Published in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science—Parallel Processing of Discrete Optimization Problems*, Vol. 22, pp. 197-232, 1995.

New Anticipatory Load Balancing Strategies for Parallel A* Algorithms*

Nihar R. Mahapatra and **Shantanu Dutt**
{mahapatra, dutt}@ee.umn.edu
Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455

Abstract

In this paper, we develop load balancing strategies for scalable high-performance parallel A* algorithms suitable for distributed-memory machines. In parallel A* search, inefficiencies such as processor starvation and search of non-essential spaces (search spaces not explored by the sequential algorithm) grow with the number of processors P used, thus restricting its scalability. To alleviate this effect, we propose a novel parallel startup phase and an efficient dynamic load balancing strategy called the quality equalizing (QE) strategy. Our new parallel startup scheme executes optimally in $\Theta(\log P)$ time and, in addition, achieves good initial load balance. The QE strategy employs near-neighbor quantitative and qualitative load balancing schemes to achieve load balance. These schemes utilize anticipatory mechanisms to detect and correct load imbalance before its actual occurrence; such mechanisms are particularly useful at lower work densities (the ratio of the problem size to P) and for lower granularity applications. The QE strategy possesses certain unique load balancing properties that enable it to significantly reduce starvation and non-essential work, and that make its performance robust across applications with different cost distributions for search-space nodes. Consequently, we obtain a highly scalable parallel A* algorithm with an almost-linear speedup. The startup and load balancing schemes were employed in parallel A* algorithms to solve the Traveling Salesman Problem on an nCUBE2 hypercube multicomputer. The QE strategy yields average speedup improvements of about 20-185% and 15-120% at low and intermediate work densities, respectively, over three well-known load balancing methods—the round-robin (RR), the random communication (RC) and the neighborhood averaging (NA) strategies. The average speedup observed on 1024 processors is about 985, representing a very high efficiency of 0.96. We also tested the effect of including an anticipatory qualitative load balancing scheme in the QE strategy and found that it reduces the average execution time by 3.32% and 8.77% on

*This research was funded in part by a Grant-in-Aid from the University of Minnesota and in part by NSF grant MIP-9210049. Sandia National Labs provided access to their 1024-processor nCUBE2 parallel computer.

256 and 512 processors, respectively, at lower work densities. Finally, we present analytical and empirical results on the scalability of parallel A* algorithms in terms of the isoefficiency metric. Our analytical results include (1) a $\Theta(P.\log P)$ lower bound on the isoefficiency function of any parallel A* algorithm, and (2) a general expression for the upper bound on the isoefficiency function of our parallel A* algorithm using the QE strategy on any topology—for the hypercube and 2-D mesh architectures the upper bounds on the isoefficiency function are found to be $\Theta(P.\log^2 P)$ and $\Theta(P.\sqrt{P})$, respectively. Experimental results validate our analysis, and also show that parallel A* search using the QE load balancing strategy has better scalability than when using the RR, RC or NA strategies.

1 Introduction

The A* algorithm [21] is a well-known, generalized branch-and-bound search procedure, widely used in the solution of many computationally demanding combinatorial optimization problems (COPs) [4, 23]. 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].

The aim of our work is to develop scalable high-performance parallel A* algorithms for solving COPs on distributed-memory machines. However, parallelization of A* introduces a number of inefficiencies. (1) First, the time required initially to split the whole search space among all P processors, i.e., the startup phase time, can be a significant fraction of the total execution time at low work densities (the ratio of the problem size to P). Therefore the startup phase needs to be executed efficiently. Also, it is desirable to have a good initial load balance to reduce idling at the beginning of parallel A*. (2) In search algorithms such as A*, the amount of work corresponding to different search subspaces is very difficult to estimate and can vary widely. Hence some form of dynamic, quantitative load balancing is crucial to reducing the idling that would otherwise occur. (3) Finally, processors performing best-first search of their local subspaces in parallel A* may search spaces that a sequential A* algorithm will not explore. This can lead to substantial "non-essential" work. To address this problem, it is imperative to perform dynamic qualitative load balancing so that at all times different processors search spaces that are comparably promising.

In addition to the above inefficiencies, duplicated work among processors can occur when the search space is a graph.[1] This problem can be tackled by using efficient duplicate pruning techniques [9, 17, 18]. However, since the focus of this paper is on load balancing strategies, we will restrict our

[1] We use the term "graph" mainly to denote graphs that are not trees, but sometimes we use it more generally to mean trees as well—this will be clear from the context.

attention to tree search spaces so that performance comparison of parallel A* algorithms employing different load balancing methods reflects the effectiveness of these algorithms in achieving load balance (rather than their efficacy in pruning duplicates). The same load balancing methods can be applied with equal effectiveness, in conjunction with any duplicate pruning techniques, to graph search problems [17] as explained briefly in Sec. 3.

While a number of innovative parallel A* methods have been proposed in past work, they have not adequately addressed the inefficiencies of slow startup and load imbalance. In previous work, a $\Theta(P)$-time sequential startup phase in which a single processor generates the P starting nodes needed for parallel search by all processors was used [13]. Also, no explicit attempts were made to obtain a good initial load balance in prior startup schemes. A number of dynamic load balancing methods for parallel A* have also been previously proposed [1, 2, 10, 11, 12, 13, 15, 20, 22, 24]. We critically analyze the effectiveness of some representative methods in Sec. 4 and point out their drawbacks.

In this paper, we propose a parallel A* algorithm with significantly better speedup and scalability than previous algorithms. Our algorithm incorporates: (1) A $\Theta(\log P)$-time optimal parallel startup scheme that achieves good initial load balance, and (2) An efficient dynamic load balancing method called the quality equalizing (QE) strategy in which processors utilize load information at neighboring processors to effect quantitative and qualitative load balance. Our schemes anticipate and correct load imbalances between neighboring processors before they actually occur. We also identify a number of unique load balancing properties in the QE strategy that enable it to effectively minimize starvation and non-essential work. Part of the work reported here appears in [7].

In Sec. 2, we first describe the A* algorithm and then its application to the Traveling Salesman Problem (TSP), which is the test problem used to determine the efficacy of our parallelization techniques. In Sec. 3, we briefly discuss our approach to the parallelization of A*. In the following section, we critically analyze the effectiveness of past approaches to achieving quantitative and qualitative load balance. In the next two sections we describe in detail the various techniques used in our parallel algorithms. Section 5 contains a description and analysis of our parallel startup scheme. Next in Sec. 6, we present the QE load balancing strategy. In the following section, we analyze the empirical performance of our schemes and compare them to the performance of previous parallel A* methods. Conclusions are in Sec. 9.

2 The A* Algorithm and its Application to TSP

First we briefly describe the sequential A* algorithm [21] and then its application to TSP. Given a COP $\mathcal{P}$, A* is used to find a least-cost solution to $\mathcal{P}$. Its operation can be viewed as a best-first search through a tree $\mathcal{T}$

of subproblems in which the original problem $\mathcal{P}$ occurs at the *root*. The children of a given subproblem are those obtained from it by breaking it up into smaller problems (branching). This branching is such that the optimal solution to an internal subproblem node u is the least expensive among the optimal solutions to the child subproblem nodes of u. The leaves of $\mathcal{T}$ represent solutions to $\mathcal{P}$. To guide its best-first search, A* estimates the "cost" of each generated node. The *cost estimate* f' of a node u is the sum of the *path cost* g from *root* to u, and a *heuristic cost* h' that is a lower bound on the cost of the best path from u to any solution node. Thus f' represents a lower bound on the cost of the best solution node reachable from u. The object of the search is to find a leaf node of minimum cost.

A list of nodes called *OPEN* is utilized to store all nodes that have been generated but not yet *expanded* (not had their children generated). Initially, *OPEN* contains just the *root* node. At each iteration, A* 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 and discards *best_node*.[2] This process is repeated until *best_node* happens to be a solution node, which is then returned as an optimal solution. We will refer to this algorithm as SEQ_A*.

We now define a few terms used in the remainder of the paper. At any time during the operation of SEQ_A*, the f' value of the current best solution node is denoted by *best_soln*. Only nodes with f' values less than the current value of *best_soln* are stored in *OPEN*. Such nodes are called *active* nodes, since these are the only nodes that can lead to an optimal solution, and *active_len* represents the number of such nodes at any time. All other nodes are *inactive* and are discarded. All nodes expanded by SEQ_A* are called *essential nodes*, and these include all nodes with cost less than the optimal solution cost and optimal-solution-cost nodes that lie in front of the optimal solution node in the *OPEN* list;[3] all other nodes are *non-essential*. Thus an active node may not necessarily be essential in which case it will become inactive at some point. An essential node will be active until it is expanded.

In our implementations we have used an improved A* algorithm [16]. In SEQ_A*, when a node is expanded (meaning that it is an essential node), all its children are generated irrespective of their costs. Since all of the generated

[2]When searching state-space graphs, *best_node* is stored in another list called *CLOSED* after it has been expanded. Also, every generated node is checked for a possible duplicate in *OPEN* and *CLOSED*. If a duplicate is found, the generated node is pruned and any cost improvement propagated to the duplicate node and its descendents.

[3]Nodes with the same f' cost are ordered in *OPEN* in terms of their likelihood of leading to an optimal solution. As we will see later, nodes that are at a deeper level in the search tree are more likely to lead to an optimal solution than same-cost nodes at shallower levels. Thus, while an ancestor node of the optimal solution node with cost equal to the optimal-solution cost will necessarily be expanded, a non-ancestor node at the same level and with the same cost may or may not be expanded depending upon its position in *OPEN* relative to the ancestor node, and hence may or may not be "essential."

children might not be essential nodes, ideally one should form only essential children in order to minimize the time for insertion of nodes in *OPEN* and also to reduce the memory requirement. We use a *partial expansion* scheme that achieves this ideal behavior. In this scheme, when a node in *OPEN* is expanded, only its best ungenerated child is selected for generation. Hence we will refer to this improved A* algorithm as SEL_SEQ_A*. Here, instead of the h' field, a node u in *OPEN* uses an h'' field that has a value defined as follows:

$$u.h'' = u.h', \text{ if u has not undergone any expansion, else} \quad (1)$$
$$= \min((v.g - u.g + v.h') \text{ over all children } v \text{ of } u \text{ remaining to be generated}) \quad (2)$$

When a node u is expanded for the first time, the h' costs of its children are computed and stored. No child node of u is formed at this time, but the h'' cost of u is updated according to Eq. 2.[4] Subsequent expansions of u result in the formation of the child with the least f' cost among all children yet to be formed, and also in the modification of $u.h''$ in accordance with Eq. 2. Furthermore, now the *best_node* picked for expansion in each iteration is the node with the minimum *modified cost estimate* $f'' = g + h''$. This ensures that no non-essential nodes are generated, thus leading to a faster and more memory-efficient A* algorithm. As a result of partial expansion, at any time during the execution of SEL_SEQ_A*, a non-solution node in *OPEN* might either be *unexpanded* or *partially expanded* (i.e., had only some children generated, but not all). All other nodes that were generated have been *completely expanded* (i.e., have had all their children generated) and are discarded. Note that a solution node by definition is completely expanded.

We now outline our application of A* to TSP which is posed as follows. Given a set $0, 1, .., N-1$ of cities and the associated inter-city distances, find the shortest tour that visits every city exactly once and returns to the start city. We have formulated TSP as a tree search problem, in which the state of a node is defined by a 1-tuple: [ordered list of cities visited]. Initially, only *root* with *root.state* $= [(0)]$, exists. An expansion of a node $u \in \mathcal{T}$ 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. In most of our experiments, we have used the following simple heuristic function. The cost h' of a node is equal to the average of two sums S_i and S_o; S_i is the sum of the costs of the least expensive incoming edge incident on each unvisited city and the start city from the set of unvisited cities and the present city, and S_o is the sum of the costs of the least expensive outgoing edge from each unvisited city and the present city to the set of unvisited cities and the start city. Since the

[4] It turns out that the savings in the number of nodes formed more than compensates the extra memory used to store the h' costs of unformed children [16].

focus of this paper is on the design and evaluation of general load balancing strategies for parallel A* algorithms that can be applied to any COP, we have employed a tree search formulation and a simple heuristic function for this purpose (as opposed to a graph search formulation [17] and a tighter heuristic function [19] that are possible for TSP). In Sec. 6.2, we show that while a different heuristic function can affect the cost-wise distribution of nodes in *OPEN* , it will not have any impact on the load balancing capabilities of the QE strategy. Performance results in Sec. 8, where we test the efficacy of our schemes using the above heuristic as well as the much stronger LMSK heuristic [14], further corroborate this assertion.

3 Parallelization of A*

Here we describe the generic high-level approach we have used to parallelize A* on distributed-memory machines. Each processor executes an almost independent SEL_SEQ_A* on its own *OPEN* list. 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 across all processors. Apart from solution broadcasts, processors interact to redistribute work for better processor utilization and to detect termination of the algorithm. Such algorithms can be characterized as *parallel local A** (PLA*) algorithms.

We now introduce a few terms used in the sequel. The *OPEN* list of processor i is denoted $OPEN_i$, the set of its neighbors as $neighbors(i)$, and its neighborhood is defined as $neighbors(i) \cup \{i\}$. For any set S of processors, let $OPEN_S$ denote the union of the *OPEN* lists of these processors. For a node $u \in OPEN_S$, define $pos_S(u)$ to be the position of node u in the list of nodes in $OPEN_S$ arranged by non-decreasing f''-cost. Let $node_{i,m}$ denote the node at position m in $OPEN_i$. Define $rank_S(u) = min(pos_S(v))$ over all nodes $v \in OPEN_S$ that have the same f''-cost as node u. For any node $u \in OPEN_i$, we will refer to $rank_{\{i\}}(u)$ as node u's *local rank*, $rank_{neighbors(i) \cup \{i\}}(u)$ as its *neighborhood rank*, and $rank_{\{0,1,\ldots,P-1\}}(u)$ as its *global rank*. Thus in Fig. 2(a), for instance, the node with cost 90 in $OPEN_i$ has a local rank of three and, assuming that processor j is the only neighbor of processor i, a neighborhood rank of five. At any time in the execution of a parallel A* algorithm, the S-*rank range* for the nodes at position m in a set S of processors is defined as $max(rank_S(node_{i,m}) - rank_S(node_{j,m}) + 1)$ over all processors $i, j \in S$. When $S = \{0, 1, \ldots, P-1\}$, the S-rank range is called the *global-rank range*, and when $S = neighbors(i) \cup \{i\}$, it is called the *neighborhood-rank range* for the neighborhood of processor i. For instance in Fig. 2(a) and Fig. 2(b), the $\{i, j\}$-rank range for nodes at position two are two and six, respectively.

In contrast to a parallel local A* algorithm, a *parallel global A** (PGA*) algorithm uses a global/centralized OPEN list or multiple lists that are kept

consistent across processors, and processes nodes in the order of their global ranks. Thus in PGA* search at any time the best nodes of all processors lie in a global-rank range of at most P, which is optimal. However, such *global-rank ordered searches* are suitable only for shared-memory machines [12] and do not scale up well with the number of processors, since contention for the global list or the cost of maintaining consistent multiple lists becomes excessive.

Since PGA* algorithms have poor scalability, we focus on PLA* algorithms that use nearest-neighbor load information and work transfers to ensure that neighboring processors expand comparably promising nodes. In such nearest-neighbor PLA* algorithms, in the ideal case processors will expand nodes such that the neighborhood-rank range for best nodes in the neighborhood of each processor i is at most $d+1$, where d is the degree of the architecture. Such an ideal search is called *neighborhood-rank ordered search*. Due to the relatively low overheads in PLA* algorithms, they scale much better than PGA* algorithms. However, the departure from strict global-rank ordered search and a distributed-memory implementation introduce a number of inefficiencies in PLA* algorithms:

1. *Starvation*: This is defined as the total time (over all processors) spent in idling, and occurs when processors run out of work.
2. *Non-essential work*: This is the total time spent in processing non-essential nodes. It arises because processors perform *local-rank ordered search* (processors expand nodes in order of non-decreasing local ranks), rather than global-rank ordered search.
3. *Memory overhead*: This is caused by the generation and storage of non-essential nodes. Therefore tackling non-essential work automatically takes care of the memory overhead problem.
4. *Duplicated work*: This is the total extra time associated with pursuing duplicate search spaces and is due to *inter-processor duplicates*, i.e., duplicate nodes that arise in different processors, when the search space is a graph.

The above inefficiencies grow with the number of processors P used thus causing the efficiency $E = T_1/(P.T_P)$ of PLA* to deteriorate; here T_1 denotes the sequential execution time and represents the *essential work* for the problem in terms of the amount of time spent processing essential nodes, and T_P denotes the execution time on P processors. We use *work density* to refer to the ratio T_1/P.

Load balancing strategies are used to tackle the inefficiencies of starvation and non-essential work (and hence that of memory overhead), while duplicate pruning strategies [9, 17, 18] are required to minimize duplicated work. Since the subject of this paper is load balancing methods, we will confine our attention to tree search spaces. However, for graph search problems, we briefly point out how any load balancing method can be applied with similar effectiveness in combination with a commonly used hashing-based duplicate

pruning technique.

The duplicate pruning technique referred to above utilizes a suitable hash function to associate an *owner* processor with each distinct node of the search space. Then duplicate nodes arising in different processors are transmitted to the same owner processor where duplicate checking and pruning takes place [9, 18]. Nodes may be transferred from their owner processor, in accordance with any load balancing algorithm, to other processors where they are expanded [17]. Thus when a node u is first generated, it is hashed to its corresponding owner processor. Subsequently, when a duplicate copy v of node u is generated in any processor, it will be hashed to the same owner processor. Node v then gets pruned in the owner processor and any cost improvement is propagated to the descendents of node u in the owner and other processors to which u has been transferred.[5] Thus the duplicate pruning technique determines where nodes are checked for duplicates and pruned, while the load balancing algorithm decides where they are expanded.

In the next section, we critique previous approaches to quantitative and qualitative load balance, and in the two subsequent sections we present our parallel startup scheme and dynamic load balancing strategy that address the inefficiencies of starvation and non-essential work. For termination detection, we have used an optimal spanning-tree based algorithm which can be found in [6].

4 Previous Load Balancing Strategies

During parallel search in PLA*, it is not only important to perform quantitative load balance across processors to reduce starvation, but also to ensure that a certain amount of qualitative load balance prevails, so that non-essential work is minimized. A number of methods have been proposed to achieve quantitative and/or qualitative load balance [1, 2, 10, 11, 12, 13, 15, 20, 22, 24]. Here we critically analyze the effectiveness of five representative schemes—two purely quantitative load balancing schemes, viz., the *round-robin* (RR) strategy [10, 13, 20] and the *neighborhood averaging* (NA) strategy [22], and three schemes directed primarily towards qualitative load balance, viz., the *random communication* (RC) strategy [11, 12], the Anderson-Chen (AC) strategy [2], and the Luling-Monien (LM) strategy [15]. In all load balancing schemes that we have implemented, unless otherwise stated, work transfer from a *donor* to an *acceptor* processor comprises alternate best-cost nodes, starting with the second best-cost node, from the *OPEN* list of the donor.

[5] Note that, because of the partial expansion scheme used in SEL_SEQ_A* on each processor, node u may be partially expanded in its owner processor and partially in other processors to which the load balancing algorithm transfers it, and so may also have descendents in these processors.

4.1 Quantitative Load Balancing Strategies

In the round-robin (RR) 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 [10, 13, 20]. The *donor* processor grants a fixed fraction (one third in our implementation) of its active nodes to the *acceptor* processor. The drawback of this scheme is that a number of decisions such as the next processor to request from and the fraction of work that should be granted are oblivious to the load distribution in the neighboring processors.

The neighborhood averaging (NA) strategy tries to achieve quantitative load balance by balancing the number of active nodes (*active_len*) among neighboring processors [22]. For this purpose, each processor reports the current *active_len* value to its neighbors when it has changed by some constant absolute amount *delta*. Let w_i denote the amount of work (in terms of *active_len*) 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 processor j with respect to $W_{avg,i}$. If a processor i determines that it is a source processor, i.e., its load w_i is greater than its average neighborhood load $W_{avg,i}$, it donates its surplus work δ_i^i one node at a time to all its sink neighbors in a round-robin fashion, so that no sink processor j receives more nodes than its deficiency $-\delta_i^j$. This attempts to bring the load of the sink neighbors to the neighborhood average. Figure 1 depicts how source processor j_2 donates its surplus work $\delta_{j_2}^{j_2} = w_{j_2} - W_{avg,j_2} = 15 - 7 = 8$ nodes, to its sink neighbors j_3, i_2 and j_1. In our implementation, we used the value of *delta* that gave the best performance (10-50), and performed work (node) transfer *en masse* rather than one node at a time to reduce work transfer overhead.

There are two main drawbacks in this scheme. First, work transfer decisions rely solely on the load distribution around the source processor, and not that around the sink processors as well. This can give rise to two types of problems. Firstly, since this is a source-initiated strategy, it can happen that multiple source neighbors (e.g., j_4, i_3 and k_3 in Fig. 1) may simultaneously attempt to satisfy the deficiencies of a sink processor (j_3), thus in all likelihood converting the latter to a "source" relative to its previously source neighbors. As a result, thrashing of work will occur. Further, as illustrated in Fig. 1, it is also possible for work transfer decisions to be contrary to the goal of good load balance. When processor j_2 determines that it is a source processor relative to its average neighborhood load W_{avg,j_2}, and that it has a sink neighbor j_3, j_2 is actually a "sink" processor relative to the average neighborhood load W_{avg,j_3} of processor j_3 ($w_{j_2} = 15 < W_{avg,j_3} = 22$, even though $w_{j_2} = 15 > W_{avg,j_2} = 7 > w_{j_3} = 2$). In this case, processor j_3 should actually receive work from its other source neighbors (j_4 and i_3), and then grant some of it to processor j_2, i.e., work transfer should take place from the heavily loaded neighborhood of processor j_3 to the lightly loaded neighborhood of processor j_2, instead of in the reverse direction as the NA

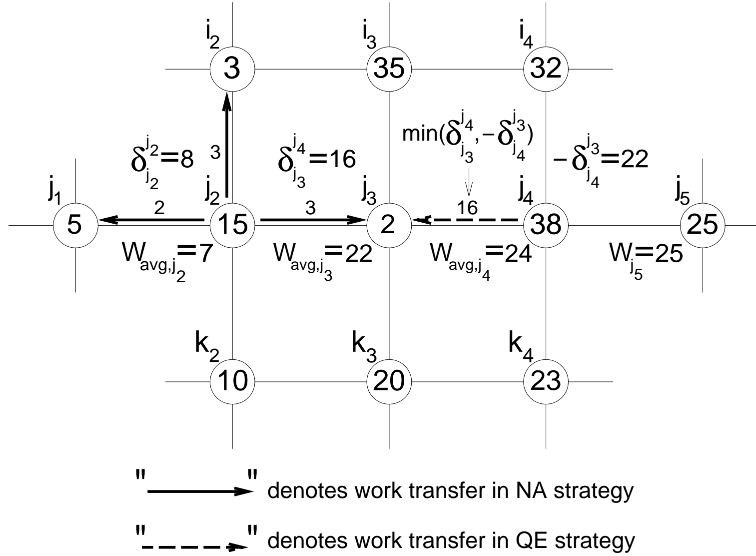


Figure 1: Comparison of the effectiveness of the NA and QE strategies in achieving quantitative load balance on a 2-D mesh architecture.

strategy would effect. The second major drawback in this strategy is that load information is disseminated when absolute changes in load occur rather than percentage changes—small load changes are more important at lower loads than at heavier loads. By not taking this into account, load reports may either be too frequent (high communication overhead) or too widely spaced (poor load balancing decisions).

4.2 Qualitative Load Balancing Strategies

We now discuss previous qualitative load balancing methods. Clearly, any scheme that performs global-rank ordered search will minimize non-essential work, but the cost of enforcing such a discipline can severely limit the scalability of parallel A*; PGA* is an example of such an approach. The challenge is to devise a low-cost approach to approximating this ideal discipline. We will analyze the effectiveness of previous load balancing methods [2, 11, 12, 15] by answering two questions: (i) Are there situations in which (redundant) work transfers take place even when there is qualitative load balance (how costly is the approach)? (ii) Are there situations in which work transfers exacerbate an existing qualitative load imbalance (how accurately does it approximate the ideal discipline)? Although these seem extreme situations, their likelihood will be indicative of the overall effectiveness of the load balancing methods.

In the random communication (RC) strategy, each processor donates the newly generated children of the node expanded in each iteration to random

neighbors [11, 12]. The fact that work transfers in this scheme are independent of load conditions, gives rise to two significant problems. First, both the foregoing situations are possible, i.e., work transfers can take place between processors that have equally good nodes, and also a processor with bad quality nodes may donate work to another with superior quality work. Second, its performance will not improve with increase in the granularity of node expansion, i.e., the speedup (T_1/T_P) for instances of two applications with the same number of essential nodes but different granularities will be approximately the same. This is in contrast to the efficacy of strategies that use load information, e.g., the two strategies discussed next and our QE strategy presented in Sec. 6. Such strategies can afford to increase the amount of load information used with increase in the granularity of node expansion to make better load balancing decisions, thus leading to enhanced overall performance.

In the scheme proposed by Anderson and Chen, herein referred to as the AC strategy, each processor i periodically reports the non-decreasing list of costs of nodes in $OPEN_i$ to its neighbors [2]. Then, for each neighbor j, processor i computes its load (relative to processor j's load) as $w_i = \sum_{u \in OPEN_i} \frac{1}{pos_{\{i,j\}}(u)}$; processor j's load w_j is computed similarly by i. If w_i exceeds w_j by some threshold, then processor i donates work to processor j (no details are given in [2] regarding which nodes are donated). In Fig. 2(a) we show a situation wherein this scheme will cause work transfer between two processors with perfectly balanced loads relative to each other.[6] A distribution of nodes among a set S of processors is said to be *perfectly load balanced* if at any time the *active_len* value for any two processors differs by at most one, and the S-rank range of the nodes at any position in the *OPEN* lists of the processors is at most $|S|$. For instance, in Fig. 2(a) nodes at any position m in $OPEN_i$ and $OPEN_j$ have ranks of $2.m-1$ and $2.m$, respectively, relative to the processor-set $\{i,j\}$, thus representing an $\{i,j\}$-rank range of two. Next, in Fig. 2(b) we depict a situation in which a processor i with essential nodes (assuming that the optimal solution has a cost of 55 and current $best_soln = 170$) will actually receive nodes from a processor j with no essential nodes, rather than donate some of its essential nodes to the latter. Another major drawback in this scheme is that the overhead of transferring, storing and merging cost-lists, and computing the work load is ($\Theta(\mid OPEN \mid)$), which is high, since the *OPEN* queue is frequently very long; this also means that the cost-list information will age substantially before it gets used, leading to even poorer load balancing decisions.

Finally, Luling and Monien have proposed a qualitative load balancing method, referred to here as the LM strategy, that defines the load of a

[6] Here we assume that the threshold value for work transfer is less than 0.5; it is easy to give an example for a larger threshold value.

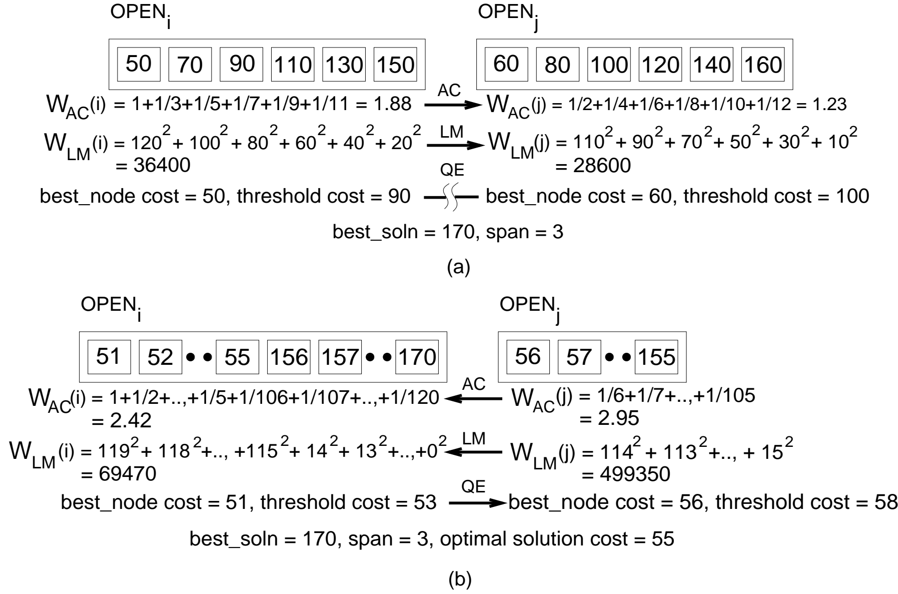


Figure 2: Comparison of the effectiveness of the AC, LM and QE strategies in achieving qualitative load balance. Work transfer operations in the three strategies (a) under a perfectly balanced load condition, and (b) under a condition of load imbalance.

processor i as $w_i = \sum_{u \in OPEN_i} (best_soln - \text{cost}(u))^2$ [15].[7] The behavior of the load balancing algorithm is determined by the following parameters: (i) Δ_{down}, Δ_{up}: a load balancing activity is initiated if the load decreases by more than Δ_{down}% or if it increases by more than Δ_{up}%. (ii) Δ: a processor participates in load balancing activity with its neighbor if their loads differ by more than Δ%. (iii) w_{min}: for a processor to participate in any load balancing activity, its load must be larger than w_{min}. From Fig. 2 we see that the LM strategy shares with the AC strategy the deficiencies of redundant work transfer (Fig. 2(a)) and ineffective qualitative load balance (Fig. 2(b)). Another major drawback in this scheme is that the load definition used to compute a processor's load and to perform load balance is dependent on the absolute cost of nodes. To achieve good qualitative load balance as depicted by the *OPEN* lists of processors i and j in Fig. 2(a), only the neighborhood ranks of nodes are important. By not taking this into account, the efficacy of the LM strategy will be sensitive to the absolute-cost distribution of nodes in *OPEN*, which depends on the application, the heuristic function being used, the input data distribution, and even the stage of execution of PLA*. Additionally, this method is highly communication intensive, since it uses only single-node work transfers.

[7] There are other definitions of w_i that are proposed, but this definition is suggested as the one best suited to qualitative load balance.

5 Static Load Balancing - Parallel Startup Phase

Here we describe and analyze a novel parallel startup phase, PAR_START, that is used in all of our parallel algorithms. The startup phase is structured as a b-ary *startup tree* of depth d (see Fig. 3). In the startup tree, each vertex corresponds to a node generation phase, the outgoing edges from a vertex to a node distribution phase (these will be described shortly), and each leaf to the nodes finally allocated to a single processor. The startup phase execution pattern for any processor is described by a unique path from the root of the startup tree to the corresponding leaf. Initially all processors start with the same *root* node (node $a1$ in Fig. 3). Then each processor asynchronously executes SEL_SEQ_A* until it has obtained $b.m$ ($m = 2$ in Fig. 3) active nodes, where m is the *multiplicity* of each branch. Thus nodes $b1$, $b2$, $b3$ and $b4$ are obtained from node $a1$. This is the *node generation phase*. Next in the *node distribution* phase, the P processors are divided into b groups labeled 1 through b, with group i being assigned m nodes, viz., the i'th, $(2.b+1-i)$'th, $(2.b+i)$'th, $(4.b+1-i)$'th, $(4.b+i)$'th, and so on, best-cost nodes in $OPEN$. Thus in Fig. 3, processors 0 and 1 (group 1), for instance, receive the 1-st ($b1$) and $(2.b + 1 - 1)$'th $= (2.2 + 1 - 1)$'th $=$ 4'th ($b4$) best-cost nodes. If we assume that the amount of essential work received by the i'th group of processors is inversely proportional to $\sum_{u \text{ is assigned to group } i} pos_{\{j\}}(u)$ and if m is even, where $OPEN_j$ is the original list from which nodes were split among processor groups, then this pattern of node assignment will effect perfect work distribution across processor groups. The above sequence of node generation and distribution alternates, until each processor has obtained its own m nodes. This completes the startup phase, and subsequently, each processor executes SEL_SEQ_A* on its starting nodes. Note that, since in each distribution phase the work load is evenly distributed across the different processor groups under the preceding assumption, at the end of the startup phase all processors will end up with the same work load. Therefore we obtain the following theorem.

Theorem 1 *If the amount of essential work corresponding to any set R of nodes in any list* OPEN_j *is assumed to be inversely proportional to* $\sum_{u \in R} pos_{\{j\}}(u)$*, and if the multiplicity m is even, then the parallel startup scheme PAR_START distributes the initial load equally among all processors.*

Furthermore, our startup scheme does not involve any communication between processors. Therefore we have:

Property 1 *The parallel startup scheme PAR_START does not involve any communication between processors.*

This property is especially important for distributed implementations of PLA* where communication latencies are high.

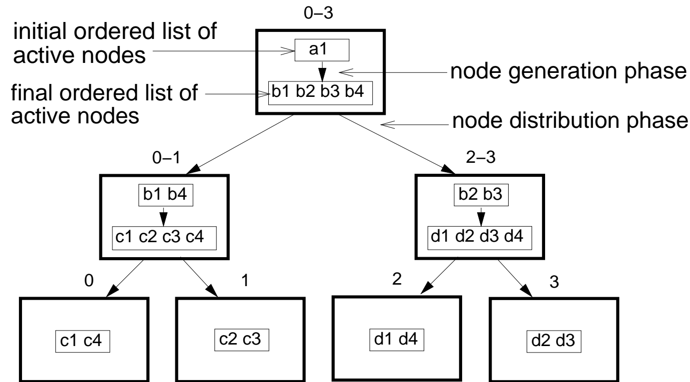


Figure 3: Structure of the startup phase represented by a startup tree for $b = 2$, $m = 2$ and $P = 4$.

Since the assumption in Theorem 1 will not hold exactly in practice, the load balance obtained in the startup phase will not really be perfect. Under more realistic conditions, the parameters b and m will affect the degree of load balance achieved by the startup phase. Before we consider the effect of b and m, we define a few terms that are useful in our subsequent discussions. The *quality* of a node refers to the amount of essential work associated with the node as reflected by its cost.[8] By *quantity* of work (nodes) in a processor, we mean its value of *active_len*. A good distribution of nodes across processors is one in which different processors have almost equal quantity and quality of nodes. Clearly, since each processor obtains the same number of starting nodes, a good quantity distribution is effected by the startup phase. To judge the impact of parameters b and m on startup phase qualitative load balance, we note that in any distribution phase, the assignment of the first set of b least-cost nodes will determine the load of a processor group more than each of the subsequent sets of nodes. Therefore processor groups receiving good nodes in the first set of b nodes will be more heavily loaded than others. Hence smaller the branching factor b, more are the number of distribution phases and more is the deterioration in the quality 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.

Now we analyze the time complexity of the startup phase. Let us call the combination of a node generation phase followed by a node distribution phase, a *step*. Then in each step, $b.m$ distinct active nodes are generated,

[8] 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.

and m of them are distributed to each of the b processor groups. Note that each processor executes a total of $\lceil \log_b P \rceil$ steps. Therefore the startup phase time T_{su} becomes:

$$T_{su} = \Theta(m.b.\lceil \log_b P \rceil) = \Theta(m.b. \left\lceil \frac{\log_2 P}{\log_2 b} \right\rceil) \quad (3)$$

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 $\Theta(\log P)$, which is optimal as we show in the following two theorems, while for a sequential startup phase ($b = P$, $m = 1$) used in previous work [13], T_{su} grows as $\Theta(P)$.

Theorem 2 *The minimum time in which the startup phase can be completed on any P-processor parallel machine is $\Theta(\log P)$.*

Proof: We first consider startup phase time complexity on a P-processor PRAM with concurrent read (CR) capability, which implies that all processors have simultaneous access to all nodes that are generated. The startup problem is then to generate P distinct nodes from $root$, so that subsequently each processor can pick a distinct starting node. Let c, a constant with respect to P, be the maximum number of children of any search-space node. We assume the generation of one child of any node in the startup problem to be an unit-time atomic task. Thus from $root$ a maximum of c nodes can be generated in unit time by c processors in parallel. Similarly, if there are $i < P$ nodes at any intermediate stage of the startup phase, in the next time unit a maximum of $i.c$ nodes can be generated. Consequently, after j time units, the maximum number of distinct nodes available will be c^j. Thus the startup time complexity becomes $\Theta(\log_c P) = \Theta(\log P)$.

Secondly, since any problem is at least as easy to solve on a PRAM with CR capability as on any other same-sized machine, it follows that the startup problem will take $\Omega(\log P)$ time on any P-processor machine with or without a CR capability. □

Theorem 3 *The parallel startup scheme PAR_START executes in optimal time on any parallel machine.*

Proof: From Eq. 3, it follows that for constants b and m, the startup time complexity is $\Theta(\log P)$, which from Theorem 2 is optimal for all parallel machines. □

6 Dynamic Load Balancing - The Quality Equalizing (QE) Strategy

In this section, we present our new load balancing strategy called *quality equalizing* (QE) strategy because of its use of a highly effective scheme

(Sec. 6.2) for balancing the quality of work between neighbors. The QE strategy comprises both a quantitative load balancing scheme to reduce starvation and load balancing overhead, and a qualitative load balancing scheme to curtail non-essential work. The two schemes are described in detail below.

6.1 Quantitative Load Balancing

In this scheme, each processor monitors its *active_len* periodically and reports any significant changes in it to its neighbors—in our implementation a change of 10% is reported to the neighbors. Also, each processor assumes that the processor space comprises its neighbors and itself only. Here we use w_i, $W_{avg,i}$ and δ_i^j to mean the same quantities as in the discussion of the NA strategy in Sec. 4. 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, when a processor i anticipates running out of work (the way this is done will be described shortly), 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 $min(\delta_i^{i_1}, -\delta_{i_1}^i)$ (with the restriction that at least 10 percent and no more than 50 percent of the work at i_1 is granted). 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. If the work request is turned down by processor i_1, say, because i_1 has already granted work to another sink processor in the meantime, then processor i requests work from the processor with the next most amount of work, and so forth, until it either receives work or has requested all its neighbors. In the latter case, it waits a certain amount of time, and then resumes requesting work as before. We will refer to work requests meant to effect quantitative load balance as *quantitative work requests*.

In Fig. 1 we show how a sink processor j_3 will request work using the above quantitative load balancing scheme. From the discussion of the NA strategy in Sec. 4, we concluded that processor j_3 should actually receive work from processor j_4 rather than from processor j_2 as in the NA strategy. This is exactly what happens in the QE strategy, and thus work flows from the heavily loaded neighborhood of j_4 to the lightly loaded neighborhood of j_2 via processor j_3. The NA strategy makes work transfer decisions for processor j_2 considering a processor space comprising its neighbors and itself, i.e., a processor space of radius one. On the other hand, our quantitative load

balancing scheme makes work transfer decisions for processor j_3 considering a processor space comprising its neighbors, its neighbors' neighbors (since the average neighborhood load of the neighbors is taken into account), and itself, i.e., a processor-space of radius two. Therefore we obtain the following property of the QE strategy.

Property 2 *The QE strategy makes quantitative load balancing decisions for any processor i considering a processor-space radius of two around it using only near-neighbor load information.*

Anticipatory Quantitative Load Balance: To reduce idling caused by latency between work request and work procurement, processors issue quantitative work requests when starvation is anticipated as follows.[9] We note that at any time the least-cost node in a processor is expanded. Therefore any decrease in *active_len* below a low threshold implies that the best nodes available are not good enough to generate active nodes and hence this decrease is likely to continue.[10] In our scheme, processors start requesting nodes when *active_len* is below a certain low 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 mitigated by pipelining the message transfer, i.e., by sending the work in batches. Basically, the first message unit should be short, so that the processor does not idle long before it receives any work. Subsequent messages (for the same work transfer) can be longer, but should not be so long that the preceding message unit gets consumed and the processor idles for an appreciable period of time. For the problem sizes we experimented with, it sufficed to use a short message (one or two nodes) followed by longer messages (each not exceeding 20 nodes). Note that we perform quantitative load balance only when a processor is about to starve, not at all times as in the NA strategy. Thus our quantitative load balancing overhead is low.

6.2 Qualitative Load Balancing

Here the objective is to minimize the total amount of non-essential work. This is done by ensuring that neighboring processors expand nodes with comparable neighborhood ranks. In this method, each processor periodically monitors the cost of its s'th best node in *OPEN*, where $s \geq 2$ is the *span*, and reports any changes to its neighbors. We will refer to this node as the *threshold node* and its cost as the *threshold cost* of the processor. In this

[9] 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.

[10] If the threshold is kept low and the expansion of best node does not yield any active nodes, it is more likely that the other nodes in *OPEN* will not yield any essential nodes, than when the threshold is high.

manner, every processor at any time has information regarding the threshold cost of its neighbors. Note that because of message latency, and due to the time spent expanding nodes, the threshold-cost information may become stale before it is processed by a processor. To simplify the discussion, such *information aging* is assumed here to be zero; later we will show how it can be taken into account by using anticipatory qualitative work requests.

A processor i requests work from the neighbor j with the least threshold cost, when the cost of its best node is more than the threshold cost of that neighbor. The work request carries processor i's best-node cost, and processor j grants only a few good nodes with better cost. In case, the work request is turned down by processor j, say, because j has already granted work to another processor in the meanwhile, then processor i requests work from the neighbor with the next least threshold cost that is less than the best-node cost of i, and so forth, until it either gets work or has requested all such neighbors.

From the condition for triggering work requests, we immediately obtain the following defining property of our qualitative load balancing method:

Property 3 *In the QE strategy, a processor i requests nodes with better neighborhood rank than that of its best node, whenever the latter's rank relative to any neighboring processor deteriorates beyond the threshold value s. Furthermore, work is requested from the neighbor that has a threshold node with the least rank relative to the set of processors $neighbors(i) \cup \{i\}$.*

From the above property, it is clear that in the worst case, each neighbor of a processor i can have exactly $s - 1$ nodes that have better neighborhood ranks than i's best node, and still not cause i to request work. Hence we obtain:

Property 4 *In the QE strategy, the neighborhood-rank range of best nodes in the neighborhood of any processor can become at most $d.(s-1)+1$ before work transfer is triggered to reduce it, where d is the degree of the target architecture.*

For a small enough span, the above worst-case neighborhood-rank range of $d.(s - 1) + 1$ for the best nodes departs by only a small factor from the ideal range of $d + 1$. Hence parallel search using the QE strategy is close to neighborhood-rank ordered search. Furthermore, since neighborhoods are connected, and since work transfers (comprising alternate best-cost nodes from the *OPEN* lists of donor processors) expedite the dispersal of nodes with good global ranks at any processor to other processors, an almost-ideal neighborhood-rank range for best nodes in neighboring processors translates to a small factor difference from the ideal global-rank range of P for the best nodes in all processors. Consequently, non-essential work is curtailed. Work requests meant to correct qualitative load imbalance will be referred to as *qualitative work requests*. 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 Fig. 2, we show work transfer decisions made in the above described qualitative load balancing scheme under a condition of perfect load balance (Fig. 2(a)) and a condition of load imbalance (Fig. 2(b)). In Fig. 2(a), no work transfer takes place because the best-node cost of none of the processors exceeds the threshold cost of the other. It is easy to see that work transfers under balanced load conditions will never take place when the span is at least two. Therefore we have:

Property 5 *If two processors i and j have perfect load balance, then the QE strategy will not perform any work transfer between them.*

In Fig. 2(b), the QE strategy transfers essential nodes from processor i (the processor with essential nodes) to processor j (the processor with no essential nodes), since the cost of the best node in processor j (56) is more than the threshold cost of processor i (53). In fact, if a processor i has no essential nodes, i.e., its best-node cost is greater than the optimal solution cost, and has at least one neighbor with s or more essential nodes, i.e., the minimum threshold cost among the neighbors of i is less than or equal to the optimal solution cost, then a work request for essential nodes will be generated from processor i. Therefore we have the following property.

Property 6 *In the QE strategy, if a processor i has no essential nodes, and has at least one neighbor with s or more essential nodes, then a work request for essential nodes will be generated from processor i.*

From Properties 3 and 6, we see that work requests in the QE strategy are triggered based on the neighborhood rank of nodes rather than on the absolute cost of nodes as in the LM strategy of Sec. 4.2. Thus the QE strategy is impervious to the absolute cost-wise distribution of nodes in the *OPEN* list, which depends on the application, the heuristic lower-bound function, the input data distribution and also the stage of execution of PLA*.

Anticipatory Qualitative Load Balance: Although Property 6 guarantees that a processor i with no essential nodes will request essential nodes when an adequate number of them are available with any neighbor, it is not sufficient to prevent non-essential work. Due to message latency, processor i will perform non-essential work during the time that it takes to fetch essential nodes. If the donor processor has already expanded or granted its essential nodes to some other processor before receiving the work request from processor i, then i will perform even more non-essential work. To tackle this latency problem, we have designed a scheme by which qualitative load imbalance is predicted and corrected before its actual occurrence by means of anticipatory qualitative work requests. The basic idea is to check for any cost deterioration in the *lead node* (instead of the *best_node* as was done without the anticipatory mechanism) of a processor i relative to the *threshold node* of any

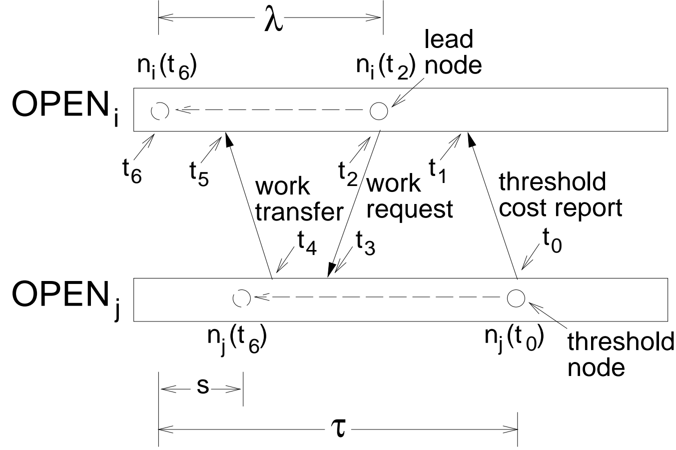


Figure 4: The *OPEN* lists of neighboring processors i and j and the positions of lead node n_i and threshold node n_j at different time instants used in the computation of the lead position λ and threshold position τ.

neighbor j, and to trigger work transfer to correct this condition just before the lead node becomes the *best_node* in processor i at a later time (see Fig. 4). Clearly, for any processor i we should have $pos_{OPEN_i}(threshold\,node) > pos_{OPEN_i}(lead\,node) \geq pos_{OPEN_i}(best_node) = 1$. The *span* is then defined as $s = pos_{OPEN_i}(threshold\,node) - pos_{OPEN_i}(lead\,node) + 1$. The choice of span will be discussed shortly, but first we determine the position λ of the lead node and the position τ of the threshold node, in terms of various time intervals defined next, in order to correctly anticipate and trigger qualitative work requests.

In our scheme, a processor repeatedly: (1) expands a node, (2) then processes all incoming messages (e.g., load information and work requests from neighbors) and outgoing messages (e.g., load information and work transfers to neighbors when required). The average time to execute these steps once is $t_s = t_e + t_m$, where t_e is the average node-expansion time and t_m is the average message-processing time. In the worst-case, each node expansion may result in d informational messages (e.g., due to a change in the threshold cost) to/from d neighbors and one work request/transfer message. Thus $t_m = O(d)$. Let t_c denote the transmission time for a work report to reach a neighbor, i.e., t_c is a known constant of the parallel machine. Since a processor processes a new set of messages every t_s time period, the average time period between the arrival and processing of a message at a processor is $t_s/2$. Hence the delay between any processor i reporting its threshold cost at time t_1 to a neighbor j, and the time t_2 when j compares that with its lead-node cost is $(t_2 - t_1) = t_d = t_c + t_s/2 = O(d)$. We use t_{al} and t_{at} to denote the average times taken to advance by one position by nodes at or before the lead-node position, and by nodes at or before the threshold-node position,

respectively. Thus, the average times required for the lead and threshold nodes to become the best node are $\lambda.t_{al}$ and $\tau.t_{at}$, respectively. Note that in any $OPEN_i$, a node u_1 will advance at a rate faster than another node u_2 that lies behind it ($pos_{OPEN_i}(u_2) > pos_{OPEN_i}(u_1)$). This is because, when *best_node* is expanded in any iteration, all its generated children that occupy positions before u_1 will necessarily occupy positions before u_2, while some of its children might lie between u_1 and u_2. Therefore, $t_s \leq t_{al} < t_{at}$, since t_s is the rate at which the *best_node* "advances". We assume that due to effective qualitative and quantitative load balance the values t_s, t_{al}, and t_{at} are approximately the same in neighboring processors.

Now the problem of computing the values of λ and τ can be stated as follows (see Fig. 4). Determine the position λ of the *lead node* in $OPEN_i$ of a processor i and the position τ of the *threshold node* in $OPEN_j$ of a neighbor j of processor i, such that:

If

1. Processor j reports the cost of its threshold node n_j at time t_0 ($pos_{OPEN_j}(n_j(t_0)) = \tau$), and
2. Processor i receives the threshold cost of processor j at time t_1, and subsequently requests m nodes with cost less than $cost(n_i)$ from j at time t_2, if $cost(n_i) > cost(n_j)$, where $m \leq s/2$ and n_i is the lead node of processor i ($pos_{OPEN_i}(n_i(t_2)) = \lambda$),

then

It should be possible for processor i to obtain m nodes with cost no greater than $cost(n_j)$ (assuming no more than $s/m - 1$ neighbors request nodes from j in the current load-balancing phase[11]) before it processes node n_i at a later time t_6. Also, at time t_6 processor j should have $s - m.l$ nodes with cost no greater than $cost(n_j)$, where $l \leq s/m-1$ is the number of requests processor j honored during the current load-balancing phase.

So, if n_i is the first non-essential node in $OPEN_i$ and n_j is an essential node, then processor i will be able to get m essential nodes from processor j (assuming no more than $s/m - 1$ neighbors request nodes from j in the current load-balancing phase), before it processes node n_i, i.e., before it performs non-essential work. Furthermore, at the time that processor i receives these m essential nodes, processor j will have at least m essential nodes (since only $l \leq s/m - 1$ work requests are granted in a given load-balancing phase), thus maintaining a good qualitative load balance. Later we will show that the probability of processor i being able to fetch essential nodes, even when there are other processors competing to get nodes from processor j, is high.

[11] A load-balancing phase is defined as the average time period between work requests by a "sink" processor. For the purpose of our analysis, we can assume that work requests by processors are synchronized. While this will not be the actual case in a MIMD machine, it certainly models the average distribution of work requests.

We are now ready to compute the lead and threshold positions λ and τ in terms of the *span* s and the time intervals t_s, t_c, t_{al}, and t_{at}. At time $t_0 = 0$, processor j in Fig. 4 reports the cost of its threshold node n_j. This message arrives at processor i at time $t_1 = t_c$ and at time $t_2 = t_c + t_s/2$ processor i compares the threshold cost of j with its lead cost $cost(n_i)$. If $cost(n_i) > cost(n_j)$, processor i requests nodes of cost better than $cost(n_i)$ from processor j. The latter receives the request at time $t_3 = 2.t_c + t_s/2$ and donates nodes at time $t_4 = 2.t_c + t_s$. Processor i receives the work message at time $t_5 = 3.t_c + t_s$, and inserts the received nodes into its $OPEN$ list at time $t_6 = 3.t_c + 3.t_s/2$. From the definition of *span*, we have:

$$pos_{OPEN_j}(n_j(t_0)) - pos_{OPEN_i}(n_i(t_0)) = s - 1 \tag{4}$$

From the above we directly obtain:

$$\begin{aligned}
\lambda &= pos_{OPEN_i}(n_i(t_2)) = pos_{OPEN_i}(n_i(t_6)) + (t_6 - t_2)/t_{al} \\
&= 1 + 2.(t_c + t_s/2)/t_{al} \qquad (5) \\
\tau &= pos_{OPEN_j}(n_j(t_0)) = pos_{OPEN_i}(n_i(t_0)) + s - 1 \\
&= pos_{OPEN_i}(n_i(t_2)) + (t_2 - t_0)/t_{at} + s - 1 \\
&= s + (t_c + t_s/2)(2/t_{al} + 1/t_{at}) \qquad (6)
\end{aligned}$$

Thus each processor monitors the cost of its threshold node and reports any changes in it to its neighbors. A processor requests work from the neighbor with the least threshold cost, when the cost of its lead node happens to be worse than the threshold cost of that neighbor. Since the requesting processor is able to obtain good nodes before any appreciable quality degradation occurs, qualitative load imbalance is prevented.

We now consider the choice of the span value s. Note that the smaller the span, the less is the non-essential work (because of tighter load balance) but more the work transfer overhead, and vice versa. Also note that the value of s determines the potential number of nodes of better cost at a source processor compared to the best-node cost of a requesting sink neighbor. Suppose we keep s constant with respect to d. Now consider a sink processor k that sends a qualitative work request at time t_1 to its source neighbor i with the least threshold cost. If at t_1 processor i receives multiple work requests, then only a constant number of them are guaranteed to be honored. If k's request is honored, the work transfer from i to k helps limit the $\{i, k\}$-rank range of their best nodes to s. However, if k's request is turned down, it will request from another source neighbor. In the worst-case, processor k may need to request from $O(d)$ source neighbors before finally receiving work from some source neighbor j at time t_2. In this case, the time elapsed between the first request and the last request is $t_2 - t_1 = O(d.t_d)$, where recall that $t_d = t_c + (t_e + t_m)/2 = O(d)$. If processor k had requested work from processor j at time t_1, it would have received at least one node u with a $\{j, k\}$-rank of no more than s. However, by requesting work at a later time

t_2, it can only be sure of receiving a node with a $\{j,k\}$-rank not more than $rank_{\{j,k\}}(u) + O(d.t_d)$. Hence, by keeping a constant span, the rank range of best nodes of neighboring processors can only be limited to $O(s + d.t_d)$. On the other hand, if we choose $s = O(d)$, all work requests will be honored the first time around, since the number of simultaneous requests to the same source processor grows as $O(d)$. Hence a best-node rank range of $O(d)$ can be maintained between any two processors. Moreover, keeping $s = O(d)$, as opposed to keeping it constant, reduces work transfer overhead leading to improved performance. In our implementation, we let s grow at a rate slightly less than $\Theta(d)$.

To gauge the effectiveness of the above scheme, we compute, during any load-balancing phase (see footnote 11), the probability r that a "sink" processor requesting good nodes from a "source" neighbor will be able to obtain such nodes; a processor is a "sink" with respect to a "source" neighbor when its best-node cost is worse than that neighbor's threshold cost. For this we assume that the span $s = d$ and that the number of nodes donated on a qualitative work request is $m = 2$. If a source processor has s' nodes of better cost than the best-node costs of sink neighbors, where $s' \geq s$, then it will be able to satisfy at most $\frac{s'}{m} - 1 \geq \frac{s}{m} - 1$ requests from sink neighbors, since for good load balance it should keep m nodes with itself. Let us assume that the fraction of all processors that are sources is α, the fraction of sinks is γ, and the fraction of processors that are neither sources nor sinks is β, so that $\alpha + \beta + \gamma = 1$.[12] Furthermore, we assume that the source and sink processors are uniformly distributed throughout the network, and, since a processor is equally likely to be a source or a sink, we let $\alpha = \gamma$.

Since a sink processor has $d.\alpha$ source neighbors that it can request from, the probability that it requests from a particular source neighbor is $\frac{1}{d.\alpha}$.[13] Therefore for any source processor p_1, the probability that a particular neighbor p_2 requests from it is the product of the probability that p_2 is a sink (γ), and the probability that p_2 requests from p_1 ($\frac{1}{d.\alpha}$). Thus the probability that a source processor receives i requests from its d neighbors during any given load-balancing phase is $\binom{d}{i} . (\frac{\gamma}{d.\alpha})^i . (1 - \frac{\gamma}{d.\alpha})^{d-i}$.[14]

[12] Strictly speaking, a processor cannot be categorized in absolute terms as a source, sink or neither, since it may be a "source" with respect to one neighbor and sink relative to another. However, such categorization simplifies the analysis with probably little loss in accuracy.

[13] Since different source neighbors of a sink processor can in general have different threshold costs, and since in the QE strategy a sink processor requests from the source neighbor with the least threshold cost, all source neighbors will not be equally likely to receive a request from it. However, because the best source grants nodes, its threshold cost will increase, thus making it more likely for the other sources to receive requests. Hence assuming all sources of a sink neighbor to be equally likely to receive requests is fair.

[14] Here $\binom{d}{i}$ denotes the number of ways in which i objects can be chosen from a set of d distinct objects.

Now consider a sink processor p_1 with a source neighbor p_2 from which it requests work in a given load-balancing phase. We want to determine the probability r that this request will be granted. If p_2 receives $\frac{s'}{m} - 2$ or less requests from its other neighbors during this phase, p_1's request is guaranteed to be honored. However, if p_2 receives $i \geq \frac{s'}{m} - 1$ other requests, the probability that p_1's request will be granted is the probability that it is among the first $\frac{s'}{m} - 1$ requests received by p_2—this probability is $\frac{(s'/m)-1}{i+1}$. Thus for $\alpha = \gamma$, $s = d$ and $m = 2$, the probability that p_1's request will be honored is:

$$
\begin{aligned}
r &= \sum_{i=0}^{(s'/m)-2} \binom{d-1}{i} \cdot \left(\frac{\gamma}{d.\alpha}\right)^i \cdot \left(1 - \frac{\gamma}{d.\alpha}\right)^{d-1-i} + \\
&\quad \sum_{i=(s'/m)-1}^{d-1} \frac{(s'/m)-1}{i+1} . \binom{d-1}{i} \cdot \left(\frac{\gamma}{d.\alpha}\right)^i \cdot \left(1 - \frac{\gamma}{d.\alpha}\right)^{d-1-i} \\
&\geq \sum_{i=0}^{(s/m)-2} \binom{d-1}{i} \cdot \left(\frac{\gamma}{d.\alpha}\right)^i \cdot \left(1 - \frac{\gamma}{d.\alpha}\right)^{d-1-i} + \\
&\quad \sum_{i=(s/m)-1}^{d-1} \frac{(s/m)-1}{i+1} . \binom{d-1}{i} \cdot \left(\frac{\gamma}{d.\alpha}\right)^i \cdot \left(1 - \frac{\gamma}{d.\alpha}\right)^{d-1-i} \\
&= \sum_{i=0}^{(d/2)-2} \binom{d-1}{i} \cdot \left(\frac{1}{d}\right)^i \cdot \left(1 - \frac{1}{d}\right)^{d-1-i} + \\
&\quad \sum_{i=(d/2)-1}^{d-1} \frac{d/2-1}{i+1} . \binom{d-1}{i} \cdot \left(\frac{1}{d}\right)^i \cdot \left(1 - \frac{1}{d}\right)^{d-1-i}
\end{aligned} \tag{7}
$$

From the above inequality we obtain:

$$
\begin{aligned}
1 - r &= \sum_{i=0}^{d-1} \binom{d-1}{i} \cdot \left(\frac{1}{d}\right)^i \cdot \left(1 - \frac{1}{d}\right)^{d-1-i} - r \\
&\leq \sum_{i=d/2-1}^{d-1} \left(1 - \frac{d/2-1}{i+1}\right) \cdot \binom{d-1}{i} \cdot \left(\frac{1}{d}\right)^i \cdot \left(1 - \frac{1}{d}\right)^{d-1-i} \\
&\leq \left(\frac{d}{2} + 1\right) \cdot \left(1 - \frac{d/2-1}{d}\right) . \frac{(d-1)^{d/2-1}}{(d/2-1).(d/2-2)} . \\
&\quad \left(\frac{1}{d}\right)^{d/2-1} . \left(1 - \frac{1}{d}\right)^0 \\
&\leq \left(1 - \frac{1}{d}\right)^{d/2-1} \ll 1 \text{ for } d \geq 8
\end{aligned} \tag{8}
$$

This means that the probability r a sink processor requesting work is granted nodes in its first attempt is high for $d \geq 8$, and that this probability increases with d. Indeed, numerically computing the lower bound on r from Eq. 7,

Procedure QE
/* Procedure QE is used in PLA*-QE to achieve load balance */
begin

Each processor i, $0 \leq i < P$, executes the following steps:

1. Report work status: Periodically monitor *active_len* and the *threshold cost*, and report any significant changes (10% and greater than 0%, respectively) in them to all neighbors.
2. Receive work report: **If** (a work status report is received from a neighbor) **then** record it.
3. Update *j_max* and *j_best*: *j_max* := neighbor with the maximum *active_len* value; and *j_best* := neighbor with the least threshold cost.
4. Work request:
   if (no previous work request from i remains to be serviced) **then begin**
   if(*active_len* $= 0$) **or** (*active_len* $<$ *acceptor threshold* and is decreasing)
   Send a quantitative work request to *j_max*, along with the information $\delta_i^{j_max}$;
   else if (*lead_node* is costlier than the threshold cost of *j_best*)
   Send a qualitative work request to *j_best*, along with the cost of *lead_node*.
   endif
5. Donate work: **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. Donate work: **If** (a qualitative work request is received from neighbor j) **then** grant a few active nodes that are cheaper than j's *lead_node*.
7. Receive work: **If** (work is received) **then** insert nodes received in $OPEN_i$.

end /* Procedure QE */

Figure 5: Algorithm for the Quality Equalizing Strategy.

we find that for $d = 6$ it is 0.928, for $d = 8$ it is 0.987, and for $d = 10$ it is 0.998. For $d = 4$, r turns out to be greater than or equal to 0.684, but the probability that a sink processor is able to obtain nodes from source neighbors in two or less attempts is $r + (1 - r).r \geq 0.9$ which is high. Hence we obtain the following result.

Theorem 4 *In the QE strategy, if the span $s = d$ and the number of nodes granted on a work request is $m = 2$, using an anticipatory qualitative work request scheme ensures with high probability (> 0.9) that a processor requesting essential nodes will be able to obtain them in at most one (for $d \geq 6$) or two ($d = 4$ or 5) attempts, and avoid non-essential work. Here d is the node degree of the target architecture.*

A formal description of the QE strategy that comprises all the schemes discussed in this section is given in Fig. 5.

In the next section, we will compare the performance of the QE strategy with that of the RR and RC strategies. Since the QE strategy performs both quantitative and qualitative load balance, while the RR and RC strategies

Algorithm PLA*($\mathcal{P}$,P,b,m)
/* Given a COP $\mathcal{P}$, and, the branching factor b and multiplicity m for the startup phase, PLA* returns an optimal solution to $\mathcal{P}$, using P processors */
begin

Each processor i, $0 \leq i < P$, executes the following steps, starting with the same *root*:

1. Parallel startup phase: Execute the parallel startup scheme PAR_START with branching factor b and multiplicity m.
   If an optimal solution is found in the startup phase **then** report solution to host and exit.
   Otherwise initialize $OPEN_i$:= my m nodes; *active_len* := m.
   If any solution has been found in the startup phase **then** *best_soln* := cost of current best solution
   else *best_soln* := ∞. /* *best_soln* holds the cost of the current global best solution */

repeat

2. Execute an iteration of SEL_SEQ_A* on $OPEN_i$.
3. **If** (a solution is generated that is better than *best_soln*) **then** update *best_soln* := current solution cost; broadcast *best_soln* to all other processors.
4. **If** (a solution update message is received) **and** (if the received solution is better than the current best solution in $OPEN_i$) **then** update *best_soln*.
5. Execute one of RR, RC, RR+RC, NA+RC and QE strategies depending on the load balancing method to be used.

until(termination is detected)

end /* Algorithm PLA* */

Figure 6: Algorithm PLA*.

are geared primarily towards a single type of load balance, we also consider two methods that combine previous quantitative and qualitative load balancing approaches to make a stronger comparison. The mixed methods are RR+RC and NA+RC, and are similar to the RR and NA strategies, respectively, except that as in the RC strategy, each processor also donates the newly generated children of the node expanded in each iteration to random neighbors. In Fig. 6, we give a formal description of PLA* that utilizes the parallel startup phase PAR_START and either one of the above load balancing methods.

7 Scalability Results

In this section, we present good upper and lower bounds on the scalability of parallel A* algorithms obtained in our previous work [7]. In that work the lower bounds on scalability were obtained under the assumption that

communication delay is negligible compared to the node expansion time (which is true for the TSP application considered in our work). However, the results remain unchanged if the anticipatory qualitative load balancing scheme of Sec. 6.2 is used in the QE strategy; this scheme was not used previously. Before we present our bounds, we define a few terms. Let W denote the total essential work in terms of the total number of essential expansions. The speedup S is defined as the ratio W/I_P, and the efficiency E as S/P, where I_P is the number of iterations required by the parallel algorithm executing on P processors. The total overhead over all processors W_o for a parallel algorithm is therefore equal to $P.I_P - W$. Hence efficiency $E = W/(W + W_o) = 1/(1 + W_o/W)$. The isoefficiency function of a parallel algorithm is defined to be the required rate of growth of W with respect to P, to keep the efficiency fixed at some value, and is a measure of the scalability of the algorithm [13]. From the expression for efficiency, we notice that W needs to grow as W_o for constant efficiency. In other words, the rate of growth of W_o with P (and other architectural parameters that change with the size of the parallel machine) is the isoefficiency function of the parallel algorithm. Lower values of the isoefficiency function like $\Theta(P)$ and $\Theta(P.\log^2 P)$ indicate that the algorithm is very scalable, while high values of this function like $\Theta(P^2)$ imply poor scalability.

First, we give a lower bound on the isoefficiency function (upper bound on the scalability) of parallel A* algorithms that is obtained by analyzing the overhead incurred by the optimal startup phase PAR_START of Sec. 5. This overhead accrues from the limited initial parallelism available in the search problem.

Theorem 5 [7] *A lower bound on the isoefficiency function of an arbitrary parallel A* algorithm on any P-processor architecture is $\Theta(P.\log P)$.*

To derive the above lower bound, we considered the overhead incurred in the initial stage (startup phase) of PLA*-QE. The upper bound on the isoefficiency function of PLA*-QE is obtained by determining the idling and non-essential work overhead incurred in the final stage of PLA*-QE—we assume that due to the availability of appreciable number of essential nodes during the intermediate stage of PLA*, the idling and non-essential work overhead during this period is not more (in order terms) than the overhead incurred in the final stage of PLA*.

Theorem 6 [7] *The isoefficiency function of PLA*-QE is upper bounded by $\Theta(W_o)$, where $W_o = \sum_{i=0}^{D-1} p_i.(D-i).d = O(P.D.d)$ is with respect to an origin processor that maximizes it. Here p_i denotes the number of processors at a distance of i from origin, and D and d are the diameter and degree, respectively, of the P-processor target architecture.*

The expression for W_o in the above theorem can be simplified for an interesting class of parallel architectures called k-ary n-cube tori, which includes

rings ($n = 1$), 2-D tori ($n = 2$), 3-D tori ($n = 3$) and hypercubes ($k = 2$) as special cases [5]. Here, n is the dimension and k the radix of the architecture. The number of processors $P = k^n$. Every processor in the k-ary n-cube has an n-digit radix-k label $(a_{n-1}, a_{n-2}, \ldots, a_i, \ldots, a_1, a_0)$, and has neighbors $(a_{n-1}, a_{n-2}, \ldots, (a_i + 1) \bmod k, \ldots, a_1, a_0)$ and $(a_{n-1}, a_{n-2}, \ldots, (a_i - 1) \bmod k, \ldots, a_1, a_0)$ along each dimension i. Therefore every processor has m neighbors along each dimension, where $m = 2$ for $k > 2$, and $m = 1$ for a hypercube ($k = 2$). Also, k-ary n-cube meshes are defined in the same way except that there are no end-around connections in any dimension; the linear array ($n = 1$) and 2-D mesh ($n = 2$) are its special cases. The upper bound on the isoefficiency function of PLA*-QE for these architectures is given next.

Theorem 7 [7] *The isoefficiency function of PLA*-QE on P-processor k-ary n-cube tori and meshes is upper bounded by $\Theta(P.k.n^2)$.*

For the linear-array/ring ($n = 1$), 2-D mesh ($n = 2$) and hypercube ($k = 2$) architectures, three popular topologies, the following three corollaries are directly obtained from the above theorem.

Corollary 1 [7] *The isoefficiency function of PLA*-QE on a linear array or ring with P processors is upper bounded by $\Theta(P^2)$.*

Corollary 2 [7] *The isoefficiency function of PLA*-QE on the hypercube architecture with P processors is upper bounded by $\Theta(P. \log^2 P)$.*

Corollary 3 [7] *The isoefficiency function of PLA*-QE on the 2-D mesh architecture with P processors is upper bounded by $\Theta(P.\sqrt{P})$.*

In the next section, we present performance results for PLA*-QE on the hypercube architecture that validate our scalability analysis.

8 Performance Results

We implemented five PLA* algorithms each incorporating the parallel startup scheme PAR_START, and either one of RR, RC, QE, RR+RC and NA+RC load balancing strategies, on an nCUBE2 hypercube multicomputer. All our algorithms use the tree search-space formulation of Sec. 2 to solve TSP. Unless otherwise stated, the simple heuristic of Sec. 2 is used as the lower-bounding function. The TSP city graphs chosen were complete, with inter-city distances either uniformly or normally distributed over the interval [1, 100]. Three merits of performance, averaged over ten TSP instances, are used: (1) Average execution time measured in milliseconds; (2) Average speedup defined as the ratio (average T_1)/(average T_P); and (3) Average isoefficiency function, which is measured as the required rate of growth of essential work (as represented by average T_1 for different values

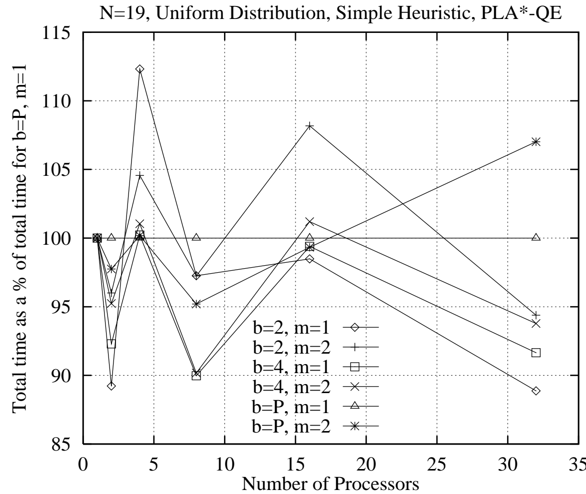


Figure 7: Effect of the parameters b, m and P on the total execution time T_P.

of N) with respect to P to keep the efficiency fixed at some value. Below we present performance results of our parallel startup scheme and load balancing strategies.

8.1 Effect of Parallel Startup Phase

In Fig. 7, we plot the execution time T_P 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 countered by decreasing T_{su} and hence smaller values of 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 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 b and m is minimal. Thus the choice of b and m is more crucial at low and intermediate work densities, i.e., for medium to large number of processors for a given problem size. Notice also that the improvement in total execution time obtained using a startup phase with maximum parallelism ($b = 2$) with respect to a completely sequential startup phase ($b = P$), grows with the number of processors used and is as high as 10% for $P = 32$ and 60% for $P = 256$ (the latter is not shown).

8.2 Effect of Load Balancing Strategies

In Fig. 8 we plot the speedup curves for PLA* using the RR, RC and QE strategies.[15] 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 QE strategy in enhancing scalability—speedups of PLA*-QE for $P = 256$, i.e., at an intermediate work density, and for $P = 1024$, i.e., at a lower work density, are about 25-50% and 30-100%, respectively, above the speedups of PLA*-RR and PLA*-RC for uniformly distributed data (Fig. 8(a)). The corresponding figures for normally distributed data are 15-120% and 20-185% (Fig. 8(b)). Note that the speedup values for normally distributed data with a small deviation are larger compared to that for uniformly distributed data for the same input problem size. The reason for this is that the number of essential nodes in the former case is greater due to there being more nodes of comparable quality because of the smaller deviation for the inter-city costs; thus the work density is larger leading to a more efficient parallel search.

Next in Fig. 9, we plot the speedup curves for the various load balancing methods using the LMSK heuristic, which is much tighter than the simple heuristic of Sec. 2; the curve for RR strategy could not be plotted because of memory overflow caused by large amounts of non-essential work. Here again we notice that the speedup of PLA*-QE is high and about 35-50% better on 1024 processors than that obtained using the RC strategy; the speedup improvement over the RR strategy for a smaller problem size ($N = 36$) with uniformly distributed data, is 123% (not shown). Furthermore, we observe from Fig. 9 that the QE strategy yields 11-22% better speedup for $P = 1024$ than the RR+RC and NA+RC methods, which combine previous quantitative and qualitative load balancing schemes. Also, note from Fig. 9(a) that we obtain an average speedup of about 985 on 1024 processors using our QE strategy, which represents a very high efficiency of 0.96. From Figs. 8 and 9, we see that the performance of the QE load balancing method remains excellent even when the heuristic function or the input data distribution are varied—the differences in speedup for the QE strategy in the various cases is due to differences in work densities, with larger speedups corresponding to higher work densities. This bears out our argument in Sec. 6.2 that the QE strategy is robust with respect to changes that affect the cost-wise distribution of nodes. Thus the QE strategy should be effective in the solution of a variety of COPs.

To test the effectiveness of our anticipatory qualitative load balancing

[15] Some large TSP instances included in the plots of Figs. 8, 9 and 10 could not be solved on processors less than a certain number P' due to memory overflow. Since the relative speedup $T_{P'}/T_{2.P'}$ at P' was found to be almost two (i.e., almost perfect), and since this relative speedup could have only improved at smaller number of processors had there been enough memory to solve these instances (because of lower overhead at smaller number of processors), we assumed the speedup at $P = P'$ to be P'.

Number of Processors	% Reduction in Execution Time
256	8.77
512	3.32

Table 1: Percentage reduction in execution time obtained using the anticipatory qualitative load balancing scheme in the QE strategy; the LMSK heuristic is used.

scheme in Sec. 6.2, we collected data for the performance of the QE strategy both with and without the anticipatory mechanism. At higher and intermediate work densities, the total overhead (idling and non-essential work) incurred by the base QE strategy (i.e., without the anticipatory mechanism) is observed to be quite low (about 2% on the average for $N = 44$ on 512 processors) and hence there is very little scope for improvement. Consequently the impact of the anticipatory scheme will not be much at these work densities. Therefore we collected performance data for lower work densities in which the idling and non-essential work overheads are relatively higher. Table 1 gives the average percentage reduction in execution time obtained using the QE strategy with the anticipatory qualitative load balancing scheme over that of the base QE strategy. This data was obtained for 11 randomly chosen instances with execution times of 10 seconds or less on 512 processors.[16] One would expect that for larger number of processors (i.e., at lower work density), the anticipatory scheme would prove more beneficial since there is more overhead to be reduced. However, the data in Table 1 suggests otherwise. The explanation for this is that when the work density decreases beyond a certain point, the amount of work available with the neighbors of a sink processor is quite less, and hence the near-neighbor anticipatory load balancing scheme is unable to fetch any work. Note that the anticipatory mechanism would prove more useful for lower granularity applications than higher granularity applications like TSP. This is because with lower node-expansion granularity, the effect of message latencies will become magnified.

Finally, in Fig. 10 we plot the isoefficiency curves for PLA*-RR, PLA*-RC and PLA*-QE.[17] 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 of $P.\log P$ and is much better than that of PLA*-RR and PLA*-RC. Also, note that the isoefficiency function of PLA*-QE is better

[16] Data for 1024 processors could not be collected due to non-availability of the machine.

[17] Since points corresponding exactly to the desired efficiency could not be obtained, we used the closest points available. The isoefficiency curves for the analytical lower and upper bounds were plotted by making them coincident with the isoefficiency curve for PLA*-QE at $P = 1$, and then letting them grow for larger values of P at the rates $P.\log P$ and $P.\log^2 P$, respectively—this determines the constants associated with the analytical bounds.

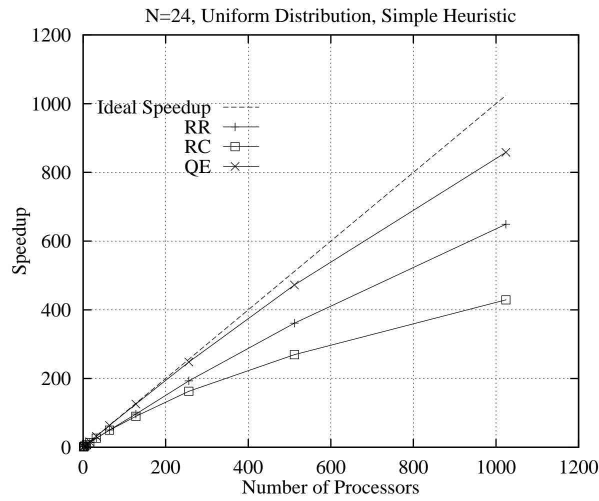
than $P.\log^2 P$ and the isoefficiency functions of all PLA* algorithms are worse than $P.\log P$. This supports our scalability analysis in the previous section.

9 Conclusions

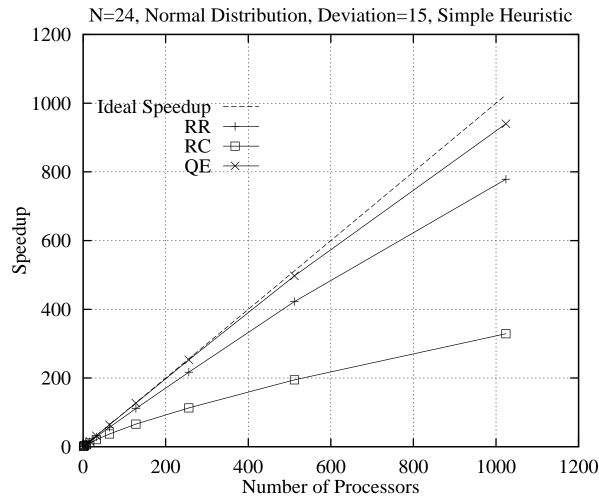
Although it is possible to obtain linear speedups in parallel A* search for sufficiently high work densities, at lower and intermediate work densities (i.e., for larger number of processors keeping problem size fixed), inefficiencies such as uneven work distribution and search of non-essential spaces gain prominence and cause the efficiency to deteriorate. In this work, we proposed a novel parallel startup scheme PAR_START and an efficient dynamic load balancing method, the quality equalizing (QE) strategy, to tackle these problems, and thus improve the scalability of parallel A* algorithms. PAR_START executes in an optimal time of $\Theta(\log P)$ and also achieves good initial load balance across processors. The QE strategy was shown to possess certain unique load balancing properties that aid in achieving good quantitative and qualitative load balance. Furthermore, the QE strategy includes anticipatory mechanisms to detect and correct load imbalances before their actual occurrence. Performance results show that the base QE strategy yields speedup improvements of 20-185% over the previously proposed RR and RC strategies, and 11-22% over the RR+RC and NA+RC strategies that combine prior quantitative and qualitative load balancing methods. For problem instances with small execution times (and hence for which idling and non-essential work overheads are high), the anticipatory qualitative load balancing scheme provides reductions in execution time of 3.3-8.8% over the base QE strategy. It is expected that for lower granularity applications like vertex cover and integer programming, the anticipatory scheme will provide even higher performance improvements, since then the effect of message latencies will become more pronounced. We used two different heuristic functions for the TSP problem, and inter-city costs that were either uniformly or normally distributed, to model the different cost-wise distribution of search-space nodes typically seen across various applications. It was observed that for comparable work densities, the QE strategy performs equally well for different node-cost distributions, and thus is a very robust load balancing method.

References

[1] T.S. Abdel-Rahman, “Parallel Processing of Best-First Branch-and-Bound Algorithms on Distributed Memory Multiprocessors,” Ph.D. Thesis, University of Michigan, Ann Arbor, 1989.

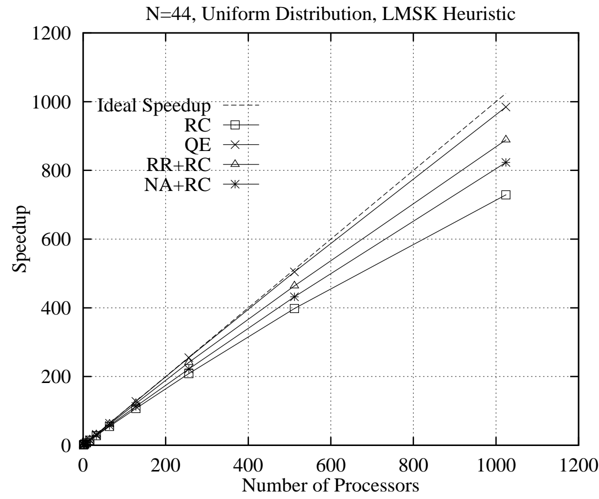


(a)

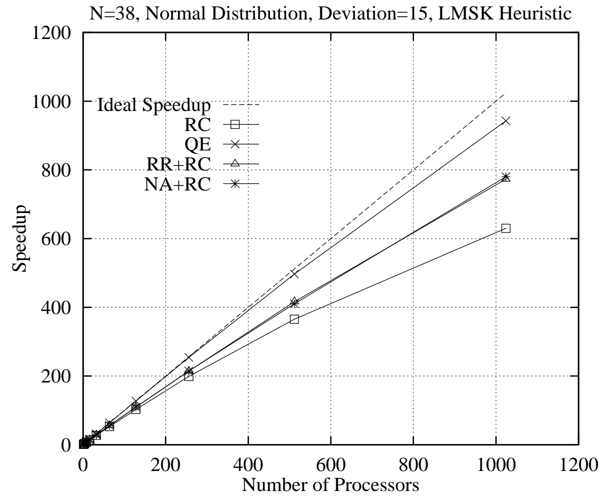


(b)

Figure 8: Speedup curves for PLA* algorithms using the simple heuristic of Sec. 2 and employing different load balancing strategies, for (a) uniformly distributed data and (b) normally distributed data.



(a)



(b)

Figure 9: Speedup curves for PLA* algorithms using the LMSK heuristic and employing different load balancing strategies, for (a) uniformly distributed data and (b) normally distributed data.

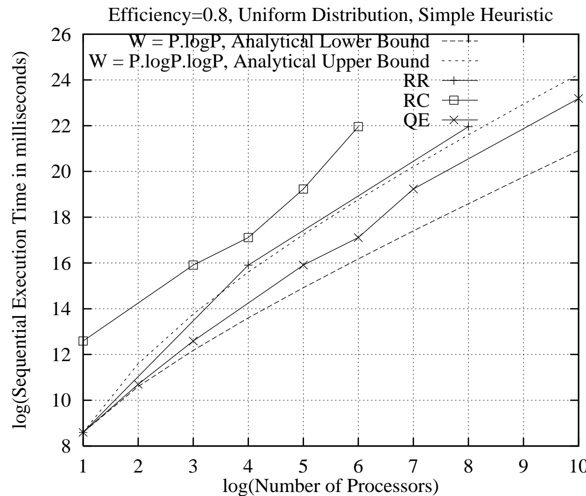


Figure 10: Isoefficiency curves for PLA* algorithms using the simple heuristic of Sec. 2 and employing different load balancing strategies, for uniformly distributed data.

[2] S. Anderson and M.C. Chen, "Parallel Branch-and-Bound Algorithms on the Hypercube," *Proc. Second Conference on Hypercube Multiprocessors*, pp.309-317, 1987.

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

[4] R.K. Brayton and F. Somenzi, "Minimization of Boolean Relations," *IEEE International Symposium on Circuits and Systems*, Vol.2, pp.738-743, May 1989.

[5] W.J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Trans. on Comp.*, Vol.39, No.6, June 1990.

[6] S. Dutt and N.R. Mahapatra, "Parallel A* Algorithms and their Performance on Hypercube Multiprocessors," *Seventh Int'l Par. Proc. Symp.*, pp.797-803, Apr. 1993.

[7] S. Dutt and N.R. Mahapatra, "Scalable Load Balancing Strategies for Parallel A* Algorithms," to be published in *Journal of Parallel and Distributed Computing*, 1994.

[8] J. Eckstein, "Parallel Branch-and-Bound Algorithms for General Mixed Integer-Programming on the CM-5", Technical Report TMC-257, *Thinking Machines Corp.*, Aug. 1993.

[9] M. Evett, J. Hendler, A. Mahanti and D. Nau, "PRA*: A Memory-Limited Heuristic Search Procedure for the Connection Machine," *Proc. Third Symp. on the Frontiers of Mass. Par. Computation*, pp.145-149, 1990.

[10] S.-R. Huang and L.S. Davis, "Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm," Proc. Eleventh Int'l Joint Conf. on Artificial Intelligence, pp.23-29, 1989.

[11] R.M. Karp and Y. Zhang, "A Randomized Parallel Branch-and-Bound Procedure," *Proc. ACM Ann. Symp. on Theory of Computing*, pp.290-300, 1988.

[12] 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.*, pp.122-126, Aug. 1988.

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

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

[15] R. Luling and B. Monien, "Load Balancing for Distributed Branch-and-Bound Algorithms," *Sixth Int'l Par. Proc. Symp.*, pp.543-548, 1992.

[16] 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.

[17] N.R. Mahapatra and S. Dutt, "Scalable Duplicate Pruning Strategies for Parallel A* Graph Search," *Fifth IEEE Symp. on Par. and Distr. Proc'g*, pp.290-297, Dec. 1993.

[18] G. Manzini and M. Somalvico, "Probabilistic Performance Analysis of Heuristic Search Using Parallel Hash tables," *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL, Jan 1990.

[19] D.L. Miller and J.F. Pekny, "Results from a Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems," *Operations Research Letters*, Vol.8, pp.129-135, 1989.

[20] V.N. Rao, "Parallel Processing of Heuristic Search," Ph.D. Thesis, University of Texas at Austin, July 1990.

[21] E. Rich, "Artificial Intelligence," *McGraw Hill*, New York, pp.78-84, 1983.

[22] V.A. Saletore, "A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks," *Proc. Fifth Distributed Memory Computing Conference*, 1990.

[23] N.A. Sherwani, "Algorithms for VLSI Physical Design Automation," *Kluwer Academic Publishers*, Norwell, Massachusetts, 1993.

[24] A.B. Sinha and L.V. Kale, "A Load Balancing Strategy for Prioritized Execution of Tasks," *Seventh Int'l Par. Proc. Symp.*, pp.230-237, Apr. 1993.

[25] D.R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *J. of the ACM*, Vol.31, No.1, pp.163-188, Jan 1984.