Published in the Proc. of the Fifth IEEE Parallel Processing Symposium, Apr. 1993, pp.797-803.

Parallel A* Algorithms and their Performance on Hypercube Multiprocessors*

Shantanu Dutt, MEMBER IEEE and **Nihar R. Mahapatra**, STUDENT MEMBER IEEE
Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455

Abstract

In this paper we develop parallel A* algorithms suitable for distributed-memory machines. In parallel A* algorithms, inefficiencies grow with the number of processors P used, causing performance to drop significantly at lower and intermediate work densities (the ratio of the problem size to P). To alleviate this effect, we propose a novel parallel startup phase and efficient dynamic work distribution strategies, and thus improve the scalability of parallel A* search. We also tackle the problem of duplicate searching by different processors, by using work transfer as a means to partial duplicate pruning. The parallel startup scheme proposed requires only $\Theta(\log P)$ time compared to $\Theta(P)$ time for sequential startup methods used in the past. Using the Traveling Salesman Problem (TSP) as our test case, we see that our work distribution strategies yield speedup improvements of more than 30% and 15% at lower and intermediate work densities, respectively, while requiring 20% to 45% less memory, compared to previous approaches. Moreover, our simple duplicate pruning scheme provides an average reduction of 20% in execution time for up to 64 processors, compared to previous approaches that do not prune any duplicates.

Keywords: A* search, branch-and-bound search, duplicate pruning, dynamic work distribution, parallel A*, scalability, startup work distribution.

1 Introduction

The A* algorithm [9] is a well-known, generalized branch-and-bound search procedure, widely used in the solution of many computationally demanding combinatorial optimization problems [1, 8]. Its operation, as detailed later, can be viewed essentially as a best-first search of a state space graph. Parallelization of branch-and-bound methods provides an effective means to meet the computational needs of many practical search problems [3, 8, 10].

Many researchers in the past have adopted a tree formulation of the search space in problems such as the Traveling Salesman Problem (TSP) [3, 7], the 15-puzzle problem [3] and test pattern generation [1, 8], where a graph formulation is applicable. A tree formulation makes A* amenable to easy parallelization; each processor explores a different part of the search space, that is disjoint from the search spaces of other processors. However, for the above problems, using a tree formulation implies that nodes that represent identical subproblems are not considered duplicates since their states are artificially defined to be different. When such identical subproblems arise during A* search, they will remain undetected, thus resulting in a duplication of search effort. A graph formulation on the other hand, will enable detection of such nodes and hence duplicated search can be avoided. However, it is not easily amenable to parallelization on distributed-memory machines, since duplicate nodes may be generated in different processors.

In this paper we develop parallel A* algorithms for searching arbitrary directed graphs and suitable for distributed-memory machines, with TSP as our test problem. In parallel A* algorithms, inefficiencies grow with the number of processors P used, causing performance to drop significantly at lower and intermediate work densities (the ratio of the problem size to P). To alleviate this effect, we propose a novel parallel startup phase and efficient dynamic work distribution strategies and thus improve the scalability of parallel A* algorithms. Our new parallel startup scheme requires only $\Theta(\log P)$ time compared to $\Theta(P)$ time for sequential startup methods used previously. To tackle the problem of duplicate searching discussed earlier we present an indirect but reasonably effective method of pruning duplicates via work transfer. Explicit schemes for inter-processor duplicate pruning are currently being investigated. We also present a $\Theta(\log P)$ termination detection scheme for our parallel A* algorithms.

2 The A* Algorithm

In this section we briefly describe the sequential A* algorithm [9]. Given a combinatorial optimization problem $\mathcal{P}$, we are interested in finding a least cost solution to $\mathcal{P}$. The A* algorithm operates by performing a best-first search for an optimal solution node from a *root* node in a state space graph $\mathcal{G}$. The *root* node corresponds to the original problem $\mathcal{P}$, while all other nodes represent subproblems derived from $\mathcal{P}$. Associated with each node u is a cost g of the current best path from *root* to u and a cost h' that is a lower bound on the cost of the best path from u to any solution node. Nodes representing the same problem state are considered to be *duplicates*.

The *cost estimate* of a node is $f' = g + h'$, i.e., it represents a lower bound on the cost of the best solu-

*This research was funded in part by a Grant-in-Aid and in part by a Faculty Summer Research Fellowship both from the University of Minnesota Graduate School. Sandia National Labs provided access to their 1024-processor nCUBE2 parallel computer.

tion node reachable from this node. Two lists of nodes are utilized: *OPEN* and *CLOSED*. *OPEN* is a list of nodes that have been generated but not yet *expanded* (not had their children generated), while *CLOSED* is a list of all nodes that have been expanded. The actual operation of the algorithm proceeds in steps. At each step, it picks a node from *OPEN* with the minimum f' value called *best_node*, generates its children and computes their cost estimates. Next, it adds each generated child to the *OPEN* list, after checking to see if any duplicates exist, in which case all except the best copy are removed. Finally *best_node* is placed in *CLOSED*. This process is repeated until *best_node* happens to be a solution node, which is then returned as an optimal solution. This sequential A* algorithm will be referred to as SEQ_A*.

Now we define a few terms used in the remainder of the paper. At any time during the operation of A*, the f' value of the current best solution node is denoted by *best_soln*. We use two classifications of nodes based on *best_soln* and the optimal solution cost. Nodes in *OPEN* that have f' values less than the current value of *best_soln* are called *active* nodes, since these are the only nodes that need to be expanded, and *active_len* represents the number of such nodes at any time. Other nodes in *OPEN* are said to be *inactive*. Nodes that have cost less than the optimal solution cost are called *essential* nodes, since they form a necessary and sufficient set of nodes that a sequential algorithm needs to expand in order to find an optimal solution. All other nodes are called *non-essential* nodes.

In SEQ_A*, when a node is expanded, i.e., it is an essential node, all its children are generated irrespective of their costs. This way a large number of non-essential nodes may be generated. In our implementations we have used an improved A* algorithm, that employs a *partial expansion* scheme to generate only the best available child from the node undergoing expansion [6]. This way very few non-essential nodes are formed. We will refer to this improved sequential A* algorithm as SEL_SEQ_A*.

3 Application of A* to TSP

The Traveling Salesman Problem can be posed as follows. Given a set $0, 1, .., N-1$ of cities and intercity distances, find the shortest tour that visits every city exactly once and returns to the start city. We have formulated TSP so that it is equivalent to a search in a state space graph $\mathcal{G}$, a problem domain most readily handled by A*. The state of a node is defined by a 3-tuple: [*start city, set of visited cities, present city*]. Initially only *root* with $root.state = [0, \{0\}, 0]$, exists. An expansion of a node $u \in \mathcal{G}$ yields a child v for each city that remains to be visited in u. This way a TSP tour is constructed by visiting an additional unvisited city, from the *present city*, in each expansion. The heuristic function used by us is similar to the one given in [5].

While our formulation of TSP reduces it to a graph-search problem (as will be shown shortly), previous sequential and parallel branch-and-bound methods employed to address TSP [3, 5, 7, 10] have used a tree search formulation. In their formulation, the state of a node is defined by either: (1) a 1-tuple: [ordered list of cities visited] or (2) a 2-tuple: [set of edges currently in the tour, set of edges excluded from the tour]. Consider two nodes u and v, whose states, according to the first tree formulation, are defined by $u.state = [(0,1,2,3,4)]$ and $v.state = [(0,2,3,1,4)]$; and according to the second tree formulation by $u.state = [\{(0,1),(1,2),(2,3),(3,4)\}, \emptyset]$ and $v.state = [\{(0,2),(2,3),(3,1),(1,4)\}, \{(0,1)\}]$ (where (i,j) denotes the edge from city i to city j). Thus, u and v would be considered to represent two different subproblems (since their states are different) according to these formulations. However, both u and v represent the same subproblem, which can be posed as: find a minimal cost path visiting exactly once all cities in the set $\{4, 5, \ldots, N-1, 0\}$, originating at 4 and terminating at 0. This statement is precisely captured by our formulation, according to which the states of both u and v are given by: $[0, \{0,1,2,3,4\}, 4]$. Duplicates are detected by the duplication check carried out in every iteration of A*. As a result, one of u and v gets pruned. It can similarly be shown that a graph search formulation is more appropriate for problems such as the 15-puzzle and test pattern generation, that have previously been tackled using a tree search formulation [1, 3, 8].

4 Parallelization of A*

Here we present a brief overview of our approach to the parallelization of A* and introduce a few terms used in the sequel. We have developed three parallel A* algorithms targeted at the hypercube architecture. Our parallel A* algorithms can be described as *parallel local A** (PLA*) algorithms, since each processor executes an almost independent SEL_SEQ_A* on its own *OPEN* and *CLOSED* lists. The starting nodes required for a processor's sequential algorithm are generated and allocated in a *startup phase*. Processors broadcast any improvements in *best_soln*, which is maintained consistent in all processors. Apart from solution broadcasts, processors interact to redistribute work for better processor utilization, to send or receive cost updates and to detect termination of the algorithm; algorithms for these tasks are presented in subsequent sections. We define *non-essential work* (*essential work*) to be the total time over all processors spent in processing non-essential (resp. essential) nodes. Let T_P denote the execution time on P processors and T_1 the sequential execution time. We use *work density* to refer to the ratio T_1/P. By *duplicated work* we mean the total extra time over all processors associated with pursuing duplicate search spaces. The total time over all processors spent in idling will be referred to as *starvation time*.

In contrast to a parallel local A* algorithm, a *parallel global A** (PGA*) algorithm uses global lists or lists that are consistent across processors. Such an algorithm is suitable only for shared-memory machines [3] and does not scale up well with the number of processors, since contention for global lists or the cost of maintaining consistent lists becomes excessive. Parallel local A* schemes on the other hand, can be implemented on distributed-memory machines and have better scalability. However, the use of multiple incon-

sistent *OPEN* lists and a distributed-memory implementation, introduce a number of inefficiencies in a parallel local A* algorithm:

1. *Starvation*: This occurs when processors run out of work and idle.
2. *Non-essential work*: Since the nodes expanded in each processor do not necessarily represent a global best selection, non-essential work may be performed.
3. *Duplicated work*: This is due to duplicate nodes that arise in different processors (*inter-processor duplicates*).
4. *Memory overhead*: This is caused by non-essential nodes and duplicate nodes formed. Work transfer also adds to the memory requirement, since both nodes that are granted and nodes that are received must be stored (in the donor and acceptor processors respectively) in a graph search algorithm to detect any possible duplicates that may arise later.

Later we present schemes that deal with the above inefficiencies very effectively.

5 Parallel Startup Phase

Here we describe and analyze a novel parallel startup phase that is used in all of our parallel algorithms. The structure of the startup phase can be represented as a b-ary tree of depth d. This is illustrated in Fig. 1 for the case $b = 2$ and $P = 4$. Each vertex of the tree in Fig. 1 corresponds to a node generation phase and the outgoing edges from a vertex correspond to a node distribution phase (these will be described shortly). Each leaf of the tree corresponds to the nodes finally allocated to a single processor. The startup phase execution pattern for each processor is described by a unique path from the root of the tree to a corresponding leaf. Initially all processors start with the same *root* node (node $a1$ in Fig. 1). Then each processor asynchronously executes SEL_SEQ_A* until it has obtained bm ($m = 2$ in Fig. 1) active nodes, where m is the *multiplicity* of each branch. This is the *node generation phase*. Next in the *node distribution* phase, the P processors are divided into b groups, with each group being assigned the same number m of nodes in *OPEN*. The assignment is done so as to effect an equitable distribution of nodes among the processor groups[1]. This sequence of node generation and distribution alternates, until each processor has obtained its own m nodes. Subsequently, processors execute SEL_SEQ_A* on these starting nodes.

Let us call the combination of a node generation phase followed by a node distribution phase, a *step*. Then in each step, bm distinct active nodes are generated, and m of them distributed to each of the b processor groups. Note that each processor executes a total of $\lceil \log_b P \rceil$ steps. Therefore assuming that no more than $\Theta(b.m)$ duplicates are encountered in any step, the startup phase time T_{su} becomes:

$T_{su} = \Theta(mb\lceil \log_b P \rceil) = \Theta(mb\lceil \frac{\log_2 P}{\log_2 b} \rceil)$

Thus T_{su} increases linearly with m and relatively more slowly with b and P. In fact, we see that for the parallel startup phase with constant values of b and m, T_{su} grows only as $\Theta(\log P)$, while for a sequential startup phase ($b = P$, $m = 1$) used in previous work [4], T_{su} grows as $\Theta(P)$.

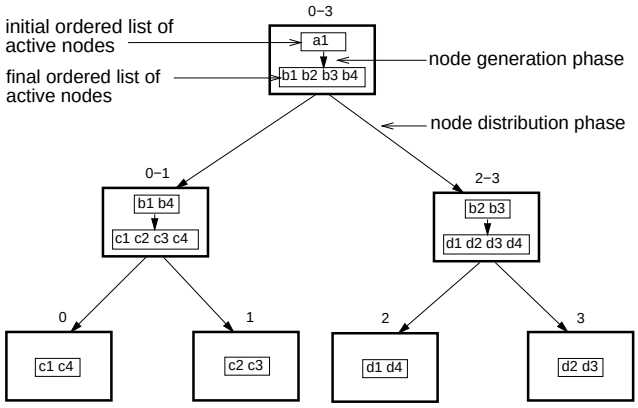

Figure 1: Structure of the startup phase for $b = 2$, $m = 2$ and $P = 4$.

We first define a few terms that will be useful in our subsequent discussions. By the *quality* of a node we mean the amount of essential work associated with the node as reflected by its cost (see footnote 1). By *uniqueness* of nodes across processors we mean the extent to which different processors are free of duplicates. We use the term *quantity* of work (nodes) to mean the number of active nodes. A good distribution of nodes across processors is one in which different processors have unique nodes, as well as almost equal quantity and quality of nodes. The startup phase parameters b and m determine the initial distribution of nodes across processors in terms of quality and uniqueness[2], as explained below.

In each distribution phase, some of the processor groups receive costlier nodes compared to the others, and the search graph gets partitioned into parts that are not necessarily distinct. Hence smaller the branching factor b, more are the number of distribution phases and worse is the quality and uniqueness distribution of starting nodes across processors. On the other hand, a larger value for m means a larger choice of nodes to expand from in the node generation phase, and hence better quality of nodes for the subsequent distribution phase. We will analyze the effect of parameters b and m on the performance of parallel A* at different work densities in Sec. 8.

6 Work Distribution

In the interest of scalability, we have employed nearest-neighbor work distribution strategies in which both work requests and work transfers are confined

[1] The distribution is made equitable in terms of the amount of work the nodes represent. A less expensive node is likely to generate more essential nodes compared to a costlier node with a comparable number of visited cities. Therefore, the amount of work a node represents can be approximately deduced from its cost.

[2] Since each processor obtains the same number of starting nodes, a good quantity distribution is effected by the startup phase.

to neighbors. First we describe two commonly used work distribution strategies for distributed-memory machines proposed in previous work, viz., the *round-robin* (RR) strategy [4] and the *random communication* (RC) strategy [2, 3]. We then present our new work distribution strategy, which we call *quality equalizing* (QE) strategy, because of its use of a highly effective scheme for balancing the quality of work between neighbors.

6.1 Previous Strategies: In the round-robin strategy, a processor that runs out of nodes requests work from its busy neighbors in a round-robin fashion, until it is successful in procuring work. The *donor* processor grants a fixed fraction (one third in our implementation) of its active nodes to the *acceptor* processor.

In the random communication strategy, each processor donates the newly generated children of the node expanded in each iteration, to random neighbors. This way a more uniform distribution of good quality nodes is achieved and is therefore helpful in reducing the amount of non-essential work.

The above strategies do not adequately address the inefficiencies identified in Sec. 4. For instance, the round-robin strategy strategy does not use any scheme to directly reduce non-essential work. The random communication strategy, on the other hand, employs an expensive and to some extent redundant approach of frequent node transfers to tackle this problem. The deficiencies in these strategies will become clear in our following discussion of the quality equalizing strategy.

6.2 Quality Equalizing Strategy (QE): The quality equalizing strategy comprises a combination of schemes that address the inefficiencies identified in Sec 4. We next describe each of these schemes in detail.

6.2.1 Anticipatory Work Request: The first scheme reduces the idling due to latency between work request and work procurement by having processors request work on imminent starvation[3]. Since at any time the least cost node in a processor is expanded, any decrease in $active\_len$ implies that the best nodes available are not good enough to generate active nodes and hence this decrease is likely to continue. In our scheme, processors start requesting nodes when $active\_len$ is below a certain threshold, the *acceptor threshold*, and it is decreasing. It is found that this prediction rule works very well in practice. Using such a look-ahead approach, we are able to overlap communication and computation. Moreover, the delay due to transfer of a long message can be reduced by pipelining the message transfer, i.e., by sending the work in batches.

6.2.2 Quantitative Load Balancing: This scheme addresses the problem of starvation. In this scheme, each processor monitors its $active\_len$ periodically and reports any significant changes in it to its neighbors. This way at any time each processor knows the quantity of work available with each of its neighbors. Also, each processor assumes that the processor space comprises its neighbors and itself only. Let w_i denote the amount of work available with processor i, and $W_{avg,i}$ the average amount of work per processor available with i and its neighbors. Let $\delta_i^j = w_j - W_{avg,i}$ denote the surplus amount of work at a neighboring processor j with respect to $W_{avg,i}$. To achieve perfect quantitative load balance between i and its neighbors, each processor should have $W_{avg,i}$ amount of work. This means that each neighbor j of i should contribute δ_i^j units of work to i, which is the common pool. A negative value for δ_i^j implies a deficiency, and in that case, j will collect $-\delta_i^j$ units of work from i, instead of contributing. Similarly, if we look at the work transfer problem from the perspective of a neighboring processor j of i, then to achieve perfect load balance between j and its neighbors, processor i should collect $-\delta_j^i$ units of work from j.

In our scheme, if a processor i runs out of work, then it requests work from the neighbor i_1 that has the maximum amount of work. A request for work from i to i_1 carries the information $\delta_i^{i_1}$, and the amount of work granted is $min(\delta_i^{i_1}, -\delta_{i_1}^i)$. The minimum of the two is taken because we do not want to transfer any extra work that may cause a work transfer in the opposite direction at a later time.

6.2.3 Qualitative Load Balancing: The next objective is to minimize the total amount of non-essential work. Note that to achieve this, it is not necessary to follow a global selection rule for expansion as has been attempted earlier in [10]. Instead we need only ensure that all processors work on essential nodes throughout the duration of the algorithm. Our scheme is based on the idea that any processor spends a reasonable period of time expanding and processing its best few nodes before it moves onto costlier nodes. Therefore as long as the best few nodes with each processor are good (ideally essential) our objective will be fulfilled. To achieve this, we use a scheme in which each processor monitors the cost of its fifth best node in *OPEN* periodically and reports any significant changes to its neighbors. In this manner, every processor at any time has information regarding the cost of the fifth best node in the *OPEN* list of each of its neighbors. A processor requests work from the neighbor with the least cost fifth best node, when the cost of its best node is more than the least cost fifth best node in any of its neighbors. The donor processor grants only a few good nodes to the acceptor. This way we ensure that neighboring processors, and eventually all processors, work on nodes that are qualitatively comparable. As a result, the amount of non-essential work gets reduced. Note that since only a few nodes are transferred, this scheme has very low work transfer overhead, and therefore is especially useful at low and intermediate work densities. In our subsequent discussions we will refer to work requests meant to effect quantitative load balance (such as in the previous scheme) as *quantitative work requests*, while those related to qualitative load balance (such as in the present scheme) as *qualitative work requests*.

6.2.4 Duplicate Pruning: In our parallel algo-

[3] This latency may be caused primarily by a lack of work with the neighbors, or if there is work, then by the neighbors being busy. Furthermore, the message transfer time might be high because of a long message.

rithms we prune all intra-processor duplicates using the duplication check test in SEL_SEQ_A*. We use an indirect method for pruning inter-processor duplicates via work transfer as follows. Consider two processors P_1 and P_2 that possess duplicate nodes u and v, respectively. In the absence of any inter-processor duplicate pruning scheme, the search space from both u and v will be explored thus contributing to duplicated work. In our scheme, if u, which can possibly be a partially expanded node, is donated by P_1 to P_2, then u is placed in the *CLOSED* list of P_1 and no further expansions are carried out from u. Furthermore, the child nodes that were already formed from u in P_1 are not permitted to be formed from u in P_2, unless they were already generated from an existing duplicate of u (v in this case) in P_2. Thus the amount of inter-processor duplicate pruning achieved corresponds to the search paths that were generated from u in P_1 but were not generated from v in P_2. Note, however, that the pruning is not complete since at the time of work transfer search paths that have been generated from u in P_1 and from v in P_2 remain duplicated in both the processors. This simple inter-processor duplicate pruning scheme is used in all the parallel algorithms, viz., PLA*-RR, PLA*-RC and PLA*-QE.

A formal description of the quality equalizing strategy that comprises all the schemes discussed in this subsection (6.2) is given in Fig. 2.

7 Termination Detection

In an A* algorithm, whether sequential or parallel, a termination condition is reached when there are no more active nodes to process. In PLA*, active nodes are either with a processor or are extraneously present in *active messages* that are potential sources of active nodes for the receiving processor. In our parallel algorithms, work transfers and cost updates are the two types of active messages—the former may carry active nodes, while the latter may cause existing inactive nodes to become active. An active message originating at processor i and destined for processor j is said to be "owned" by i until an acknowledgement is received from j. A processor "stops" when it has neither active nodes to process nor owns any active messages. A stopped processor "resumes" when it receives an active message that becomes a source of active nodes. The purpose of the acknowledge signal is to allow the acceptor processor to resume before the donor processor can stop. Therefore to correctly detect termination, i.e., to ensure that there are no more active nodes or unacknowledged active messages, we need to only ascertain that all processors have stopped. For this purpose, a spanning tree of depth $\log P$ rooted at processor 0, is mapped onto the hypercube. STOP messages are passed upward in the spanning tree starting at stopped leaf processors. Non-leaf processors send a STOP message upward only after they have stopped and have received STOP messages from all their child processors. Thus in $\Theta(\log P)$ time the root processor receives all STOP messages and determines that a termination barrier has been reached by all processors. Subsequently, processor 0 signals termination to all processors.

Procedure QLTY_EQUALIZING_STRATEGY(i)
/* Procedure QLTY_EQUALIZING_STRATEGY is used in PLA*-QE to achieve load balance, and i is the processor that executes it */
begin

Processor i executes the following steps:

1. Work status report: Periodically monitor $active\_len$ and the cost of the fifth least cost node, and report any significant changes (10% and 2% respectively) in them to all neighbors.
2. **If** (a work status report is received from a neighbor) **then** record it.
3. Let j_max := neighbor with the maximum $active\_len$ value; and j_best := neighbor with the best fifth least cost node.
4. Work request:
   if (no previous work request from i remains to be serviced) **then begin**
   if($active\_len = 0$) **or** ($active\_len < 5$ and is decreasing)
   Send a quantitative work request to j_max, along with the information $\delta_i^{j\_max}$;
   else if ($best\_node$ is costlier than the fifth least cost node in j_best)
   Send a qualitative work request to j_best, along with the cost of $best\_node$.
   endif
5. **If** (a quantitative work request is received from neighbor j) **then** grant $min(\delta_j^i, -\delta_i^j)$ (but at least 10% and not more than 50% of $active\_len$) active nodes in a pipelined fashion.
6. **If** (a qualitative work request is received from neighbor j) **then** grant at most 2 active nodes that are cheaper than j's $best\_node$.
7. **If** (work is received) **then** check for duplicates of nodes received;
   If (no duplicates are found) **then** insert in $myOPEN$;
   else perform appropriate duplicate pruning and propagate cost improvements if any.

end /* Procedure QLTY_EQUALIZING_STRATEGY */

Figure 2: Algorithm for the Quality Equalizing Strategy

Two additional signals RESUME and ACKNOWLEDGE are used to signal a resume caused by an active message, and to acknowledge the receipt of an active message, respectively. If an active message originating at processor i causes j to resume, then j sends a RESUME signal upward in the spanning tree. The RESUME signal is sent to nullify a STOP signal previously transmitted along this path from j[4]. If the RESUME is no longer needed to be transmitted upward at the ancestor processor k of j, then k signals an ACKNOWLEDGE to i. On receiving the ACKNOWLEDGE signal, i "relinquishes" ownership of the active message originally sent to j. Now processor i can stop if it has neither active nodes nor owns any active messages.

8 Performance Results

Algorithm PLA* utilizes a parallel startup phase, either one of the three work distribution strategies discussed earlier, viz., RR, RC and QE strategies, the duplicate pruning scheme of Sec. 6.2.4, and the termination detection algorithm. The three versions of PLA* employing the different work distribution strategies are called PLA*-RR, PLA*-RC and PLA*-QE. We implemented our parallel algorithms on an nCUBE2 hypercube multicomputer to solve TSP and averaged all data over 25 random samples. Four merits of performance are used: (1) Execution time measured in terms of the number of clock ticks on the nCUBE2. (2) Speedup defined as the ratio T_1/T_P. (3) Memory utilization factor (MUF) defined to be the ratio of the memory required by the parallel algorithm to that required by the sequential algorithm. (4) Isoefficiency function, which is the required rate of growth of T_1 with respect to P, to keep the efficiency fixed at some value, and is a measure of the scalability of the algorithm [4].

8.1 Effect of the Startup Phase: In Fig. 3, we plot the execution times for various b and m combinations, as a percentage of the execution time of the case $b = P$ and $m = 1$ (sequential startup). The amount of startup phase time T_{su} affects the performance at different work densities in the following ways: (1) At low work densities (roughly $P > 16$ for $N = 19$) the fraction of the time $1 - T_{su}/T_P$ spent in completely parallel execution is small; this can be counterbalanced by decreasing T_{su} and hence smaller values for b and m yield better performance. (2) At intermediate work densities (roughly $4 < P \leq 16$ for $N = 19$) the total time $T_P - T_{su}$ available for load balancing and duplicate pruning is insufficient; this can be alleviated by a good distribution of starting nodes and hence larger (though not necessarily the largest) values of b and m prove to be more useful. (3) Finally, at high work densities (roughly $P \leq 4$ for $N = 19$) $T_P \gg T_{su}$, so that the effect of the above two factors is minimal. Hence the choice of b and m is crucial at low and intermediate work densities.

8.2 Effect of Graph Search Formulation and Duplicate Pruning: Next in Fig. 4 we plot for PLA*-QE the percentage improvement obtained using a graph formulation (duplication check on) over a tree formulation (duplication check off) for uniformly distributed and normally distributed random inputs. Two effects are apparent from these plots: (1) The performance benefit from duplication check gradually tapers off with larger number of processors, i.e., with larger number of partitions of the search graph, until it is no longer useful. This is because with larger number of partitions of the search graph, more duplicates are inter-processor rather than intra-processor, and hence checking for intra-processor duplicates actually becomes a penalty, since there are fewer such duplicates. (2) The performance gain due to duplication check is more pronounced (on an average 20% better for up to 64 processors) when the data distribution is normal and the variance is low than when it is uniformly distributed. When the variance of the data is low, it becomes more likely that many search paths will be equally competitive and hence will be explored to an equal extent; this increases the chances of duplication. In practice, data is most often distributed normally, and thus an explicit duplicate pruning scheme should prove very useful.

8.3 Effect of Work Distribution Strategies–Speedup and Isoefficiency Results: In Fig. 5(a) we plot the average speedup and 100 times the average memory utilization factor versus number of processors for PLA*-RR, PLA*-RC and PLA*-QE. The fact that PLA*-QE performs significantly better than PLA*-RR and PLA*-RC at lower and intermediate work densities corroborates our predictions regarding the utility of the quality equalizing strategy in enhancing scalability—speedups of PLA*-QE for $P = 64$, i.e., at an intermediate work density, and for $P = 256$, i.e., at a lower work density, are about 15% and 15 to 35%, respectively, above the speedups of PLA*-RR and PLA*-RC. From the same figure we also note that PLA*-QE has a very low memory overhead, about 20 to 45% less for $P = 256$, in comparison to PLA*-RR and PLA*-RC. Finally, in Fig. 5(b) we plot the isoefficiency curves for PLA*-RR, PLA*-RC and PLA*-QE. A lower bound on the isoefficiency of any load balancing scheme for the hypercube architecture is $\Omega(P \log P)$ [4]. Although not many data points are available, we notice that the general trend of the isoefficiency function for PLA*-QE is close to the lower bound and is much better than that of PLA*-RR and PLA*-RC.

9 Conclusions

For most parallel search algorithms, it is possible to obtain linear speedups for sufficiently high work densities. At lower and intermediate work densities, inefficiencies such as uneven work distribution, search of non-essential as well as duplicate spaces, and overheads due to communication and increased memory utilization, gain prominence and cause performance to deteriorate. In this paper, we proposed a parallel startup scheme and dynamic work distribution strategies to tackle these problems. Our new parallel startup phase requires only $\Theta(\log P)$ time compared to $\Theta(P)$ time for sequential startup methods used previously. Moreover, we presented efficient work distribution schemes based on a qualitative analysis of the inefficiencies that exist in parallel A* algorithms.

[4]Note that the root processor will not signal termination, since processor i has not yet stopped.

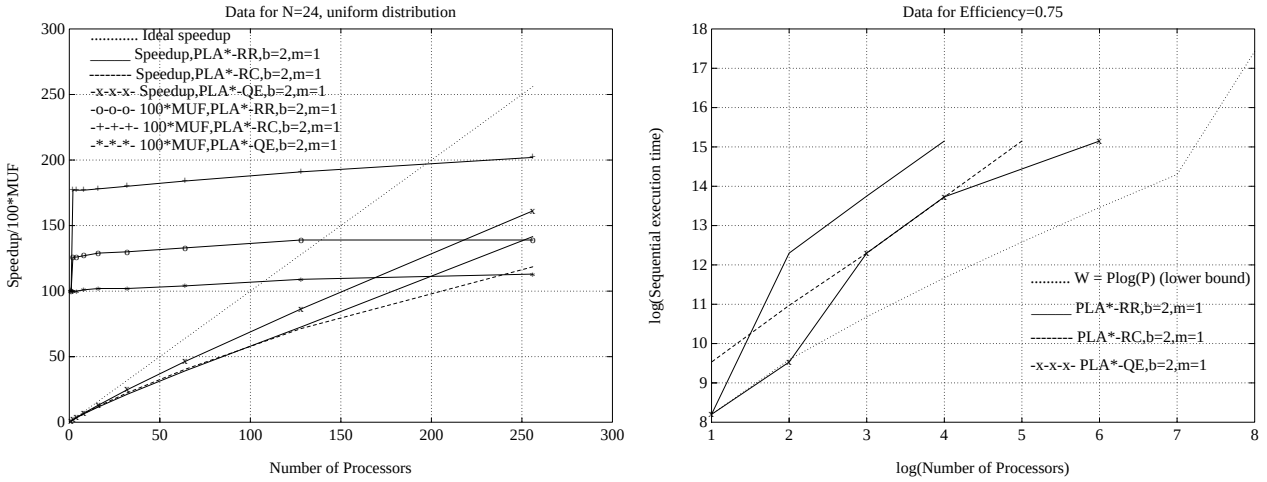

Figure 5: (a) Speedup and (b) Isoefficiency curves for PLA*-RR, PLA*-RC and PLA*-QE.

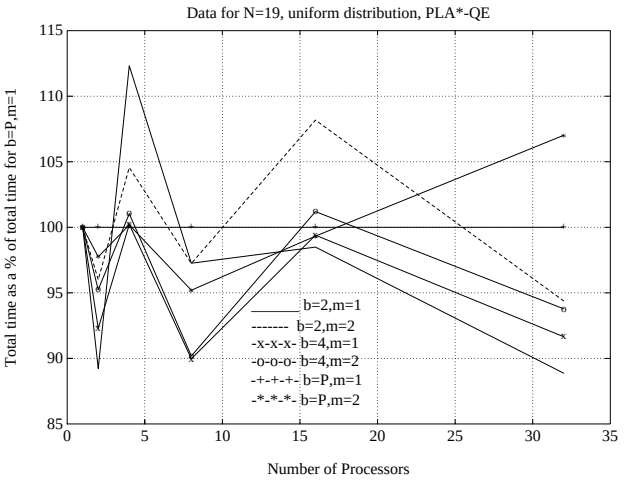

Figure 3: Effect of the parameters b, m and P on the total execution time T_P.

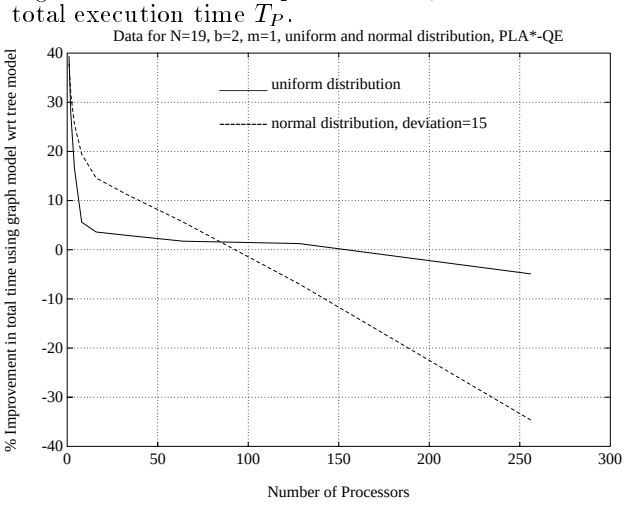

Figure 4: Effect of the duplication check test on the total execution time T_P.

For a fixed problem size, the above strategies enable us to use a larger number of processors more effectively, thus enhancing the scalability of our parallel algorithms. Furthermore, we observed that for search spaces that are actually graphs and not simple trees, duplicate pruning is critical in obtaining good performance. There appears to be a good scope for improving the performance of our parallel A* algorithm further by using explicit duplicate pruning strategies that will be more effective with a larger number of processors; we are currently investigating such pruning strategies. Lastly, we presented an optimal termination detection scheme for our parallel A* algorithm. Performance results of our parallel algorithms reveal the utility of all our schemes and corroborate their analyses. Our schemes should prove very useful in practice, since A* is a generalized branch-and-bound algorithm used to solve a large class of optimization problems.

References

[1] S. Arvindam, V. Kumar and V.N. Rao, "Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD," *Proc. 3rd Symp. Mass. Par. Comp.*, Oct. 1990.

[2] R.M. Karp and Y. Zhang, "A Randomized Parallel Branch-and-Bound Procedure," *J. of the ACM*, 1988.

[3] V. Kumar, K. Ramesh and V.N. Rao, "Parallel Best-First Search of State-Space Graphs: A Summary of Results," *Proc. 1988 Nat'l Conf. Artificial Intell.*, 1988.

[4] V. Kumar and V.N. Rao, "Load Balancing on the Hypercube Architecture," *Proc. Hypercubes, Concurrent Comp., Appli.*, Mar 1989.

[5] J.D. Little, et. al., "An Algorithm for the Traveling Salesman Problem," *Operations Research*, Vol.11, 1963.

[6] N.R. Mahapatra and S. Dutt, "Improvement and Analysis of the A* Algorithm," Technical Report *in*

preparation, Electrical Engineering Dept., Univ. of Minnesota, Minneapolis, MN, 1993.

[7] J. Mohan, "Experience with Two Parallel Programs Solving the Traveling Salesman Problem," *IEEE Conf. Par. Proc'g*, pp.191-193, 1983.

[8] S. Patil and P. Banerjee, "A Parallel Branch and Bound Algorithm for Test Generation," *IEEE Trans. Computer-Aided Design*, Vol.9, pp.313-322, Mar 1990.

[9] E. Rich, *Artificial Intelligence*, McGraw Hill, New York, 1983.

[10] B.W. Wah and Y.W. Ma, "MANIP - A Parallel Computer System For Implementing Branch And Bound Algorithms," *Proc. 8th Annu. Symp. on Comp. Arch.*, pp.239-262, 1982.