Parallel State-Space Search for a First Solution with Consistent Linear Speedups

L. V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Vikram A. Saletore
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

91-80-1

Parallel State-Space Search for a First Solution with Consistent Linear Speedups[*]

L. V. Kalé[†] and Vikram A. Saletore[‡]

ABSTRACT

Consider the problem of exploring a large state-space for a goal state where although many such states may exist in the state-space, finding any one state satisfying the requirements is sufficient. All the methods known until now for conducting such search in parallel using multiprocessors fail to provide consistent linear speedups over sequential execution. The speedups vary between sublinear to superlinear and from one execution to another. Further, adding more processors may sometimes lead to a slow-down rather than speedup, giving rise to speedup anomalies reported in literature. We present a prioritizing strategy which yields consistent speedups that are close to P with P processors, and that monotonically increase with the addition of processors. This is achieved by keeping the total number of nodes expanded during parallel search very close to that of a sequential search. In addition, the strategy requires substantially smaller memory relative to other methods. The performance of this strategy is demonstrated on a multiprocessor with several state-space search problems.

KEY WORDS: Parallel algorithms; parallel depth-first search; first solution; state-space trees; linear speedup.

[*]This research has been supported in part by the National Science Foundation under Contract No. CCR-89-02496.

[†]Authors Address: Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois 61801.

[‡]Author's Current Address: Department of Computer Science, Oregon State University, Corvallis, Oregon 97331.

List of Figures

List of Tables

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 (also called a 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. Frequently, the desired solution is 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.

Search is a major computational paradigm in Artificial Intelligence, and with developments and advances in AI, sophisticated AI computations are taking longer and longer times to run even on the new faster processors. If AI research is to achieve its long term, ambitious objectives, it seems clear that it must use parallel processing techniques [1, 2, 3, 4]. Secondly, many 'real-life' applications such as in planning (plan construction) [5, 6], symbolic integration [7, 8, 9], synthesis paths for organic compounds in Gelernter's SYNCHEM [10], test generation for VLSI circuits [11], etc. require finding an *adequate* solution rather than an optimal one. Another example is theorem proving, which requires only one proof, although many proofs may exist.

The search space involved in search problems tends to be very large, and is a highly computationally intensive problem. Search may appear to be naturally parallel computation. At any state one may apply multiple operators to obtain successor states, and then subtrees beneath each node can be searched by different processors. The parallelism between alternative subtrees is called *speculative parallelism* [1, 12, 13, 14]. This simple illusion of parallelism is shattered by the following observation. If a solution is found in the first subtree, the work done in the other subtrees is wasted. This phenomenon was observed early on [15], and it was noted that due to this, parallel search may present anomalous results i.e. the addition of processors is not guaranteed to increase speedups. Speedup performance of parallel state-space search is given by the ratio of the time taken by a sequential depth-first search algorithm to the time taken by the parallel version. The problem we address here is: how should we conduct a parallel search efficiently for finding *any one solution* to such problems.

What criteria should be used to evaluate parallel search execution schemes in this context? We believe that the following two criteria are essential.

1 The first and foremost criterion is the time required to find a solution. A parallel scheme

[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.

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 in the system. Also, speedups must increase monotonically with the addition of processors.

2 A second important performance criterion is the amount of *memory* required to conduct a search. The complexity of a state-space search problem may be expressed in terms of two parameters: the average branching factor B of the search tree, and the depth D at which a solution is found. The branching factor B is the number of new states that can be generated by the application to a given state. The state space to be searched is exponential in the depth of the tree, given by $O(B^D)$. The memory required to conduct the search is dependent on the search strategy and may vary from linear to exponential function of the depth. With parallel processing techniques, the memory requirement may increase proportionately to the number of processors P. It is therefore important to consider the memory usage of different schemes.

In this paper we present a series of strategies that achieve these objectives. In Section 2, we examine stack-based parallel execution schemes and show how they lead to anomalous and inconsistent speedups, and relatively high memory usage. Our scheme is described in Section 3 as a progression of improvements to a basic priority based scheme. The *Chare-Kernel* [16], a run time execution environment for parallel programming within which these schemes are implemented, is reviewed in Section 4. The performance results in Section 5 demonstrate that our schemes indeed achieve the objectives stated above. Application of these techniques to the IDA* [17, 18] search technique to obtain first optimal solution is described in Section 6. In Section 7, we extend the techniques developed earlier to improve the performance of parallel IDA*. We discuss some of the limitations of our scheme and the future work in this area in Section 8.

2 Stack Based Search Techniques

A sequential depth-first search begins by expanding the root of an OR tree into its children. At each successive step, the most recently generated node is expanded into its descendents. This is continued until a goal node is found. A sequential depth-first search can be efficiently implemented using a last-in-first-out *(LIFO)* stack of active nodes. The depth of the stack is the depth of the node currently being searched. The worst case stack length defines the storage requirement for the depth-first search.

The advantage of the sequential stack-based depth-first search over other search techniques is its low storage requirement. For other search techniques such as the best-first and breadth-first searches, the storage requirements is exponential $O(2^D)$ in the depth of the tree, whereas for sequential depth-first search, the storage requirement is linear in the depth of the tree $O(D)$ [6]. Memory required by a depth-first search defines the lower bound on the memory usage by a search algorithm.

A parallel stack-based depth-first search algorithm is an extension of the serial algorithm. Two approaches have been taken by other researchers [19, 20, 21, 22]. One approach is to have a single stack shared by all processors. This approach is suitable and easily implemented on small shared-memory machines. In the other approach there is a separate stack for each processor. The multiple stack model is suitable for distributed-memory machines, but has been implemented on shared-memory machines as well [22].

2.1 The Multiple Stack Model

In the multiple stack model, a processor is initially given the root node. It expands it and inserts the descendents into its local stack. Several load balancing algorithms [21] can be used to distribute work among processors. Each processor searches a disjoint part of the search tree using its local stack in a depth-first manner. When a processor completes its search without finding a solution, it tries to get active nodes from other processors and executes a depth-first search. When any one processor finds a solution all of them quit.

Significant earlier work on parallel search includes schemes proposed by Kumar et al. [20, 21, 22]. They have implemented the multiple stack model for their parallel version of Iterative Deepening-A* (IDA*) [17, 18], on the Sequent Balance shared-memory multiprocessor and on the Intel Hypercube iPSC/2, a distributed-memory multiprocessor. In this model the strategy attempts to divide the search tree at the top equally among the processors. A stack splitting strategy is used to rebalance work, if any processor exhausts its subtree. The IDA* algorithm is used to obtain optimal solutions to the *15-puzzle* problem. They report speedups in [22] that range from 3.46 to 16.27 using 9 processors for the *first optimal* solution to the same problem instance on the shared-memory multiprocessor. The inconsistency in the speedups is due to the anomalies that exist due to the asynchronous nature of parallel stack-based search. The scheme is shown to lead to linear speedups for *all optimal* solutions to the 15-puzzle problem on the shared-memory machine. However, in our context, these strategies are not satisfactory, as they do not consistently give close to linear speedups *for times to the first solution.*

Speedup Anomalies

Since our objective is to find any one solution, the search terminates whenever any processor encounters a solution. As the search space searched by different processors is determined dynamically, the speedups can differ greatly from one execution to another. In this model, since processors search disjoint parts of the search space, a solution may be found by visiting fewer or more nodes than a sequential depth-first search resulting in superlinear (speedup $> P$) or extreme sublinear (speedup much $< P$) speedups. Since all processors run asynchronously, the order in which the active nodes are selected and expanded is random and very different than that for a sequential search; it may also vary from one execution to another.

It is possible that a processor may find a solution more quickly by searching a smaller search space than the space searched with a sequential search leading to an acceleration anomaly (a very large increase in speedup with an increase in the number of processors). Deceleration anomalies (a decrease in speedup with an increase in the number of processors) are also possible [23], because there is no guarantee that the work performed by the addition of a processor will contribute to finding the first solution, and such work may generate more futile work for other processors. In this model [22], *detrimental anomalies* (speedup of less than 1) do not exist, assuming all processors have equal speed. This is because there will be at least *one* processor at any time working on a node n so that everything to the left of n in the tree will have been searched. Kumar and Rao's splitting strategy for load redistribution ensures that when the local stack is split, the node that was leftmost in the original stack is still leftmost in one of the two new stacks.

Many researchers [19, 15, 23] have reported this phenomenon of speedup being superlinear or extremely sublinear in isolated executions of parallel depth-first search using P processors. Lai and Sahni have shown that it is possible for a parallel branch-and-bound algorithm using n_2 processors, to take more time than one using n_1 processors, although $n_1 < n_2$; furthermore, it is also possible to achieve superlinear speedups in excess of the ratio n_2/n_1. Although their result is obtained in the context of branch-and-bound algorithms, it applies to pure search algorithms as well.

Memory Requirement

To determine the worst case memory required for the multiple stack model consider a search tree and let B be the uniform node branching factor and D be the depth of the tree. Since each processor has its own separate stack, the memory needed for the search will be proportionate to the worst case sum of the individual stack lengths of all the processors in the system. We also assume for the worst case that each processor has equal speed and that all node expansions take about the same amount of time. Initially, a processor picks up the root node and expands into its B descendents. The B descendents are then put on its local stack, making the stack length equal to B. Idle processors will try to acquire untried nodes from this processor. Now as long as the number of active nodes in the system is less than P, a breadth-first search will ensue until each processor is assigned to one active node. This will occur at a depth d equal to $\lceil \log_B P \rceil$ when there are P active nodes in the system, one node in each processor's local stack. The sum of the stack lengths will therefore be equal to P. Each processor will pick up a node from its stack and conduct a depth-first search of subtrees of depth $(D - \lceil \log_B P \rceil)$. The worst case stack length for a tree of depth x and branching factor b with a sequential depth-first search is given by $1 + x(b-1)$. Therefore, each processor will require a worst case stack length of $(1 + (D - \lceil \log_B P \rceil)(B-1))$. Since there are P such stacks, the stack length for the parallel system in the worst case will be given by

$$Stack\ Length_{worst\ case} = P + P * (D - \lceil \log_B P \rceil) * (B-1) \ \approx \ P * D * (B-1)$$

The worst case memory required will be equal to $k * StackLength_{worst\ case}$, where k is the node size (usually in *bytes/node*). The expression shows that the memory needed for parallel depth-first search is roughly proportional to the product of the number of processors P in the system, the branching factor B of the problem, and the maximum depth D of the search space.

2.2 The Shared Stack Model

In the single stack model [19], all processors share a global single stack. Processors pick up nodes from the shared stack and expand them and push the descendents onto the stack. Since the global stack is shared among all processors, a *locking* mechanism is needed to avoid multiple access hazards during insert and delete operations. As before, when a goal node is reached, all processors quit.

Speedup Anomalies

In the shared stack model, as all processors run asynchronously, the set of nodes examined by the time a solution is generated may be very different from run to run as well as from the one processor case. This behavior causes the parallel search to expand fewer or more nodes than a serial search causing anomalies [15, 23].

In this model, detrimental anomaly is also possible even assuming all processors have equal speed. This is because one can not guarantee that a processor is working on the leftmost unexplored node at any time during the search. If nodes generated to the right of a node n are the most recent and are inserted last in the stack, then these nodes will be the first ones to be selected and expanded. A situation can occur where node n is selected after all the subtrees to its right are explored. If all the solutions to the problem exist to the left of node n, then processors may first complete the search space to the right of n and then search the subtrees to the left of n, resulting in detrimental speedup anomalies to the first solution.

Memory Requirement

To obtain an expression for the worst case memory needed for the single stack model, again assume that all processors are identical and have equal speed. As before, let B be the uniform branching factor and D be the depth of the search space. Initially, a processor picks up the root node and expands into its B descendents. The B descendents are then put on the stack causing the stack length to equal B.

As long as the stack length remains less than the number of processors P, all the nodes from the stack get picked up and expanded, and the tree is explored in a *breadth-first* fashion until the number of active nodes in the stack becomes greater than P. This occurs at a depth d equal to $\lceil \log_B P \rceil$ when there are at least P active nodes in the shared stack. From this point onwards, each processor may pick up a node from the stack, expand it and insert the B descendents onto the stack. In the worst case (assuming all processors work synchronously), the P processors may pick P nodes at depth x, for example and put $(P * B)$ nodes on the stack all at depth $x + 1$, a net increase of $P * (B - 1)$ nodes in the stack. At each successive iteration the stack length will increase by $P * (B - 1)$. At depth D the worst case stack length will become

$$\mathit{Stack\ Length}_{worst\ case} = P + P * (D - \lceil \log_B P \rceil) * (B - 1) \ \approx\ P * D * (B - 1)$$

A similar expression has also been derived by Imai et al. in [19]. For a uniform tree this expression gives the worst case stack length required to conduct a parallel depth-first search using a single shared stack. The expression derived here is identical to that for a multiple stack model. The worst case memory requirement is $k * \mathit{Stack\ Length}_{worst\ case}$, where k is the average node size. The space requirement is again proportional to the product of number of processors P in the system, the branching factor B for the problem, and the depth D of the search space.

As shown by the expressions above, the space usage presents another problem. Although it is relatively easy to prevent exponential memory usage with a stack, memory usage tends to increase proportionately with the number of processors, in the worst case. In fact, empirical experiments (see Section 5) confirm that the average case also performs in the same proportion. In the following sections, we will describe techniques that attempt to eliminate the dependence of memory usage on the branching factor B and the product $P * D$ in the above expression.

3 A Priority Based Search

We associate *priorities* with work (computations) in a parallel depth-first 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 using any local (value ordering) heuristic. A sequential depth-first search algorithm searches the tree left-to-right and in a sense gives higher priority to alternatives on the left than to the alternatives on the right. The parallel scheme we develop achieves a similar behavior by examining nodes from left-to-right and the set of nodes expanded is very close to that expanded by a sequential depth-first search.

In the context of a first solution, the work to the right of the solution path does not contribute to the solution and therefore constitutes *wasted work*. In a parallel search environment, since we want to focus the processors towards the first solution, the alternatives at a choice point must have priorities, with the work in the leftmost alternative having the highest priority. The work on other alternatives (speculative work) can only speed up in finding the solution to the problem if work under left subtrees fails to find a solution; otherwise, it is wasted work. As explained earlier, the anomalies in a parallel search are caused by the random selection of nodes for expansion that could possibly result in an increased wasted work. The wasted work can be reduced if the nodes are expanded roughly in the same order as they are expanded during a sequential depth-first search.

Figure 1 shows the work performed (left of the solution path) to reach a first solution with a sequential depth-first search. A parallel depth-first search may expand nodes both to the left and to the right of the solution path. To obtain *consistent speedups* with parallel execution, one must try to reduce work performed on the right of the solution path. This also implies that the system resources must *focus* on work towards the left in the search space. In a sequential depth-first search, the subtree under a node l on the left is explored completely before expanding a node r on the right and exploring subtrees beneath r. To mimic the sequential order of node expansions, every descendent of node l receives higher priority than node r and all its descendents. This policy is applied again recursively to nodes in the smaller subtrees within this larger subtree. Any deviation from this policy that causes nodes to the right of the solution path to be expanded, when nodes to its left are available, results in increased wasted work, if a solution is obtained from the left subtree.

Consider a root node R of a subtree expanded into its descendents A, B, C, and D, with node A being the leftmost, as shown in Figure 2. If the nodes are *ranked* from left-to-right with node A having the highest priority and node D the lowest, then assuming all processors share a single priority queue (a priority queue is now needed instead of a stack to manage nodes with priorities), node A will be selected first for expansion. If there are idle processors in the system, then nodes B, C, and D will be selected, in that order. Let node A be expanded into its children E, F, G, and H. Let another processor expand node B into its children I and J. All the nodes generated are inserted into the queue. The serial behavior of node expansions dictates that with the current set of nodes in the queue the order of node selection and expansion should be E, F, G, H, I, J, C, and D. This suggests that node E being the leftmost must have the highest priority. Also, the set of descendents E, F, G, and H of node A must have higher priority than descendents I, J, of node B. In addition, all descendents of leftward nodes A and B must have higher priority than their parent's sibling nodes C and D. To achieve these goals, descendent nodes must be *ranked* from left-to-right and the *priority* of descendents of high priority nodes must have higher priority than the descendents of low priority nodes.

Associating priorities this way 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, 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 2, nodes A and B are picked up for execution when 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. In the following cycle after the children of nodes E and F are put back in the queue, then nodes K and L are picked up next for execution. We describe next a scheme to generate and assign priorities dynamically to the nodes of a search tree that would schedule them for execution to accomplish our objectives.

At times it is possible that a more favored (higher priority) node may be waiting for a resource while the less favored node is being executed. This occurs when a processor expands a high priority node and puts its descendents into the pool of work while another processor is still executing a low priority node. Assigning priorities to nodes only suggests to the resource management strategy that, in a given small window in time, the highest priority node in a common pool of work must always get a resource, if one is available.

3.1 Bit-Vector Priorities and Priority Assignment

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 and are not modified once assigned. A node with a priority bit-vector P_1 is defined 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 of Figure 2. 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) descendents of the node R represent the m alternatives for solving the subproblem. The descendents are assigned priority bit-vectors by extending the parent's priority based on their rank. Nodes may be ranked using any local (value ordering) heuristics. If a descendent node is ranked nth among m siblings, its *relative priority* (or rank) among its siblings is represented as an encoding of n as a $\lceil \log m \rceil$ bit binary number say p_{child}. Let the priority vector of the parent node be P_{parent}. Then the priority vector of the child node P_{child} is obtained by appending p_{child} to P_{parent} (The idea of associating a similar sequence using path numbers with nodes of OR trees has appeared in [24, 25]). In Figure 2, consider node A with a priority $P_{parent} = p00$ and if we assume a left to right ranking of its children E, F, G, and H, the relative priority (ranking) of the second child F out of the four siblings will be $p_{child} = 01$. Then the priority vector of node F is obtained as $P_{child} = p0001$ (p is priority of node R). The bit-vector length of P_{child} increases correspondingly. It can be shown that lexicographic ordering of these priorities corresponds to left-to-right ordering of the nodes in the tree. However, 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 levels (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. 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. Assigning bit-vector priorities this way achieves two goals.

1 The relative priority or the rank of the sibling nodes preserves the left to right order by ensuring that a node on the left is always at a higher priority than all its siblings on the right. Thus, local ranking heuristics are honored.

2 Appending the relative priority of the child to the priority inherited from its parent ensures that descendents of high priority nodes get higher priority than the descendents of low priority nodes. (For example, in Figure 2 node M (priority $p00010$) has higher priority than node J (priority $p011$) as $p00010$ is lexicographically smaller than $p011$).

Since the *leftmost* node has the highest priority, it always gets a resource, thus preventing detrimental anomalies. Some acceleration anomalies are preserved because it is possible that when there is not enough work in the left subtree, idle processors that are expanding the right subtrees may find a solution faster. With such a prioritizing scheme, since processing effort is focussed leftward, the parallel depth-first search behavior is similar to that of a sequential depth-first search. This results in a decrease in wasted work and the elimination of deceleration anomalies. Therefore, this prioritizing strategy guarantees monotonic, almost linear, speedups.

3.2 Eliminating Space Dependence on Branching Factor

We have shown earlier that the worst case memory requirement for a parallel search using single or multiple stacks is proportional to the stack length given by the following expression:

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

When bit-vector priorities are used in conjunction with a shared priority queue, the search behavior using P processors is very similar to that of a sequential case using a stack. However, the worst case queue length for a *prioritized* search is given by the same expression. This is because there cannot be more than $P * B$ active nodes at any level in the tree, and as in the single-stack, this upper bound can be realized by having the P processors pick the leftmost P nodes at each level, in lock-step.

The branching factor B is problem dependent. For example, in the *N-Queens* problem where one must place N Queens on an $N \times N$ chessboard such that no queen attacks another, the average number of choices where a queen can be placed is close to N in the shallower parts of the search space (a 100-Queens problem will have close to 100 alternatives for placing a queen). For large values of B, the memory requirement can be prohibitive by the expression above. A sequential depth-first algorithm selects the current leftmost 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. The successor node is usually generated from the parent node by making appropriate modifications to the parent node. The difference of information between the parent and the child is stored on a 'trail' to create the next child after backtracking to the choice point [26]. With priority based parallel depth-first search, one can mimic this behavior by generating the first descendent node by modifying the parent node in the usual manner, and copying the parent node with appropriate modifications for *lumping* work for generating remaining descendent nodes into an independent single *lumped-node*. We call this technique as the ***binary decomposition*** of a search tree.

Figure 3 shows the binary decomposition of a search tree, where node S is expanded into its child S_1 and a lumped node $(S_2, S_3, .., S_b)$ with a branching factor b. When this lumped-node is picked up for expansion, it generates the next sibling node E (S_2) and a lumped-node F that represents work for generating the rest of the sibling nodes $(S_3, S_4, .., S_b)$. 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 B to 2 and effectively eliminates the $(B - 1)$ factor in the memory usage expression $P * D * (B - 1)$. In doing so, we have increased the maximum depth of the tree by a factor of B towards the right, but if the solution is much closer to the left in the search space, as is frequently the case, this effect is small. Note that the derivation of $P * D * (B-1)$ is 'worst-case'. When the leftmost subtree is being searched, the $(B-1)$ right siblings are on the stack, but when the rightmost sibling is being searched, there are no other nodes (for that level) on the stack. Therefore, although the depth increases, the memory requirement does not increase. This is because only one *lumped* sibling node occupies the storage at any time. (The lumped nodes are analogous to the choice-points used in OR-parallel Prolog systems [27].)

Therefore, the binary decomposition of a uniform tree with branching factor B transforms it to a *skewed* binary tree with a depth increasing from D on the left to $D * B$ on the right. The number of internal nodes increases by a factor of $(B-1)$, since for each internal node $(B-1)$ lumped nodes are created. Thus, the total number of nodes in the search tree *doubles*. However, the amount of work does not increase much.

The major advantage of this technique is that since the all the descendents of a node are not produced until absolutely needed, the *wasted work* is reduced considerably. Moreover, using the priority scheme of Section 3.1 with this technique ensures that the leftmost nodes are picked up for execution. The bit-vector priorities associated with a binary decomposed tree are shown in Figure 3.

One might argue that by converting a *B-ary* tree into a binary tree, we have reduced the available parallelism of the problem. This is true only when the total number of nodes in the queue (or stack) is less than the available number of processors in the system. This happens at the initial and relatively small stages of the search as the tree is explored. Once the number of nodes in the queue becomes greater than the number of processors, the available parallelism is more than sufficient to keep all processors busy. Without binary decomposition, unnecessary work may be performed to produce *all* the children of a node that is picked up for execution resulting in an increase in the memory requirement to store the active nodes. Binary decomposition technique results in a decrease in the memory requirement and as well as reduced wasted work.

3.3 Reducing Space Dependence on Number of Processors: Delayed-Release

As shown by the expression for the stack length in Section 2, the memory usage in a parallel search also depends on the number of processors in the system. In both the multiple and shared stack models, the increase in the memory usage with processors occurs because of the availability of possibly a large number untried alternatives (nodes) at each level of the search space. This *immediate availability* of parallelism (alternatives) at shallower levels in the search space is the major cause for increased wasted work.

With bit-vector priorities, processors pick the leftmost P nodes. Although this eliminates anomalies, the memory requirement is still proportionate to $(P * D)$ for a search space with depth D. This is because at every level of the search tree, P processors may pick P nodes from the shared queue and create $(P * B)$ children. Thus, there is an increase of at most $P * (B - 1)$ nodes at every successive level of the search space. This can be avoided by delaying the nodes at shallower levels of the search space and making them available to processors at the bottom level when a leaf node is encountered. This delaying effect can be achieved if processors *skip* intermediate levels in the search space and after reaching the deepest level (leaves) of the tree explore nodes from the *bottom* of the search tree. This *bottom-first* strategy gives rise to a search behavior called the *broomstick* sweep of the search space as shown in Figure 4. The set of nodes searched 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 shape of a *broom*.

We achieve this *broomstick* behavior by a technique called *delayed-release* as described below. The search begins at the root in the usual manner. When a node is expanded, all its children except the leftmost child l are put in a list accessible from l. This list is local to l and the children nodes in the list are not immediately available to other processors at the time they are generated. The local list inherited by a node (parent node) is appended to the local list of its leftmost child l. Therefore, each processor effectively creates only *one* child when a node is expanded, which is inserted into the shared priority queue. This technique is applied to every node that is picked up for execution until a leaf node is encountered. At this point, all the nodes kept in the list are released as active nodes and made available to be picked up by other processors. Parallelism is suppressed due to the delaying of nodes to the bottom of the search space. However, this delay is insignificantly small and parallelism is quickly available and exploited from the bottom of the search space. Figure 5 gives the algorithm for the delayed-release scheme.

Figure 6 shows the state of a binary search tree using the delayed-release technique. When node R is expanded, only one *supernode* consisting of the linked list $[L_1, R_1]$ is produced. A supernode may contain several regular nodes as well as lumped-nodes. Node R_1 may represent a single right child for a binary tree or a *lumped-node* representing the remaining right children for a *b-ary* tree. When this supernode is picked up for execution, only the first node L_1 from the list is expanded, and a supernode $[L_2, R_2, R_1]$ is released. Eventually, when the supernode consisting of $[A, B, R_5, R_4, R_3, R_2, R_1]$ is picked up for expansion, it is discovered that node A is a leaf; thus, 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, nodes A, B, C, D, E, R_2, and R_1 form the set of frontier nodes. The nodes linked by solid arrows within a frontier node are the nodes created but not yet released to other processors. The bit-vector priorities assigned in the usual manner ensure that the P processors in the system pick the bottom P nodes for execution.

3.4 Memory Usage with Delayed-Release

We now determine if there is any saving in memory usage in exploring the search space in a delayed-release manner. Let D be the depth of the uniform search tree and B its branching factor. At each level, only one node is made available for execution and the remaining $(B-1)$ nodes are collected in a list which is passed down from the parent to its leftmost child. Therefore, when a leaf node is encountered at depth D, $(B-1)*D$ nodes are released for execution. Thus, memory is needed to store $(B-1)*D$ active nodes in the queue. Since the bottom nodes have higher priorities, the processors pick the P highest priority nodes for execution and proceed in a similar manner. We assume that all processors take the same amount of time to expand a node to create its children. We also assume that the time to insert and delete nodes in a shared pool of work is small, compared to a node expansion time and therefore is ignored for this analysis.

Let $P = k(B-1)$ be the number of nodes picked up from the depth D to depth $D-k$ where k is some integer greater than or equal to 1. If $k=1$, then $P \leq (B-1)$ and only the nodes from the depth D are picked up for execution. Also, let n be the least i such that $P \leq (B-1)^i$; i.e. for some i, the number of nodes $(B-1)^i$ becomes larger than P. If there are sufficient nodes then P nodes are picked up for execution each cycle. The $(B-1)$ nodes created at depth D are the leaf nodes and thus do not produce any children. The $(B-1)$ nodes created at depth $(D-1)$ generate $(B-1)^2$ children one cycle later. This is because each of $(B-1)$ children at depth $(D-1)$ release $(B-1)$ nodes at depth D one cycle later. Similarly, $(B-1)$ nodes created at depth $(D-2)$ will generate $(B-1)^3$ at least 2 cycles later. If there are sufficient nodes available at depth D, then nodes at shallower depths will not be picked up for execution, thus limiting further expansion of nodes. This happens at the ith iteration when $(B-1)^i$ becomes larger than P.

Therefore, the multiplicative factor of P in the worst case stack length is reduced to an additive factor and the stack length is given by

$$Maximum\ Stack\ Length \approx\ (B-1)*D - P*i + max((B-1)^i, P) + P*B*i$$

Objectives Achieved

We summarize the following objectives achieved by the delayed-release technique.

1 The wasted work is virtually eliminated as processors are forced to skip node expansions at intermediate levels and focus on the nodes deeper in the search space.

2 The memory required by the search algorithm is now proportional to sum of P and D as opposed to their product.

3 As wasted work is reduced, it results in consistent and good linear speedups to the first solution.

4 Since the number of active nodes in the priority queue at any time is reduced, the overhead of managing the queue is reduced, thus marginally improving the time to the first solution.

3.4.1 A Refinement: Delayed Partial Release

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 P is the number of processors in the system, then at most P highest priority nodes will be picked up for execution. If the number of nodes released is larger than the number of processors in the system, then the nodes released in excess of P constitute excess parallelism that can not be exploited at the time of release since, there will be enough work to keep all processors busy. These excess nodes can always be released at a later time if no solution is generated from the first P nodes picked up for execution.

The *delayed partial release* technique works as follows. Whenever a processor needs to release nodes, a maximum of P highest priority nodes and a supernode that is comprised of the remaining nodes are released, i.e., a maximum of P nodes are released. In this way, parallelism is controlled and limited to P at every release. If a processor completes its task and cannot find other high priority nodes (possibly created by expanding leftward nodes) it can expand the supernode comprising the excess nodes and proceed in the usual manner. This technique does not result in a reduction in memory space, since memory space is still needed to store the *unreleased* excess nodes in the list. Since now there are even fewer entries in the queue than with the *delayed-release* strategy, the overhead of managing the priority queue is further reduced.

4 Chare-Kernel: The Run Time Execution Environment

The above strategies are implemented in the *Chare-Kernel* parallel programming system [28, 29, 16], with a base language similar to C. The Chare-kernel provides the user with a run time environment for machine independent parallel programming. Using the base language, one can break down the execution of a program into *small processes or tasks* (called chares) of medium-grain size. The Chare-Kernel enables the programmer to implement process models for implicitly parallel High

Level Languages (HLLs) or to write explicitly parallel programs (called *Chare-Kernel programs*) directly.

The *Chare-Kernel* is a message driven execution environment. A message is either a *seed* for a medium-grained process (called a chare) or a *response* to a (parent) chare. Chares are activated upon creation or receipt of a message. Once active they may create other chares, generate and assign priorities to them or send response messages to other (or parent) chares. However, they cannot receive messages in a specific order, nor can they wait on messages of specific types. A message specifies an *entry point*, a *chare-id*, and a *priority* in addition to the message data. Activating a chare involves jumping to the entry point associated with it and executing instructions, until either it suspends itself or creates new chares or messages and completes. A suspended chare is awakened when a message meant for it is selected for execution. There is no preemption and the chares are allowed to run till they either suspend or terminate.

The *Chare-Kernel* maintains a pool of messages and is responsible for supplying work to the processors from this pool. It manages and schedules messages, provides different queueing strategies for message selection, performs load distribution, and memory management on both shared-memory and message-passing machines. Currently it supports bit-vector priorities of arbitrary lengths in addition to integer priorities on several shared memory multiprocessors such as the Encore Multimax, Sequent Balance and Symmetry and Alliant FX/8. Response messages from children chares to other chares that carry partial solution are assigned the same priority as that of the children chares so that response messages also get executed in a prioritized fashion [30]. The message selection, queue and memory management are all *invisible* to the application. It therefore provides a clean separation between decomposition of computation into parallel parts and selection and allocation of parallel actions to processors.

When an idle processor requests for a chare or a message for execution, the *queueing strategy* decides which message to schedule next from a pool of messages. Also, when load needs to be balanced in the system the queueing strategy selects which message needs to be sent to a remote processor based on their priorities. The load balancing decisions affect the processor utilization, and the execution time of a program, while memory utilization and time to first solution depend on the task decomposition techniques, priorities associated, and the queueing strategies that manage the priorities in an efficient manner.

The queueing strategy in the shared-memory implementation of the Chare-Kernel currently provides a global message queue shared by all processors and separate response queues for each individual processor. The response message carries either the partial solution or a failure to its parent chare. Since processors share the queue the simplest way to provide access under mutual exclusion is to associate a lock with the queue. This scheme is feasible for shared-memory multiprocessors with up to 20-30 processors.

The Chare-Kernel provides its own memory management that avoids the contention in the allocation of shared-memory space. Two memory management schemes, a *quick fit allocator* and a *buddy system allocator* are provided on the shared-memory machines. We have used the quick fit allocator in all our experiments. For a detailed explanation of memory management schemes see [31].

4.1 Grain Size Control

If the cost of making work available to another processor exceeds the cost of executing it at the local processor, then it does not make sense to decompose and parallelize work beyond a certain size or *granularity* of work. The ideal grain size for a shared-memory system depends on the actual overhead involved in picking up new work from the shared pool, creating parallel *new chares* or sending *responses* to parent chares, and putting them back into the work pool. In a distributed-memory system, the overhead includes the cost of distributing work to other processors, and therefore depends on the load balancing strategies used. If the granularity is too large, the loss in parallelism and uneven load distribution may cause processors to idle. And if the granularity is too small, then the cost of decomposing work to parallel subcomputations may exceed the cost of executing it sequentially.

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. The search space beneath a node at the cut-off depth is searched sequentially by a processor. Other techniques that attempt to gauge the size (granularity) of subtrees beneath a node and make a decision to either expand into parallel subtasks or explore it sequentially, are also possible. With grain size control, the overhead cost of parallelization is amortized over the total execution time. We found that such simple techniques using cut-off depths were sufficient to prevent the priority queue from being a sequential bottleneck. With a large number of processors one may increase the grain size to maintain the frequency of access to the shared queue. In terms of processing time the average grain size is defined as

$$Average\ Grain\ Size\ =\ \frac{Total\ Sequential\ Execution\ Time}{Total\ Number\ of\ Messages\ Processed}$$

To retain similar speedup properties the grain size and the number of granules must both increase proportionally to 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

[32, 33, 34, 35]. More importantly, as absolute adherence to priorities is not essential, techniques using multiple heaps with load balancing techniques that enforce processors to pick high priority work are possible. Experiments conducted on shared-memory machines of up to 30 processors using the Chare-Kernel have shown that with an average grain size of about 10-100 millisecond, the total overhead for parallelizing computations is insignificantly low compared to the total execution time.

5 Performance on Shared Memory Architectures

We implemented the above search techniques for parallel search on a shared-memory machine with a bus-based architecture and studied their performance. The Sequent Symmetry was used as the shared-memory architecture. The strategies were implemented using the Chare-Kernel [36] parallel programming system.

We present and discuss the performance data on a few OR-parallel search problems to obtain the first solution. We discuss the performance on the *N-Queens*, *Knights-Tour*, and *Magic Squares* problems below.

5.1 N-Queens Problem

The *N-Queens* problem is an established benchmark for OR parallel search. The goal is to place N Queens on an $N \times N$ chessboard such that no two queens are placed on the same row, column or diagonal (i.e., no two queens attack each other). This problem has a large number of solutions (e.g., the 8-Queens problem has 92 solutions), where any one solution may be acceptable. The search space can be represented by an OR-tree with the OR branches specifying the different choices for placing the next queen. To place the next queen, the search algorithm uses the 'most constrained' heuristic of selecting a row that has the fewest placement choices [37, 38]. Since at every placement there are at most N ways of placing the queen, the average branching factor at shallow depths of the search space is close to N. At deeper levels of the search space, as queens are placed, the number of queens that can be placed decreases since many of the positions are pruned.

We obtained performance data for the first solution to the *126-Queens* problem. The problem size N was chosen to be large to show that a large problem can be solved within a reasonable amount of memory usage and time. The sequential depth-first search to a first solution expanded 35,248 nodes. A grain size of 20-Queens was used, i.e., the first 106 queens are placed by breaking down computations and parallelizing work with the last 20 queens placed using a sequential algorithm within a processor. This led to an average grain size of 45 milliseconds.

Performance Data and Comparison

The experiments were carried out on the Sequent Symmetry with up to 20 processors. For the *126-Queens* problem it is not possible to run the experiment with a complete decomposition at every level since the memory requirement overflows the available memory. Therefore, all the search strategies used the binary decomposition technique of Section 3. In a later experiment we show how memory usage increases as the decomposition factor is increased.

The speedup plots in Figure 7 show that with bit-vector priorities the wasted work is reduced, resulting in linear, clearly monotonic speedups. The delayed-release technique further improves the speedup performance. The performance data is obtained from a single run of the *126-Queens* problem and is very consistent over different runs. Thus, it supports our claim that priority bit-vectors do indeed eliminate anomalies and consistent linear speedups are obtainable. The stack-based strategies yield speedups that vary from run to run, as documented by Kumar et al. in [22]. The speedups with the shared stack model also varied wildly between highly superlinear and extreme sublinear and thus are not reported here. The sequential execution time for the first solution to the *126-Queens* on Sequent Symmetry took 201 seconds. Figure 8 shows the maximum queue lengths measured in number of messages obtained for different strategies. The plot for worst-case queue length obtained from the expression derived earlier is shown in the figure for comparison. The plots in the figure show that the maximum queue lengths increase proportionately to the number of processors with the binary decomposition technique using bit-vector priorities. This dependence on the number of processors is eliminated with the delayed-release technique. With the delayed partial release technique the queue lengths are reduced even further.

Comparing the two figures for maximum queue lengths in Figure 8 and maximum memory used (in MBytes) in Figure 9, we observe that memory usage is proportional to the to the maximum queue lengths obtained. The average message size (including the size of the node state) is approximately 1000 bytes for the stack and 1200 bytes for the priority queue. The increase in memory space for messages is needed to store the bit-vector priorities and related information to manage messages with priorities. The memory plots also show that the memory usage is roughly the same for the two delayed-release techniques (see Section 3.4). Both Figures 8 and Figure 9 support our claim that memory usage can be reduced considerably with our techniques.

Table I shows the total number of nodes expanded for the three binary decomposition schemes using bit-vector priorities. It shows that (a) with priorities, the work performed with P processors is not significantly more than with 1 processor (for example, for the delayed-release technique 36,507 nodes are expanded using 18 processors compared to 35,248 nodes with a sequential search: an increase of 3.57%) and (b) the wasted work is reduced when delayed-release techniques are employed. Table II shows that the maximum queue lengths and memory usage increase considerably when the branching factor is allowed to increase. In this experiment instead of using the binary decomposition, we have used *k-ary* decomposition where k is called the *decomposition factor*, such that if no solution is found from the left $k-1$ nodes, the kth node (lumped-node) is expanded to create another set of at most k children nodes. The corresponding speedup figures in Table II shows that an increase in the decomposition factor increases wasted work resulting in a performance degradation.

5.2 Knights-Tour Problem

The *knights-tour* of an $N \times N$ chess board is another example of state-space search. The knight must visit each position on the chess board *once* and return to its starting position. Many solutions exist for the knights-tour problem (the knights-tour is an instance of the Hamiltonian circuit problem). The knight must move according to the following conditions. The knight may move from its current position to another position on the board by moving 2 positions along a row (or column) and 1 position along the column (or row). Therefore, there are at most 8 distinct positions a knight can reach from a given position.

We used an $8X8$ chessboard for this problem. The knight starts the tour from a corner of the board returning to the same position. To improve performance, the search algorithm uses the following heuristic. From any position the knight looks ahead to see if out of the 8 possible positions there exists a position that can be reached from the current position and only one other position. If such a position exists, it moves to that new position discarding the remaining choices. If there is no such position, then the possible moves are ranked left-to-right arbitrarily (in clockwise order).

Performance Data and Comparison

Figure 10 shows the speedup plots to the first solution of the knights-tour problem on the Sequent Symmetry multiprocessor. All of the strategies used bit-vector priorities. The sequential execution examines approximately 30,000 nodes before obtaining the first solution. Again, consistent linear speedups are obtained. The maximum queue lengths in Table III shows that the stack length increases considerably with full decomposition of the search tree. Considerable improvement in performance is obtained with delayed-release techniques. The performance figures were obtained from a single run and are very consistent over different runs.

5.3 Magic Squares Problem

The *Magic Squares* is also an example of a state-space search. The problem is to place integers from *1 to* N^2 on a $N \times N$ square board, one integer in each smaller square such that the sum of the integers along any row, column or diagonal is identical. For any N, the sum of the integers along a row (column or diagonal) must be equal to $N * (N^2 + 1)/2$.

Performance Data and Comparison

We used a 6×6 board for the Magic-Square problem. Figure 11 shows the speedup plots to the first solution using bit-vector priority strategies. Very consistent and linear speedups were obtained up to 18 processors that improved with delayed-release schemes. Table IV shows the improvement in maximum queue length and memory usage with our search techniques. The data reflects the performance from a single run and is very consistent over different runs.

6 Applicability to Iterative Deepening-A* (IDA*)

In the earlier sections we demonstrated the effectiveness of priority based parallel search to eliminate anomalies and obtain consistent linear speedups to first solution for state-space searches. In this section we extend those ideas and apply to Parallel Iterative Deepening A* algorithm for obtaining optimal solutions.

Heuristic state-space search [39, 6] is an important problem-solving method that is used to solve a large variety of problems in Artificial Intelligence. A large class of such problems has multiple solutions. Sometimes, one is interested in an *optimal solution* based on certain cost criteria. Best-first search algorithms are generally used to find optimal solution to such problems. If an admissible heuristic is available, one can use the A* algorithm [39], which ensures that the first solution found is the optimal one. The drawback of A* algorithm is that it requires an exponential amount of memory to store the set of active nodes. The Iterative Deepening A* (IDA*) algorithm [17, 18], on the other hand, requires memory linear in the depth of the search space.

IDA* is a state-space search algorithm for finding an *optimal* solution. It works by iteratively conducting a cost-bounded depth-first search over the search space with increasing bounds. Similar to other heuristics search procedures (e.g. A* algorithm), it uses two functions g and h, where for each node n, $g(n)$ is the cost of reaching n from the initial node (root) and $h(n)$ is a conservative estimate of the cost of reaching the closest goal node from n (i.e. $h(n)$ is an admissible heuristic). In each iteration, IDA* expands nodes in a depth-first manner until the branch is cut off when the total estimated cost ($f(n) = g(n) + h(n)$) of the last node on that path exceeds the cost threshold for that iteration. The threshold for the first iteration starts at the estimate of the cost of the initial state, and increases for successive iteration of the algorithm. For each succeeding iterations, the threshold is set to the minimum f value of all node costs that exceeded the previous threshold in the previous iteration. Successive iterations continue until a goal node is selected for expansion.

If the heuristic function is *admissible*, then IDA* (similar to A*) is guaranteed to find an optimal solution to the problem. Furthermore, Korf in [18] has shown that IDA* expands the same number of nodes, asymptotically, as in A* for exponential tree searches. Also, since at any point it is performing a depth-first search, and never searches deeper than depth D of the search-space, IDA* requires storage $O(D)$ [40], and is highly space efficient. A seeming disadvantage of IDA* is that it performs duplicate computations of all but the last iteration prior to reaching the goal depth. Korf in [18] has shown that under reasonable assumptions, the duplicate computations do not affect the asymptotic growth of the run time for exponential tree searches.

Parallelizing the IDA* search algorithm to obtain the first optimal solution in a multiple solution domain has the same objectives as for a parallel depth-first search stated earlier. In addition the overhead must be low, so that (a) the speedups compared to a *sequential* algorithm are significant, and (b) small sized problems can be effectively parallelized.

6.1 Parallel Iterative Deepening A*

The Parallel IDA* (PIDA*) scheme was proposed by Kumar et al. [20, 21, 22] to obtain optimal solutions to random instances of the *15-Puzzle* problem. They implemented a multiple stack model for shared-memory and distributed-memory machines. They report anomalous speedups (from 3.46 to 16.27 with 9 processors) in [22] for the *first optimal* solution to the 15-Puzzle problem. Since PIDA* performs a stack-based depth-first search, the memory space needed by PIDA* increases proportionately to the number of processors in the system. Kumar et al. in [22] have reported good speedup performance for large to very large problem sizes (sequential execution times of 900 seconds to 36000 seconds). However, since each successive iteration of IDA* does not begin until the previous one is completed, processor idling for small problem sizes leads to performance degradation, even for all solutions. This phenomenon is not so visible for the large problem sizes that Kumar et al. have selected, but can become more pronounced if the number of processors is increased, keeping the problem size fixed.

Another approach by Powley and Korf using the IDA* algorithm in [41] employs different *processes* to search different thresholds (windows) simultaneously, hoping that one of them will find a solution, i.e., each process performs a separate iteration of the IDA* algorithm, except that some processes may do work beyond the goal iteration. Their objective is to find quickly any solution, optimal or non-optimal. Since the search time is dominated by the last (goal) iteration, even if others are performed in parallel, this strategy leads to poor utilization of processors, even for finding a *nonoptimal* solution. This is because even when there is enough work in iterations preceding the goal iteration, processors may be assigned to iterations beyond the goal iteration, and the processing effort may not contribute to a first solution. The IDA* algorithm computes several iterations of increasing sizes. Due to processor idling between successive iterations and insufficient work in the initial iterations, there may be a loss of speedup and efficiency. Running multiple iterations in parallel may improve the processor utilization but can be problematic because work done in iterations beyond the goal iteration (one with an optimal solution) is wasted.

We will show that with our techniques, speedup anomalies and the dependence of memory requirement on the number of processors are virtually eliminated, resulting in reduced wasted work and consistent linear speedups to the first optimal solution. In addition our techniques improve the speedup performance that scales up as the number of processors is increased.

6.2 PPIDA* - Prioritized PIDA*

To eliminate anomalies and achieve linear speedups to a first optimal solution, we use *priorities* with the Parallel IDA* algorithm. We call this technique Prioritized Parallel IDA* (PPIDA*). Each successive iteration of IDA* is essentially a depth-first search of the cost bounded search tree and thus can be easily parallelized. We use the priority schemes described earlier in Section 3.1 to search each of the cost-bounded tree in parallel. Priorities are dynamically assigned to the nodes when they are created. In Parallel IDA* the root node of each new iteration of the cost bounded search tree is assigned a *null* (priority bit-vector of length 0) priority at the start of the search process. The m children of a node are assigned priority bit-vectors by extending the parent's priority, based on their rank as described earlier Section 3.

6.3 Performance of Prioritized PIDA* on Multiprocessors

To test the effectiveness of our Prioritized Parallel IDA* scheme, we used the 15-Puzzle problem, a larger 4×4 relative of the 8-Puzzle. The 15-Puzzle problem consists of a 4×4 square board containing 15 square tiles numbered from 1 through 15. The sixteenth square on the board is a blank. Any tile adjacent to the blank space can be slid into that space thus constituting a move. The problem is to start with any given board position where the numbered tiles occupy the square spaces in a random fashion and to slide the tiles around to reach a desired board configuration. We used the Manhattan distance heuristic (the $h(n)$ function) [39] to estimate the cost of a particular board configuration from the goal configuration. The average branching factor for this problem is 2. Depending on the initial board configuration, the 15-Puzzle problem can have a large number of solutions. We are interested in searching for a first optimal solution to the 15-Puzzle problem.

We implemented both the Parallel IDA* (PIDA*) and the Prioritized Parallel IDA* (PPIDA*) to solve the 15-Puzzle problem on shared-memory machines. Sequent Symmetry was selected as the shared-memory multiprocessor. For our experiment a problem instance from [18] was selected that expanded approximately 10,500,000 nodes and to a depth of 53 to the first optimal solution. For the same problem instance sequential IDA* expands 9,982,569 nodes to the first optimal solution. The difference is due to the order in which the tiles are slid into the blank space. The sequential execution to the first optimal solution for this problem instance takes 1100 seconds on the Sequent Symmetry. We used grain size control to decide when to stop breaking a computation into parallel subcomputations and executing it sequentially on a processor. Since IDA* searches a cost-bounded search space iteratively, we stop decomposing the computations when the difference between the current bound and the cost ($f(n)$) of a node n drops below the specified grain size; i.e., when $bound - f(n) \ \leq grain\ size$, the cost-bounded subtree under node n is searched sequentially. In our experiments for the 15-Puzzle problem, a grain size of 6 was selected, which led to an average grain size of 40 milliseconds.

The performance data was obtained using a last-in-first-out (LIFO) stack to show that anomalous speedups are obtained and which vary from one execution to another. We also obtained performance with *delayed-release* techniques using bit-vector priorities. The speedup plot in Figure 12 shows the anomalous behavior to the first solution using a LIFO stack with PIDA* on the Sequent Symmetry. The speedup plots obtained with our scheme (Prioritized PIDA*) scheme show that linear and monotonic speedups are obtained up to 18 processors. Table V gives the maximum memory used to obtain the first optimal solution. The memory usage increases rapidly with a stack-based technique as the number of processors increase. With our schemes using priorities, memory usage is reduced that does not increase proportional to the number of processors (for example, memory usage of 0.27 MBytes with priorities compared to 0.403 MBytes with a stack based scheme using 18 processors). We again emphasize that the performance data represents results of an experiment by running a problem instance *once* as opposed to the average over several runs. We ran our algorithm using bit-vector priorities on different problem instances. The results obtained from multiple runs of any problem instance are the same. This supports our claim that bit-vector priorities do indeed eliminate anomalies and very consistent speedup performance can be obtained.

7 Prioritized Iteration Overlapped Parallel IDA*

Although we obtained consistent and good linear speedups to large problems on shared-memory multiprocessors, the speedups performance on small problems are not as high as they could be. In PIDA*, processors pick up nodes from the queue and search in a depth-first manner within the *same* cost-bounded space. The threshold starts at the estimate of the cost of the initial state and increases for each iteration of the algorithm. The cost-bounded trees correspondingly increase in size. Therefore, in the parallel version of IDA*, all processors search in parallel, an iteration k with a lower threshold *completely* before proceeding to the *next* iteration $k+1$ with a higher threshold.

The difficulty is that parallelism increases and decreases in *waves* with each successive iteration. At the beginning of each iteration the parallelism is low. It increases quickly as the search space is explored to occupy as many processors as possible, and then trails off as the iteration winds down before the next iteration begins. If enough work is not generated to keep all processors busy, the processors idle. Thus, processors may idle because of diminishing work during the completion phase of one iteration until enough work can be generated in the next iteration which causes the performance to degrade.

The problem size selected in the previous section was large (sequential execution ≈ 1100 seconds) and expanded close to 10 million nodes. Since the computation is dominated by the last iteration the effect of processor idling in between iterations with a small number of processors (of up to 20) is not very apparent. If we were to select a small problem size, we would witness the degradation in performance even with a small number of processors. Figure 13 shows the speedup performance of another instance of the 15-Puzzle problem expanding 300,000 nodes to the first optimal solution (sequential execution time of 42 seconds). Although we obtain consistent performance to the first optimal solution, the speedup performance drops rapidly as the number of processors increase. Also, if a large number of processors is used for the large problem size, then a situation similar to that shown in Figure 13 could exist as processor idling may occur.

We now show that the processor idling can be reduced if the idle processors are allowed to start the next iteration in a *speculative* manner, hoping that in case no solution is found from the current iteration, then work performed speculatively in the successive iterations will not be wasted and may possibly contribute to the solution if one of the successive iterations contains the optimal solution. In general, the idle processors are allowed to start not *one* but *several* iterations concurrently using priorities, as shown in Figure 14. This technique is called *Prioritized Iteration Overlapped PIDA** (PIO-PIDA*) [12]. However, this technique requires that if the threshold at an iteration is known, then one must be able to compute the threshold for the next several iterations.

For the 15-Puzzle problem, if the threshold at an iteration k is σ, then the threshold for the next iteration $k+1$ is given by $\sigma + 2$. In problems such as these it is possible to know the threshold for the next iteration without completing the current iteration. The number of iterations that can exist at any given time is called the *Speculative Computing Factor* (SCF) for this search strategy. Since several iterations may be executing concurrently, there may be computations (nodes) from different iterations in the work pool at any given time. To obtain an optimal solution, it is necessary to ensure that work from the iteration with the smallest threshold receives the highest priority. We achieve this using bit-vectors priorities. Each root node of a successive iteration is assigned a nonempty and *decreasing* bit-vector priority using a novel scheme described below.

If the total number of iterations is known a priori, we can set aside a fixed number of bits to encode each iteration distinctly. If we use n bits, called the *static bits*, to encode priorities, a maximum of 2^n iterations can be assigned distinctly encodings. For iterations greater than 2^n the encoding is set to all *ones* giving the same priority to any further iterations.

As the total number of iterations is generally not known a priori, one cannot use a fixed number of bits to assign priorities distinctly to each iteration. Instead we use *dynamic bits* [30] to encode an unbounded number of iterations that are started. The *dynamic bits* encode each iteration dynamically and distinctly. The *dynamic bits* are of the form $X0Y$, where X is a string of n 1's and Y is a string of n bits ranging from 00...0 to 11...1. The first iteration is encoded with a priority 0 ($n = 0$). The second and third iterations are encoded in the form $X0Y$ as 100 and 101. To obtain the encoding bits for next iteration, Y is incremented unless Y is already all 11...1, in which case X is extended by one bit and Y is set to (n+1) 0's. An iteration number N can be represented with this scheme by 1^k0Y where $k = \lceil \log N \rceil$ and $Y = N - 2^k + 1$ is a k-bit binary number.

The work in any particular iteration would still need priorities as explained in Section 6.2 to order the nodes left-to-right to eliminate anomalies and reduce wasted work. In this strategy, the root nodes of each new iteration will have the priority of that iteration (as explained above) instead of a null priority as in the Prioritized PIDA* scheme of Section 6.2. Note that due to the prefix property satisfied by the dynamic bits, all nodes in an iteration with a smaller threshold are at higher priority than nodes in iterations with larger thresholds. If an idle processor can not find work in iterations that have already started, it *leapfrogs* and begins the next successive iteration using the dynamic count. Since there are now many iterations running in parallel it is possible that a nonoptimal solution may be found before an optimal one, despite the priorities. Therefore, optimality is guaranteed by completing all earlier iterations (an iteration 'completes' if it either finds a solution or exhausts its search space) with smaller thresholds once a solution is found.

The idea of allowing multiple iterations in parallel to search different thresholds (windows) was used by Powley and Korf in [41] with a different objective. In Powley and Korf's implementation, each of the P *processes* is assigned an iteration each with different thresholds to search. Each process expands the root node to a fixed number of nodes, called the fixed frontier set. When all the nodes in the fixed frontier set are examined, the minimum h value for each node, its associated path information and the value of the threshold are broadcast to other processes to order their nodes. This ordering information from one iteration to the next is therefore sequentialized. Their analysis shows that the minimum number of processes required in the best case is $D/2$ where D is the length of the optimal path. The disadvantage of this scheme is that if there are more than $D/2$ processes running concurrently, work performed in iterations with thresholds much larger than the one that generates the optimal solution may be wasted. If there are less than $D/2$ processes, then work may not be sufficient to keep all processors busy. Moreover, since D is not known a priori, one can not determine the minimum number of processes needed accurately. However, Powley and Korf's scheme can be supplemented with our prioritizing strategy, which will ensure that the iterations with lower thresholds receive the maximum resources to improve performance.

Performance of PIO-PIDA* Scheme

To test the effectiveness of our technique we ran experiments using PIO-PIDA* (Prioritized Iteration Overlapped Parallel IDA*) to solve instances of the 15-Puzzle problem. We used two problem instances of different sizes. One expanding 300,000 nodes (small problem size: sequential execution time of 42 seconds) and the other expanding 2,000,000 nodes (larger problem size: sequential execution time of 116 seconds). We obtained performance data for the two different problems with different values of speculative computing factors (SCF) on the Sequent Symmetry. The SCF was varied from 1 to 4 to observe the effect of running several iterations concurrently on the performance. Running an experiment with SCF=1 corresponds to the Parallel IDA* (PIDA*) using priorities, i.e., the next larger iteration is not started before the previous one is completed.

The speedup plots for the two problem sizes are shown in Figure 15 and Figure 16. For the smaller problem instance (Figure 15), the speedups improve considerably as the number of concurrent iterations are allowed to increase from 1 to 4 (speedup of 11.2 for SCF=1 to speedup of 18.5 for SCF=4 using 20 processors). This is because with a smaller problem size the percentage of time that processors remain idle (mainly between iterations and earlier non-parallel iterations) is large compared to the time to the first optimal solution. The improvement in speedup for the larger problem (Figure 16) from 1 iteration to 4 iterations in parallel is also significant.

8 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. We also extended these techniques for Parallel IDA*. The extensions were needed because IDA* involves a series of increasing depth-first searches. An important extension to our basic scheme allows successive iterations to run concurrently with minimum wastage. This improves the speedups for small problems or for systems with a large number of processors.

To the best of our knowledge, no other method proposed to date consistently achieves monotonically increasing speedups for a first solution. In addition to speedups, our methods also substantially reduces the memory requirements. Compared to other methods, which require memory proportionate to the product of depth D of the search tree and number of processors P, our technique is proportional to their sum. It was demonstrated that in practice, with as low as 10-20 processors, the savings were substantial. 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. The limitation of our technique is that priorities once assigned do not change and therefore the scheme cannot be generalized to other searches such as the best-first search techniques.

Recently, Rao and Kumar have derived an interesting result in [42] 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 reasons. 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 search-spaces for more consistent speedups[2].

The use of priorities effectively *decouple* the parallel search *algorithm* from the scheduling strategy [43]. (In contrast, schemes such as [44, 45] 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. Synchronization is much

[2]Their results prove that depth-first search is not the best sequential algorithm for these search domains. Further, it suggests a possibly best sequential algorithm: to use k stacks each with a node from a frontier of size k around the root, and to conduct a depth-first search within each stack by time-slicing between nodes on all stacks. k is problem dependent, but usually small according to empirical evidence of [42]. The synthesis we suggest essentially applies our prioritization strategy to this modified sequential algorithm.

simpler, as the interactions between processors are limited to accessing the common pool of work on small shared-memory machines. Most important, the prioritization 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.

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 a more complex situation than the pure state-space (OR-tree) search discussed in this paper, if AND-parallelism is also to be exploited. AND/OR trees, and their extensions such as the Reduce/Or Trees [46], are used to represent such computations. Unlike the OR nodes, all the children of an AND node must have a solution for the AND node to succeed. We have proposed and implemented a prioritization scheme for AND/OR parallel execution of Logic Programs [30]. The techniques developed in this paper, such as the delayed-release technique, can be incorporated in such schemes with appropriate modifications, to ensure monotonic speedups and reduced memory requirements [47].

Acknowledgements

This research was supported in part by the National Science Foundation under grant number CCR-89-02496. We would like to thank Ross Overbeek for providing access to the Sequent Multiprocessors at Argonne National Laboratory and David Padua for allowing us to use the Alliant FX/8 at CSRD, University of Illinois. Our thanks also to W. Fenton, B. Ramkumar, and Amitabh Sinha for their contribution to the *Chare-Kernel II.* Thanks also to Kevin Nomura for the earlier shared-memory version of the Chare-Kernel and to W. Fenton for some of the benchmark programs. We would also like to thank the anonymous referees for reviewing the manuscripts and for their comments, critiques and suggestions for improving the quality of the paper.

References

[1] Halstead R., "Parallel Symbolic Computing," *Computer*, pp. 35–43, Aug. 1986.

[2] Kale L. V., "Parallel Problem Solving," in *Parallel Algorithms for Machine Intelligence and Vision*, pp. 146–181, Springer-Verlag New York Inc., 1990. invited paper.

[3] Stolfo Salvatore J., "Initial Performance of the DADO2 Prototype," *Computer*, pp. 75–83, Jan. 1987.

[4] Wah B.W., "New Computers for Artificial Intelligence Processing," *Computer*, pp. 10–15, Jan. 1987.

[5] McDermott D. and Davis E., "Planning Routes through Uncertain Territory," *Artificial Intelligence*, vol. 22, no. 2, pp. 105–135, 1984.

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

[7] MathLab Group, LCS, MIT, *The MACSYMA Reference Manual*, Jan. 1983.

[8] Moses J., "Symbolic Integration: The Stormy Decade," in *Communications of the ACM*, pp. 548–560, Aug. 1971.

[9] Nilsson N.J., *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, Inc., 1971.

[10] Gelernter H., Sridharan, Hart, Yen, Fowler and Shue, "The Discovery of Organic Synthesis Routes by Computer," *Topics in Current Chemical Research*, 1973.

[11] Arvindam S., Kumar V, Rao V. Nageshwara and Singh V., "Automatic Test Pattern Generation on Multiprocessors: A Summary of Results," in *Second Conference on Knowledge Based Computer Systems*, pp. 41–51, Dec. 1989.

[12] Saletore Vikram A., *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1990.

[13] Hausman B., *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, Royal Institute of Technology, 1990.

[14] Burton F. W., "Controlling Speculative Computation in a Parallel Functional Language," in *International Conference on Distributed Computer Systems*, pp. 453–458, Nov. 1985.

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

[16] Kale L.V. and Shu W., "The Chare-Kernel Language for Parallel Programming: A Perspective," Tech. Rep. UIUCDCS-R-88-1451, Dept. of Comp. Sc., Univ. of Illinois, Urbana-Champaign, Aug. 1988.

[17] Korf R.E., "Depth-first Iterative Deepening: An Optimal Admissible Tree Search," in *Artificial Intelligence*, pp. 97–109, 1985.

[18] Korf R.E., "Optimal Path-Finding Algorithms," in *Search in Artificial Intelligence*, pp. 223–267, Springer-Verlag New York Inc., 1988.

[19] Imai M., Yoshida Y. and Fukumara T., "A Parallel Searching Scheme for Multiprocessor Systems and its Applications to Combinatorial Problems," in *International Joint Conference on Artificial Intelligence*, pp. 416–418, 1979.

[20] Kumar Vipin and Rao V. Nageshwar, "Parallel Depth First Search. Part 2: Analysis," *International Journal of Parallel Programming*, pp. 501–519, Dec. 1987.

[21] Rao V. Nageshwar, Kumar V. and Ramesh K., "A Parallel Implementation of Iterative-Deepening-A*," in *National Conference on Artificial Intelligence (AAAI)*, Aug. 1987.

[22] Rao V. Nageshwara and Kumar Vipin, "Parallel Depth First Search. Part 1: Implementation," *International Journal of Parallel Programming*, pp. 479–499, Dec. 1987.

[23] Li G.J. and Wah B.W., "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," in *IEEE Transactions on Computers*, pp. 568–573, June 1986.

[24] Li G.J. and Wah B.W., "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," in *International Conference on Parallel Processing*, (St. Charles, IL), Aug. 1984.

[25] Li G.J. and Wah B.W., "How Good are Parallel and Ordered Depth-First Searches," in *International Conference on Parallel Processing*, (St. Charles, IL), Aug. 1986.

[26] Charniak E., Riesbeck C.K., McDermott D., and Meehan J., *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, 1987.

[27] Hausman B., Ciepielewski A. and Haridi S., "OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *Symposium on Logic Programming*, pp. 69–79, Sept. 1987.

[28] Fenton W., Ramkumar B., Saletore V., Sinha A. and Kale L.V., "Supporting Machine Independent Programming on Diverse Parallel Architectures," in *submitted to the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[29] Kale L.V., "The Chare-Kernel Parallel Programming Language and System," in *International Conference on Parallel Processing*, vol. 2, pp. 17–25, Aug. 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*, pp. 390–406, Oct. 1989.

[31] Nomura Kevin R., "Memory Management in the Shared Memory Implementation of Chare Kernel," Master's thesis, University of Illinois at Urbana-Champaign, May 1989.

[32] Biswas Jit and Brown J.C., "Simultaneous Update of Priority Structures," in *International Conference on Parallel Processing*, (St. Charles, IL), pp. 124–131, Aug. 1987.

[33] Deo N. and Prasad S., "Parallel Heap," in *International Conference on Parallel Processing*, vol. 3, (St. Charles, IL), pp. 169–172, Aug. 1990.

[34] Jones D.W., "Concurrent Operations on Priority Queues," in *Communications of the ACM*, pp. 132–137, Jan. 1989.

[35] Rao V. N. and Kumar Vipin, "Concurrent Access of Priority Queues," in *IEEE Transactions on Computers*, Dec. 1988.

[36] Kale L.V., "The Design Philosophy of the Chare Kernel Parallel Programming System," Tech. Rep. UIUCDCS-R-89-1555, Dept. of Comp. Sc., Univ. of Illinois, Urbana-Champaign, Nov. 1989.

[37] Bitner J. and Reingold E.M., "Backtracking Programming Techniques," in *Communications of the ACM*, pp. 651–655, 1975.

[38] Stone Harold S. and Stone Janice M., "Efficient search techniques- An empirical study of the N-Queens Problem," *IBM Journal of Research and Development*, pp. 464–474, July 1987.

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

[40] Korf R.E., "Real-Time Heuristic Search: New Results," in *National Conference on Artificial Intelligence (AAAI)*, pp. 128–132, Aug. 1988.

[41] Powley C. and Korf R.E, "Single Agent Parallel Window Search: A Summary of Results," in *International Joint Conference on Artificial Intelligence*, pp. 36–41, Aug. 1989.

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

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

[44] Kale L.V., Padua D.A., and Sehr D.C., "OR-Parallel Execution of Prlog with Side Effects," *The Journal of Supercomputing*, 1988.

[45] Lusk E., Warren David H. D, Haridi Seif *et. al*, "The Aurora OR-parallel Prolog Sýstem," in *International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan), pp. 819–830, Nov. 1988.

[46] Kale L.V., "A Tree Representation for Parallel Problem Solving," in *National Conference on Artificial Intelligence (AAAI)*, (St. Paul), Aug. 1988.

[47] Saletore Vikram A., Ramkumar B., and Kale L.V., "Consistent First Solution Speedups in OR-Parallel Execution of Logic Programs," Tech. Rep. UIUCDCS-R-90-1725, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Apr. 1990.

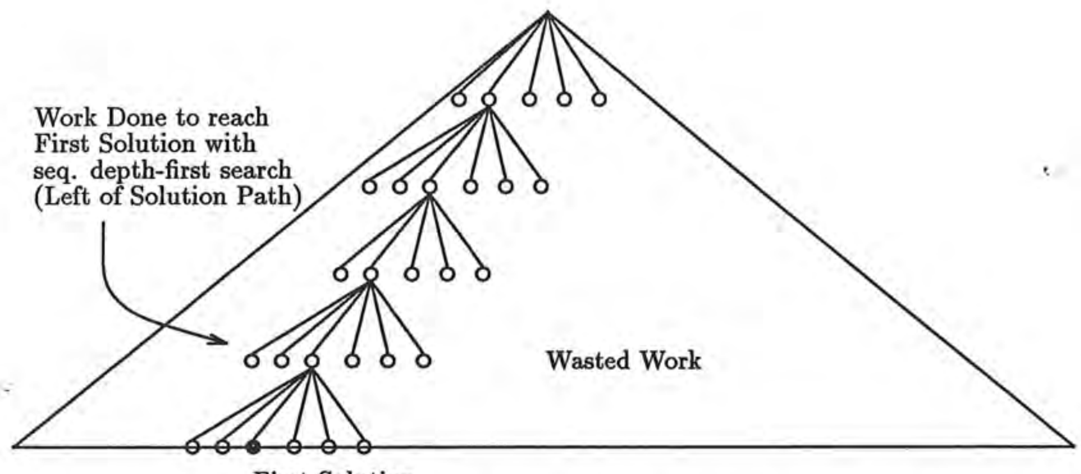

Figure 1: Sequential depth-first search of a search tree and *wasted work* with parallel execution.

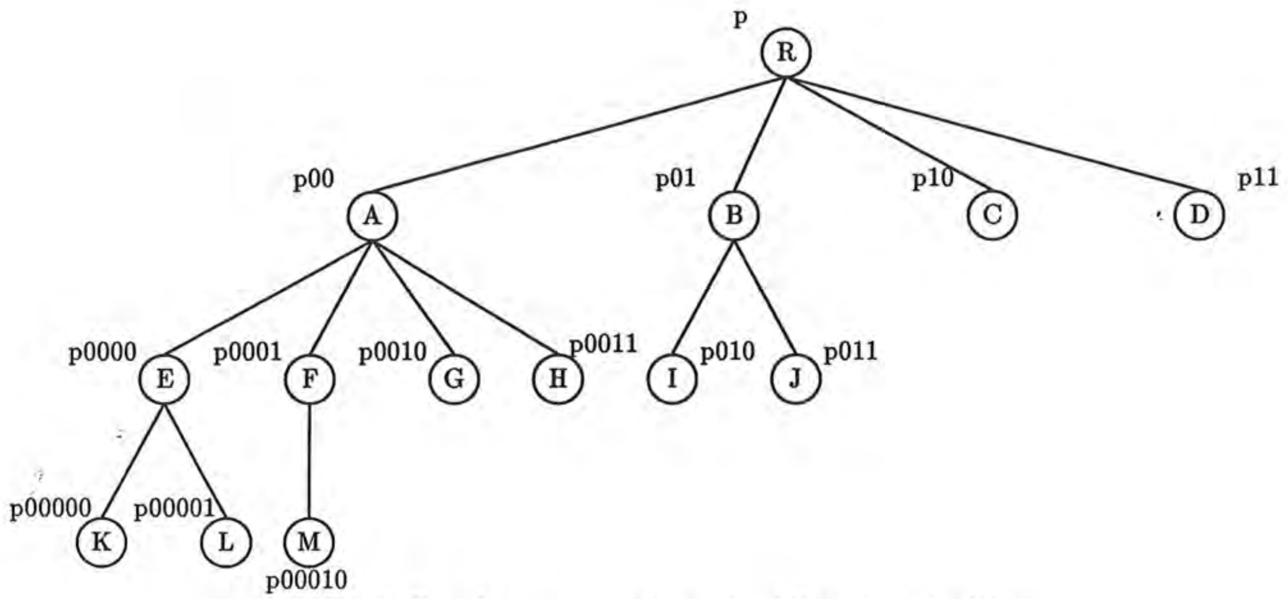

Figure 2: Search tree expansion using bit-vector priorities.

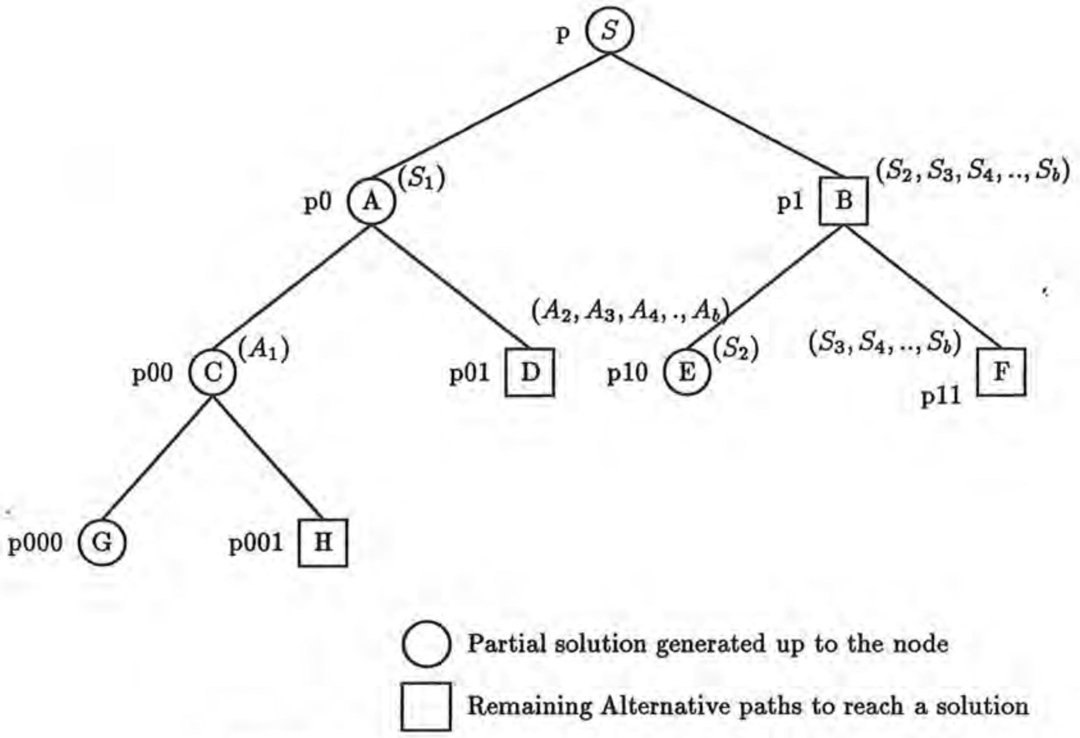
p
S
p0
A
(S_1)
p1
B
$(S_2, S_3, S_4, .., S_b)$
p00
C
(A_1)
p01
D
$(A_2, A_3, A_4, ., A_b)$
p10
E
(S_2)
$(S_3, S_4, .., S_b)$
p11
F
p000
G
p001
H
Partial solution generated up to the node
Remaining Alternative paths to reach a solution

Figure 3: Binary decomposition of a search tree.

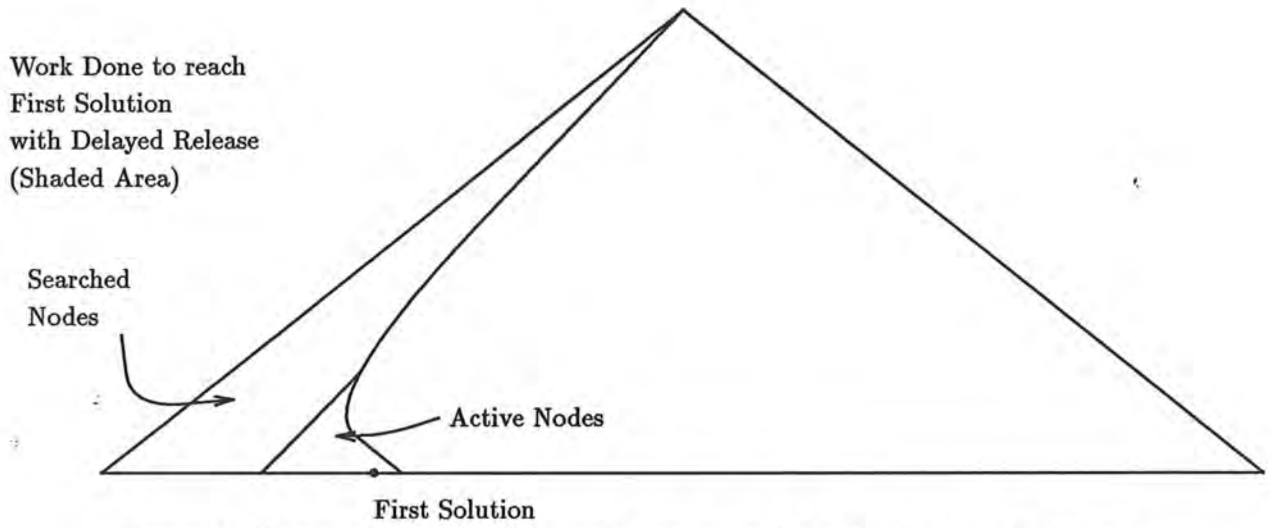

Figure 4: *Broomstick* sweep of a search tree using the *delayed-release* technique.

```
Delayed_Release(parentNode)

     if (leaf(parentNode)) /* parentNode is leaf node */
         /*
           release() releases all nodes in the list
           pointed by parentNode
         */
         release(parentNode);
     else
         /*
           expand() returns pointers to the leftmost and rightmost
           nodes of the list of children nodes created as a
           result of expansion of parent node.
           append() appends the list of children nodes to the
           list of nodes pointed by parent node.
         */
         expand(parentNode, leftmost, rightmost);
         append(parentNode, leftmost, rightmost);
         Enqueue(leftmost);
```

Figure 5: Algorithm for the implementation of the *delayed-release* technique.

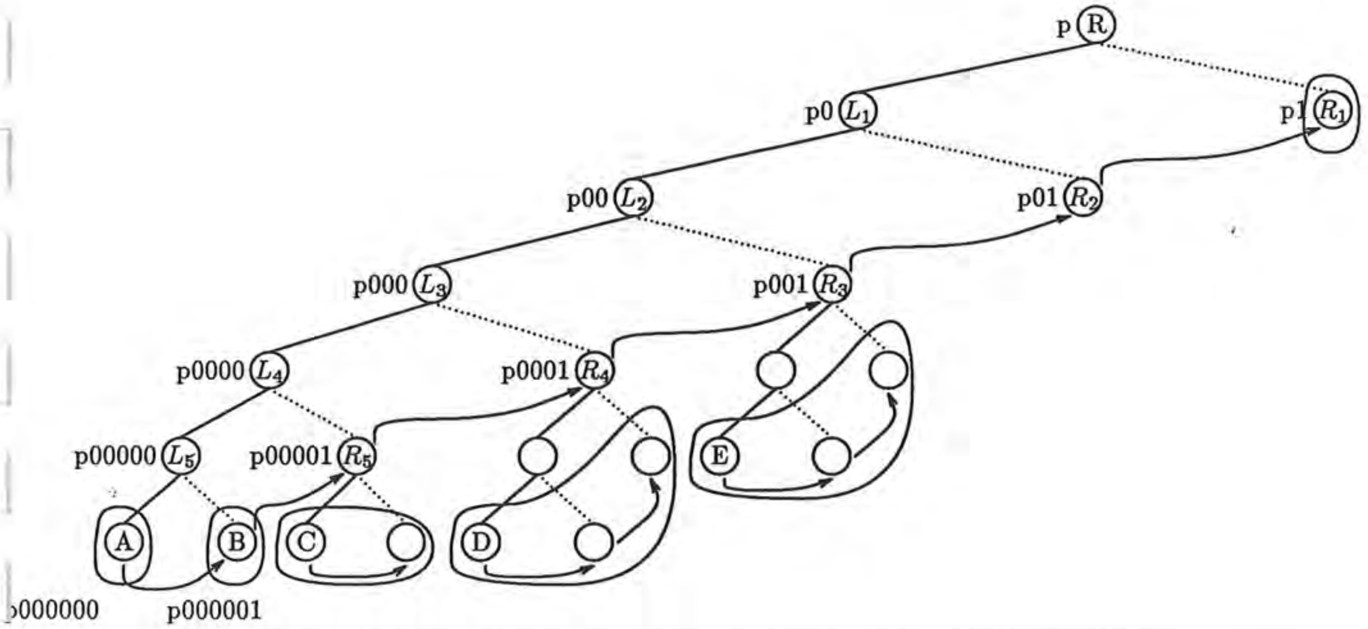

Figure 6: Expansion of a search tree using the *delayed-release* technique.

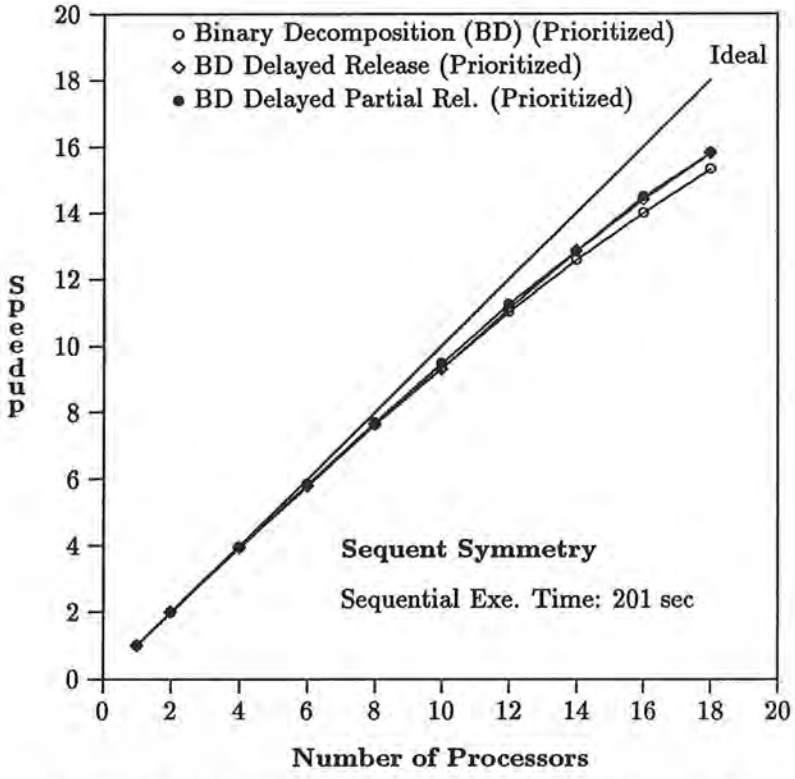

Figure 7: 126-Queens: Speedups to the first solution on Sequent Symmetry.

Table I: 126-Queens: Number of nodes expanded to the first solution.

126-Queens: Number of Nodes Expanded on Sequent Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
BD	35248	35338	35574	35917	36164	36371	36578	36637	36795	36944
BD-DR	35248	35241	35306	35532	35601	35738	35987	36051	36229	36507
BD-DPR	35248	35265	35357	35439	35644	35786	35870	36077	36219	36448

BD: Binary Decomposition BD-DR: BD Delayed Release

BD-DPR: BD Delayed Partial Release

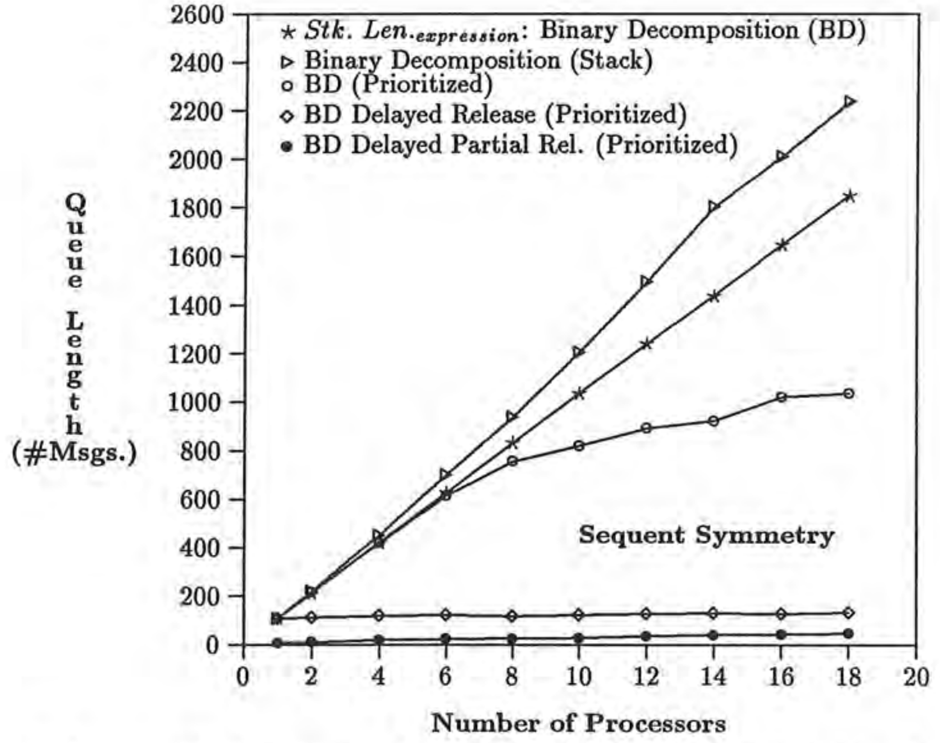

Figure 8: 126-Queens: Maximum queue lengths to the first solution on Sequent Symmetry

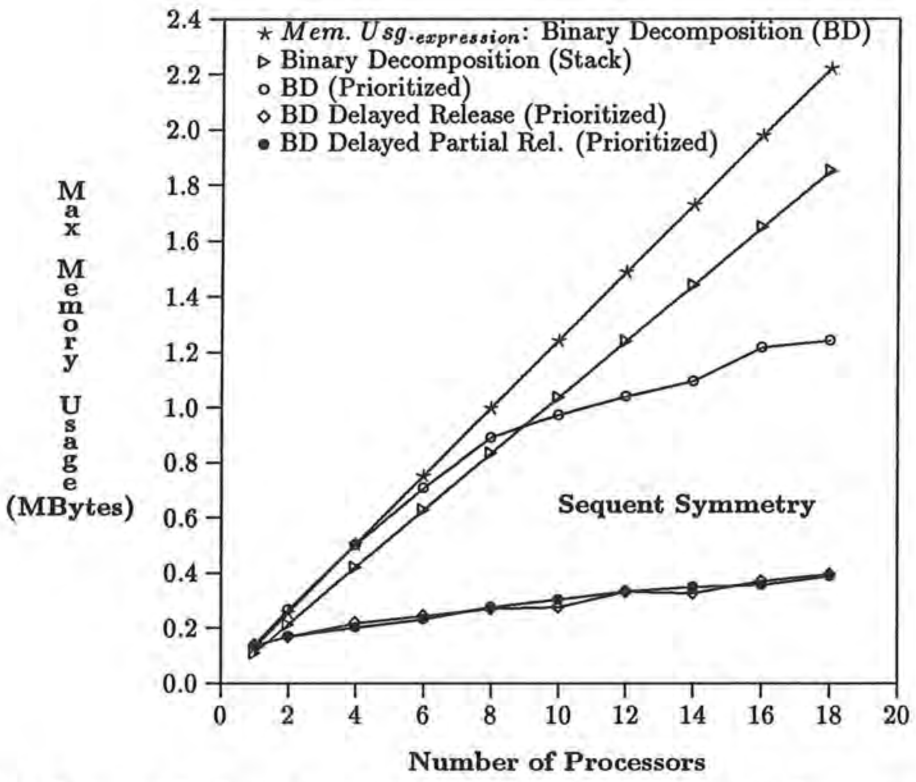

Figure 9: 126-Queens: Maximum memory usage to the first solution on Sequent Symmetry

Table II: 126-Queens: Performance to the first solution with increasing branching factor.

126-Queens: First Solution Speedup						
PEs	1	2	4	6	8	10
Branching Factor=2	0.99	1.99	3.93	5.82	7.62	9.33
Branching Factor=3	0.99	1.99	3.89	5.41	7.43	9.05
Branching Factor=4	0.99	1.97	3.85	5.61	7.28	8.10

126-Queens: Maximum Queue Lengths (Msgs.)						
PEs	1	2	4	6	8	10
Branching Factor=2	108	215	417	613	755	819
Branching Factor=3	213	416	801	1154	1413	1571
Branching Factor=4	317	714	1213	1797	2176	2397

126-Queens: Maximum Memory Usage (MBytes)						
PEs	1	2	4	6	8	10
Branching Factor=2	0.138	0.266	0.501	0.705	0.890	0.972
Branching Factor=3	0.238	0.498	0.907	1.278	1.587	1.737
Branching Factor=4	0.344	0.714	1.318	1.938	2.343	2.580

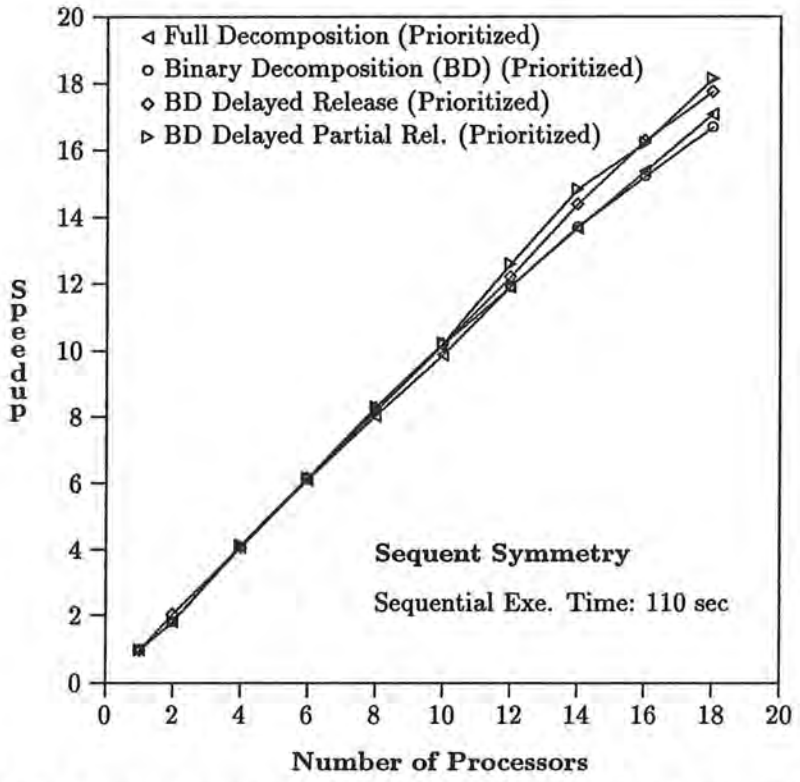

Figure 10: 8 × 8 Knights-Tour: Speedups to the first solution on Sequent Symmetry.

Table III: 8 × 8 Knights-Tour: Maximum queue lengths to the 1st solution.

Knights-Tour: Max. Queue Size (# Msgs.) on Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
FD	78	138	226	285	298	400	405	403	395	471
BD	35	57	88	108	126	149	175	181	197	217
BD-DR	34	36	39	43	49	59	51	46	47	47
BD-DPR	7	13	20	20	26	21	27	30	35	37

FD: Full Decomposition BD: Binary Decomposition

BD-DR: BD Delayed Release BD-DPR: BD Delayed Partial Release

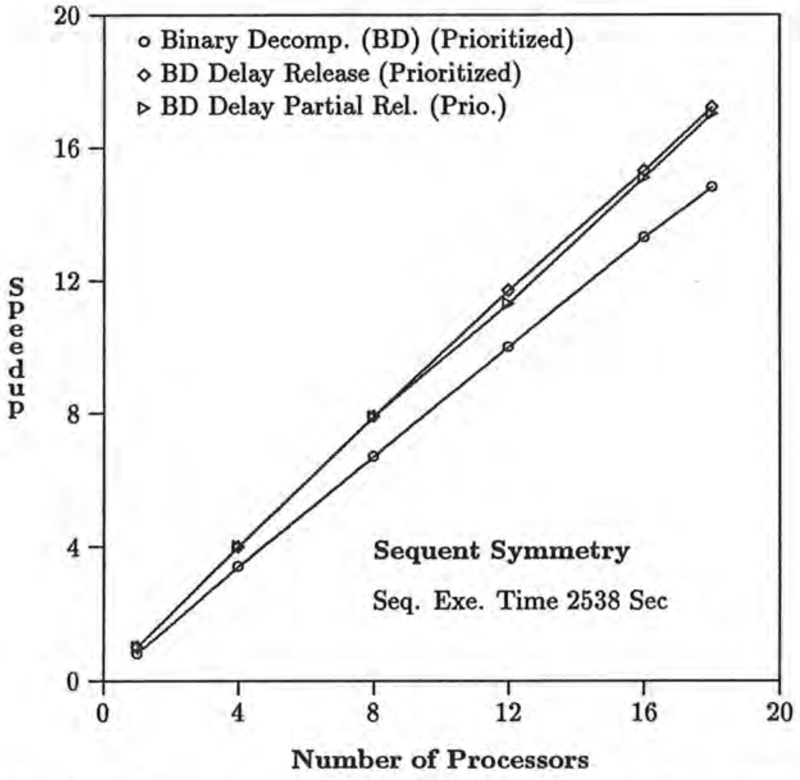

Figure 11: 6 × 6 Magic-Squares: Speedups to the first solution on Sequent Symmetry.

Table IV: 6 × 6 Magic Squares: Queue lengths and memory usage to the 1st solution.

Max. Que. Lengths (Msgs.) on Sequent Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
BD	17	35	50	68	74	68	74	84	99	143
BD-DR	15	17	19	22	25	24	34	38	48	54
BD-DPR	4	9	15	18	20	24	28	39	41	42

Maximum Memory Usage (KBytes) on Sequent Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
BD	13.0	27.6	38.0	52.6	58.8	59.2	64.1	72.7	84.8	114.2
BD-DR	12.3	15.9	26.6	38.5	43.8	43.9	52.8	64.9	70.0	79.6
BD-DPR	11.7	16.0	26.0	34.8	36.8	48.7	55.2	65.8	71.2	78.6

BD: Binary Decomposition (Prioritized)

BD-DR: BD Delayed Release (Prioritized)   BD-DPR: BD Delayed Partial Release (Prioritized)

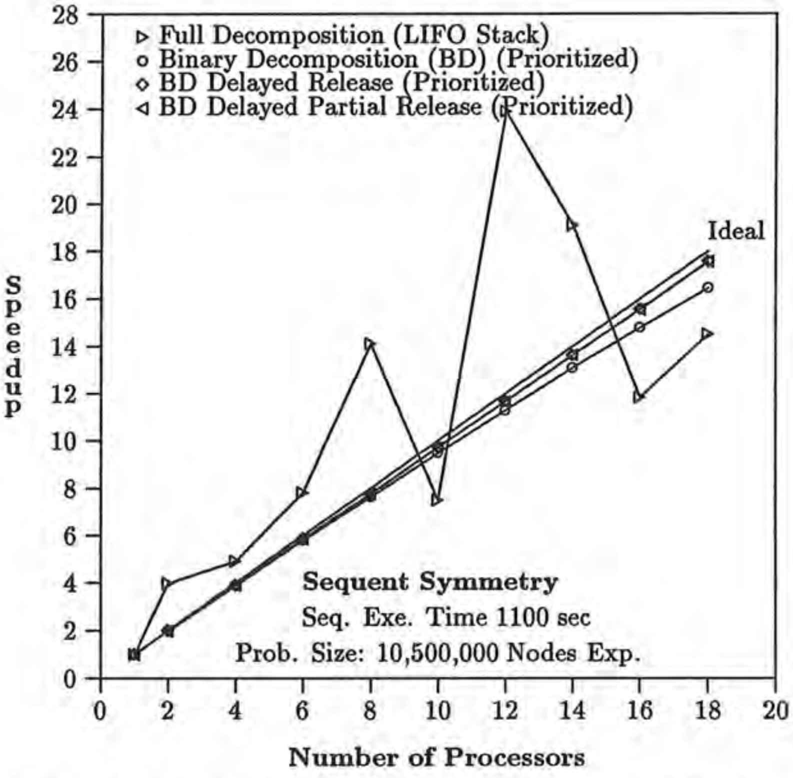

Figure 12: Parallel IDA*: Speedups to the first optimal solution of the 15-Puzzle problem.

Table V: Parallel IDA*: Queue lengths and memory usage to the 1st optimal solution of the 15-Puzzle problem.

Max. Que. Lengths (Msgs.) on Sequent Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
FD-LIFO	21	22	21	36	36	55	52	67	55	73
BD	26	27	31	42	43	45	48	53	59	64
BD-DR	26	27	33	36	38	42	44	48	55	55
BD-DPR	14	19	28	33	35	45	46	48	55	56

Maximum Memory Usage (MBytes) on Sequent Symmetry										
PEs	1	2	4	6	8	10	12	14	16	18
FD-LIFO	0.116	0.121	0.116	0.198	0.198	0.305	0.287	0.370	0.303	0.403
BD	0.144	0.170	0.203	0.216	0.229	0.245	0.250	0.258	0.258	0.267
BD-DR	0.143	0.174	0.196	0.218	0.228	0.243	0.249	0.257	0.265	0.277
BD-DPR	0.143	0.167	0.201	0.235	0.247	0.253	0.247	0.253	0.259	0.270

FD-LIFO: Full Decomposition with Stack   BD: Binary Decomposition

BD-DR: BD Delayed Release   BD-DPR: BD Delayed Partial Release

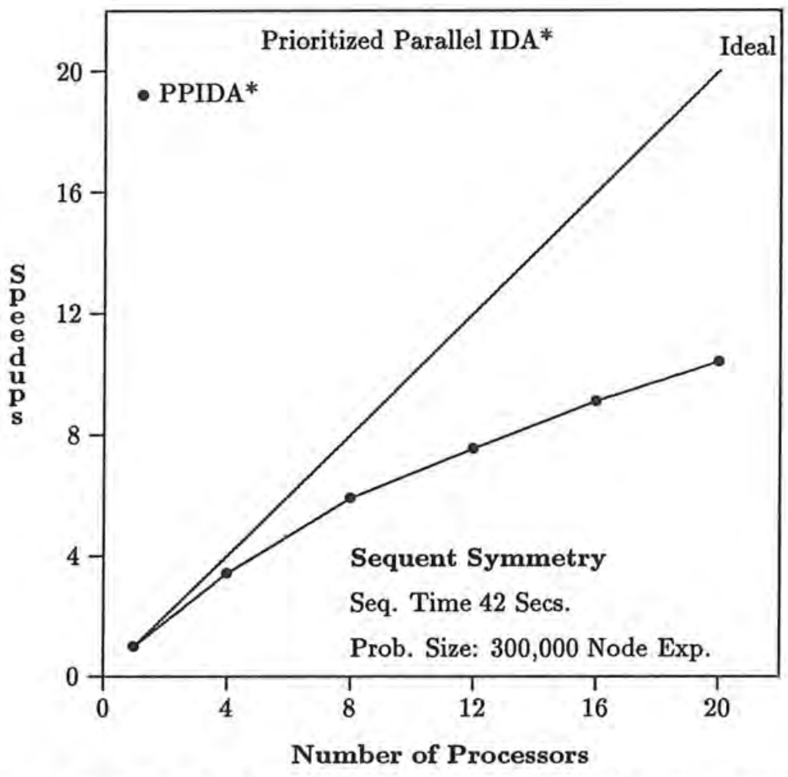

Figure 13: PPIDA*: Speedups to 1st optimal solution of the 15-Puzzle. Problem size: Small

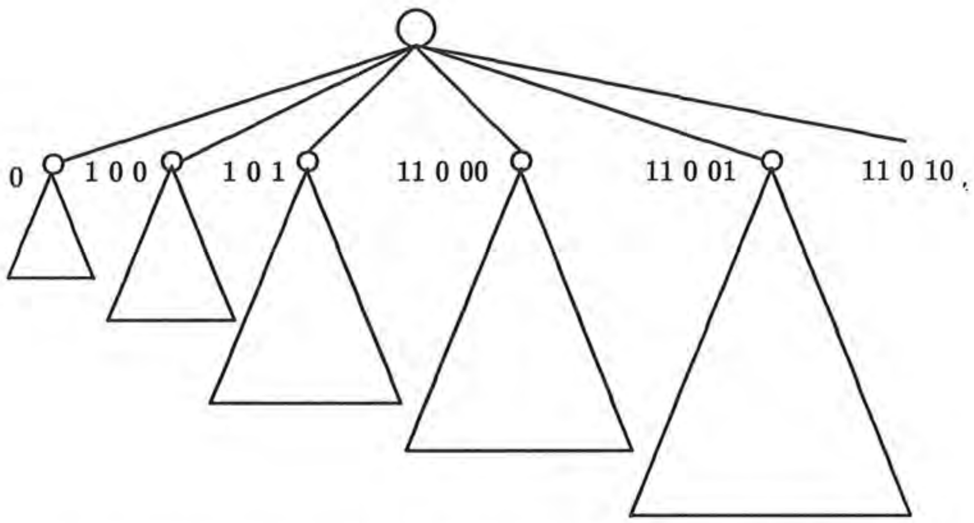

Figure 14: Concurrent execution of prioritized iterations in PIO-PIDA*.

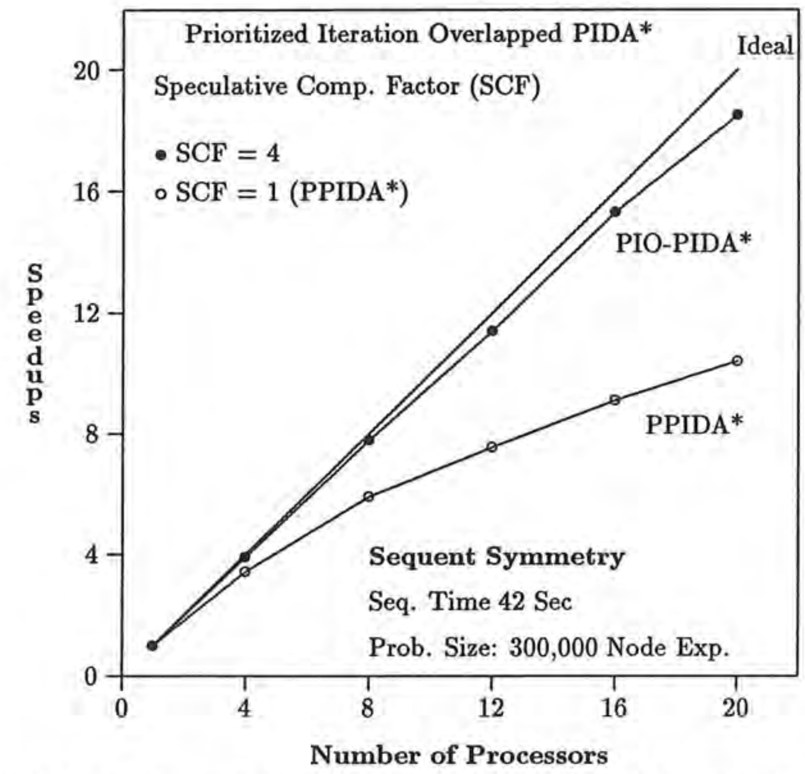

Figure 15: PIO-PIDA*: Speedups to 1st optimal solution of the 15-Puzzle. Problem size: Small

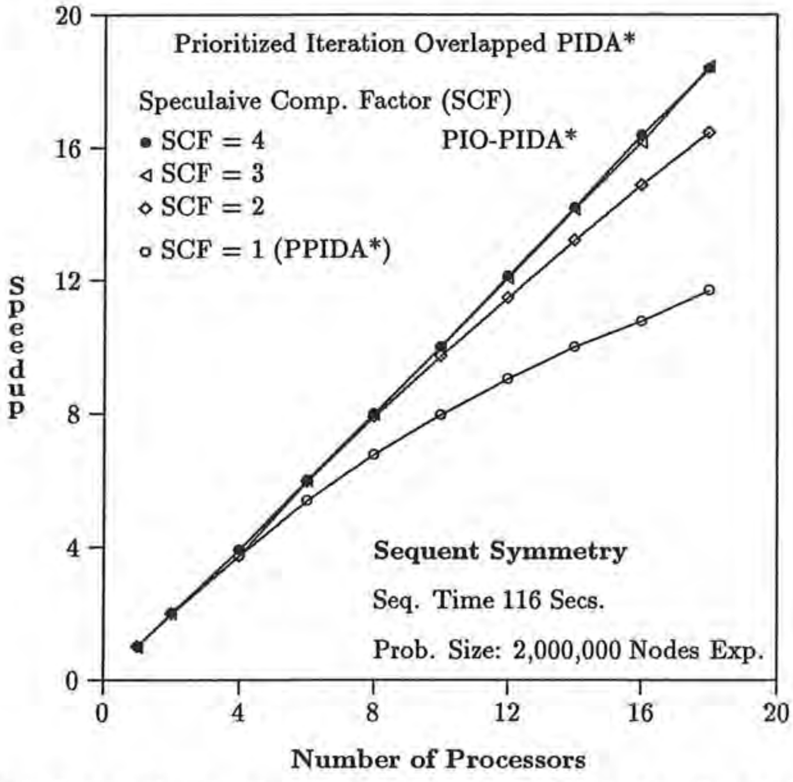

Figure 16: PIO-PIDA*: Speedups to 1st optimal solution of the 15-Puzzle. Problem size: Large