Prioritization in Parallel Symbolic Computing

L. V. Kale
Department of Computer Science
University of Illinois at Urbana Champaign
1304 W. Springfield Ave., Urbana, IL-61801

Abstract:

It is argued that scheduling is an important determinant of performance for many parallel symbolic computations, over and above the issues of dynamic load balancing and grainsize 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.

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, 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 enable 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 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. 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. 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 (such as state-space search, game tree search, etc.) and describe how specific priority-based scheduling strategies can be used to effectively parallelized them. This section includes descriptions of some regimes for which effective pritiritization strategies are not yet known, and should be areas of active research. 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 3. 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 structures 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 distributed memory one) where pointers 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) grainsize control, (2) dynamic load balancing and (3) scheduling.

Grainsize 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 soon 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. 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.

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 overal CPU-time available in the system. When a process forks sub-processes, it assigns to them fractions of the fraction that were 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. The difference is 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 this 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 is a parallel programming system that we have developed over the past 6 years 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 and non-shared memory machines without change. It currently runs on Intel iPSC/860, NCUBE, Sequent Symmetry, Multimax, Alliant FX/8 and networks of workstations, and will be ported to machines including the CM-5 in near future.

Charm is one of the first systems to support an asynchronous, message driven, execution model. 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 (or when the initial message containing the "seed" for a new process is picked up). 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 (different) prioritization schemes in different 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 describe 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 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. This are then pushed onto the stack, possibly using some local value-ordering heuristics so that the child most likely to lead to a solution is kept at the top of the stack. Instead of an explicit stack, depth-first search is sometimes expressed recursively.

```
Search(state)
    if isGoal(state)
        recordSolution(state)
    else
        for each successor state $S_i$
            search($S_i$)
```

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 grainsize control. Earlier, we worked on the all-solution problem using the Chare Kernel machine independent parallel programming system [3], 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 [12, 8].

When one is interested in any one solution, this parallelization technique leads to problems. 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 under 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 Sahani [9].

One may get deceleration anomalies where adding a processor may 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 wasted there time on. In extreme cases, this may lead to detrimental anomalies, where p processors perform slower than 1 processor doing 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 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 [12].

We started with the dual objectives of (1) ensuring that speedups are consistent - i.e. do not vary from run to run 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 [15] is based on bit-vector priorities. 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 $\lceil logk \rceil$-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.

The complete scheme involves a few additional subtle points of strategy, and is described in [4]. 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 $D * P$. The report also describes performance results on first solution speedups to a 126-queens problem, a knight's tour problem, and a magic-square construction problem.

The scheduling induced by this strategy sweeps the search tree from left to right. Moreover, at any moment, the set of "active" nodes form a characteristic shape resembling a broom with a long stick.

4.2 Iterative Deepening

Sometimes, one is interested in an optimal solution to a search problem. If an admissible heuristics is available [10] 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 can not 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 [7], and is called IDA*. As in A*, the first solution found is an optimal solution in IDA* too. Very successive iterative 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 [12]. were the first to demonstrate parallel schemes for this problem. Their results did exhibit speedup anomalies, and they reported speedups to all solutions (as their primary interest was to demonstrate the efficacy of their load balancing scheme). Note that there may be multiple optimal solutions in the last level. Even when there is one, it may be found before the whole tree up to that level is explored. Thus the notion of speculativeness prevails in this context too.

Our prioritization techniques described above were successful at getting 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. At the beginning of each iteration the parallelism is low. It increases quickly to occupy all the processors, and then trails off toward the end of the iteration. A substantial improvement was obtained by allowing multiple iterations to run together. This has to be done with some care, as work in iterations beyond the iteration containing the optimal solutions contributes further to wasted work. This was handled by assigning a non-empty increasing bit-vector priorities to the root nodes of successive iterations using an interesting encoding scheme. (See [4] for details.).

This encoding scheme solves the problem of assigning increasing priorities to successive roots, without knowing an upper-bound on how many iterations there will be. The encoding must also ensure that each node in one iteration receives higher priority than all nodes in the next iteration. With that, 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-size problems.

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 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 grainsize control (This is just one of many possible methods for grainsize 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, than the message carrying the 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 pre-prending a "0" to the priority of all solution messages, and a "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 Salesman Problem (TSP) is a typical example of an optimization problem solved using branch&bound techniques. In this problem the salesman 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 have a cost C_{ij} associated with them (if $i = j$, then C_{ij} is assumed to be of infinite cost).

In the branch&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 prioritized scheduling mechanism, we associate the lower bound of a node as the priority of the corresponding process, with lower values signifying higher priorities. (The parallelization scheme also needs an ability to propogate the cost of the best-known solution at current time, so that is accessible from all processors. The *monotonic variables* supported in Charm provide this capability).

This prioritization scheme was implemented and tested on many versions of Traveling Salesperson Problem. With appropriate choice of priority balancing strategies (described later), we were able to demonstrate good speedups even on a 512-processor NCUBE machine. The major challenge here, 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 strict adherence to priorities, while still preserving good load balance, and avoiding any bottlenecks.

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 add as breadt-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 linear to exponential (in the depth of the tree), one can expect memory overflows on many problem instances. This can be avoided by adopting a prioritization strategy that adapts to the current memory usage. When memory utlization 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 U+x. When we detect that the memory usage is high, we assign priorities differently. Each node at depth d is a assigned a priority d. 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. U+x). The system will still finish all nodes created prior to this point in a depth-first (LIFO) fashion, but then revert back to a best-first pattern.

The memory usage is also high in a prioritized strategy, because of 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 certain priority level. This can be used whenever a new better solution is found, to clean up the queue.

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 Programs, 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).

4.5.1 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 [5, 1]. The past and ongoing work related to this model in our group includes development of a binding environment [6] and a compiler [11]. 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 distributed memory machines. The compiled system currently runs on iPSC/2 hypercube, as well as 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.

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 in Section 3.2. The work described in Section 2 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 in Section 3.3.

4.5.2 Speedups for a First Solution

The REDUCE/OR process model is based on REDUCE/OR trees [2], which is an alternative to the traditional AND/OR trees. 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 [1]. 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 solution. 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:

$$h(I,T)\ :-\ p(I,X)\ \rightarrow\ (q(X,Y)\ //\ r(X,Z))\ \rightarrow\ s(Y,Z,T).$$

when an instance of this clause is activated, an initial binding tuple with 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 p, one q and one r process is fired immediately.

Thus there may be multiple q (and r) processes active at one time. Each value of Y returned by q is combined with compatible values of Z (i.e. those that share the same X value) returned by the corresponding r process, and for each consistent combination, a s 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 response to its 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 system, this was done using a central shared stack, whereas on distributed 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 an 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 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 clause 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 child, by appending the child's rank to X (as described in the section on state-space search). A REDUCE process used a more complex method for assigning priorities. The AND parallel literals, such as p and q in the example above, received identical priorities. The literals closer to the end of the dependence graph received higher priority than those that preceded them in the graph. In addition, multiple instances of a single literal fired were prioritized so that the one fired earlier has higher priority than the ones fired later. For brevity, we will leave the description of this scheme at that, and refer the reader to [13] for a full description.

Intuitively, the scheme represented 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 has 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, and without affecting the performance on pure AND and pure OR parallel problems.

4.5.3 Consistent Speedups

The method described above is not free from anomalies. As 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 started to apply these techniques to the parallel Prolog system. The description of the process structure for ROPM described above should make 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 involves tagging responses with their priorities, and using the priorities of 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 in the sub-section 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 [14]. We only note that consistent and excellent first solution speedups were obtained for pure OR parallel Logic Programs with this scheme.

4.6 Dealing with AND Parallelism

The scheme described in Section 4.5.2 improves first solution speedups in AND/OR parallel programs (but has anomalies) whereas that in Section 4.5.3 the scheme yields consistent speedups but works only for OR parallel programs. A synthesis of these is needed. We developed a simple scheme [14] 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.

4.7 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 strategies that lead to reasonably good performances. These strategies illustrate another use of priorities.

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 elsewhere.

4.8 Other problems:

We believe that many other domains in AI, including search with duplication check, bi-directional search, genetic algorithms, truth maintenance systems, neural nets, and productions systems can be parallelized effective via the use of priorities. These represent fertile areas of future research.

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).

We will describe one 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 [14].

Centralized strategies provide good priority 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. In this strategy, 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. As in the load manager strategy, 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 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 (we have experimented with a strategy in which one half of the top priority tasks were exchanged, but this resulted in a degradation in performance, perhaps because of the cost of determining the top half elements). We can intuitively explain why exchanging the top priority tasks might be alright: 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.

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 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.

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. New work created on managees is stored in hash-tables on the processor itself, while a token containing the priority of the new work is sent to the load managers. The load managers balance tokens and priorities among themselves by exchanging their high priority tokens — a fixed number of tokens is always exchanged to accomplish priority balancing, while some more tokens may by exchanged to balance the number of tokens on the load managers. 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 on it, and sends a request to the processor storing the work corresponding to the token asking for the work to be sent to M.

Acknowledgements: Most of the research reported in this paper was conducted with different (then) graduate students. Much of such work 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 Saletore, the work on parallel Prolog with B. Ramkumar, and Vikram 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 V. Saletore, and more recently with Amitabh Sinha.

References

[1] 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.

[2] Kale L.V. A Tree Representation for Parallel Problem Solving. In *National Conference on Artificial Intelligence (AAAI)*, St. Paul, August 1988.

[3] Kale L.V. The Chare-Kernel Parallel Programming Language and System. In *International Conference on Parallel Processing*, August 1990.

[4] Kale L.V. and Saletore Vikram A. Parallel State-Space Search for a First Solution with Consistent Linear Speedups. Technical Report UIUCDCS-R-89-1549, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1989.

[5] 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.

[6] 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.

[7] Korf R.E. Optimal Path-Finding Algorithms. In *Search in Artificial Intelligence*, pages 223–267. Springer-Verlag, 1988.

[8] Kumar Vipin and Rao V. Nageshwar. Parallel Depth First Search. Part 2: Analysis. *International Journal of Parallel Programming*, pages 501–519, December 1987.

[9] Lai T.H. and Sahni Sartaj. Anomalies in Parallel Branch-and-Bound Algorithms. In *Communications of the ACM*, pages 594–602, June 1984.

[10] Nilsson N.J. *Principles of Artificial Intelligence.* Tioga Press, Inc., 1980.

[11] 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.

[12] Rao V. Nageshwara and Kumar Vipin. Parallel Depth First Search. Part 1: Implementation. *International Journal of Parallel Programming*, pages 479–499, December 1987.

[13] 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.

[14] Saletore Vikram A. *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, In preparation, Dept. Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, September 1990.

[15] Saletore Vikram 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.