand, the application programmer prefers a simple interface well suited for a narrower class of applications. The application interface layer can mediate between these two aims by providing a number of specialized interfaces. For example, a generic depth-first search algorithm can be adapted to backtracking, depth-first branch-and-bound, iterative deepening A* and variants regarding the way solutions should be treated (find all, find any, find best, ...).

2.7 Application layer

The way the application dependent parts are parallelized, depends on the original situation. If the application is implemented from scratch, the most elegant way is to use the generic search algorithm. But often there will already be a highly tuned sequential version which is to be ported. In this case, it can be easier to use the interface to the load balancer directly. It is not very complicated and it does not require any specific knowledge of parallel computing. This also makes it possible to use application specific knowledge for tasks like splitting subproblems or packing them for transmission.

Even more tuning may be useful if the application uses heuristics not (yet) implemented by the library which depend on global information. For example, duplicate elimination may be useful when the graph to be traversed is not really a tree. This requires a distributed hash table which can be implemented using the machine interface layer or even the native machine specific primitives.

3 Implementation experiences

Analogous to the previous section, we now explain the implementation experiences level by level, bottom up. We stress the application layer because this is the place where performance can be measured directly.

3.1 Machine layer

The implementations have been done on our local Workstation Cluster with PVM, Transputers (Parsytec SuperCluster and GCel-3/1024) with the parallel operating system Cosy and on the Power-PC based Parsytec machines Power X'plorer and GC/PP with the run-time system Parix.[3] An MPI-based implementation for the IBM SP-2 and other machines will follow soon.

3.2 Machine interface layer

The implementation for PVM and Cosy is very simple. The largest individual piece of code is responsible for making sure that the same code is executed on all PEs and that all PE IDs are available everywhere. In Parix this "SPMD-mode" is the default, but it does not support asynchronous communication between arbitrary PEs. This functionality can be implemented using the quite efficient multi-threading system of Parix. All implementations use a generic collective communication sublayer which turned out to be more efficient than the builtin functionality of the the abstract machine.

3.3 Load balancing layer

So far, there is only a load balancer for tree shaped computations. Its heart is the random polling load balancing algorithm described in [9] – an almost penetrantly simple algorithm. Every PE works on a single subproblem at a time. When this subproblem is exhausted, the PE asks a randomly chosen other PE to split its subproblem. When the requested PE is also idle, another random PE is choosen. It can be shown [19] that for a large class of problems, it is unlikely that any PE has to issue more than $O(\log P)$ requests overall (Let P denote the number of PEs). So, if the per PE load is larger than the cost for communicating $O(\log P)$ subproblems, arbitrarily high efficiency can be achieved.

The load balancer is further enhanced by an initialization scheme which broadcasts the root problem to all PEs, and then splits it locally, based on the PE number. Furthermore, the tree based double-counting termination detection algorithm [12, Section 4.6.1] is adapted to random polling: Incoming and outgoing work transfers are counted on each PE. When all PEs have been idle once, these counters are added and a new cycle is started. When both counters remain the same for two subsequent cycles, all PEs must be idle and no work can be in transit. This scheme can be implemented portably and efficiently using the asynchronous `reduceAdd`-function of the machine interface layer. It is more scalable than the ring based schemes used by other implementations and requires less messages than tree based schemes previously proposed [3, 9].

There is also an efficient support for the branch-and-bound heuristics. When a new solution is found, the new bound needs to be quickly disseminated to all PEs. We do this by indirectly sending the value to PE 0 along a binary reduction tree. Values which are only locally optimal are discarded as soon as possible. Only when a new value has reached PE 0, it is broadcast to all PEs. If locally improved solutions were immediately broadcast, this would result in severe network contention for applications like the knap-

[3] We would like to thank the Paderborn Center for Parallel Computing (PC^2) for making the GC-machines available.

sack problem where many suboptimal solutions are found initially. This method is also described in [6].

3.4 Sequential search layer

Implementing a useful generic search module proved to be quite difficult. The reason is that there are much more ways of handling solutions and tree pruning heuristics than one would expect from the neat textbook algorithms. Interestingly a large step towards a general solution was to *simplify* the basic search routine by stripping it of all treatments of heuristics or solution handling leaving only the bare tree traversal functionality. In addition, there are separate functions for reporting new results which the application can call at any time. The interface overhead between generic search algorithm, application interface layer and application is small because most of the functionality can be implemented as macros.

The main asset of providing the search layer may also be unexpected. Splitting a search space defined by a search stack is in principle quite easy (refer to [16] for example) but in practice there are many “opportunities” for introducing bugs which only become apparent when several PEs are involved. For an application programmer who does not know the internals of the library, these bugs are difficult to find. A thoroughly tested generic splitting function can save much of this trouble. Future versions of the library may contain components for black box testing user defined splitting functions. Another possibility is to provide a new interface between load balancer and application which is based on a generic splitting function and an application-function for converting between a generic and an internal representation of the search space.

3.5 Application interface layer

The application interface layer is very thin yet. So far, there are only some auxiliary modules. For example, one simplifies packing and unpacking functionality in case search tree nodes contain no dynamic data structures.

3.6 Application layer

The following three sections describe experiences made with the Golomb ruler problem, the 0/1 knapsack problem and some preliminary results on the 15-puzzle.

Golomb rulers

A *Golomb ruler* [1] 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 Golomb ruler with minimal m. Figure 2 shows a Golomb ruler of length 17 with 6 marks. The problem comes from discrete mathematics but it has applications in radio astronomy and coding theory.

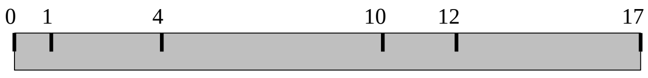

Figure 2: Golomb ruler with $k = 6$, $m = 17$.

It is solved using backtracking. On level i of the search tree, mark i is placed. Starting from a trivial approach, we introduced a number of heuristics which reduce the sequential execution time by two orders of magnitude. The search tree therefore has a quite irregular shape, but it is not very deep and it remains wide and bushy. The computations associated with a search-tree node are sufficiently coarse-grained to warrant using a generic search algorithm. But it is critical that the generic search algorithm decouples tree traversal and handling heuristics. Else it would not be possible to efficiently implement the three types of heuristics used. Before a node expansion takes place, an approximation of a branch-and-bound value is computed. After the node expansion, a more accurate bound is computed. Sometimes, all subsequent successors of a node are pruned without further computations. Incorporating all these options into a single “text-book style” generic search strategy would be diffucult.

For real applications, we want to find Golomb rulers for as large k as possible. It turns out that parallelization with random polling is no problem then. Even large numbers of loosely coupled workstations can achieve almost perfect PE

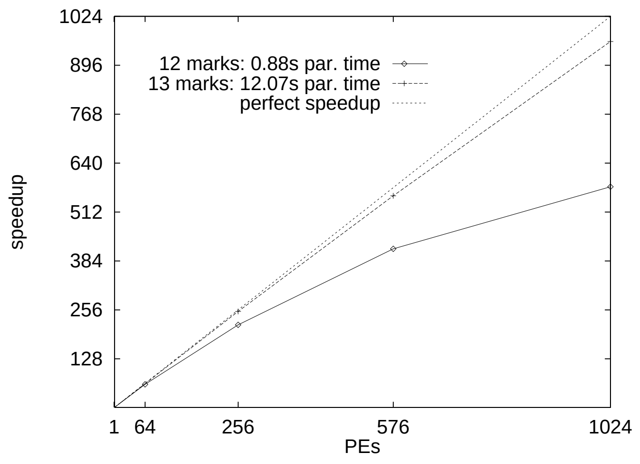

Figure 3: Speedup for Golomb rulers.

utilization. The parallel algorithm expands few more nodes than the sequential algorithm. Here we use Golomb rulers as a benchmark problem for studying scalability issues. We are therefore interested in good speedups for small problems. A load balancer performing well in this setting can also be expected to perform well for applications with very irregular search trees.[4]

Figure 3 shows speedups for the task of verifying that the shortest known rulers with 12 respectively 13 marks are indeed optimal. For such instances, the search tree is independent of the order in which subtrees are evaluated. We therefore get smooth, well reproducible speedup curves. (Small fluctuations due to the randomized load balancer have been damped by averaging over 5 measurements). Even for the quite small problem with 12 marks we get a speedup of 578 at a parallel execution time of $0.88s$. For even larger problems we achieve almost perfect speedup. (Compared to the specialized sequential algorithm.)

The 0/1 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 [11].

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 data structures would be too high. Parallelizing the best first approach is also difficult. In [13] the speedup on 16 PEs remains below 6.

For the same reason we choose to implement the application directly on top of the load balancer.[5] The search space is split by evenly dividing open subproblems on all tree levels between the subproblems [16]. 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

[4]For example, this relation can be observed in game tree search. Heuristics which reduce search overhead lead to frequent drops in usable parallelism [5].

[5]In an early version we tried to use the generic search algorithm. It took a factor of about 1.7 more time (Gnu-C). We were not able to fully resolve the sources of overhead. Perhaps one problem is that optimizers of compilers are written with unjustified assumptions about how code has to look like. This optimizer problem was much more pronounced in experiments with C++ (Also Gnu). Just making `Node` a class (not to speak of the other data types) incurred an additional overhead factor of two. Even templates with inline functions seem to be slightly slower than macros.

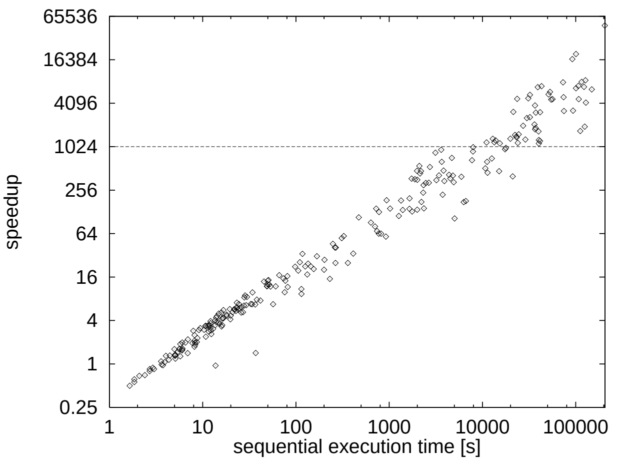

Figure 4: Speedup for 256 instances of the knapsack problem on 1024 PEs.

close to m – the search tree is almost a linear list. Clearly, no speedup for parallel tree search is possible here.[6] 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 4 shows the relation 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.

[6]Speedups reported in [10] for this class of instances are an artifact of a very inefficient node evaluation function.

The 15-Puzzle

The 15-Puzzle is a standard benchmark problem from AI. The best sequential algorithm for finding shortest solutions is iterative deepening depth-first search (IDA* [8]). There are also successful parallelizations [16, 15, 7, 17]. Our implementation uses PIGSeL for an individual iteration of the search. Since there are a number of ways to keep the number of required iterations for small instances small and because the search tree turns out to be not too irregular, we expect a scalability similar to the Golomb ruler problem. (We have no measurements on large machines yet.)

The 15-puzzle is interesting due to its extremely fine granularity. On a PPC 601 our implementation needs only about 50 machine cycles per node expansion. We therefore do not use the generic search algorithm. It is not even advisable to use the data type `Subproblem` directly for searching. For example, the Motorola C-compiler never puts slots of C-`struct`s into registers. We therefore copy the more important parts of a subproblem description into local scalar variables of the sequential search function. Only those are reliably put into registers by the optimizer.

4 Conclusions

We have demonstrated that it is possible to efficiently exploit large parallel machines for searching irregularly shaped trees using a reusable portable library. The random polling algorithm, which was known to be successful on low diameter networks [9], is equally effective on high latency machines with software routing. Our implementation is particularly efficient due to a broadcast based initialization scheme, the double-counting termination detection algorithm and a bottleneck free implementation of the branch-and-bound heuristics. The results for small Golomb ruler problems demonstrate an efficient parallel execution time which is at least an order of magnitude smaller than previous results on comparable machines [18, 23]. Much of this effectivity carries over to more challenging problems like the knapsack problem which requires fine-grained depth-first branch-and-bound with a deep thin search tree and frequent bound updates.

The design of PIGSeL makes it possible to incorporate a wide range of machines, load balancers and applications into a single framework. Building the library on a thin machine interface layer with portable sublayers for higher-level services proved to be effective on three different abstract machines. Tree shaped computations form a simple and useful interface between the load balancer and the search algorithm for depth-first search. Using generic search algorithms to separate the search strategy and the application is more difficult and the overhead of this approach is only tolerable for sufficiently coarse grained applications.

References

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

[2] G. Brassard and P. Bratley. *Algorithmics Theory and Practice.* Prentice Hall, 1988.

[3] S. Dutt and M. R. Mahapatra. Parallel A* algorithms and their performance on hypercube multiprocessors. In *International Parallel Processing Symposium*, Newport Beach, 1993.

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

[5] H. Hopp and P. Sanders. Parallel game tree search on SIMD machines. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 349–361, Lyon, 1995. Springer.

[6] L. V. Kale and A. B. Sinha. Information sharing mechanisms in paralllel programs. Technical report, University of Illinois, Urbana Champaign, 1991.

[7] G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1057–1072, 1994.

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

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

[10] 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.

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

[12] F. Mattern. *Verteilte Basisalgorithmen.* Number 226 in Informatik-Fachberichte. Springer, 1987.

[13] 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.

[14] D. S. Nau, V. Kumar, and L. Kanal. General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, 23:29–58, 1984.

[15] C. Powley, C. Ferguson, and R. E. Korf. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.

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

[17] A. Reinefeld. Scalability of massively parallel depth-first search. In *DIMACS Workshop*, 1994.

[18] A. Reinefeld and V. Schnecke. Work-load balancing in highly parallel depth-first search. In *Scalable High Performance Computing Conference*, pages 773–780, Knoxville, 1994.

[19] 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.

[20] P. Sanders. Emulating MIMD behavior on SIMD machines. In *International Conference Massively Parallel Processing Applications and Development*, Delft, 1994. Elsevier.

[21] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 379–393, Lyon, 1995. Springer.

[22] S. Tschöke and N. Holthöfer. A new parallel approach to the constrained two-dimensional cutting stock problem. In *Workshop on Algorithms for Irregularly Structured Problems*, LNCS, pages 285–299, Lyon, 1995. Springer.

[23] S. Tschöke, M. Räcke, R. Lüling, and B. Monien. Solving the traveling salesman problem with a parallel branch-and-bound algorithm on a 1024 processor network. Technical report, Universität Paderborn, 1994.