Prioritization in Parallel Symbolic Computing *

L. V. Kale[1], B. Ramkumar[2], V. Saletore[3], and A. B. Sinha[1]

[1] Department of Computer Science, University of Illinois at Urbana Champaign,
1304 W. Springfield Ave., Urbana IL-61801
[2] Department of Electrical and Computer Engineering,
University of Iowa, Iowa City, IA-52242
[3] Department of Computer Science,
Oregon State University, Corvallis, OR-97331

Abstract. It is argued that scheduling is an important determinant of performance for many parallel symbolic computations, in addition to the issues of dynamic load balancing and grain size control. We propose associating unbounded levels of priorities with tasks and messages as the mechanism of choice for specifying scheduling strategies. We demonstrate how priorities can be used in parallelizing computations in different search domains, and show how priorities can be implemented effectively in parallel systems. Priorities have been implemented in the Charm portable parallel programming system. Performance results on shared-memory machines with tens of processors and nonshared-memory machines with hundreds of processors are given. Open problems for prioritization in specific domains are given, which will constitute fertile area for future research in this field.

1 Introduction

The field of Artificial Intelligence — in at least one of its interpretations — sets itself an ambitious goal: that of building computational systems that are capable of intelligent behavior that is on par with the best humans, and better. Building such systems will require a clear understanding of the structure and organization of intelligent systems, and their specific abilities, such as inductive learning, inference, planning and so on. Much research has been carried out (and is going on) on this front. As this work progresses, it is also becoming clear that a significantly large computational power will be required to integrate the strategies derived from the research into a system that can attain a desirable level of performance. Fortunately, recent advances in computer architecture have enabled construction of massively parallel machines with unprecedented levels of performance. Viewed in this light, it seems inevitable that parallel processing technology will be used to build AI systems of the future.

Many research issues must be dealt with before this technology can be used successfully in AI applications. One such issue — the one that we will deal with

* Some of the research reported here was supported in part by NSF under the grants CCR-89-02496, and CCR-91-06608.

in this paper — involves scheduling. In a parallel AI computation, there will be many computational subtasks that can be performed in parallel at a given moment. As the number of such tasks can be expected to vastly outnumber the number of processors, one must decide which of the tasks to execute next on which processor. These scheduling decisions have a very significant impact on the performance of many AI applications. This will also be illustrated with examples later in this paper. What mechanism should be used to specify a scheduling strategy? What scheduling strategies are useful in specific contexts? How should the chosen mechanism be implemented on parallel machines?

We believe that an appropriate framework for such applications is one in which subtasks are modeled as medium grained processes that are capable of creating new processes dynamically, and which communicate with each other mainly via messages. In such a framework, we argue that associating *priorities* with messages and processes is a good mechanism for implementing scheduling strategies. Section 2 includes descriptions of some regimes for which effective prioritization strategies are not yet known, and should be areas of active research. In Section 3, we describe *Charm*, a portable parallel programming system that provides such a framework, and supports priorities. This system was used to carry out the experiments described in this paper. In Section 4, we examine several search regimes (e.g. state-space search and game tree search) and describe how specific priority-based scheduling strategies can be used to effectively parallelize them. Section 5 describes techniques for supporting priorities in parallel systems.

2 Alternate Scheduling Mechanisms

Although the scope of AI strategies in general is quite broad, we will focus on a subset that can be characterized as "tree structured computations" or "search computations". This by itself is quite a large class, as illustrated by the list of specific subclasses discussed in Section 4. Such computations can be viewed as a process of developing a tree (starting with a single-node tree, usually), by adding nodes to it, pruning subtrees, and propagating information up and down the tree.

One method for parallelizing such computations is to think of the tree as a shared data structure on which the processors "walk". Such an approach is often used on shared-memory machines with only a few processors. Scheduling strategies can then be expressed as tree traversals. For example, a strategy component might be: "if you are at a leaf node, traverse upward in the tree upto the first node that has an unexplored child". Although this mechanism is sometimes intuitive, it incurs substantial performance penalties on large machines (particularly the ones with nonshared-memory) where pointers between nodes in the tree may often cross processor boundaries. Also, implementation of such strategies is complicated by the intricacies of sharing memory — to avoid deadlocks and race conditions, for example.

A process-based parallelization of such computations is conceptually simple.

Each node of the tree is implemented as a process. New nodes are added to the tree by creating new processes, and propagation of information up and down the tree is implemented via messages.

As the number of nodes in the tree at any moment during its computations may be (and often is) much larger than the number of processors, the underlying system must support multiple processes per processor. In addition, a parallel implementation of such a process-model must deal with the issues of (1) grain size control, (2) dynamic load balancing and (3) scheduling.

Grain size control deals with the problem of amortizing the overhead of process creation and message-passing, typically by combining many processes into a single process. Dynamic load balancing techniques are needed to ensure that processors are effectively utilized, to complete the execution of the overall computation as fast as possible.

Scheduling, in contrast, deals with the question of which messages and processes, among the many available ones, to execute next. It is particularly important for *speculatively* parallel computations, where some of the tasks that can be carried out in parallel may turn out to be futile or unnecessary [29]. Here, the order in which tasks are executed has an impact on the total amount of computations performed — by affecting the number of messages/processes that must be processed before the problem is solved. Speculative computations in parallel functional languages have also been investigated in [1].

What mechanisms can be used to specify and implement scheduling strategies?

1. Assign fractions: In this strategy, each process is given a promise of a certain fraction of the overall CPU-time available in the system. When a process forks sub-processes, it assigns to them fractions of the fraction that was allocated to it. Such a strategy can be implemented in many different ways: for example, a process given 10% of the cpu-time may be given 10% of the processors in the system, or be allowed to run on all the processors in the system for 10% of the time. Many other shades in between are also possible.
2. Assign times: Each process is give a certain fixed cpu-time. It may assign some of its time to its child processes as above. This mechanism is different from the above in that the quanta of cpu-time assigned is a consumable resource — a process is terminated when the quanta assigned to it finishes.
3. Assign priorities to processes and messages. The parallel system must then try to adhere to the priorities to the extent possible.

We will explore the last option, although the first two have their merits in specific situations. In this paper, we will describe (a) how these mechanisms can be used effectively in different search domains, and (b) how priorities can be implemented effectively in parallel systems. First, we will introduce Charm, a portable parallel programming system which supports dynamic creation of small grained processes, and which was used in the implementations described in the paper.

3 Charm: The Parallel Programming Framework

Developing parallel application programs is currently difficult due to (a) the diversity of parallel architectural platforms, (b) the inherent difficulty of parallel programming and (c) the difficulty of reusing parallel software. Due to the diversity, programs written for one parallel machine don't usually run on another machine by a different vendor.

Charm [7] is a parallel programming system that we have developed to address this problem. Charm provides portability and supports features that simplify the task of parallel programming. Programs developed with Charm run efficiently on different shared-memory and nonshared-memory machines without change. It currently runs on Intel iPSC/860, NCUBE, CM-5, Sequent Symmetry, Multimax, Alliant FX/8, networks of workstations, and will be ported to machines including the Intel Paragon in near future.

Charm is one of the first systems to support an asynchronous, message driven, execution model [7]. This allows for maximum overlap between computation and communication and facilitates modular program organization. Recognizing that parallel programs involve distinct modes of sharing information, it supports six modes in which information may be shared between processes. These modes are implemented differently on different machines for efficiency. The system supports (static and) dynamic load balancing and prioritization. Charm was designed to support reuse of parallel software modules and includes specific features for promoting it.

From the point of view of this paper, the important features of Charm are that it supports dynamic creation of processes (called *chares*) and allows multiple processes per processor. A process is activated when a message for it is picked up for execution (or when an initial message containing the "seed" for a new process is picked up for execution). The process is allowed to complete the processing of this message before the system picks up another message — for the same or different process — for execution.

4 Applying Prioritization

In the following subsections, we will describe how we were able to obtain effective speedups by using (specific) prioritization schemes in various search domains. In some of the domains, there still remain open problems signifying the need for better prioritization strategies. The domains discussed include: state-space search, iterative deepening, divide-and-conquer, best-first and branch-and-bound search, AND-OR trees and problem reduction, game trees, and bi-directional search.

4.1 State Space Search

In a state-space search problem, one is given a starting state, a set of operators that can transform one state to another, and a desired state. The task is to find

a sequence of operations that transform the start state to the desired goal state. The desired state may be described by a set of properties it must satisfy, and there may be many such states in a given state-space. One can imagine a tree with the starting state as its root, and for any state S in the tree, all the children states that can be obtained by one application of any rule to S. This tree is called the search tree, and is usually implicit, in that it is not explicitly represented in the computer program or data. A state-space search program can be thought of as traversing this tree. A depth-first search strategy is usually implemented using a stack of states in sequential programs. The search begins with the starting state on the stack. From then on a node from the top of the stack is picked up and examined. If it is a goal state, the solution may be recorded. If it is a dead-end state, it can be discarded. Otherwise all possible operators are applied to the state to produce the set of its children states. These are then pushed onto the stack, possibly using some local value-ordering heuristic so that the child most likely to lead to a solution is kept at the top of the stack.

Parallelizing depth-first search may therefore seem simple: instead of searching successor states one after the other, search them in parallel. As the search tree grows exponentially in the depth of the tree, one may also expect to generate a large degree of parallelism. Indeed, if one is looking for all solutions, this is as simple as that, except for the important problems of load balancing and grain size control. Earlier, we worked on the all-solution problem using the Chare-Kernel machine independent parallel programming system [12], which provides dynamic load balancing among other facilities. With this, we were able to obtain very good speedups for many depth-first search problems. Other work on dynamic load balancing for this problem includes that of Kumar and Rao [28, 18], who describe an idle-processor initiated load-balancing scheme which splits (i.e. divides the nodes on) the stack of the donor processors, and [4] which relies on a hierarchical load balancing scheme.

When one is interested in any one solution, such parallelization techniques lead to difficulties. If we search two successors of a state (assume there are only two for simplicity), the solution may lie in the sub-tree of either node. If it lies in the sub-tree of the first node, the work in the second sub-tree will be wasted. Exploring the two subtrees in parallel is thus speculative — we may not need both those sub-computations. This fact, and the resultant speedup anomalies were noted in a branch-and-bound search which is closely related to depth-first search, by Lai and Sahni [19].

One may get deceleration anomalies where adding a processor may actually slow down the search process in finding a solution. This may happen because the added processor may create some "red herring" work that other processors end up wasting their time on. In extreme cases, this may lead to detrimental anomalies, where p processors perform slower than 1 processor performing the search. It is also possible to get acceleration anomalies: a speedup of more than p with p processors. This can happen because the added processor picked a part of the search tree that happened to contain the solution. Kumar *et. al.* noted this in the context of parallel depth-first search. They reported a speedup varying

between 2.9 to 16 with 9 processors for a 15-puzzle problem [28].

We started with the dual objectives of (1) ensuring that speedups are consistent — i.e. do not vary from one execution to another and (2) ensuring that the speedups increase monotonically with the number of processors, preferably being as close to the number of processors as possible. With that objective, it is clear that all the work that is done by the sequential program is "mandatory" whereas all the other nodes not explored by the sequential algorithm are "wastage".

Our scheme, described in [13, 31] is based on bit-vector priorities, and builds upon an idea in [20]. Each node in the search tree is assigned a priority. Priority bit-vectors can be of arbitrary length, and their ordering is lexicographic — the lexicographically smaller bit-vector indicates higher priority. The priority of the root is a bit-vector of length 0. If the priority of a node is X, and it has k children, then the priority of the i'th child is X appended by the $\lceil log k \rceil$-bit binary representation of i. Thus, if a node with priority 01101 has three children, their priorities will be 0110100, 0110101, and 0110110, from left-to-right. It can be shown that lexicographic ordering of these priorities corresponds to left-to-right ordering of the nodes in the tree. To be sure, there is a loss of information in the bit-vector representation: A node with priority 0110110 may be at level 7 of a binary tree, or level 3, with the top-level branching factor of 2, and the next two (grand-parent and parent of this node) with a branching factors of 7 and 5 respectively, among many other possibilities. Fortunately, this loss of information does not destroy the left-to-right ordering in a specific tree, and saves much in storage and comparison costs over a scheme that assigns a fixed number of bits to each level. Figure 1 shows an example of how such priorities are assigned to nodes of a search tree.

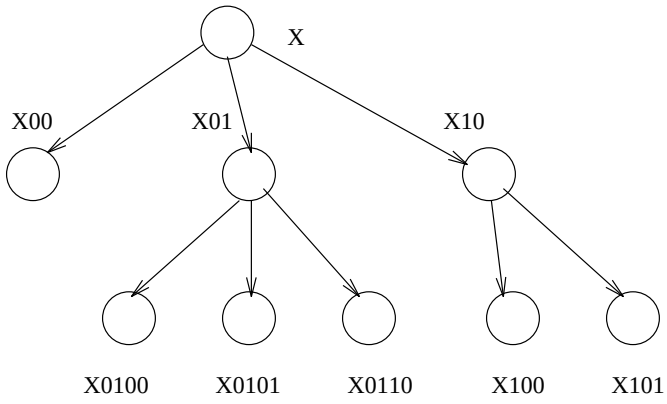


Fig. 1. Illustration of the assignment of bit-vector priorities to nodes. The priority of the topmost node is assumed to be X.

The complete scheme, described in [13], involves a few additional subtle points of strategy. In particular, a technique called *delayed-release* is used to further reduce the wasted work, and reduce the memory requirement to roughly a sum of $D + P$ where D is the depth of the tree, and P is the total number of processors. Most other parallel schemes for depth-first search require storage proportional to roughly $D * P$. Delayed-release works as follows: A process responsible for expanding a node would normally fire a new process for each child node. Instead, it now simply stores the nodes in a list, and goes on to expand the leftmost child. It continues this process until it reaches a node (say N) deemed to be small enough (by the domain specific grain size control heuristic) to search sequentially. At this point, it fires (releases) processes for each of the nodes accumulated in the list, and then goes on to carry out the sequential search under the node N.

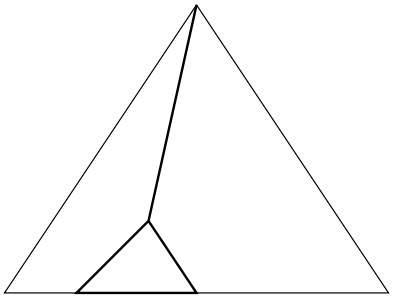


Fig. 2. The prioritization strategy leads to a characteristic broom-stick sweep of the search space.

The scheduling induced by this strategy sweeps the search tree from left-to-right. Moreover, at any moment, the set of "active" nodes — i.e. all nodes being expanded by the processors and their ancestors — form a characteristic shape resembling a broom with a long stick, as shown in Figure 2.

This strategy was implemented and tested with various state-space search problems on shared-memory and small nonshared-memory machines. A sample of the performance data, taken from [29], is shown in Table 1.

This strategy was also applied successfully in obtaining consistent speedups to one solution in test pattern generation for sequential circuits by Ramkumar and Banerjee [27]. In test generation for sequential circuits, we once again have a state space where the nodes in the search space represent assignments to primary inputs or inputs to flip-flops (called *pseudo-inputs*) in the circuit being tested. Typically, in the one processor case, heuristics are used to determine the order in which assignments are made to the inputs. These heuristics were supplemented by our prioritizing scheme for parallel execution. For any given heuristic, our

Table 1. Performance of the prioritization strategy on Sequent Symmetry. All times are in Seconds.

Processors	1	2	4	8	12	16	18
126-Queens	202.0	100.5	51.0	26.3	18.1	14.0	12.7
8x8 Knight's Tour	113.0	52.9	26.5	13.1	8.9	6.6	6.1

scheme was able to consistently speedup the execution time as the number of processors were increased.

The test generator is initially provided with a list of faults in a given circuit for which test vectors need to be detected. A test vector sequence is a solution for a fault if it can successfully propagate the fault to a primary output in the circuit. Whenever a sequence of test vectors is found to detect a given fault, a fault simulator is invoked to determine whether this test vector serves as a solution for other faults in the fault list. All such covered faults are dropped from the fault list. Each fault is assigned a time limit which bounds the computation time that can be devoted to finding a solution for a single fault. If no solution can be found within the time limit, the test generator has failed to find a solution that may exist in the search space. If the entire search space has been explored unsuccessfully, the fault is called a *redundant* fault. The efficiency metric in Table 2 reports the percentage of faults which are redundant or for which test vectors have been detected.

In Table 2, we quote some of the results presented in [27] for an 8-processor Intel i860 hypercube. The test generator, called ProperTEST, was developed using the Charm system and, as a result, ran unchanged on a variety of machines, including a network of Sun Sparc I workstations, a Sequent Symmetry, an Intel i860 hypercube and an Encore Multimax.

In column 1 of Table 2, the benchmark circuits used are identified. In column 2, the number of PEs used in the experiment is listed. In columns 3 and 4, the time spent in test generation and fault simulation phases of the computation is reported. Finally, in column 5, the efficiency of the test generator is presented. The efficiency metric reports the quality of the solution obtained. It is important that speedup is not obtained at the cost of poor fault coverage by the test generator. As can be seen from the results, the use of priorities was instrumental in obtaining consistent speedups *without* significant loss in quality.

There is an interesting postscript to the research on state-space search. As we began experimenting with specific applications, such as the N-queens problem, we attempted to improve the heuristics used in the search, to ensure that the speedups were measured against the "best possible" sequential algorithm. For the N-queens problem, this led to such a good heuristics that it almost always led to a first solution, without much search [11]. This was a true heuristic, as distinct from the well-known closed form solutions to the N-queens problem, in that it can be used continually to generate multiple solutions beyond the first

Table 2. Execution times (in seconds) of the ProperTEST test pattern generator for sequential circuits on selected ISCAS89 sequential benchmark circuits on the Intel i860 hypercube. All reported execution times are in seconds.

Intel i860 hypercube (Message Passing)

Circuit	#PEs	Test gen. time	Fault sim. time	Efficiency
s386	1	184.4	2.7	100
	8	28.8	1.3	100
s713	1	27.0	3.7	98.8
	8	6.6	1.0	98.8
s5378	1	6016.5	184.8	75.3
	8	901.7	38.6	72.7

one. (The 126 queens results quoted in Table 1 is prior to the use of this newly discovered heuristic). A similar experience was obtained for the 3-satisfiability problem [3]. As long as a solution existed, the heuristic we developed was able to find it without much search! We plan to experiment with other heuristics for NP-complete problems.

4.2 Iterative Deepening

Sometimes, one is interested in an optimal solution to a search problem. If an admissible heuristic is available [22] one can use the A* algorithm, which ensures that the first solution found is the optimal one. However, A* requires large memory space on the average, and degenerates to breadth-first search in the worst case. An iterative deepening technique can be used in such a situation: due to admissibility property, we know that the cost of the solution cannot be less than the heuristic value of the root. So, we can conduct a depth-first search, but restrict ourselves to not search below nodes that exceed the bound given by the heuristic value of the root. If no solution is found, we can search for the next possible bound. This can be obtained by keeping track of the heuristic values of the unexplored children (of the explored nodes), and picking the minimum from these. Alternatively, in some problems, the increasing sequence of bounds is clear by the nature of the problem definition itself. For example, in the well-known fifteen puzzle problem, if the cost measure is the number of steps required to produce the goal state, it can be shown that the bound must increase by two in every successive iteration. This process is continued until a solution is found. This algorithm was defined by Korf [17], and is called IDA*, for Iterative Deepening A*. As in A*, the first solution found is an optimal solution in IDA* too. Each successive iteration duplicates all the work done by the previous iteration. However, as the tree-size increases exponentially in the depth of the tree, the cost of the last iteration dominates, and this duplication is not too expensive. Even with a binary branching factor, the duplication cost is at most 100%, which is tolerable considering the significant memory savings.

As each iteration of IDA* is a depth-first search, it can be parallelized using the techniques described above. Kumar *et. al.* in [28] were the first to demonstrate parallel schemes for this problem. Their results did exhibit speedup anomalies for single solutions, and they reported speedups to all solutions (as their primary interest was to demonstrate the efficacy of their load balancing scheme). It is true that all the optimal solutions exist in the last level of the tree (and therefore all the previous layers are completely explored irrespective of the order in which the nodes are explored.) However, the last iteration — complete exploration of the last level — is typically larger than all the previous ones combined. The order in which nodes are explored in the last iteration affects when the first solution is found, and thus the notion of speculativeness prevails in this context too.

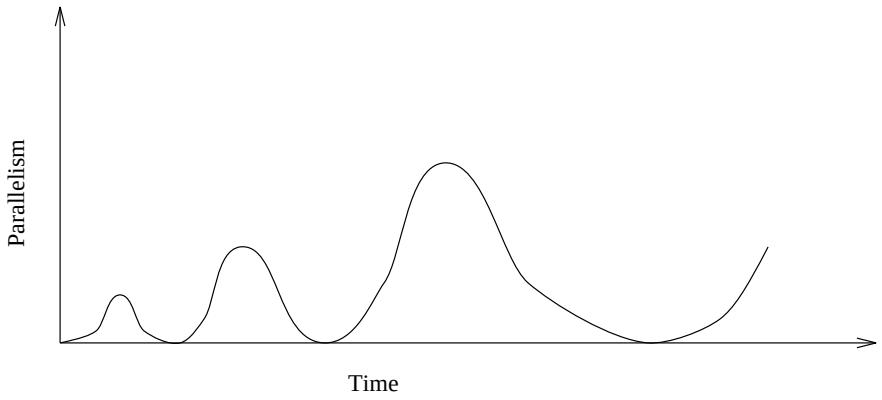


Fig. 3. The nature of inherent parallelism in IDA*.

Our prioritization techniques described in Section 4.1 were successful at obtaining consistent and monotonic speedups for this problem. However, the speedup with these techniques alone are not as high as they could be, although for each *iteration*, we obtained close to the best possible speedups. The difficulty is that the parallelism in this problem increases and decreases in waves with each iteration, as shown in Figure 3. At the beginning of each iteration the parallelism is low. It increases quickly to occupy many processors, and then trails off toward the end of the iteration.

If we knew that $n+1$'th iteration was necessary, we could start it concurrently with n'th iteration, so that by the time n'th iteration finishes, the next iteration is already running full-swing, thereby keeping the processors busy. Even with this knowledge, we would have to make sure that the next iteration did not generate work that kept processors from working on the previous iteration. However, seen from this perspective, this problem seems amenable to prioritization. Just

assign higher priority to earlier iterations, and allow multiple iterations to run concurrently. This scheme is described in [32].

The fact that we do not know which iterations are necessary can be handled as follows: We start K iterations in parallel. Whenever any one of them finishes without reporting a solution, we start the next iteration. Whenever a solution is reported from the i'th iteration, we (a) do not start any new iterations, (b) terminate all the iterations larger than i, and (c) wait for all iterations smaller than i to finish, as they may have a better solution. This scheme might seem wasteful, because we are generating a new class of speculative, and potentially wasteful, computations (viz. the iterations beyond the optimal-solution depth). However, with proper use of priorities, our experiments suggest that work on subsequent iterations is done only if no work on the current iteration is available.

To assign higher priorities to earlier iterations, we assign a non-empty bit-vector as the priority of the root node in every iteration, as opposed to an empty one in the baseline algorithm. This priority must be so chosen that every descendant of the root of an iteration has lexicographically smaller bit-vector compared to any node in a subsequent iteration. If the maximum number of iterations is known, this can be accomplished easily by assigning a binary representation of the iteration number as the priority of its root. However, in general, this number is not known. A binary coding scheme we developed solves this problem neatly. The first iteration is given a representation: "0", the second and third one "010" and "011" respectively, the 4'th through 7'th are assigned "00100", "00101", "00110", and "00111", and so on. It can be verified that this representation satisfies the required property — any extension of 011, for example, is lexicographically smaller than all extensions of 00100.

With this scheme we were able to "soak up" the computing resources during the previously idle periods without increasing the wastage, and produce almost perfect speedups even for small-sized problems. The improvement obtained can be seen in Table 3, taken from the data in [29]. The slight superlinearity seen in the data is not surprising, and it occurs because the optimal solution is reported before all the nodes to the left of the solution node are explored — other processors may still be working on such nodes. See [13, 32] for details of this scheme, and additional performance data.

Table 3. Performance of IDA* solving a 15-puzzle instance on Sequent Symmetry. The numbers shown are speedups. The sequential execution time for this instance was 116 seconds.

Processors	1	4	8	12	16	18
Basic Parallel IDA*	1	3.8	6.7	8.9	10.8	11.7
2 Concurrent Iterations	1	4.0	7.9	11.5	15.0	16.5
3 Concurrent Iterations	1	4.0	8.0	12.0	16.2	18.3

4.3 Divide And Conquer: Memory Usage

A divide-and-conquer is a deterministic computation, without any speculative parallelism. A problem is divided into two or more subproblems. This subdivision continues recursively until subproblems are "small enough" to be directly solved. Solutions to the subproblems are "combined" to form a solution to their parent problem. The computation can thus be seen as the process of growing the tree downwards, and then passing information up the tree, combining it at the intermediate nodes, until the root node forms the final solution. In a process-based parallel formulation, each node of the tree is made into a process, with the last few levels of tree being combined into one process for the sake of grain size control (this is just one of many possible methods for grain size control that can be used in tree-structured computations).

Without any speculative parallelism, it may seem to be futile to attach priorities to processes. However, significant savings in memory usage can be obtained by using the left-to-right priorities (as used in state-space search). Because of the broom-stick sweep, the memory used with P processors is proportional to $D + P$, for a D-deep tree, instead of $O(D * P)$, which would have been the memory usage without the use of priorities. The $O(D * P)$ could be obtained by using a LIFO strategy for dealing with new processes and messages. Further reduction in memory usage can be obtained by giving a higher priority to messages carrying solutions to subproblems, in relation to the messages carrying subproblems to be solved. While the former may result in reduction in memory usage by finishing the parent subproblem, the latter often results in creation of more processes. This reduction can be effected by attaching the prefix "0" to the priority of all solution messages, and "1" to that of the new processes. Note that if 1X was the priority attached to a message carrying a subproblem, then the message carrying a solution to it should bear priority "0X".

4.4 Branch-and-bound and Best-First Search

The Traveling Salesperson Problem (TSP) is a typical example of an optimization problem solved using branch-and-bound techniques. In this problem the salesperson must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with it (if $i = j$, then C_{ij} is assumed to be of infinite cost).

In the branch-and-bound computation one starts with an initial partial solution, and an infinite upper bound. New partial solutions are generated by *branching* out from the current partial solution. Each partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. For every partial solution, a lower bound on the cost of any solution that can be found by extending the partial solution is computed. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. Two (or more) new partial solutions are obtained from the current partial solution by including and excluding

the "best" edge (determined using some selection criterion) not in the partial solution. The upper bound is updated whenever a solution is reached.

Note that the left-to-right tree-traversal strategy that we used for state-space search is inappropriate for this problem. To maximize pruning, we would like to process nodes with lowest lower-bounds first. This leads to a form of best-first search. To parallelize this computation with a prioritized scheduling mechanism, we associate the lower bound of a node with the priority of the corresponding process, with lower values signifying higher priorities. (The parallelization scheme also needs an ability to propagate the cost of the best-known solution at the current time, so that it is accessible from all processors. The *monotonic variable* abstraction supported in Charm provides this capability).

This prioritization scheme was implemented and tested on many versions of the Traveling Salesperson Problem. The major challenge for the priority balancing strategy was to ensure that the number of nodes expanded in a parallel search is not much more than those expanded in sequential search. This implies trying to implement a good degree of adherence to priorities, while still preserving good load balance, and avoiding any bottlenecks. With appropriate choice of priority balancing strategies (described in section 5), we were able to demonstrate good speedups even on 512 processors of an NCUBE machine, as shown in Table 4.

Table 4. The figure shows the execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 with upto 512 processors using the tokens strategy to balance load. In this case the cluster size is 16 processors.

Processors	1 (estimated)	64	128	256	512
Time	19,366	302.6	151.1	86.2	42.1
Estimated Speedup	1	64	128	225	460
Number of Nodes expanded	-	85,165	84,030	93,816	85,420

It should be noted that the lower bounding methods we used are simple, almost naive, methods. Much more sophisticated methods exist today which would cut down the number of nodes generated for this problem by many orders of magnitude. The simple algorithm was sufficient for our purpose here, as we simply wanted to demonstrate how speculative parallelism can be controlled in this context. With better algorithms, one can use our strategies and attempt to solve much larger problems.

As even larger branch-and-bound problems are attempted, we anticipate memory overflow problems. This is because, in the worst case, best-first search can be as bad as breadth-first search for memory usage, and never as good as the depth-first search. As the spread between memory requirements of depth-first and breadth-first varies from a linear to exponential function of the depth of the tree, one can expect memory overflows on many problem instances. Again, a

priority based scheme can be used to formulate a solution to this problem. The memory overflows can be avoided by using a prioritization strategy that adapts to the current memory usage. When memory utilization is very high (say more than 90%), one may switch to a depth-first strategy for all new nodes being generated. This can be accomplished with priorities alone as follows: let U be the maximum value the priority make take in the normal strategy, and let D be the maximum depth of the tree. Instead of using a lower-bound x as the priority of a node, we will use $D + x$. When we detect that the memory usage is high, we assign priorities differently. Each node at depth d is assigned a priority $D - d$. Thus, the priority of a node generated after the memory threshold has been reached is always higher (i.e. numerically smaller) than any node generated earlier, as $D - d < D + x$. This will have the effect of finishing off the nodes from the "original" priority queue by completing the depth-first search under each node. Once this strategy reduces memory usage below the preset threshold, we can switch back to assigning the lower-bound based priorities as above (i.e. D+x). The system will thus finish an adequate number of nodes created prior to this point in a depth-first (LIFO) fashion, but then revert back to a best-first pattern. Controlled use of memory has also been used in [35] by combining good features of A* and IDA*.

Another reason for higher memory usage in a prioritized strategy is the potentially large number of low priority nodes that may "rot" in the queue. These nodes represent work that is pruned due to some solution found earlier. However, until they are examined, they won't be discarded; and as we are proceeding in priority order, they will not be examined for a long time. A solution to this problem is to provide a "flush" primitive in the system that would delete all messages below a certain priority level. This can be used whenever a new better solution is found, to clean up the queue. An alternate solution is to switch to a "garbage-collection" mode globally (across all processors) on some trigger — such as high memory utilization on some processor, or as a periodic cleanup phase. Under this mode, all processes simply examine the lower bound of the node they are meant to expand, and discard it (if its lower bound is larger than the current upper bound, as usual), or else simply store it in another repository process. When all the nodes have been cleared in this fashion (and this condition can be detected by the quiescence detection algorithm in Charm), the repository processes on all the processors are awakened, and they create new processes for all the non-garbage nodes.

4.5 Logic Programming: AND-OR and REDUCE-OR trees

A Pure Logic Program is a collection of predicate definitions. Each predicate is defined by possibly multiple clauses. Each clause is of the form: $H :- L_1, L_2 ... L_n$, where the L_i's are called the body literals. (A literal is a predicate symbol, followed by a parenthesized list of terms, where a term may either be a constant or a variable, or a function symbol followed by a parenthesized list of terms). A clause with no body literals is called a fact.

A computation begins with a query, which is a sequence of literals. A particular literal can be solved by using any of the available clauses whose heads unify with the goal literal. In the problem-solving interpretation of a Logic Program, each literal corresponds to a (sub) problem, and different clauses for a predicate correspond to alternate methods for solving the problem. Also, it is possible to have multiple solutions for a given problem. So, again, when one is interested in only one solution, the problem of speculative parallelism arises. This is further complicated by the presence of AND parallelism, which is the parallelism between multiple literals of a clause (or, in problem-solving terminology: that between multiple subgoals of a particular method).

REDUCE/OR Process Model: Our work on speculative computations in Parallel Logic Programming was conducted in the context of the REDUCE/OR process model (*ROPM*), proposed and developed in [9, 14, 8]. The past and ongoing work related to this model in our group includes development of a binding environment [15] and a compiler [26, 25]. The REDUCE/OR process model exploits AND as well as OR parallelism from Logic programs, and handles the interactions of AND and OR parallelism without losing parallelism. It is also designed so that it can use both shared and nonshared memory machines. The compiled system executes on the NCUBEs and Intel's hypercubes, as well as on many shared-memory machines such as: Sequent Symmetry, Encore Multimax, Alliant FX/8, etc. Thus, when we started working on first-solution speedups in Logic Programs (i.e. with speculative computations), it was clear to us that we must work within the framework of *ROPM* to retain its advantages. This added one more constraint on the possible schemes. Speculative work in OR-parallel Prolog has also been investigated by Hausman in [6].

We first worked on simply improving the first solution speedups in *ROPM* compared with the then prevalent scheduling scheme, which was a *LIFO* scheme, with each processor having its own stack. This is described below. The work described in Section 4.1 on pure state-space search came later, and encouraged by those results we set a new objective of consistent and monotonic speedups. The resultant work is described subsequently.

Speedups for a First Solution: The REDUCE/OR process model is based on the REDUCE/OR tree [10], which is an alternative to the traditional AND/OR tree. It overcomes the limitations of AND/OR trees from the point of view of parallel execution. The detailed description of the process model can be found in [9]. What concerns us here is the process structure generated by ROPM. Each invocation of a clause corresponds to a process, called a REDUCE process (with the exception of clauses and predicates explicitly marked sequential: these are used for granularity control). The REDUCE process uses a dependence graph representation of the literals in the clause. It starts with a tuple of initial bindings to its variables, and fires OR processes for each literal that can be fired without waiting for any other literal, according to the graph. Each OR process may send multiple solutions. Each solution results in a new binding tuple, which may

trigger firing of other OR processes for dependent literals. For example, consider a clause with four literals, with the dependence graph represented by:

$$h(I,T) \ :- \ t(I,X) \ \rightarrow \ (u(X,Y) \ // \ v(X,Z)) \ \rightarrow \ w(Y,Z,T).$$

When an instance of this clause is activated, an initial binding tuple with variable I bound to some value, and other variables unbound, is created. One OR process for solving p with this initial binding of I is then fired. For every value of X returned by t, one u and one v process is fired immediately. Thus, there may be multiple u (and v) processes active at one time. Each value of Y returned by u is combined with compatible values of Z (i.e. those that share the same X value) returned by the corresponding v process, and for each consistent combination, a w process instance is fired. Each OR process, given an instantiated literal, simply fires off REDUCE processes for each clause that unifies with the literal, and instructs them to send responses directly to the OR-process's parent REDUCE process. Thus, the process tree looks similar to a proof tree, rather than to an OR tree (or SLD tree). This fact is important in understanding (as well as designing) the scheme we proposed.

In the compiled implementation of ROPM, the requests for firing processes were stored and serviced in LIFO fashion. On (small) shared-memory systems, this was done using a central shared stack, whereas on nonshared-memory machines, a separate stack was used on each processor, and a dynamic load balancing scheme moved such requests from one processor's stack to another's. Although this strategy resulted in good use of memory space, it had one drawback (if one is interested in just one solution): all the solutions tended to appear in a burst toward the end of the computation, for problems that involve AND as well as OR parallelism. It is easy to see why, with a different and a simpler, example. Suppose there is a clause with two AND-parallel (i.e. independent) literals, p and q. When the clause fires, it pushes p and q process-creation requests on the stack. Assume p is on top, without loss of generality. Literal p may have a large sub-tree, with many solutions, and so all the processors in the system may be busy working on p. This will result in production of all solutions to p before any solutions to q. (Of course, toward the end a few processors will be working on q while others are finishing up p). However, from the point of view of reporting the first solution faster, the system should focus attention on q as soon as one solution from p is obtained. In addition, if there are two alternative computation-intensive clauses for p, we should have the system concentrate its resources on one clause (and its subtree) rather than dividing them arbitrarily among the two.

The solution we proposed used bit-vector priorities, with the root having a null-priority. An OR process with priority X assigned a priority to each of its children, by appending the child's rank to X (as described in the section on state-space search). A REDUCE process uses a more complex method for assigning priorities. If it contains only AND parallel literals, such as p and q in the example above, they receive identical priorities. When the literals form a more complex dependence graph, such as the clause consisting of t,u,v and

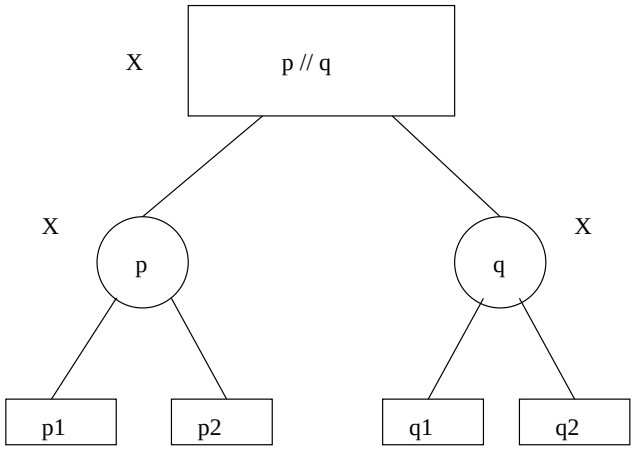


Fig. 4. The Prioritization Scheme applied to a simple and-or tree

w above, priorities are assigned such that the literals closer to the end of the dependence graph receive higher priority than those that precede them in the graph. This is done by assigning "distance bits" to the priorities, which signify the distance from the end of the dependence graph. Thus, for example, the u and v processes receive priority X01, which is higher than the priority of the t process (X10), but lower than that of the w processes (X00). Thus, when there is an s process (and its subtree) available, the system focuses its attention on completing a solution to s (and thereby a solution to the reduce process) instead of finding additional solutions to t (or u or v). In addition, multiple instances of a single literal fired are prioritized so that the one fired earlier has higher priority than the ones fired later. This necessitates addition of "instance bits" in addition to the "distance bits" used above, as shown in Figure 5.

Intuitively, the scheme represents the strategy of supporting the subcomputations that were closer to yielding a solution to the top level goal. ("Support the Leader" strategy). It solved the "all-solutions-in-a-burst" problem mentioned above, because p's and q's subtrees now have identical priority, and so compete for resources with each other, thereby ensuring that some p and some q solutions will be produced in parallel. We were able to demonstrate good first solution speedups for problems involving both AND and OR parallelism, with very little overhead; This was accomplished without affecting the performance on pure AND and pure OR parallel problems, or on all-solution searches. For further information about this scheme, we refer the reader to [30].

Table 5 below shows the performance of our strategy on a benchmark. This program involves finding a prime that can also be expressed as a sum of a Fibonacci number and a perfect number. The first benchmark problem was:

" fib(F,20000) // perfect(P,3000) $\rightarrow$ X is F+P $\rightarrow$ prime(X)."

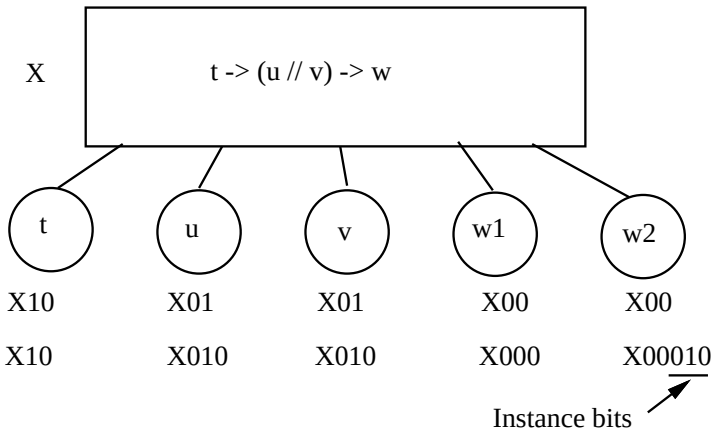


Fig. 5. Prioritization for clauses with dependence graphs. The first row shows priorities with distance bits only, while the second row shows priorities with instance bits appended.

I.e. it first searches for a Fibonacci number and a perfect number (in the specified range) in parallel, and for every pair obtained, checks if its sum is a prime. The second problem increases the problem size and requires that the primes so found be larger than 50,000.

" fib(F,80000) // perfect(P,3000) $\rightarrow$ X is F+P $\rightarrow$ X $>$ 50000, prime(X)."

As seen in the table, the time to first solution for the second instance was reduced almost ten-fold, at the cost of a small overhead (as represented by the overall time to completion, which increased from 27.9 to 28.0 to 29.6).

Table 5. Performance of two benchmark problems with AND and OR parallelism on Sequent Symmetry with 16 processors. For each case, time to first solution is shown, followed by the time to completion in parenthesis, in seconds.

	Depth-First	Prioritized: distance bits	distance and instance bits
Problem 1	2.4 (15.0)	1.4 (14)	0.8 (16.0)
Problem 2	23.2 (27.9)	3.5 (28.0)	2.8 (29.6)

Consistent Speedups: The method described above is not free from anomalies. The results from state-space search made it clear to us that consistent non-anomalous and monotonic speedups can be obtained in that domain. We then

applied these techniques to the parallel Prolog system, while restricting ourselves to pure OR parallel programs without any AND parallelism. The description of the process structure for ROPM described above should make it clear why the application is not straight-forward. The OR tree (search tree) used in state-space search is now folded into the REDUCE/OR tree.

The scheme we developed in [34] to address the speedup anomaly involves tagging responses with their priorities, and using the response's priority to decide the priority of any processes fired due to it. For example, a REDUCE process with priority X may have two dependent literals p and q, with q being dependent on p. A solution returned for p would have a priority indicating its place in the tree beneath p, say XY. The priority of the q instance fired using this binding returned by p will then be XY also. (Compare this with the scheme described above in which the priority of q would have been X0). If this q instance sends a solution tagged with a priority vector XYZ, the resultant binding is sent as a response to the parent of the REDUCE/OR process with priority vector XYZ attached to it. (As opposed to just X in the previous scheme).

The complete details of this scheme can be found in [29]. We only note that consistent and excellent first solution speedups were obtained for pure OR parallel Logic Programs with this scheme. As an example, the following table shows the performance of the Prolog compiler with priorities, running a Prolog program to find a Knight's tour on a 6x6 board. The speedups observed were very consistent from run to run, and can be seen to increase well with processors. The degree of wastage can be estimated from the number of messages processed, which is proportional to the number of processes created, and is seen to be well controlled.

Table 6. Performance of prioritized Parallel Prolog Compiler on a 6x6 Knight's tour program on Sequent Symmetry.

Processors	1	4	8	12	16	20
Execution Time (Secs.)	1245	337	183	127	97	80
Messages Processed	9348	9464	9526	9617	9624	9694
Wasted Work	0%	1.2%	1.9%	2.8%	2.9%	3.7%

Dealing with AND Parallelism: The first scheme described above improves first solution speedups in AND/OR parallel programs, but suffers from anomalies, whereas the second scheme yields consistent speedups but works only for OR parallel programs. A synthesis of these is needed. We developed a simple scheme [29] that is sufficient to ensure consistent and linear speedups for many (but not all) AND/OR problems. We believe that schemes that involve dynamic changing of priorities are necessary to handle this class of problems.

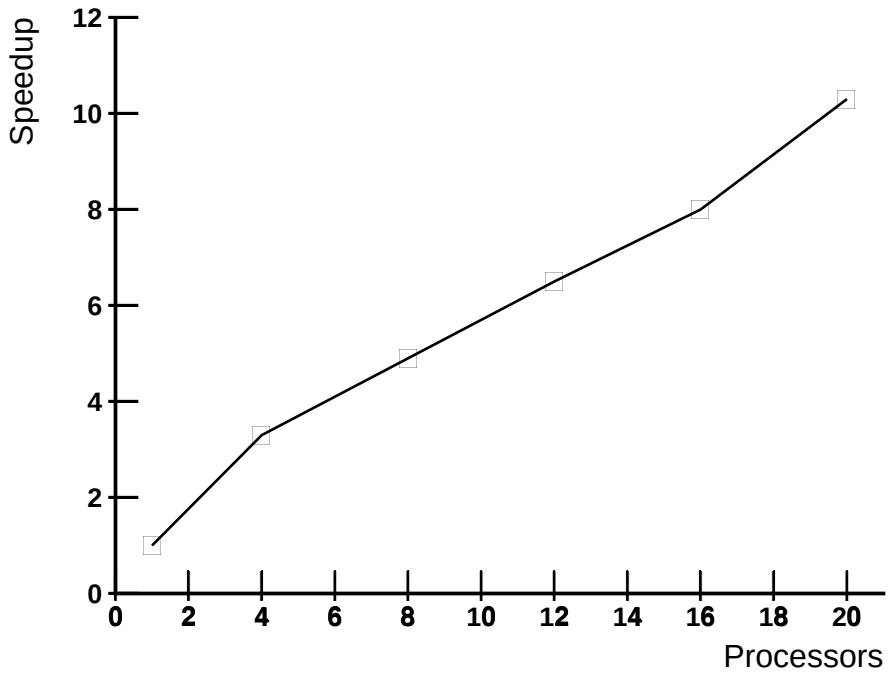


Fig. 6. Performance of a prioritization scheme for game tree search, on Sequent Symmetry. Speedups are relative to one processor speeds.

4.6 Game Trees

Alpha-beta search is an efficient game tree search procedure. However, when trying to conduct the search for the best move in parallel, it imposes a sequential bottleneck, as it requires information generated by left subtrees to be used within right subtrees. Attempts to obviate this bottleneck may reduce the amount of pruning, and thus increase the number of nodes examined, thus undermining potential gains of parallel processing. We investigated a parallel method that is symmetric in the sense of not requiring a strict left-to-right information flow. Each node (process) maintains a lower and upper bound on the value of the position it represents. As these bounds change during computation, their new values are sent to the parent processes. The pruning rule in this context becomes: "any child, for which the best it can do is worse than the worst I can do" is pruned. (For a max node: "Any child whose upper bound is smaller than my lower bound" is pruned.) Notice that this rule may prune children even when none of them have final values, unlike the alpha-beta. So, it is potentially possible that

this method, under a proper prioritization strategy, may need to explore fewer nodes than the alpha-beta strategy. Finding such a strategy remains an open problem at the moment. However, we have explored some simple prioritization strategies that lead to reasonably good performances [21]. Figure 6 below shows the performance of a simple strategy from [21] on a game position for the board game Othello, with a 8-ply search.

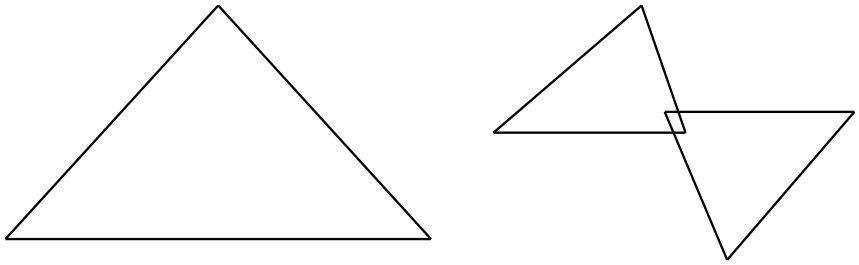
Notice that the above formulation leads to many different types of messages — those carrying new nodes to be expanded, those carrying updates to lower/upper bounds, those carrying termination messages (telling a child it should terminate its subtree and then itself), etc. Assigning differential priorities depending on the type of messages becomes at least as important as assigning different priorities to messages of the same type. The specific strategies we employed for this purpose are described in [21].

4.7 Bi-directional Search

When the goal state of a state-space search is fixed, and the operators for transforming states are invertible, it becomes feasible to search backwards from the goal state to the start state. Such search is not feasible for the N-queens problem because a goal state is specified only implicitly, by the constraints it must satisfy. It is feasible for the 15-puzzle or Rubik's cube, for example, because the desired state is concretely known.

For such problems, it is then possible to search in both directions - a forward search from the starting state and a backward search from the goal state. This can lead to potentially tremendous reduction in search space, as illustrated in Figure 7. Assuming a uniform branching factor of b and a depth of d for the search tree, the size of the search space reduces from b^d to $2b^{d/2}$. So, with a branching factor 2, and a depth of 30, one can reduce the search space (approximately) from a billion states to only 65,000 states. The promise of such a reduction makes bi-directional search very attractive.

A few details must be dealt with before attempting to realize these gains. Korf *et. al.* [16, 17] have explored these issues in the past. The forward and backward search must intersect in time, so as to make sure the solution states don't miss each other. This rules out the space-efficient depth-first search for at least one of the two directions. We must store the states in the backward (or symmetrically, the forward) search, and then search in depth-first manner in the forward (backward) direction. Given the memory limitations, it may be necessary to limit the depth of backwards search to less than half of the total depth. Secondly, the depth d may not be known apriori. Again, as suggested by Korf *et. al.*, we can employ an iterative deepening search when d is not known in advance. For simplicity, we can carry out a single backward search up to a depth b from the goal state once, store all the states so generated, and then carry out multiple iterations of the forward search with increasing depth-bounds d (see Figure 8). It suffices to store only the last (top) layer in the backward search, as any solution path must pass through a state in this layer. We will call this layer the "goal layer" in the following.



(a) forward search only (b) bi-directional search

Fig. 7. Potential reduction in search-space with Bi-directional search.

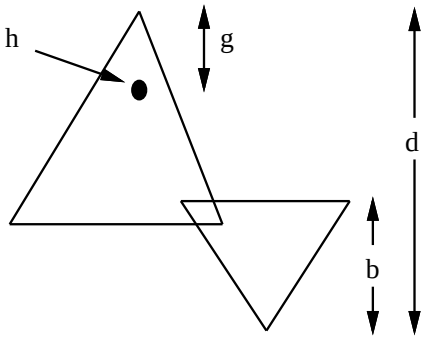


Fig. 8. bi-directional search for an IDA* iteration with depth-bound d

Parallelization of this algorithm requires a decision on how to store the goal layer — it can either (a) be replicated on all processors or (b) distributed across processors. With (a), it uses up more memory and so can store a shallower tree, but avoids message passing overheads for checking the goal layer for every state generated in the forward search. The choice depends on the number of processors — with more processors, you can store more layers with (b) and the gains in reduction of forward search-space start to outweigh the message passing overheads. The crossover point depends on the work done per state, and on the message passing overhead, and so is different for different applications, and possibly on different machines. We chose to use the distributed storage of the goal layer.

The problem we chose for this experiment was the 15-puzzle. We used the Manhattan distance heuristic (the minimum number of moves from any state to the goal state is at least the sum of the distance each tile is away from its

"home"). Each node in the forward search has a g value — which is the number of moves from the start state required to reach it — and an h value, which is the heuristic value for the node as defined above. A node in the forward search is checked for occurrence in the goal layer if its g value equals $d - b$. Also, a node a pruned (i.e. discarded) if $g + h > d$, because there is no prospect of finding a solution of depth d under it.

There is an issue of speculative computation in the last iteration, when one is looking for one optimal solution. To understand the issues in bi-directional search, we decided to isolate that issue by assuming that we are looking for all optimal solutions. As we carried out experiments with varying degrees of bi-directional search on 32, 64 and 128 processors of an NCUBE, a surprising result emerged, as seen in Table 7. The performance actually became worse when we used bi-directional search! The variant with no backward-search was the fastest. What happened to the tremendous promise of bi-directional search?

Table 7. Performance is worse with bi-directional search! Timing results for a 15-puzzle instance (Korf Problem #2) on NCUBE/2. All times are in seconds.

Depth of backward search	32 processors	64 processors	128 processors	States Explored
0	461	233	119	81,958,206
12	475	239	125	81,075,244
18	518	266	148	71,819,789

The solution to the mystery can be found by looking at the shape of the forward search tree. By counting the number of nodes at each level in the forward search, it can be seen that this shape is roughly as shown in Figure 9. I.e. the branching factor is not uniform, and it varies with the depth of the tree. It has an expanding phase followed by a "stagnant" phase, followed by a shrinking phase. (Stone & Stone also report similar shapes of search trees in [37].) It turns out that the lower bounding heuristic — the Manhattan distance one — is a strong pruning device, which discards a majority of the new states being generated at deeper levels in the tree. For such problems, backward search is essentially useless, as the size of the search space with bi-directional search is essentially similar to that with forward search alone. This is confirmed by counting the number of states explored (the last column of the table above), which are seen to be reduced from 82 million to 72 million, a much smaller reduction, with a much steeper price to pay in terms of communication.

To confirm the hypothesis further, we explored the 15 puzzle problem further by intentionally weakening the heuristic. The heuristic lower bound on each node was set to be w times the Manhattan distance, with $0 \leq w \leq 1$. With such weakening, the size of the search space explodes, and we were forced to

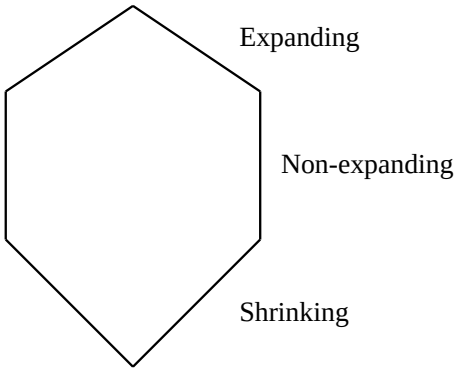


Fig. 9. A schematic view of the shape of the forward search space.

consider a smaller problem instance. With $w = 0.7$, for example, we were able to demonstrate the advantages of bi-directional search, as shown in Table 8. Thus, assuming that there will be other problems where such a strong heuristic is not available, bi-directional search can be still useful.

Table 8. Performance with weakened heuristics — Bi-directional search has its advantages.

Depth of backward search	32 processors	64 processors	128 processors	States Explored
0	952	480	241	177,263,862
12	975	505	270	159,092,443
18	605	301	184	50,810,983

Thus, our aim of exploring first-solution speedups with bi-directional search was beset, at least for 15 puzzle, at this point as even for all solutions, bi-directional search wasn't very promising. For problems (with weaker pruning heuristic) where it is promising, the following steps must be taken for first-solution speedup: First, the forward search must be prioritized, with bit-vector priorities as in simple state-space search. Secondly, to avoid backward search consuming more time than the forward one, the two should be overlapped in time. We plan to explore these issues further with other search problems. Some preliminary results on the "Peg solitaire" game are in [2]. That report also deals with "duplication detection" which is relevant for simple as well as bi-directional searches where states may recur independently in different parts of the search tree.

5 Implementing Priorities on Parallel Systems

Most prioritization strategies for parallel systems can be considered to be variants of either a centralized or a fully distributed scheme. In the centralized scheme, work is allocated to requesting processing elements from a central pool of work, where the work is sorted exactly according to their priorities. In a simple fully distributed scheme, each individual processing element maintains its own pool of work (sorted according to priorities). Any new work generated is sent to a randomly selected processor. The variations in these strategies arise with the differences in schemes used to balance work and balance priorities. In case of the centralized strategies, the central pool of work becomes a source of bottleneck. Fully distributed strategies on the other hand suffer from an inability to adhere to priorities on the global level (low priority work might be done on some processing elements, even though there exist higher priority work on other processing elements).

Our earlier work [29] involved a fully distributed approach to prioritization in a parallel system. The initial distribution of the work onto various processing elements is random. Subsequently, processors periodically exchange information about load and priorities with their neighbors, and attempt to distribute priorities and load by moving work around. These strategies still have the drawback (to a smaller extent) we discussed earlier — priorities are not distributed uniformly over processors, hence low priority (wasteful) work gets done. For additional variants of this strategy and their performance on various machines we refer the reader to [29].

Centralized strategies provide good priority adherence and load balancing. Their weakness is that the central pool of work becomes the bottleneck. We can solve the problem of a bottleneck by splitting the pool of work amongst a few processing elements — essentially creating some sort of a semi-distributed strategy as described in [36, 33]. In the strategy in citeSinhaIPPS93, the processors in the system are partitioned into clusters. One processor in each cluster is chosen as the load manager, the remaining processors in the cluster being its managees. Managees send all new work created on themselves to be queued in a centralized pool (sorted according to priorities) at their corresponding load manager. Each load manager has two responsibilities:

1. It must distribute the work among its managees. The managees inform their load managers of their current work load by sending periodic load information and piggybacking load information with every piece of new work they send to the manager. The load manager uses load information from its managees to maintain the load level within a certain range for all its managees.
2. It must balance both load and priorities over all the load managers in the system. This is accomplished by an exchange of some high priority tasks between pairs of managers. Each manager communicates with a defined set of neighboring managers. An exchange of tasks between a pair of managers occurs in two steps. In the first step, the managers exchange their load information. In the second step each manager sends over some tasks to the

other manager. A fixed number of tasks are always sent — this does the priority balancing. In addition, more (again, a fixed number) tasks depending on the task-loads of the managers involved are sent by the manager with greater load to the manager with the lesser load — this contributes to the load balancing. Note that the tasks exchanged are the highest priority tasks on each manager It might seem that by sending its highest priority tasks to another load manager, the sending manager is not distributing its priorities correctly. However this strategy performed well. We can intuitively explain why exchanging the top priority tasks might be sufficient: the managees of each manager work on the top priority elements on their load managers, so (in some sense) their work represents the top priority elements on the manager. Therefore an exchange of work between managers causes a distribution of the top priorities between two managers and their managees. We experimented further by implementing a strategy in which priorities were balanced by having pairs of managers exchange one-half of their top priority tasks. But this strategy resulted in a degradation in performance. We attribute the degradation in performance to the cost of determining the top half elements.

There is an imbalance in the memory requirements of the load managers and the managees in the hierarchical strategy. The imbalance arises because all newly created work is queued up at the load managers. This poses problems because the amount of new work that can be created becomes limited by the number of managers and their available memory, even though there is a larger amount of memory available on the managees (assuming all processors in the system have an equal amount of memory, and that there is more than one managee for each manager). We can balance memory requirements of processing elements using the following variant of the above strategy, developed in [36].

As in the above scheme, the processing elements in the system are split up into clusters — one processor in each cluster is chosen as the load manager, the remaining processors are its managees. However, now new work created on managees is stored in hash-tables on the processor itself, while only a *token* containing the priority of the new work is sent to the load managers. The load managers balance these prioritized tokens among themselves by exchanging their high priority tokens similar to the above. Each managee informs its manager of its load by (1) piggybacking load information with each token it sends to the manager, and (2) periodically sending load information. When a manager decides that one of its managees (say M) needs work, it selects the highest priority token from its (the manager's) pool of tokens, and sends a request to the processor storing the work corresponding to the token for the work to be sent to M.

6 Discussion

We argued that associating priorities with new processes and messages is an effective method for controlling the "focus" of a parallel symbolic computation. Although integer priorities are sufficient for some domains, other domains

are seen to require unbounded levels of priorities specified via arbitrary-length bit-vectors. Priorities are particularly effective for speculatively parallel computations, where part of the parallel work may be wasted, and so it is important to identify and focus on specific parts of the overall computation. Prioritization strategies for several tree-structured computations including state-space search, iterative deepening game tree search, branch-and-bound, and logic programming (which is the same as problem reduction based problem solving and planning from this paper's point of view) were developed and their success demonstrated. There still remain many open problems in prioritization particularly for game-tree search, AND-OR tree search, and bi-directional search.

All the computational experiments were carried out in the framework of the Charm parallel programming system. The system's modular organization allows one to plug different load balancing and queuing strategies without having to re-code the rest of the mechanics of parallel processing including support for message driven execution, portability across shared and nonshared memory machines, etc. Therefore, the Charm runtime system is an excellent testbed for such research. Several prioritization strategies were implemented as Charm modules, including a scalable yet effective variant that balances the three objectives of load balance, priority balance and memory-utilization balance. The modules are now part of the Charm system, where the users can select any one of them to link with their programs.

One of the interesting themes that came up repeatedly in different components of the work described here involves the importance of heuristic. In many state-space search problems, good value ordering heuristics combined with effective pruning strategies were able to obviate the need for parallel processing by essentially homing on to the solution along the leftmost branches of the search tree. This was true for N-queens problem, almost all instances of the 3-SAT problem that we tried, graph-coloring problems, etc. Good pruning heuristics also were seen to nullify the benefits of bi-directional search in many cases (e.g. 15 puzzle). This may seem somewhat discouraging, as it takes away the motivation for parallel processing. However, not all state-space search problems are as easy as the N-queens, and only a right combination of effective heuristics and parallel processing combined with prioritization will allow one to tackle some of the more difficult problems that we may wish to tackle in future. In addition, there are domains, such as game tree search, AND-OR tree search, and branch-and-bound, where the size of the search-space is very large, and which cannot usually be reduced simply by heuristic alone. (Although again lower bounding heuristics do lead to dramatic search-space reduction in branch-and-bound problems.) These problems represent a fertile area for future research, which will have a significant overall impact on real-life applications.

Integration of priorities with *futures* [5] particularly in connection with the *sponsor* model [23, 24] proposed for scheduling futures, represents another interesting problem. Attaching priorities to futures when they are spawned is straightforward; however, the sponsor model requires that when a process touches a future, the future assumes the priority of the touching process. Implementation

of this strategy has similarities with the problem encountered in state-space searches in connection with duplication detection [2], and also with the game tree searches, where priorities need to be dynamically propagated [21]. So some of the strategies developed in these works may be applicable in this context.

Acknowledgements: Most of the research reported in this paper was conducted by the first author with different (then) graduate students. One exception is the research by Ramkumar and Banerjee [27], who apply some of our earlier results on priorities to parallel test pattern generation. Much of the research discussed in this paper has been reported in separate papers. This paper should be seen as an overview, which brings together the applications of prioritization from different contexts. In particular, the work on state-space search and IDA was conducted with Vikram A. Saletore, the work on parallel Prolog with B. Ramkumar, and Vikram A. Saletore, the work on branch-and-bound with Amitabh Sinha, and some preliminary work on game-tree searches was conducted with Chin-Chau Low, and additional state-space search problems were implemented and studied by Wayne Fenton. Design and implementation of various priority balancing strategies was carried out earlier with Vikram A. Saletore, and more recently with Amitabh Sinha. We are also grateful to Argonne and Sandia National Laboratories for the use of their parallel machines.

References

1. Burton F. W. Controlling Speculative Computation in a Parallel Functional Language. In *International Conference on Distributed Computer Systems*, pages 453–458, November 1985.
2. Einarsson, Thorr T. Bidirectional Search in Parallel. Master's thesis, Dept. of Comp. Sc., University of Illinois at Urbana-Champaign, July 1993.
3. Fenton, Wayne. Additions to the chare kernel parallel programming system, and its usefulness for parallel state-space search. Master's thesis, Dept. of Comp. Sc., University of Illinois at Urbana-Champaign, 1991.
4. Furuichi, M., Taki, K. and Ichiyoshi, N. . A Multi-level Load Balancing Scheme for OR-parallel Exhaustive Search Programs on the Multi-PSI. In *PPOPP*, pages 50–59, March 1990.
5. Halstead R. Parallel Symbolic Computing. *Computer*, August 1986.
6. Hausman B. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, Royal Institute of Technology, 1990.
7. L. V. Kale and W. Shu. The Chare Kernel language for parallel programming: A perspective. Technical Report UIUCDCS-R-88-1451, Department of Computer Science, University of Illinois, August 1988.
8. L.V. Kale. The REDUCE OR process model for parallel execution of logic programs. *Journal of Logic Programming*, 11(1):55–84, July 1991.
9. Kale L.V. Parallel Execution of Logic Programs: The REDUCE-OR Process Model. In *International Conference on Logic Programming*, pages 616–632, Melbourne, May 1987.
10. Kale L.V. A Tree Representation for Parallel Problem Solving. In *National Conference on Artificial Intelligence (AAAI)*, St. Paul, August 1988.

11. Kale L.V. An Almost Perfect Heuristic for the N-Queens Problem. In *Information Processing Letters*, April 1990.
12. Kale L.V. The Chare-Kernel Parallel Programming Language and System. In *International Conference on Parallel Processing*, August 1990.
13. Kale L.V. and Saletore V.A. Parallel State-Space Search for a First Solution with Consistent Linear Speedups. *International Journal of Parallel Programming*, August 1990.
14. Kale L.V. and Warren D.S. Class of Architectures for a PROLOG Machine. In *International Conference on Logic Programming*, pages 171–182, Stockholm, Sweden, June 1985.
15. Kale L.V., Ramkumar B. and Shu W. A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs. In *The 5th International Conference/Symposium on Logic Programming*, pages 1223–1240, Seattle, August 1988.
16. Korf R.E. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. In *Artificial Intelligence*, pages 97–109, 1985.
17. Korf R.E. Optimal Path-Finding Algorithms. In *Search in Artificial Intelligence*, pages 223–267. Springer-Verlag, 1988.
18. Kumar Vipin and Rao V. Nageshwar. Parallel Depth First Search. Part 2: Analysis. *International Journal of Parallel Programming*, pages 501–519, December 1987.
19. Lai T.H. and Sahni Sartaj. Anomalies in Parallel Branch-and-Bound Algorithms. In *Communications of the ACM*, pages 594–602, June 1984.
20. Li G.J. and Wah B.W. Coping with Anomalies in Parallel Branch-and-Bound Algorithms. In *IEEE Transactions on Computers*, pages 568–573, June 1986.
21. Low, Chin-Chau. Parallel game tree searching with lower and upper bounds. Master's thesis, Dept. of Comp. Sc., University of Illinois at Urbana-Champaign, 1991.
22. Nilsson N.J. *Principles of Artificial Intelligence.* Tioga Press, Inc., 1980.
23. R. Osborne. Speculative computation in multilisp. In *Lecture Notes in Computer Science*, number 441. Springer-Verlag, 1990.
24. R. Osborne. Speculative computation in multilisp: An overview. In *ACM Conference on Lisp and Functional Programming*, 1990.
25. B. Ramkumar and L.V. Kale. Machine independent AND and OR parallel execution of logic programs: Part II - compiled execution. *To appear in IEEE Transactions on Parallel and Distributed Systems*, 1991.
26. Ramkumar B. and Kale L.V. Compiled Execution of the REDUCE-OR Process Model on Multiprocessors. In *North American Conference on Logic Programming*, pages 313–331, October 1989.
27. Ramkumar, B., Banerjee P. Portable Parallel Test Generation for Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.
28. Rao V. Nageshwara and Kumar Vipin. Parallel Depth First Search. Part 1: Implementation. *International Journal of Parallel Programming*, pages 479–499, December 1987.
29. Saletore V.A. *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, Dept. Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, September 1990.
30. Saletore V.A. and Kale L.V. Obtaining First Solutions Faster in AND-OR Parallel Execution of Logic Programs. In *North American Conference on Logic Programming*, pages 390–406, October 1989.

31. Saletore V.A. and Kale L.V. Consistent Linear Speedups to a First Solution in Parallel State-Space Search. In *The Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, Mass.*, July 1990.
32. Saletore V.A. and Kale L.V. Efficient Parallel Execution of IDA* on Shared and Distributed Memory Multiprocessors. In *Proceedings of the Sixth Distributed Memory Computing Conference (DMCC6), Portland, OR*, April 1991.
33. Saletore V.A. and Mohammed M.A. A Hierarchical Load Distribution Scheme for Branch-and-Bound Computations on Distributed Memory Machines. Technical Report 93-80-04, Dept. of Computer Science, Oregon State University, January 1993.
34. Saletore V.A., Ramkumar B., and Kale L.V. Consistent First Solution Speedups in OR-Parallel Execution of Logic Programs. Technical Report UIUCDCS-R-90-1725, Dept. of Computer Science, University of Illinois at Urbana-Champaign, April 1990.
35. Sen A.K. and Bagchi A. Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory. In *International Joint Conference on Artificial Intelligence*, pages 297–302, August 1989.
36. A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, April 1993.
37. Stone Harold S. and Stone Janice M. Efficient search techniques- An empirical study of the N-Queens Problem. *IBM Journal of Research and Development*, pages 464–474, July 1987.