Consistent Linear Speedups to a First Solution in Parallel State-Space Search*

Vikram A. Saletore and L. V. Kalé
Department of Computer Science,
1304 West Springfield Avenue,
University Of Illinois at Urbana–Champaign,
Urbana, Illinois 61801, USA
email: saletore@cs.uiuc.edu and kale@cs.uiuc.edu

Abstract

Consider the problem of exploring a large state-space for a goal state. Although many such states may exist, finding any one state satisfying the requirements is sufficient. All methods known until now for conducting such search in parallel fail to provide consistent linear speedups over sequential execution. The speedups vary between sublinear to superlinear and from run to run. Further, adding processors may sometimes lead to a slow-down rather than speedup, giving rise to speedup anomalies. We present prioritizing strategies which yield consistent linear speedups and requires substantially smaller memory over other methods. The performance of these strategies is demonstrated on a multiprocessor.

1 Introduction

Consider the problem of searching for a solution in a large state space, starting from a given initial state. The state space is usually structured as a tree, with operators that can transform one state 'node' to another forming arcs between different states[1]. In a large class of such problems, the computations tend to be unpredictably structured and have multiple solutions. The desired solution is usually specified by certain properties, and any state satisfying these properties is an acceptable solution. Sometimes one is interested in *optimal* solution(s) based on certain cost criteria. However, many times, one is interested in just any solution. We focus on parallel exploration of search spaces in the latter context. This is an important problem: search is a major computational paradigm in Artificial Intelligence. For AI to achieve its long term, ambitious objectives, it seems clear that it must use parallel processing techniques [Halstead 1986]. Secondly, many 'real-life' applications such as Planning (plan construction), Symbolic Integration, VLSI Test Generation, Theorem Proving, etc. require finding an *adequate* solution rather than an optimal one.

A parallel scheme must be able to *consistently* generate a solution faster than the best sequential scheme, and preferably close to P times faster, where P is the number of processors used. Also, speedups must increase monotonically with the addition of processors. Another important performance criteria is the amount of *memory* required to conduct a search, which may vary from a linear to an exponential function of the depth of the tree. With parallel processing, it may also increase proportionately to P.

2 Search Techniques

A sequential depth-first search begins by expanding the root of a pure OR tree and can be efficiently implemented using a last-in-first-out *(LIFO)* stack of active nodes. The advantage of the sequential stack based depth-first search over other search techniques is its linear storage requirement $O(BD)$ whereas for best-first and breadth-first searches, it is exponential $O(B^D)$ [Pearl 1985] for a tree with branching factor B and depth D. A parallel stack based search algorithm is an extension of the serial algorithm that uses either a shared global stack or multiple stacks.

In the *shared stack model*, all processors share a global stack. Processors pick up nodes from the shared stack and expand them and push the descendents

*This research has been supported in part by the National Science Foundation under grant number CCR-89-00988.

[1]When it is possible to go from one state to another via two distinct sequences of operators, the state-space is a graph rather than tree. However, we will confine ourselves to state-space trees in this paper.

back onto the stack. In Kumar and Rao's multiple stack model [Kumar & Rao 1987], a processor is initially given the root node. A stack splitting scheme is used to distribute work. Processors search disjoint parts of the search space using their local stack in a depth-first manner. They report speedups in [Kumar & Rao 1987] that range from 3.46 to 16.27 using 9 processors for the *first optimal* solution to a 15-Puzzle problem using Iterative Deepening A* (IDA*) [Korf 1988] algorithm [2] on a shared memory multiprocessor. The inconsistency in the speedups is due to the anomalies that exist in parallel stack based search. The reason for speedup anomalies is that the parallel search may expand fewer or more nodes than a serial search.

In a multiple stack model, since processors search disjoint parts of the search space asynchronously, anomalies are possible because a processor may find a solution by searching a smaller or larger search space than the space searched with a sequential search. In a shared stack model as all processors run asynchronously, the set of nodes that are picked up for execution and put back on the stack may be very different from run to run as well as from the one processor case. This randomness in the selection of nodes is exactly the reason for anomalies reported in [Lai & Sahni 1984]. Therefore, both *acceleration anomalies* (speedup greater than number of processors P) and *deceleration anomalies* (decrease in speedup with increase in P) are possible. Also, there is no guarantee that the work performed by the addition of a processor will contribute in finding the first solution and such work may generate more futile work for other processors.

To determine the worst case memory required for the multiple stack model, consider a search tree of uniform branching factor B and depth D. The memory needed for the search will be proportionate to the worst case sum of the individual stack lengths of all the P processors in the system. Initially a processor picks up the root node and expands into its B descendents that are put on its local stack. Idle processors will try to get untried nodes from this processor. A breadth-first search will ensue until each processor is assigned to one active node which occurs at a depth of $\lceil \log_B P \rceil$ when there are P active nodes in the system. Each processor now conducts a depth-first search of subtree of depth $(D - \lceil \log_B P \rceil)$. The maximum stack length for the system is given by:

$$\textit{Stack Length}_{\textit{worst case}} \approx P * D * (B - 1)$$

[2]Each iteration of the IDA* algorithm is essentially a depth-first search of the cost bounded search space such that the first solution found is also one of the optimal solutions.

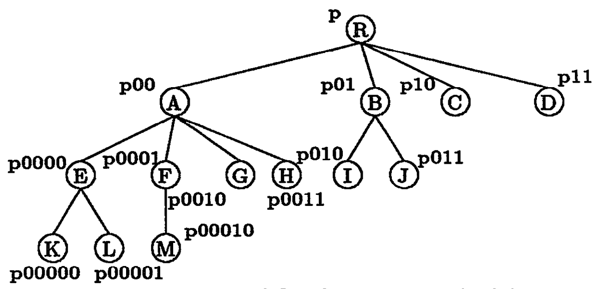


Figure 1: Or Tree with Bit Vector Priorities.

Assuming a constant node size, the memory needed for parallel search is roughly proportional to the product of P, B, and D.

For a shared stack model, a ***breadth-first*** search may continue until a depth of $\lceil \log_B P \rceil$ is reached. From this point onwards P processors *may* pick P nodes at level i from the stack and put $P * B$ newly created nodes at level $i + 1$ back on the stack, an increase of $P * (B - 1)$ nodes at each successive iteration. The worst case stack length for a shared stack model is given by an expression identical to the one above.

3 A Priority Based Search

We associate *priorities* with work in a parallel search and show how this would eliminate the anomalies and achieve linear speedups to a first solution. In a pure search, any of the alternatives at a choice point may lead to a solution. The alternatives may be ordered left to right, either using any local (value ordering) heuristic if available or using the default order. In a sequential depth-first search, a subtree under a node on the left is explored completely before expanding a node on the right. Thus, it gives higher priority to nodes (alternatives) on the left than the nodes on the right. In the context of first solution, the work to the right of the solution path does not contribute to the solution and therefore constitutes *wasted work*. To obtain *consistent speedups* with parallel search, we try to *focus* the search effort towards first solution by giving priorities to the alternatives at a choice point, with leftmost alternative having the highest priority (Work on other alternatives or *speculative work* can only speed up the solution to the problem if work under left subtrees fails to find a solution; otherwise, it constitutes wasted work.). Also, to mimic sequential order of node expansions, we must assign priorities such that every descendent of a leftward node has higher priority than all the descendents of rightward nodes.

A *priority bit vector* (also referred to as priority vector) is a sequence of bits of any arbitrary length. Priorities are dynamically assigned to the nodes when they are created. A node with a priority vector P_1 is de-

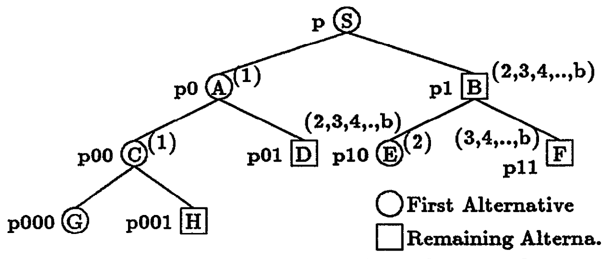
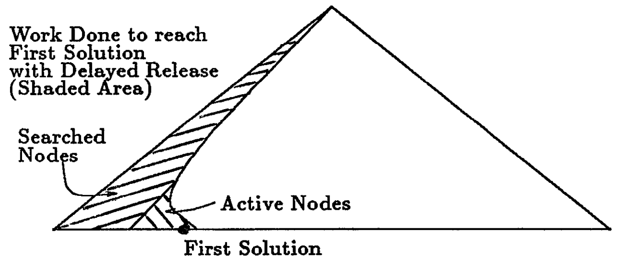


Figure 2: Binary Decomposition of a Search Tree.


Work Done to reach First Solution with Delayed Release (Shaded Area)
Searched Nodes
Active Nodes
First Solution
Figure 3: 'Broom Stick' sweep of a search tree.

fined to be at a higher priority than another node with priority P_2 if P_1 is *lexicographically smaller* than P_2.

Consider the OR tree in Figure 1. Let the root of this subtree R, have a priority p represented as a bit vector. The root node of the entire search space is assigned a *null* (priority bit vector of length 0) priority at the start of the search process. The m (m=4) children of the node R representing the m alternatives are assigned priority bit vectors by extending the parent's priority based on their rank. The rank of an *nth* child among its m siblings is represented as an encoding of n as a $\lceil \log m \rceil$ bit binary number. The priority vector of a child node is obtained by appending its rank to its parent's priority. Priorities assigned this way have the *prefix property*: no two children of a node have priority such that one is a prefix of the other (The idea of associating a similar sequence using path numbers with nodes of an OR trees has appeared in [Li & Wah 1986]).

Assigning bit vector priorities this way achieves two goals. (a) The relative priority (rank) of the sibling nodes preserves the left to right order of the sibling nodes. (b) Appending the priority of the child to the priority inherited from its parent ensures every descendent of a high priority node gets higher priority than all the descendents of low priority nodes. This reflects the policy that until there is no prospect of a solution from the left subtree beneath a node, the system should not spend its resources on the right subtrees, *unless* there are idle processors. I.e. if for a time period, if the work available in the left subtree is not sufficient to keep all the processors busy, the idle processors may expand the right subtrees. But as soon as high priority nodes become available in the left subtree, the processors must focus their efforts in that left subtree. For example, if two processors search the tree in Figure 1 then nodes A and B are picked up for execution once node R is expanded. When nodes A and B are expanded the processors explore nodes E and F (of higher priority) in the next cycle.

With priorities some acceleration anomalies are preserved because it is possible that when there is not enough work in the left subtree, processors that are exploring the right subtrees may find a solution faster. Also, since processing effort is always focussed leftward, the parallel search behaviour is similar to that of sequential search. This results in a decrease in wasted work and elimination of deceleration anomalies. As the wasted work is decreased to an insignificant proportion (see Section 3.2) this prioritizing strategy produces monotonic, almost linear, speedups.

The use of priorities to nodes requires the need for an efficient priority queue management, which can otherwise become a bottleneck if a large number of processors are used. We use *grain size* control to decrease the number of accesses to a shared queue and to spread the overhead of parallelisation. In terms of processing time the average grain size is defined as $\frac{\textit{Total Sequential Execution Time}}{\textit{Total Number of Messages Processed}}$. The ideal grain size for a shared memory system depends on the actual overhead involved in fetching nodes from the queue, creating parallel tasks and depositing new nodes back into the queue. Granularity control is used to determine when to stop breaking down a computation into parallel computations at a frontier node, treating it as a leaf node and executing it sequentially. A simple technique to control grain size, in state-space search, is to decide a cut-off depth beneath which the region is explored sequentially. Other techniques that attempt to gauge the size (granularity) of subtrees beneath a node are also possible. We found that such simple techniques were sufficient to prevent the priority queue from being a sequential bottleneck (see Section 4). With a large number of processors one may increase the grain size to maintain the frequency of access to the shared queue.

To retain similar speedup properties the grain size and the number of granules must both increase proportionally with the number of processors P, therefore, the overall problem size will have to increase with P^2. This can be alleviated somewhat by using concurrent heap access techniques. More important, as absolute adherence to priorities is not essential, techniques such as multiple heaps are also possible.

3.1 Binary Decomposition

The worst case stack length for *prioritized* search with a *shared queue* is given by the same expression as that for a parallel stack based search -namely $P * D * (B - 1)$. This is because there cannot be more than $P * B$ available nodes at any level in the tree, and like the single-stack, this upper bound can be realised by having the P processors pick up the leftmost P nodes at each level, in *lock-step*.

A depth-first sequential algorithm selects the current left most unexplored node at each level of the search space, backtracking to the choice point for remaining alternatives if no solution is found from the current alternative. In a parallel processing context, one can mimic this behaviour by generating the first descendent node say A in the usual manner, and *lumping* the work for generating remaining descendent nodes $(2, 3, .., b)$ into a single *lumped-node* B (see Figure 2). When this lumped-node is picked up for expansion, it generates the next sibling node E and a lumped-node F that lumps work for generating the rest of the sibling nodes $(3, 4, .., b)$. Thus, the lumped-node represents the remaining available parallelism in the subproblem in a form that is *extractable* whenever needed. This *binary decomposition* technique reduces an arbitrarily large branching factor to 2. (We have increased the depth of the tree towards right by this, but if the solution is much closer to the left in the search tree, as is frequently the case, this effect is small. Other techniques can be used to limit the depth with binary decomposition). Another advantage is that since the descendents of a node are not produced until needed, the wasted work is further reduced. (The lumped nodes are analogous to the choice-points used in OR-parallel Prolog systems e.g. [Hausman et al. 1987].).

Although we have reduced the parallelism in the initial stages of the search but once the number of nodes become greater than the number of processors, the available parallelism is more than sufficient to keep all processors busy.

3.2 Delayed Release

Although associating priorities eliminates anomalies, the memory requirement is still proportionate to P as given by the expression earlier. In both, stack and priority based models the increase in the memory usage with number of processors occurs because of the availability of large number untried alternatives at each level of the search. This *immediate availability* of parallelism at shallower levels in the search space is the major cause for increased memory usage. Notice that in the worst case all P processors pick nodes at level l, produce $B * P$ (or $2 * P$ for binary decomposition) children at level $l + 1$, and proceed to pick P nodes at level $l + 1$. To avoid the worst case, processors must *skip* intermediate levels. To achieve this, the parallelism in the problem available at shallower levels of the search space should be delayed and nodes at deeper levels of the search space should be made available to processors *first*. This *bottom-first* strategy gives rise to a search behaviour shown in Figure 3 termed the *Broom Stick* sweep. The set of searched nodes is represented in figure by a long narrow *stick* and the parallelism (active nodes) exploited at the bottom of the tree gives rise to the *broom effect*.

We achieve this *broom stick* behaviour by a technique called *Delayed Release*. The search begins at the root in the usual manner. When a node is expanded, all its children except the leftmost child are put in a list accessible only from the leftmost child. This list is local to the leftmost node and the nodes in the list are not available to other processors at the time they are generated. Also, the local list inherited by a node (parent node) is appended to the local list of its leftmost child. Therefore, each processor effectively creates only *one* child when a node is expanded, which is inserted into the shared priority queue. The search continues this way until a leaf node is encountered when all the nodes kept in the list are released as active nodes and made available to be picked up by other processors, subject to priorities, as before. This has the effect of *delaying* the availability of shallower nodes and making available the higher priority nodes to other processors from the bottom of the search space.

Figure 4 shows the state of a search tree using delayed release technique. When node R is expanded, only one *super-node* consisting of the linked list $[L_1, R_1]$ is produced. When this super-node is picked up, only the first node on the list, L_1, is expanded, and one super-node $[L_2, R_2, R_1]$ is released. Eventually, when the super-node $[A, B, R_5, R_4, R_3, R_2, R_1]$ is picked up for expansion, it is discovered that node A is a leaf, and so all the nodes in the list are released as individual nodes. Processors in the system now pick the highest priority nodes and explore the search space in a similar fashion. (Note that this delayed the expansion of the shallower, low priority node R_1.) With the current state of the search tree depicted, nodes A, B, C, D, E, R_2, and R_1 form the set of frontier nodes. The bit vector priorities ensure that the leftmost P nodes are picked up for execution. The delayed release technique achieves the following three objectives. (a) Most significantly, since processors are forced to skip node expansions at intermediate levels and focus on the nodes deeper in the tree, the wasted work is eliminated almost completely thus yielding linear speedups. (b) The memory required by the search algorithm is reduced considerably, as supported by empirical data.

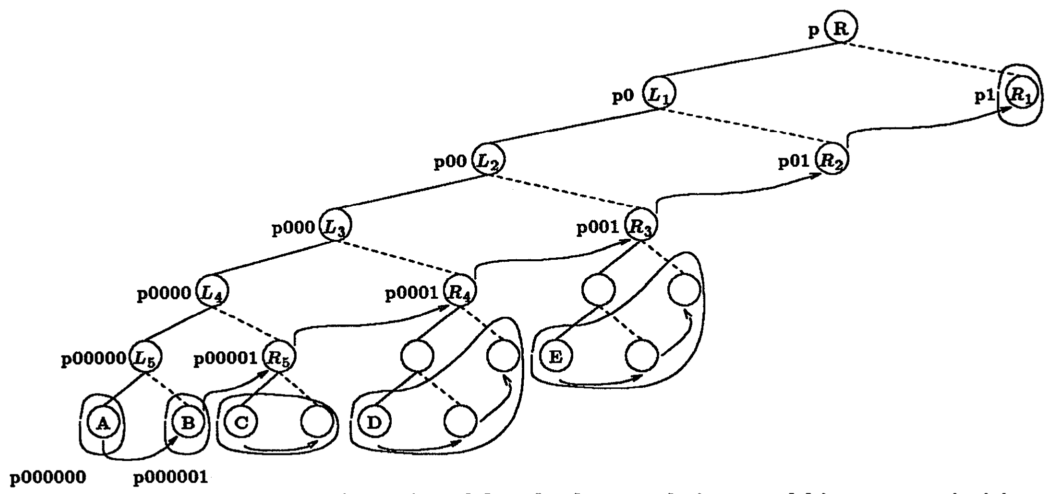


Figure 4: Tree expansion using *delayed release* technique and bit vector priorities.

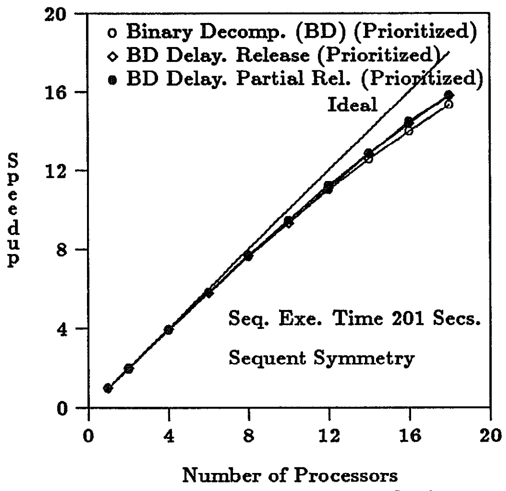


Figure 5: 126-Queens: First Solution Speedups.

(c) Since the number of active nodes in the priority queue at any time is reduced the overhead of managing the queue is reduced.

In the above strategy when a processor picks up a leaf node it releases all the nodes created and stored thus far in the local list of the leaf node. If the number of nodes released is larger than P, then the nodes in excess of P released constitute excess parallelism that can not be exploited at the time of release. In the *delayed partial release* technique, when a leaf node is encountered, the processor releases a maximum of P highest priority nodes and a single node that comprises the remaining nodes. If an idle processor cannot find other high priority nodes it expands the node comprising the excess nodes and proceeds in the usual manner. This technique does not result in a reduction in memory space, since memory space is still needed to store the *un-released* excess nodes in a list. However, since now there are even fewer entries in the queue than the *delayed release* strategy, the overhead of managing and restoring the priority queue is further reduced.

4 Multiprocessor Performance

We implemented the above search techniques for parallel search on Sequent Symmetry shared memory multiprocessor with up to 20 processors and studied their performance. The parallel search strategies were implemented using the *Chare-Kernel* [Kale 1990], a machine independent parallel programming system that currently runs on several multiprocessors. We present performance data on a few state-space search problems. Speedup performance is obtained by comparing to the best sequential program for the problem.

Nodes Expanded on Sequent Symmetry					
PEs	1 (seq)	4	8	16	18
BD	35248	35574	36164	36795	36944
BD-DR	35248	35306	35601	36229	36507
BD-DPR	35248	35357	35644	36219	36448

BD: Binary Decomposition (BD)
BD-DR: BD Delayed Release
BD-DPR: BD Delayed Partial Release

Table I: 126-Queens: Number of Nodes Expanded.

In the **N-Queens** problem the goal is to place N non-attacking Queens on a NXN chess-board. This problem has a large number of solutions where any one solution may be acceptable. We obtained performance data for the 126-Queens (N=126) problem. N was chosen to be large to show that such a large problem can be solved within a reasonable amount of memory usage and time. Binary decomposition technique of Section 3.1 was used since a complete decomposition at every level resulted in memory overflow.

The speedup plots in Figure 5 show that with bit vector priorities the wasted work is reduced resulting in linear, clearly monotonic speedups. The perfor-

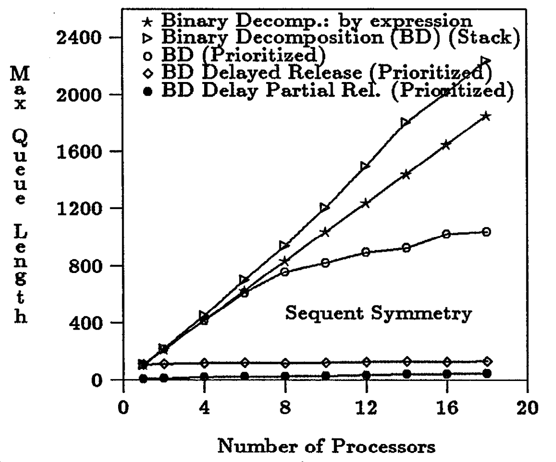


Figure 6: 126-Queens: Maximum Queue Lengths.

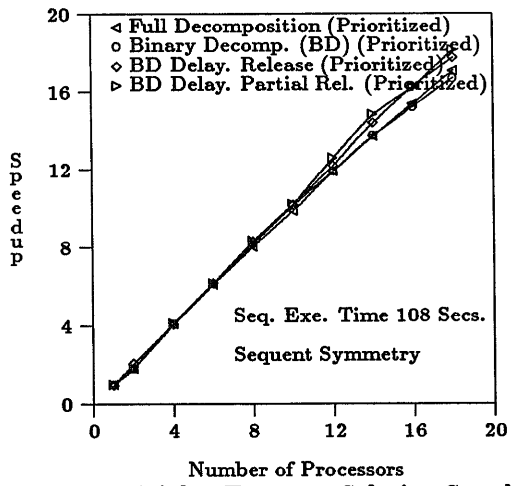


Figure 7: 8X8 Knights-Tour: 1st Solution Speedups

mance data is obtained from a single run of the 126-Queens problem and is highly consistent over different runs. The multiple stack strategy yields speedups that varies from run to run, as documented by Kumar et al. in [Kumar & Rao 1987]. The speedups with the shared stack also varied wildly between highly super-linear (few seconds) to extreme sublinear (aborted after few hours) and so are not reported here. Table I shows the total number of nodes expanded for the three schemes using bit vector priorities. It shows that (a) with priorities, the work with P processors is not significantly more than with 1 processor (e.g. an increase of 3.57% node expansions over sequential search with $P = 18$ using delayed release technique.) and (b) the wasted work is reduced with delayed release techniques. The plots in Figure 6 show that for a *LIFO* stack the maximum queue length (thus, the memory usage) increase proportionately to the number of processors P. This dependence on P is eliminated with the delayed release technique. With delayed partial release technique, the queue length decreases further (the memory usage remains unchanged for reasons explained in Section 3).

In a **Knights-Tour** problem, the knight must visit each position on a NXN chess board *once* and return to its starting position. Many solutions exist for the knights-tour problem. Speedups obtained were highly consistent from run to run, and increase linearly, as shown in Figure 7.

In the **Magic Square** problem the goal is to place integers from *1 to* N^2 on a NXN square board such that the sum of the integers along any row, column or diagonal is identical. Figure 8 gives the speedups to first solution to the problem with $N = 6$. The speedups improve with delayed release strategies. Again, the data reflects performance from a single run and is very consistent over different runs.

5 Discussion

We demonstrated the effectiveness of priority based parallel search techniques to eliminate anomalies and obtain consistent linear speedups to first solution in state-space searches. To the best of our knowledge, no other method proposed to date consistently achieves monotonically increasing speedups for a first solution. Our techniques also reduce the memory usage and it does not increase with the number of processors. It may be argued that this is unimportant as the amount of memory available grows linearly with the number of processors. However, this argument misses the point: on a P processor system our scheme requires a small fraction of the memory required by a stack based scheme. With 18 processors, *126-Queens* required 0.4 MBytes of memory compared to 1.8 MBytes with a shared stack. With a large number of processors and large problems, our scheme will be able to solve problems that the stack based scheme cannot solve due to a memory overflow.

An advantage of our scheme is that it adheres to local value-ordering heuristic, which are very important for first solution searches. However, even when good ordering heuristic is not available, our scheme is still valuable, because of its consistent and monotonic speedups.

Recently, Rao and Kumar have derived an interesting result in [Rao & Kumar 1988] concerning the multiple-stack model for parallel search, where the solution density are highly non-uniform across the search space. Although the speedups are anomalous, they show that *on the average* (i.e. averaged over several runs), the speedups tend to be larger than P, compared with the standard backtracking search, where P is the number of processors. In such types of search spaces our result is still valuable for the following rea-

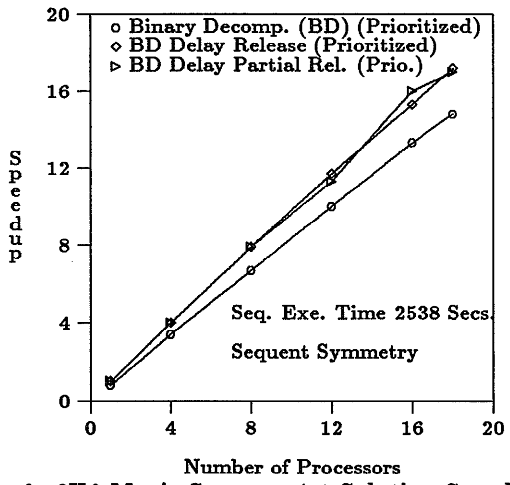


Figure 8: 6X6 Magic Squares: 1st Solution Speedups.

sons. First, the prioritizing scheme we presented *ensures* speedups close to P in all runs, and for varying values of P (assuming, of course, that there is enough work available in the part of the search tree to the left of the solution). Second, their results also show that the 'superlinearity' of speedup is not further enhanced beyond a few processors. Therefore, in a system with many processors we can exploit the non-linear solution densities better by setting the priorities of the top few nodes in the search tree to be null (i.e. empty bit vectors). This retains the advantage of exploring different regions of the search-space, hoping to exploit the non-uniform solution densities (probabilities) while still focusing the processors within these regions for more consistent speedups.

The use of priorities effectively *decouple* the parallel search *algorithm* from the scheduling strategy [Kale & Saletore 1989]. (In contrast, schemes such as [Lusk et al. 1988] for OR-parallel execution of Prolog use an explicit tree representation shared by all processors). This decoupling has several advantages. Scheduling strategies can be chosen independently of the search algorithm itself and synchronisation is much simpler. Most important, the prioritisation strategies can be extended to distributed memory machines such as the Intel iPSC/2 and BBN Butterfly, by providing a priority-balancing strategy in conjunction with the load balancing scheme. We are developing such a strategy.

We have extended these techniques for for Parallel IDA* in [Kale & Saletore 1989]. The extensions were needed because IDA* involves a series of increasing depth-first searches. Problem reduction based problem solving and Logic Programming are related areas in which, frequently, one is looking for one solution while many solutions may exist. However, this is substantially more complex situation than the pure state-space (OR-tree) search, if AND-parallelism is also to be exploited. The techniques developed in this paper can be incorporated beneficially in such schemes with appropriate modifications.

Acknowledgements

We would like to thank anonymous referees for improving the quality of the paper. We also thank Ross Overbeek at Argonne National Lab. for providing access to the Sequent Symmetry multiprocessor.

References

[Halstead 1986] Halstead R. 1986. Parallel Symbolic Computing, *IEEE Computer.*

[Hausman et al. 1987] Hausman B., Ciepielewski A. and Haridi S. 1987. OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors, *Proceedings of the Symposium on Logic Programming.*

[Kale 1990] Kale L.V. 1990. The Chare-Kernel Parallel Programming Language and System, *Proceedings of the International Conference on Parallel Processing.* Forthcoming.

[Kale & Saletore 1989] Kale L.V. and Saletore V. A. 1989. Parallel State-Space Search for a First Solution with Consistent Linear Speedups, Technical Report UIUCDCS-R-89-1549, Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign.

[Korf 1988] Korf R. E. 1988. Optimal Path-Finding Algorithms, *Search in Artificial Intelligence*, 223-276. Kanal L.K., Kumar V. eds.

[Kumar & Rao 1987] Kumar V. and Rao V. N. 1987. Parallel Depth First Search, *International Journal of Parallel Programming.* 479-519.

[Lai & Sahni 1984] Lai T.H. and Sahni S. 1984. Anomalies in Parallel Branch-and-Bound Algorithms, *Communications of the ACM.* 594-602.

[Li & Wah 1986] Li G.J. and Wah B.W. 1986. How Good are Parallel and Ordered Depth-First Searches, *Proceedings of the International Conference on Parallel Processing.*

[Lusk et al. 1988] Lusk E., Warren D.H.D, Haridi S. *et. al* 1988. The Aurora OR-parallel Prolog System, *Fifth Generation Computer Systems.* 819-830.

[Pearl 1985] Pearl J. 1985. Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Inc.

[Rao & Kumar 1988] Rao V.N. and Kumar V. 1988. Superlinear Speedup in State-Space Search, *Proceedings of the Foundation of Software Technology and Theoretical Computer Science.*