Parallel Astar Search on Message-Passing Architectures

Z. Cvetanovic and C. Nofsinger

Digital Equipment Corporation
Mid-Range Systems Engineering
85 Swanson Rd.
Boxborough, MA 01719

Abstract

In this paper we compare three parallel Astar search algorithms: in the *Shared-List* algorithm a search space is shared among processors, in the *Static Distribution* algorithm a search space is distributed once to all processors, and in the *Continuous Diffusion* algorithm a search space is continuously redistributed. Our results indicate that the Continuous Diffusion algorithm outperforms the other two algorithms on message-passing architectures. Additionally, the *Grid-Flow* technique developed for Continuous Diffusion is of general importance for nearest-neighbor algorithms that need to share some global information.

1 Introduction

Heuristic search is an important technique for solving a variety of problems in artificial intelligence and operations research. Examples of such problems include planning, learning, decision making, natural language understanding, theorem proving, and discrete optimization. Execution times for sequential heuristic search algorithms typically increase exponentially with the linear addition of information to the problem. Multiple-processor systems have the potential to enhance search efficiency by distributing the exponential work among a number of processors.

To achieve a reduction in search time that is linear in the number of processors, we must distribute the search evenly across processors. In order to statically dividê the search evenly among processors, however, one would have to use knowledge about the location of the solution in the search space. In most cases such knowledge does not exist. Thus, in a given static distribution, load will be unbalanced and some processors will work on parts of the search space not considered by the sequential algorithm. The resulting load imbalance and extra search will lead to sub-linear speedups. Dynamic techniques can potentially overcome the problems of static partitioning.

A dynamic technique uses knowledge of a scope greater than local to one processor. The more inclusive the scope the more effectively the technique will perform. In fact, if we maintain a global scope, we can mathematically guarantee an optimal work distribution [10]. However, implementations of these algorithms must ultimately map to the physical distribution of computing elements. Communication of information between elements has an associated cost. Therefore, there is a tradeoff between minimizing (1) the load imbalance and extra search and (2) the cost of communicating information that the dynamic distribution technique requires.

Recent research of parallel search problems has focused on minimizing this tradeoff function for a variety of different architectures[11][8][6]. Kumar *et al.* [6] present several techniques for distributed and shared memory architectures. They conclude that global techniques lead to better performance of parallel search. We present a distributed algorithm for message-passing architectures (Continuous Diffusion) that uses nearest-neighbor communication to provide a global perspective in time proportional to the diameter of the network.

We also considered the performance of an algorithm in which the search space is shared (Shared-List) and an algorithm in which the search space is distributed (Static Distribution). The unexpected poor performance of the Shared-List algorithm led us to create the Distributed algorithm. We found the Continuous Diffusion algorithm to outperform the other two algorithms for message-passing architectures due to its ability to distribute search space and dynamically propagate global information.

The paper is organized as follows: In Section 2, we describe the architectures, programming language, and search spaces considered. In Section 3, we present the parallel algorithms and analyze their performance. In Section 4, we consider general application of the Grid-Flow technique. In the final section, we summarize the results and suggest directions for future research.

2 Background and Preliminaries

We implemented the parallel search algorithms on both message-passing and shared-memory architectures. The architectures were simulated using the simulator developed by Delagi *et al.* [4]. The simulator models an architecture consisting of a number of processors that are interconnected by a direct communication network. The network supports a low-latency, cut-through, point-to-point routing protocol that includes multicast facilities [2]. Each processor consists of: an *evaluator*, a processor that executes an application code, an *operator*, a dedicated co-processor that handles message transfers, *interface buffers*, *memory*, *network ports*, and a dedicated *router*. The communication subsystem is simulated at approximately the register-transfer level. The operator is functionally simulated and the evaluator uses the machine running the simulation to actually execute and time the application code. Most components are parameterized, allowing a wide range of architectures to be modeled.

Most multiprocessor components are defined with a corresponding set of probes providing the ability to monitor critical aspects of that component's operation. The instrumentation supports various panels to allow visualization of a concurrent application. The visualization is used to understand latencies, bottlenecks, granularity, load balance, and the resource utilization in the systems we study.

2.1 Architectures

We simulated a message-passing architecture with an octorus interconnect. An octorus is a torus mesh in which each processor is connected with eight of its neighbors. Figure 1 illustrates an example octorus interconnection. Processors have associated local memories and they communicate by exchanging messages across communication links.

We also simulated a shared-memory architecture in which processors and memories are connected by a single shared bus. Figure 1 shows an example of the bus configuration. Processors may communicate directly or through a mutually shared location in global memory.

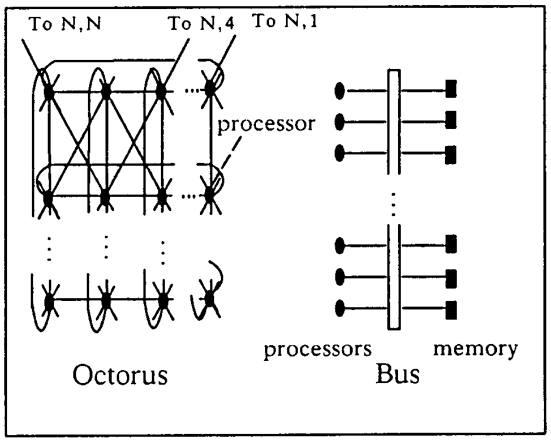

Figure 1 Architectures

2.2 Programming Language

To enable a programmer to develop and execute parallel algorithms, the simulator recognizes a set of extensions to LISP for expressing concurrency. These extensions are called LAMINA, a concurrent object-oriented language.

LAMINA's message-passing interface provides an object-oriented syntax for creating and communicating with objects that have associated state variables and methods. Objects reside at processors where they can send and receive messages to and from objects at other processors.

LAMINA's shared-memory interface implements the conventional shared-memory constructs. These include shared data-access operations for lists and arrays, and spin-locks and semaphores for synchronization.

2.3 Search Spaces

We executed search on a randomly generated Traveling Salesman Problem (TSP). TSP is a well-known member of the NP-complete class of problems and has been considered by other researchers [6] [11] [14]. Due to memory limitations of our simulation host we were not able to simulate problems of size comparable to those considered by Korf [7]. We do not consider this a liability, since our results (Section 3.3.2) indicate that the speedup increases for larger search spaces. The search spaces we experimented with are as follows:

Search Space	# Cities	# Nodes	Branching
0	5	82	4
1	10	3770	4
2	11	13000	4
3	15	210955	4

3 Search Algorithms

In this section, we describe three parallel search algorithms. A sequential Astar search proceeds by expanding the node in the search space with the lowest estimated distance to the solution and placing its successors on a queue of open nodes, if they have not already been expanded. The reader is referred to [14] for a detailed description of the sequential Astar algorithm.

3.1 Shared-List Astar

The theoretical analysis of Shared-List Astar predicts a linear speedup [10]. Thus it is an attractive candidate for parallelization. Irani and Shih [10] predict a speedup for Shared-List Astar as follows:

$$Speedup = p - \frac{p^2N - pN}{pN + (g-1)(2N - N_n)}$$

where p is the number of processors, N is the cost of the optimal solution, g is the number of suboptimal solutions, and Nn is the average cost of a suboptimal solution. Due to a large number of suboptimal solutions, the second term in the equation above is negligible for the search spaces and architectures we considered. Therefore, the expected speedup should be a linear function of the number of processors.

3.1.1 Implementation

In our Shared-List Astar implementation, the open queue and the closed list are in global memory, shared by all processors. In addition, the third list that we call the *successor list* is used to take maximal advantage of the available parallelism. Two types of search processes are created, *expanders* and *servers*. Expanders take a node from the open queue and place its children on the successor queue. Servers take nodes from the successor queue and update the open and closed lists.

3.1.2 Simulation Results

Our simulation results did not match speedup predicted by the equation above. Figure 2 shows the speedup of Shared-List Astar on the message-passing octorus and the shared-memory bus architectures. Note from this figure that Shared-List Astar execution time is optimal for four processors but degrades as the number of processors increases. Also, the octorus performed worse than the bus due to larger number of intermediate nodes necessary to access shared queues. We will explain the speedup degradation by breaking down the execution time into several components.

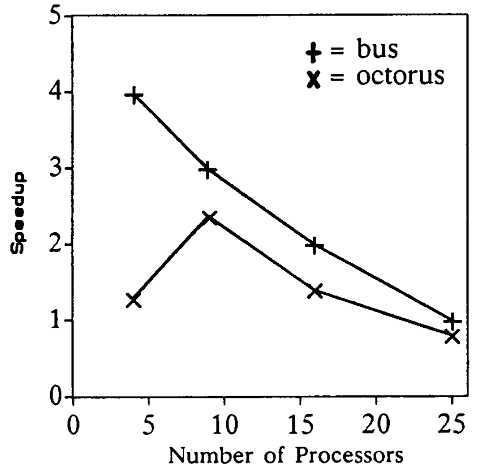

Figure 2 Speedup of Shared Astar.

3.1.3 Breakdown of Execution Time

We considered the following components of the total execution time: Process Creation, Communication, Synchronization, and Extra Search.

- *Process Creation*

  Process Creation is the time spent on spawning parallel processes. We measured this time to be less than .1% of total execution time and thus was not a significant performance limitation.

- *Communication*

  Communication is the time needed to access data from global memory. Our experiments showed communication delays to be significant for more than four processors as the volume of data transferred to and from global memory exceeded the capacity of the bus.

- *Synchronization*

  Synchronization is the time a process spends accessing a synchronization mechanism. We implemented synchronization mechanisms using both *spin-locks* (a process spin-waits for a lock to be released) and *semaphores* (a process is queued waiting for a signal from another process). We found that the synchronization was a significant portion of the execution time in Shared-List Astar. As the number of processors increased beyond four, the contention for shared queues became a major performance limitation. Also, for smaller number of processors, the spin-locks were more efficient than semaphores, while semaphores became more efficient as the number of processors increased.

- *Extra Search*

  Extra Search is the time needed to expand nodes not expanded by the sequential algorithm. Due to ordering of the global open queue, this time is low for Shared-List Astar.

3.2 Static Distribution Algorithm

The bottleneck in the Shared-List Astar was contention for the shared nodes. The Static Distribution algorithm (Distribution for short) is a straightforward approach that minimizes such contention by distributing nodes from the open queue.

Let us assume that the solution will fall with equal probability in any of p partitions of S. Thus an algorithm that partitions S into p partitions and distributes them to p processors should have a speedup of p. However, this is true only if each processor expands nodes that are expanded by the sequential algorithm. If this is not the case, then the parallel algorithm may have a speedup less or greater than p. The cause of sublinear and superlinear speedup are discussed in [9].

3.2.1 Implementation

In our implementation of Distribution, a search process is created at every processor. Each process is given the initial node. They work redundantly until there are more nodes in each open queue than processors. At this point each processor selects a portion of the open queue and continues working on this portion independently of other processors until it finds a solution.

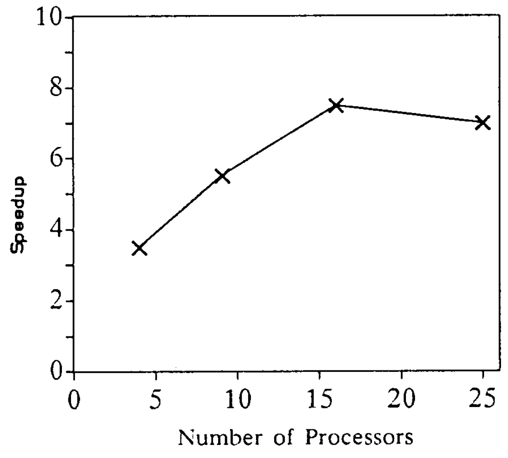

Figure 3 Speedup of Distribution Astar.

3.2.2 Simulation Results

Figure 3 shows the speedup of Distribution Astar on a message-passing architecture. Simulation results show that the efficiency is about 50% for less than 16 processors. We see, however, that speedup degrades as the number of processors increases beyond 16.

3.2.3 Breakdown of Execution Time

In order to recognize performance bottlenecks, we analyze the following components of the total execution time: Process Creation, Communication, Extra Search, Prepartitioning Bottleneck, and Solution Verification.

· *Process Creation*

The time to create processes is not significant for this algorithm since it was measured to be approximately 1% of the total execution time. The creation time is negligible since our implementation of a startup procedure creates all processes in time proportional to the diameter of the network. This startup procedure for 4x4 processors with octorus interconnect is illustrated in Figure 4.

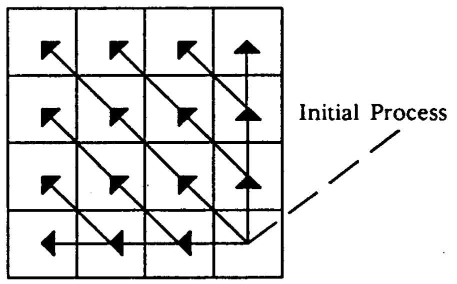

Figure 4 Diagram of Optimal Startup Technique

· *Communication*

The Distribution algorithm requires no communication for the greater portion of execution. The communication becomes a bottleneck in the Solution Verification.

· *Extra Search*

The Distribution algorithm expands a different set of nodes than the sequential algorithm. It can lead to either desirable *acceleration anomalies*, or a undesirable *deceleration anomalies* due to extra search (see [9]). Without knowledge of the location of the solution, a static partitioning has no ability to apply a heuristic to global information and thus avoid extra search.

· *Prepartitioning Bottleneck*

The prepartitioning bottleneck is the time spent in the sequential section prior to partitioning. For the search spaces and numbers of processors we considered, the prepartitioning bottleneck was not significant. For smaller problems and larger numbers of processors, however, the prepartitioning bottleneck would have more effect on the algorithm performance.

· *Solution Verification*

Once a solution is found, we must verify that it is better than the best node in all open queues. A verifying processor requests results from all other processors in order to find the optimal solution. We estimated that this technique introduced a bottleneck causing the speedup to decrease as the number of processors increased beyond 16, as indicated in Figure 3.

3.2.4 Discussion of Distribution

The Distribution algorithm can only perform well if an optimal search space partitioning corresponds to the simple distribution of the open queue. Although some speedup can be gained by this technique for small number of processors, extra search and load imbalance dominate the runtime for large number of processors and large search spaces.

As an aside, we noted that if we allow a solution to be within a certain distance from the optimal solution, a significantly better speedup can be gained. For search problems where an optimal solution is not necessary, Distribution offers a simple implementation with good results. We observed nearly linear speedup for up to 25 processors for solutions within 10% of optimal, as indicated in Figure 5.

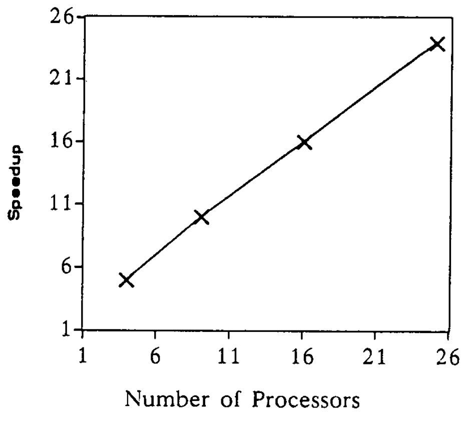

Figure 5 Sub-Optimal Distribution.

3.3 Continuous Diffusion

In order to reduce the extra search and load imbalance in Distribution, Continuous Diffusion continuously diffuses the search space among processors.

3.3.1 Implementation

The algorithm creates a set of processes as Distribution does, except that after expanding a certain number of nodes a process exchanges best nodes with four of its nearest neighbors. The algorithm is called Continuous Diffusion because near-optimal nodes continually move from the current global and local minima to processes that do not have optimal nodes.

The number of nodes a process expands between exchanges is a parameter δ. The conflicting goals of minimizing extra search and load imbalance versus minimizing communication overhead determine an optimal value for δ. When a process finds a solution, it must verify that the solution is better than the current global minimum. We can use the communication pattern of this algorithm to "share" the current minimum in the number of iterations proportional to the diameter of the network. This is discussed in more detail in the section on the Grid-Flow technique.

3.3.2 Simulation Results

Figure 6 shows a comparison of Distribution and Continuous Diffusion. Note from this figure that Continuous Diffusion performs well and outperforms Distribution when number of processors exceeds 9.

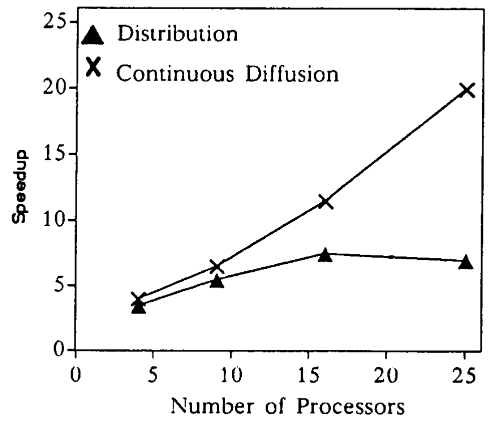

Figure 6 Speedup of Diffusion and Distribution.

Figure 7 shows the speedup of Continuous Diffusion for different search spaces (results discussed so far were for eleven cities). Note that the speedup approaches its optimal value as the problem complexity grows.

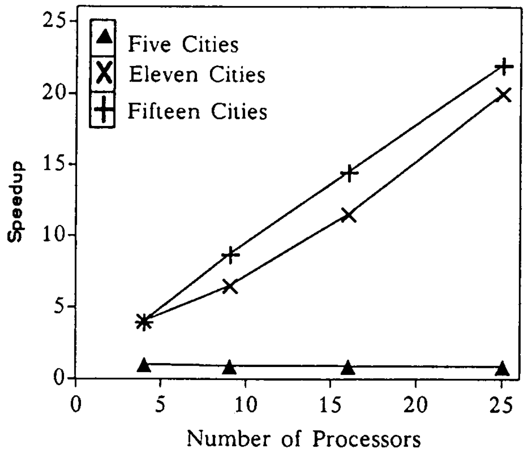

Figure 7 Speedup of different search spaces.

3.3.3 Breakdown of Execution Time

Again we characterize the overheads by the following components of the execution time: Process Creation, Communication, and Extra Search.

· *Process Creation*

This time is insignificant, due to the optimized startup technique described in the previous section.

· *Communication*

Communication time is a variable portion of the execution time dependent on the value of δ. We believe that the simulator overestimates the communication overhead compared to the existing message-passing architectures. The simulator does not allow the arrival of new message to interrupt a process. Therefore, a process must wait at regular intervals to receive information. This unnecessary overhead is reflected in the communication time. On a machine such as the Intel iPSC hypercube [5], where a message arrival can interrupt computation, we would expect performance to improve compared to our results.

· *Extra Search*

We measured that extra search is a substantially smaller portion of execution time for Continuous Diffusion than for Distribution.

3.3.4 Discussion of Continuous Diffusion

There are two conflicting goals for selecting an optimal value of δ. The amount of extra search and load imbalance are proportional and the communication overhead is inversely proportional to δ. Figure 8 shows that when added, the conflicting goals establish an optimal value for δ. The curve in Figure 8 represents the combination of two different curves. At small values of δ, communication represents a large portion of execution time, while extra search and load imbalance are minimal because the perspective is as global as possible. At large values of δ, communication represents a very small portion of the execution time while extra search and load imbalance are dominant.

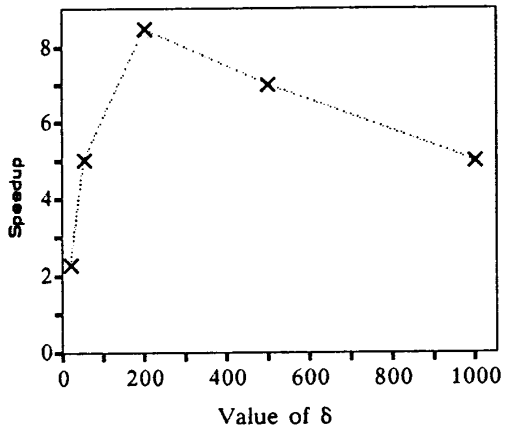

Figure 8 Speedup vs δ Value for 9 processors.

Kumar *et al.* [6] have implemented a parallel IDA* that achieves impressive speedup. While speedup is an important performance metric, it can be misleading when used to compare the performance of different algorithms applied for solving the same problem. IDA* expands many more nodes than A*, and although it results in impressive speedup due to straighforward decomposition, its total elapsed time may be longer compared to parallel A*. Korf [7] claimed that, although IDA* expands more nodes, it is faster than A* because it does not have to update open and closed lists. His two assumptions are that expanding the nodes is cheap and updating the lists expensive. In fact, we found that overheads for maintaining the open and closed list can be reduced substantially by placing open nodes in a heap and closed nodes in a hash table with heaps in the buckets. Applying the transition and heuristic functions, the major computation of expanding a node is expensive for many search spaces.

In some ways Continuous Diffusion is similar to the random strategy proposed in [6]. However, it differs in that information is guaranteed to flow at a certain rate from processor to processor, minimizing the amount of extra search and load imbalance. In addition, we found that checking for convergence, which was not considered in [6], can dominate the execution time, as we discovered in other distributed algorithms we studied [3]. We propose the Grid-Flow technique to minimize this overhead.

4 The Grid-Flow Technique

Our experience with implementing various algorithms for message-passing architectures indicates that the underlying model used in Continuous Diffusion to verify a solution is of relevance to other algorithms for message-passing architectures. This technique, that we call Grid-Flow, allows processes in nearest-neighbor algorithms to share global information in time proportional the diameter of the network.

Many nearest-neighbor algorithms also need to share some global knowledge [3]. One technique for accomplishing data sharing is to send all data to a master processor which performs some computation and then redistributes results to the slave processors. This technique introduces a bottleneck at the master processor which increases with more iterations or larger number of processors. However, sharing global information can be accomplished by using nearest-neighbor communication and allowing processors update global data before redistributing it. This is the premise of Grid-Flow.

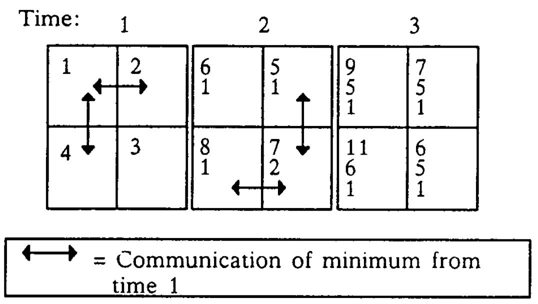

Figure 9 Example of Grid-Flow.

Figure 9 illustrates the Grid-Flow technique for Continuous Diffusion Astar. In this figure, each box represents a grid of 4 processors. Let d be the delay before all processors have the minimum value (d=2 in our example). It is proportional to the diameter of the network. Each processor keeps a list of: (1) its current minimum, (2) the minima from the d previous time steps. It sends this list to its nearest neighbors. After receiving its neighbors' lists, each process takes the minimum from its neighbors' lists for each time step and updates its list. The last item on the list is the global minimum d time steps ago. Therefore, all processors know about the global minimum in the time proportional to the diameter of the network.

5 Summary and Conclusions

We have presented three parallel Astar search algorithms: Shared-List Astar, Distribution, and Continuous Diffusion. We have simulated execution of these algorithms on two architectures, the message-passing octorus and the shared-memory bus.

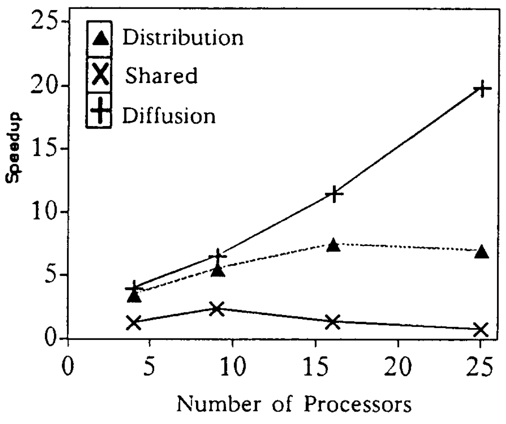

Figure 10 Algorithms comparison.

Figure 10 shows a comparison of all three algorithms on a message-passing octorus architecture. Our results indicate that for the search spaces and architectures we examined, distributed search-space algorithms outperform shared search-space algorithms for larger number of processors. We found that shared search-space algorithms did not perform as well as expected due to contention for shared data. Recently, several researchers have proposed special purpose shared-memory data structures [13]. We did not experiment with these techniques and assume that they have the potential to improve the performance of the Shared-List algorithm discussed in this paper.

The Continuous Diffusion algorithm we propose achieves 75-100% efficiency across the range of problem sizes. In addition, we believe that Continuous Diffusion is competitive with parallel IDA* because of the greater number of nodes expanded by

IDA*. The Grid-Flow technique we applied here can be used in many nearest-neighbor algorithms to share global information in time proportional to the diameter of the network.

As a future direction, we propose a demand-driven version of Continuous Diffusion. Instead of exchanging nodes automatically at each iteration, we can envision an algorithm in which nodes share the knowledge of the current global minimum via grid-flow. They can then request work from the process containing the global minimum. This algorithm may reduce communication overhead without sacrificing performance. We consider this a good candidate for future research.

Bibliography

[1] Aho, Hopcroft, Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

[2] G. T. Byrd, R. Nakano, and B. A. Delagi, "Multicast Communication in Multiprocessor Systems", *1989 Conf. on Parallel Processing*, 1989.

[3] Z. Cvetanovic, C. Nofsinger, and E. Freedman, "Parallel PDE Solvers, FFT, Monte Carlo Simulations, Simplex, and Sparse Solvers on Shared-memory Architectures", submitted to the *1990 Int. Conf. on Supercomputing*.

[4] B. Delagi, N. P. Saraiya, and G. Byrd,"LAMINA: CARE applications interface," *3rd Int. Supercomputing Conf.*, 1988.

[5] Intel Scientific Computers, "iPSC User's Guide".

[6] V. Kumar, K. Ramesh, and N. Rao, "Parallel Best-First Search of State-Space Graphs: A Summary of Results", *Proceedings of the 1988 National Conference on Artificial Intelligence (AAAI-88)*.

[7] R. E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence 27* (1985) 97-109.

[8] G. Li and B. W. Wah," Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms",*Proceedings of the 1984 International Conference on Parallel Processing*.

[9] T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Proceedings of the 1983 International Conference on Parallel Processing*.

[10] K. B. Irani and Y. Shih," Parallel A* and AO* Algorithms: An Optimality Criterion and Performance Evaluation", *Proceedings of the 1986 International Conference on Parallel Processing*.

[11] J. Mohan, "Experience with Two Parallel Program Solving the Traveling Salesman Problem", *Proceedings of the 1983 International Conference on Parallel Processing*.

[12] C. Nofsinger and Z. Cvetanovic," Parallel Monte Carlo Algorithms on Shared-memory Architectures", *Fourth SIAM Conf. on Parallel Processing for Scientific Computing, 1989*.

[13] N. Rao, and V. Kumar, "Concurrent Insertions and deletions in a priority queue", *Proceedings of the 1988 International Conference on Parallel Processing*.

[14] E. Rich, "Artificial Intelligence", University of Texas at Austin, McGraw-Hill Inc., 1976.

[15] R. Sedgewick, "Algorithms," Brown University, Addison-Wesley Co., 1983.,