Parallel Best–First Search of State-Space Graphs: A Summary of Results *

Vipin Kumar†, K. Ramesh, and V. Nageshwara Rao
Artificial Intelligence Laboratory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

This paper presents many different parallel formulations of the A*/Branch-and-Bound search algorithm. The parallel formulations primarily differ in the data structures used. Some formulations are suited only for shared-memory architectures, whereas others are suited for distributed-memory architectures as well. These parallel formulations have been implemented to solve the vertex cover problem and the TSP problem on the BBN Butterfly parallel processor. Using appropriate data structures, we are able to obtain fairly linear speedups for as many as 100 processors. We also discovered problem characteristics that make certain formulations more (or less) suitable for some search problems. Since the best-first search paradigm of A*/Branch-and-Bound is very commonly used, we expect these parallel formulations to be effective for a variety of problems. Concurrent and distributed priority queues used in these parallel formulations can be used in many parallel algorithms other than parallel A*/branch-and-bound.

1 Introduction

Heuristic search is an important technique that is used to solve a variety of problems in Artificial Intelligence (AI) and other areas of computer science[7; 17; 18]. Search techniques are useful when one is able to specify the space of potential solutions, but the exact solution is not known before hand. In such cases a solution can be found by searching the space of potential solutions. Clearly, if many processors are available, then they can search different parts of the space concurrently. Investigation of parallelism in different AI search procedures is an active area of research [9; 23; 6; 14; 15; 2].

For many problems, heuristic domain knowledge is available, which can be used to avoid searching some (unpromising) parts of the search space. This means that parallel processors following a simple strategy (such as divide the search space statically into disjoint parts and let each one be searched by a different processor) may end up doing a lot more work than a sequential processor. This would tend to reduce the speedup that can be obtained by parallel processing. If the amount of work done by a sequential processor is W_s and the total amount of work done by P parallel processors is W_p, then the **redundancy factor** due to parallel-control-strategy is given by W_p/W_s and the upper bound on the speedup is $\frac{P}{W_p/W_s}$. However, due to other factors such as communication overhead, etc., the actual the speedup may be less than $\frac{P}{W_p/W_s}$.

We have been investigating the use of parallel processing for speeding up different heuristic search algorithms [9; 23; 21; 11]. In this paper, we discuss a number of parallel formulations of the A* state-space search algorithm. As discussed in [10; 16], A* is essentially a "best-first" branch-and-bound algorithm. The parallel formulations presented in this paper are also applicable to many other best-first branch-and-bound procedures. The parallel formulations primarily differ in the data structures used to implement the OPEN list (priority queue) of the A* algorithm. Some formulations are suited only for shared-memory architectures, whereas others are suited for distributed-memory architectures as well. The effectiveness of different parallel formulations is also strongly dependent upon the characteristics of the problem being solved. We have tested the performance of these formulations on the 15-puzzle[17], the traveling salesman problem(TSP), and the vertex cover problem (VCP) [1] on the BBN Butterfly multiprocessor. The results for the 15-puzzle and VCP are very similar; hence we only present the results for the VCP and TSP. Although both TSP and VCP are NP-hard problems, they generate search spaces that are qualitatively different from each other. We also present a preliminary analysis of the relationship between the characteristics of the search spaces and their suitability to various parallel formulations.

BBN Butterfly is composed of up to 256 processor memory pairs. Each processor's local memory is accessible to other processors via a fast switch; hence it is essentially a shared-memory multiprocessor. It is easy to emulate distributed memory multiprocessors on a shared-memory multiprocessor. We study the suitability of different parallel formulations for both shared-memory and distributed-memory multiprocessors.

---

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

†arpanet:kumar@sally.utexas.edu

2 The A* Algorithm

We assume familiarity with the A* algorithm. See [17] for a good introduction to A*. We will also use the terminology presented in [17]. Here we provide a brief overview of the algorithm.

A* is used to find a least-cost path between a start state and a (set of) goal state(s) of a given state-space graph. The state-space graph is implicitly specified by the start state, a move generator (a procedure that can generate successors of any given state in the state-space graph) and a function to recognize the goal-state(s). A* maintains two lists OPEN and CLOSED. OPEN contains those nodes whose successors have not been generated yet. CLOSED contains those nodes whose successors have been generated. The process of generating successors of a node m is also referred to as "expanding m". For a node m in OPEN, g(m) is the cost of the current best path from start state to m, h(m) is a heuristic estimate of the cost of the shortest path between m and a goal state, and f(m) = g(m) + h(m) is the overall cost of the node m. In each iteration, A* selects a most promising node n (i.e., the node with the smallest f-value) from the OPEN list for expansion, generates its successors and puts the node n into CLOSED and its successors into OPEN. [1] Whenever a goal-node is chosen for expansion, A* terminates with n as the solution. It was proved in [17] that if the heuristic estimate h is admissible, then A* would terminate with an optimal solution (if a solution exists). Since the only operations done on OPEN are deletions of smallest cost element and insertion of elements, OPEN is essentially a priority queue, and is often implemented as a heap[1]. The heap implementation allows insertions and deletions in $O(\log N)$ steps, where N is the size of OPEN.

3 A Centralized Parallel Search Strategy

Given P processors, the simplest parallel strategy is to let each parallel processor work on one of the current best nodes in the OPEN list. We shall call it a centralized strategy because each processor gets work from the global OPEN list. As discussed in [5], this strategy should not result in much redundant search. There are two problems with this approach.

(1) The termination criterion of sequential A* does not work any more; i.e., if a current processor picks up a goal-node m for expansion, then the node m is no longer guaranteed to be the best goal node. But the termination criterion can be easily modified to ensure that termination occurs only after a best solution has been found[14; 19].

(2) Since OPEN will be accessed by all the processors very frequently, it will have to be maintained in a shared memory that is easily accessible to all the processors. Hence distributed-memory architectures such as the Hypercube[25] are effectively ruled out. Even on shared memory architectures, contention for OPEN limits the performance to T_{exp}/T_{access}, where T_{exp} is the average time for one node expansion, and T_{access} is the average time spent in accessing OPEN per node expansion [4]. Note that the access to CLOSED does not cause contention, as different processors would manipulate different nodes.

We have implemented this scheme for solving the Traveling Salesman Problem (TSP) and the vertex cover problem(VCP). Next we discuss these implementations and present performance results.

3.1 Performance Results for the TSP

The Traveling Salesman Problem can be stated as follows : Given a set of cities and inter-city distances, find a shortest tour that visits every city exactly once and returns to the starting city. TSP can be solved using the A*/Branch-and-Bound algorithm. A number of heuristics are known for the TSP. We have used the LMSK heuristic[13] in our experiments. Although the LMSK heuristic is quite powerful, it is not as good as the assignment heuristic[26]. We chose LMSK primarily because it was easy to implement and was adequate to show the power of different data structures and parallel control strategies discussed in this paper. We implemented parallel A* (with the LMSK heuristic) using the centralized control strategy on BBN Butterfly. OPEN was implemented as a heap. We tested the parallel version for up to 100 processors on Butterfly. The cost matrices for TSP instances were generated by a uniform random number generator. We found the average redundancy factor due to the centralized control strategy to be over .95 even for 100 processors.

Figure 1 gives the actual speedup obtained for problems of different granularities (T_{exp}). T_{exp} is the time needed to compute the LMSK heuristic of the generated nodes in each iteration of A*. This grows as $O(M^2)$, where M is the number of cities in the TSP. The speedup is fairly linear for small number of processors, but saturates at $\frac{T_{exp}}{T_{access}}$. This shows that the centralized parallel strategy is quite effective for parallelizing the TSP instances of large granularity. On problems with smaller granularities, the contention for OPEN shows up. To reduce the contention for OPEN, we implemented it as a concurrent heap[22]. On a concurrent heap, OPEN needs to be locked only for $O(1)$ time, which allows $O(\log N)$ processors to access OPEN simultaneously (N is the number of nodes in OPEN). Fig. 2 shows the improvement in performance due to the concurrent heap.

3.2 Performance Results for the Vertex Cover problem(VCP)

The vertex cover problem can be stated as follows: Given an undirected graph G = (V,E) (V denotes the set of vertices, and E denotes the set of edges), find the smallest subset of vertices such that they cover all the edges in E. The start state of the state space of the VCP is a null

[1] Since there can be more than one path by which a particular node can be reached from the start node, this step is a bit more complicated. See [17] for details.

cover. Each state is a partial cover of the graph. From any state, its two successors can be created by including or excluding the next vertex. If a vertex is excluded, then all of its neighbors are included in the partial cover. For any state n, g(n) is the number of vertices already included in the partial cover n, and h(n) is the minimum number of vertices that must be added to n to create a cover. The h function for the VCP is readily computed[2].

We implemented parallel A* to solve the VCP on BBN Butterfly and tested it on many randomly generated instances of the vertex cover problem. These instances were chosen to have 50 to 80 vertices to ensure that the search trees of these instances are reasonably large. The VCP is prone to speedup anomalies, as there are a lot of nodes in its state-space tree that have the same cost as that of its least-cost solution. Hence, the speedup depends upon when the actual solution is encountered by the search (sequential or parallel). The phenomenon of speedup anomalies in best-first branch-and-bound has been extensively studied in [12; 20]. Our recent work[24] shows that it is possible to expect superlinear speedup on the average.[3] To study the speedup behavior in absence of anomaly, we modified the A* algorithm to find all optimal solutions of the VCP.

The redundancy factor due to the centralized control strategy for the vertex cover problem is consistently around 1. But the speedup obtained is very poor and tapers off around 8. The reason for the poor performance is that the node expansion in the vertex cover problem is very cheap; hence all the processors spend a good part of their time adding or removing elements from OPEN causing contention for the shared data structure. Even if OPEN is implemented as a concurrent heap, the speedup would taper off around 24. This clearly shows that the centralized strategy is not good for small granularity problems such as the VCP.

Next we present many different decentralized control strategies that work even for problems for which the node expansion time is small. In these strategies, the OPEN list is implemented as a distributed priority queue.

4 Distributed Strategies

One way to avoid the contention due to centralized OPEN is to let each processor have its own local OPEN list[4]. Initially, the search space is statically divided and given to different processors (by expanding some nodes and distributing them to the local OPEN lists of different processors). Now all the processors select and expand nodes simultaneously without causing contention on the shared OPEN list as before. In the absence of any communication between individual processors, it is possible that some processors may work on a good part of the search space, while others may work on bad parts that would have been pruned by the sequential search. This would lead to a high redundancy factor and poor speedup. The communication schemes discussed in the next three sections try to ensure that each processor works on a good part of the search space.

4.1 The Blackboard Communication Strategy

In this strategy, there is a shared BLACKBOARD through which nodes are switched among processors as follows. After selecting a (least f-value) node from its local OPEN list, the processor proceeds with its expansion only if it is within a "tolerable" limit of the best node in the BLACKBOARD. If the selected node is much better than the best node in the BLACKBOARD, then the processor transfers some of its good nodes to the BLACKBOARD. If the selected node is much worse than the best node in the BLACKBOARD, then the processor transfers some good nodes from the BLACKBOARD to its local OPEN list. In each case, a node is reselected for expansion from local OPEN.

The choice of tolerance is important, as it affects the number of nodes expanded as well as the amount of node switching between local OPEN lists and the BLACKBOARD. If the tolerance is kept low then nodes will be switched frequently between local OPEN lists and the BLACKBOARD unless the best nodes in all the OPEN lists happen to have the same cost. If the tolerance is high then the node switching would happen less frequently, thus reducing contention on the global BLACKBOARD. But in this case a processor can possibly expand nodes that are inferior to nodes waiting to be expanded in other processors.

4.2 The Random Communication Strategy

In this strategy, each processor periodically puts the newly generated successors of the selected node into the OPEN list of a randomly selected processor. This ensures that if some processor has a good part of the search space, then others get a part of it.[5] This strategy can be easily implemented on distributed-memory systems with low diameter (such as Hypercube[25], Torus[3]) as well as shared memory multiprocessors such as the Butterfly. If the frequency of transfer is high, then the redundancy factor can be small; otherwise it can be very large. The choice of frequency of transfer is effectively determined by the cost of communication. If communication cost is low (e.g., on

[2]the computation of h(n) for a node is done using an algorithm given in [28].

[3]Although the work reported in [24] deals with average superlinear speedup in depth-first search, it is also applicable to best-first search. If many nodes in the state-space graph have the same cost, then heuristic function does not provide any discrimination among them, and the search tend to become depth-first.

[4]these OPEN lists can be implemented as heaps to allow $O(\log N)$ access time

[5]This strategy is very similar to the one in which periodically, a processor puts some of its best nodes into the OPEN list of a randomly selected processor.

shared-memory multiprocessors) then it would be best to perform communication after every node expansion.

4.3 The Ring Communication Strategy

In this strategy, different processors are assumed to be connected in a virtual ring. Each processor periodically puts the newly generated successors of the selected node into the OPEN list of one of its neighbors in the ring.[6] This allows transfer of good work from one processor to another. This strategy is well suited even for distributed-memory machines with high diameter (e.g., ring). Of course, it can be equally easily implemented on low diameter networks and shared memory architectures. As in the previous scheme, the cost of communication determines the choice of frequency of transfer.

4.4 Performance Results

We implemented the three communication schemes to solve the TSP and VCP on the Butterfly parallel processor. Experiments were run on the same problem instances that were used with the centralized scheme. In the case of the ring and random communication schemes, the exchanges were done after each node expansion. In the case of the blackboard strategy, the tolerance factor was kept quite low. Results are shown in Figures 3 and 4. The blackboard scheme does very well for both problems. The random communication scheme does very well for the VCP and only moderately well for the TSP. The ring communication scheme has a reasonable performance on the VCP but does very poorly on the TSP. The performance drop for the ring communication and the random communication scheme is primarily due to the increased redundancy factor. If nodes are transferred less frequently in the ring and random communication strategies, or if the tolerance factor for the blackboard strategy is made high, then the speedup drops significantly in all cases.[7] Hence it seems that a tightly coupled architecture (such as the Butterfly) would perform much better than loosely coupled architectures on all the formulations.

5 Analysis of Performance

Here we present a discussion of a certain feature of the state spaces of the TSP and VCP that explains the difference in performance of distributed communication strategies on the two problems.

In A*, if the heuristic is consistent[17], then the cost of the nodes expanded in successive iterations never goes down (it either goes up or stays the same). Let V_i be the set of nodes expanded by A* after the cost has gone up ith time but before it has gone up i+1 th time. Clearly the cost of each node in V_i (for any i) is the same, and the heuristic function does not provide any discrimination among different nodes in V_i. V_0 represents the expanded nodes that have the same cost as the start node. If the cost goes up L times in the search, then V_L is the set of nodes expanded whose cost is the same as the optimal solution. Note that the heuristic functions used in the TSP and the VCP (and most other problems solved by branch-and-bound) are consistent. Figure 5 plots V_i for an instance of the VCP and an instance of the TSP. Plots for the other instances are very similar in each case. Clearly, for the VCP, V_i grows very rapidly, and for the TSP it grows very slowly. For the VCP, expansions of nodes in V_L represents a very large fraction (nearly 75 percent) of the total work done by A*. Since all the nodes in V_L have the same cost, the heuristic function does not provide much discrimination between these nodes, and the loose coupling of the random and ring communication schemes seem to be good enough. For the TSP, there are only a few nodes at each cost (L is 54, and most of the V_i have between 50 and 400 nodes); hence the communication scheme should be "tightly-coupled" to be able to effectively utilize the heuristic guidance. Note that the rapid growth of V_i does not mean that the heuristic is bad. In a 65-node VCP, it reduces the search space from 2^{65} to around 11300 nodes. The LMSK heuristic used for a 25-city TSP reduces the search space from $25 * 2^{25}$ to roughly 3600 nodes. Interestingly, even for the 15-puzzle V_i grows very rapidly, and its performance on the distributed communication schemes is very similar to that of the VCP.

It is easy to see that IDA*[8] outperforms A* on those problems for which V_i grows very rapidly. We have already presented a parallel implementation of IDA* that is able to provide virtually unlimited speedup (for large enough problems) on a variety of architectures[23; 11]. Also IDA*, unlike A* requires very little memory, hence can solve large problem instances without running out of memory.

The speedup anomalies on the VCP are fully explained by the fact that a large number of nodes have the the cost equal to that of the optimal solution. Hence, the amount of work done by any search scheme (sequential or parallel) depends upon when the set of nodes leading to the optimal solution are expanded. Although a number of researchers have investigated the phenomenon of speedup anomalies in best-first branch-and-bound, all of them hypothesized that the phenomenon is unlike to occur in real problems[12; 20]. Since, for the VCP (and the 15-puzzle), V_i grows very rapidly, and the length of the solution grows linearly with problem size, for large problem instances the speedup anomaly can be very pronounced.

6 Related Research

Many of the parallel formulations of A*/Branch-and-Bound presented in this paper have been investigated by other researchers as well. The centralized scheme has been studied in [?; 20; 4]. Parallel A* with the centralized scheme for solving the TSP is essentially the same as Mohan's parallel algorithm for TSP in [?]. Mohan reported

[6] This strategy is very similar to the one in which periodically, a processor puts some of its best nodes into the OPEN list of one of its neighbors in the ring.

[7] These results are not shown in the speedup graphs.

a speedup of 8 on 16 processors on the Cm*. Our results show that for high granularity problems such as TSP, this scheme can provide several orders of magnitude speedup on commercially available multiprocessors. The use of concurrent heap further extends the upper limit on the speedup obtained using the centralized approach. We have also investigated various means of artificially increasing the granularity of the problem (i.e., increase T_{exp}).[8] These results are not presented in this paper.

A number of researchers have suggested distributed strategies similar to the random communication scheme [28; 3], and the ring communication scheme [27; 29]. Wah and Ma [29] found the ring communication scheme to give good speedup on the vertex cover problem and hypothesized that this could be a good strategy for best-first Branch-and-Bound in general. Our work has clearly shown that these strategies are effective only for those problems in which the search space has many nodes of the same cost. To the best of our knowledge the blackboard communication strategy for parallel A* has not been investigated before.

7 Concluding Remarks

We have presented many different ways of parallelizing the A* algorithm, and have studied their performance on the Vertex Cover problem (VCP) and the Traveling Salesman Problem (TSP). The performance of different formulations depends on the characteristics of the problems.

The centralized scheme has a very low redundancy factor, but causes contention for the centralized OPEN (implemented as a simple heap or as a concurrent heap) unless the granularity of the problem is large. In the distributed schemes, each processor has its own OPEN list (the OPEN list is implemented as a distributed heap); hence there is no contention for shared data structures. But the redundancy factor can be large, as some processors may have all the good nodes while others may have only bad nodes. The communication strategies (blackboard, ring, random) try to make sure that all of the local OPEN lists (priority queues) have even distribution of good nodes. Contrary to the belief of many researchers, the random and ring communication strategies are not very effective evenly distributing good nodes. They appear to perform well only on those problems in which the search space has many nodes of the same cost (e.g., the 15-puzzle, the VCP). For other problems (such as the TSP), they have a large redundancy factor, and give poor speedup. The blackboard strategy clearly outperforms the other two distributed strategies for both kinds of problems. A major drawback of the blackboard strategy is that it requires a shared-memory architecture, which is more expensive to construct than the distributed memory architectures such as ring or hypercube. Also, contention for the blackboard limits the ultimate scalability of the strategy. We are currently investigating strategies that do not suffer from these drawbacks.

It is expected that all the parallel control strategies presented in this paper would be applicable to many other problems solvable by A*/branch-and-bound. Concurrent and distributed priority queues used in these parallel formulations can be useful in many parallel algorithms other than parallel A*/branch-and-bound. Our work has demonstrated that it is possible to exploit parallelism in search to get several orders of magnitude speedup on commercially available multiprocessors. Given that each processor in these systems is an off-the-shelf microprocessor, these parallel processors can be cost effective high performance computing engines for solving AI search and optimization problems.

Acknowledgements : We would like to thank Prof. Larry Davis (Center for Automation Research, University of Maryland) for access to the BBN Butterfly parallel processor. Alan Gove implemented an earlier version of parallel A* for solving the Vertex Cover problem on the Sequent parallel processor. Rich Korf and Dan Miranker made useful comments on an earlier draft of this paper.

[8] One such scheme is: pick one node from OPEN, generate a large number of nodes, and then put them back into OPEN.

References

[1] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] J. S. Conery and D. F. Kibler. Parallelism in ai programs. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 53–56, 1985.

[3] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Boston, MA, 1987.

[4] S.-R. Huang and Larry S. Davis. A tight upper bound for the speedup of parallel best-first branch-and-bound algorithms. Technical report, Center for Automation Research, University of Maryland, College Park, MD, 1987.

[5] Keki B. Irani and Yi Fong Shih. Parallel a* and ao* algorithms: An optimality criterion and performance evaluation. In *Proceedings of International Conference on Parallel Processing*, pages 274–277, 1986.

[6] L. N. Kanal and Vipin Kumar. Branch-and-bound formulations for sequential and parallel game tree searching. *Proceedings of International Joint Conference on Artificial Intelligence*, pages 569–571, 1981.

[7] L. N. Kanal and Vipin Kumar. *Search in Artificial Intelligence.* Springer-Verlag, New York, NY, 1988.

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

[9] V. Kumar and L. N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions Pattern Analysis and Machine Intelligence*, PAMI–6:768–778, 1984.

[10] Vipin Kumar. Branch-and-bound search. In Stuart C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence:*

Vol 2, pages 1000–1004. John Wiley and Sons, New York, NY, 1987. Revised version appears in the second edition 1992.

[11] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, December 1987.

[12] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, pages 594–602, 1984.

[13] E. L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14, 1966.

[14] D. B. Leifker and L. N. Kanal. A hybrid sss*/alpha-beta algorithm for parallel search of game trees. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1044–1046, 1985.

[15] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

[16] D. S. Nau, Vipin Kumar, and L. N. Kanal. General branch-and-bound and its relation to a* and ao*. *Artificial Intelligence*, 23, 1984.

[17] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[18] Judea Pearl. *Heuristics-Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[19] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, NY, 1987.

[20] Michael J. Quinn and Narsingh Deo. An upper bound for the speedup of parallel branch-and-bound algorithms. *BIT*, 26,No 1, March 1986.

[21] V. Nageshwara Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, December 1987.

[22] V. Nageshwara Rao and V. Kumar. Concurrent insertions and deletions in a priority queue. In *Proceedings of the 1988 Parallel Processing Conference*, 1988.

[23] V. Nageshwara Rao, V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 878–882, 1987.

[24] V. Nageshwara Rao and Vipin Kumar. On the efficicency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993. Also available as Technical Report TR 90-55, Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/suplin.ps.

[25] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28–1:22–33, 1985.

[26] H. S. Stone. *High-Performance Computer Architectures.* Addison-Wesley, Reading, MA, 1987.

[27] Olivier Vornberger. Implementing branch-and-bound in a ring of processors. Technical Report 29, University of Paderborn, FRG, 1986.

[28] Olivier Vornberger. Load balancing in a network of transputers. In *2nd International Workshop on Distributed Parallel Algorithms*, 1987.

[29] Benjamin W. Wah and Y. W. Eva Ma. Manip—a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, c–33, May 1984.