Scalable Global and Local Hashing Strategies for Duplicate Pruning in Parallel A* Graph Search*

NIHAR R. MAHAPATRA and SHANTANU DUTT

mahapatr@eng.buffalo.edu
Dept. of Electrical & Computer Eng.
State University of New York at Buffalo
Buffalo, NY 14260

dutt@ee.umn.edu
Dept. of Electrical Eng.
University of Minnesota
Minneapolis, MN 55455

Abstract

For many applications of the A* algorithm, the state space is a graph rather than a tree. The implication of this for parallel A* algorithms is that different processors may perform significant duplicated work if inter-processor duplicates are not pruned. In this paper, we consider the problem of duplicate pruning in parallel A* graph-search algorithms implemented on distributed-memory machines. A commonly used method for duplicate pruning uses a hash function to associate with each distinct node of the search space a particular processor to which duplicate nodes arising in different processors are transmitted and thereby pruned. This approach has two major drawbacks. First, load balance is determined solely by the hash function. Second, node transmissions for duplicate pruning are global; this can lead to hot spots and slower message delivery. To overcome these problems, we propose two different duplicate pruning strategies: (1) To achieve good load balance, we decouple the task of duplicate pruning from load balancing, by using a hash function for the former and a load balancing scheme for the latter. (2) A novel search-space partitioning scheme that allocates disjoint parts of the search space to disjoint subcubes in a hypercube (or disjoint processor groups in the target architecture), so that duplicate pruning is achieved with only intra-subcube or adjacent inter-subcube communication. Thus message latency and hot-spot probability are greatly reduced. The above duplicate pruning schemes were implemented on an nCUBE2 hypercube multicomputer to solve the Traveling Salesman Problem (TSP). For uniformly distributed inter-city costs, our strategies yield a speedup improvement of 13-35% on 1024 processors over previous methods that do not prune any duplicates, and 13-25% over the previous hashing-only scheme. For normally distributed data the corresponding figures are 135% and 10-155%. Finally, we analyze the scalability of our parallel A* algorithms on k-ary n-cube networks in terms of the isoefficiency metric, and show that they have isoefficiency lower and upper bounds of $\Theta(P \log P)$ and $\Theta(Pkn^2)$, respectively.

Index Terms: A* algorithm, branch-and-bound search, communication delay, duplicate pruning, graph search, isoefficiency function, k-ary n-cubes, parallel A*, scalability, traveling salesman problem.

*S. Dutt was supported by NSF grant MIP-9210049, and this research was done while N. Mahapatra was a Ph.D. student at the University of Minnesota.

1 Introduction

The A* algorithm [23] is a well-known generalized branch-and-bound search procedure that is widely used in the solution of many computationally demanding combinatorial optimization problems (COPs) [26]. 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 [7].

Many researchers have adopted a tree search-space formulation in problems like the Traveling Salesman Problem (TSP) [12, 22], the 15-puzzle problem [12], and the vertex cover problem [12], where the natural formulation is a graph. This implies that nodes representing identical subproblems are artificially defined to have different states. Thus identical-subproblem nodes remain undetected resulting in duplication of search. This leads to an inefficient, although easily parallelizable, sequential A*/best-first search algorithm: each processor explores a different part of the search space that is disjoint from the search spaces of other processors. Thus a "misleading" good speedup can be obtained. A graph formulation, on the other hand, will enable detection of identical subproblems in A* and hence avoid duplicated search. However, it is not as easily amenable to parallelization on distributed-memory machines as a tree formulation, since duplicate nodes may be generated in different processors.

A commonly used method for pruning inter-processor duplicates in graph search utilizes a suitable hash function to associate an "owner" processor with each distinct node of the search space. Every node upon generation is sent to its owner processor. Thus duplicate nodes arising in different processors are transmitted to the same owner where they are pruned [8, 20]. There are two significant shortcomings in this approach. First, load balance is determined solely by the hash function and hence may not be very effective. Second, node transmissions for duplicate pruning are global; this can lead to higher message contention (hot spots) and latency.

The overall goal of our work is to develop scalable high-performance parallel A* algorithms for distributed-memory machines that can be applied to most COPs. In our previous work on parallel A*, we developed efficient load balancing strategies [5, 6] that are equally useful for both tree- and graph-search problems, and demonstrated their superior performance over other competing methods [1, 11, 13, 17, 24]. In this paper, we develop efficient inter-processor duplicate pruning methods, and incorporate them in parallel A* algorithms to obtain high speedup over sequential A* graph search.

In Sec. 2, we first describe A* and then present an improved version used in our implementations. Next, in Sec. 3 we outline our application of A* to TSP, the test problem used to determine the efficacy of our parallelization techniques. In Sec. 4, we briefly discuss a generic approach to the parallelization of A*. In the following section, we describe the simple hashing-only scheme for duplicate pruning. Next, in Secs. 6 and 7, we present in detail two new duplicate pruning algorithms that overcome the aforementioned problems in the hashing-only scheme. The algorithm in Sec. 6 derives its efficiency from decoupling the task of duplicate pruning from that of load balancing by using a hash function for the former and a load balancing scheme for the latter, but involves global communication. The algorithm in Sec. 7 uses a novel search-space partitioning that allocates disjoint parts of the search space to disjoint processor-groups in the target machine for the purpose of duplicate pruning. As a result, any set of duplicate nodes arise in only a particular subcube. Hence communication for duplicate pruning is either confined to a subcube or occurs only between neighboring subcubes. This makes hot spots more unlikely thus enhancing the scalability of the parallel algorithm. In Sec. 8, we analyze in detail the new duplicate pruning algorithms, and derive good upper and lower bounds on their scalability on a very general class of architectures, the k-ary n-cubes. Section 9 presents empirical performance results of our algorithms. Conclusions are in Sec. 10.

2 The A* Algorithm

The A* algorithm is used to find a least-cost solution to a COP $\mathcal{P}$ [23]. Its operation can be viewed as a best-first search through a graph $\mathcal{G}$ of subproblems in which the original problem $\mathcal{P}$ occurs at the *root*. Other nodes of $\mathcal{G}$ correspond to subproblems derived from their parent nodes with leaf nodes representing 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.

Two lists of nodes are utilized: *OPEN* and *CLOSED*. Initially, *OPEN* contains just the *root* node, and *CLOSED* is empty. 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 *OPEN*, after checking to see if any duplicates exist in *OPEN* or *CLOSED*, in which case the generated node is pruned and any cost improvement is propagated to the duplicate node and its descendents. Finally, *best_node* is placed in *CLOSED*. Thus at any time, *OPEN* stores nodes that have been generated but not yet *expanded* (not had their children generated), while *CLOSED* contains nodes that have been expanded. The above 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*. Any time during SEQ_A*'s execution, the f' value of the current best solution node is denoted by *best_soln*. All nodes expanded by SEQ_A* are called *essential nodes*, and these include all nodes with cost less than or equal to the optimal solution cost;[1] all other nodes are *nonessential*. This method of using directly computed lower-bound cost estimates f' of nodes is the most prevalent way of guiding A* search. There are, however, other ways of guiding A* search that may reduce execution time in some applications as, for example, the search-space abstraction method used in [10]. In this latter method, instead of directly solving the COP $\mathcal{P}$ in the original search space, the problem is mapped into and solved in an "abstract" search space whose solution is then used to guide the search for a solution in the original space. In this paper, we are concerned with the former method of A* search.

In SEQ_A*, when a node is expanded (meaning that it is an essential node), all its children, essential as well as nonessential, are generated. We have devised a *partial expansion* scheme that generates only essential nodes. In this scheme, when a node in *OPEN* is expanded, only its best ungenerated child is selected for generation. Hence we will refer to the A* algorithm employing this scheme 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:

$$u.h'' = u.h', \text{ if u has not undergone any expansion, else} \tag{1}$$

$$u.h'' = \min((v.g - u.g + v.h') \text{ over all children } v \text{ of } u \text{ remaining to be generated}). \tag{2}$$

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 generated at this time, but the h'' cost of u is updated according to Eq. 2.[2] Subsequent expansions of u result in the generation of the child with the least f' cost among all children yet to be generated, 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 nonessential nodes are

[1]To be precise, some nodes with cost equal to the optimal solution cost may not be expanded by SEQ_A*.

[2]It turns out that the savings in the number of nodes generated more than compensates the extra memory used to store the h' costs of children that are not generated.

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., it might have only some children generated, but not all). All other nodes that were generated have been *completely expanded* (i.e., they have had all their children generated) and are discarded. Note that a solution node by definition is completely expanded.

Although the partial expansion scheme above does reduce the number of nodes generated during A* search substantially, for very large search spaces some technique for searching with limited or fixed amount of memory may become necessary; this may be the case even in a parallel setting when more memory is available in the multiple processors. In this situation, one can use any of the limited-memory schemes proposed for best-first search (BFS)/A* in previous work like MREC [25], MA* [2], and PRA* [8]. In these schemes, initially a best-first search is conducted, and on running out of memory, either the search switches to a depth-first search at the tip nodes (MREC) or some nodes from the search graph are retracted (MA* and PRA*). Since the focus of this paper is on duplicate pruning in parallel A* algorithms, we have not specifically addressed the problem of limited-memory search. However, as will become clear later, our duplicate pruning schemes are independent of whether or not a limited-memory scheme is applied, and hence they can be easily used in conjunction with the limited-memory schemes above as has been done, for instance, in [8].

3 Application of A* to TSP

The Traveling Salesman Problem can be posed as follows. Given a set $0, 1, .., N-1$ of cities and the associated inter-city distances, find the shortest tour that visits, starting from any arbitrary city, every city exactly once and then returns to the start city. We have formulated TSP as a graph-search problem, in which the state of a node is defined by a 3-tuple: [*start city, set of visited cities, present city*]. Initially only *root* with *root.state* $= [0, \{0\}, 0]$, exists. An expansion of a node $u \in \mathcal{G}$ yields a child v for each city that remains to be visited in u. This way a TSP tour is constructed by visiting an additional unvisited city, from the *present city*, in each expansion. We have used two different heuristic functions in our experiments to model different computational granularities and different extents of duplicate-node occurrence that may arise in various applications. In the first 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. The other heuristic function is the well-known LMSK heuristic [16].[3]

While our formulation of TSP reduces it to a graph-search problem (as will be shown shortly), previous sequential and parallel branch-and-bound methods employed to address TSP have used a tree-search formulation [12, 16, 22, 28]. In their formulation, the state of a node is defined by either: (1) a 1-tuple: [ordered list of cities visited], or (2) a 2-tuple: [set of edges currently in the tour, set of edges excluded from the tour]. Consider two nodes u and v, whose states according to the first tree formulation are defined by $u.state = [(0,1,2,3,4)]$ and $v.state = [(0,2,3,1,4)]$; and according to the second tree formulation by $u.state = [\{(0,1),(1,2),(2,3),(3,4)\},\emptyset]$ and $v.state = [\{(0,2),(2,3),(3,1),(1,4)\},\{(0,1)\}]$ (where (i,j) denotes the edge from city i to city j). Thus, u and v would be considered to represent two different subproblems (since their states are different) according to these formulations. However, both u and v represent the same subproblem, which can be posed as: find a minimal-cost path visiting exactly once all cities in the set $\{4,5,\ldots,N-1,0\}$, originating at 4 and terminating at 0. This statement is precisely captured by our formulation, according to which the states of both u and v are given by: $[0,\{0,1,2,3,4\},4]$. Duplicates are detected by the duplication check carried out in every iteration of A*. As a result, one of u or v gets pruned. It can similarly be shown that a graph-search formulation is more appropriate for problems such as the 15-puzzle and the vertex cover problems that have previously been tackled using a tree-search formulation [12].

4 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* and *CLOSED* lists. Initially, processor 0 has the *root* node. Subsequently, the child nodes generated by expanding nodes at a processor are hashed to specific processors determined by a suitable hash function. In time, all processors receive nodes. This time period from the start of execution till the time by which all processors receive at least one node is termed the *startup phase*. Pro-

[3]Since the focus of this paper is on the design and evaluation of general duplicate pruning strategies for parallel A* algorithms that can be applied to most COPs, we have employed simple heuristic functions as opposed to a tighter heuristic function possible for TSP [21]. In Secs. 6 and 7, we will see that while a different heuristic function can affect the total number of duplicate nodes generated, it will not have any impact on the pruning capabilities of our strategies since they prune all duplicates.

cessors broadcast any improvements in *best_soln*, which is maintained consistent in all processors. Apart from solution broadcasts, processors interact to redistribute work for better processor utilization, to send or receive cost updates, to detect and prune *inter-processor duplicates* and to detect termination of the algorithm. Such algorithms can be characterized as *parallel local A** (PLA*) algorithms.

In contrast to a parallel local A* algorithm, a *parallel global A** (PGA*) algorithm uses global *OPEN* and *CLOSED* lists or multiple lists that are kept consistent across processors, and processes nodes in a strictly global best-first manner. Such an algorithm is suitable only for shared-memory machines [12] and does not scale up well with the number of processors, since contention for the global lists or the cost of maintaining consistent lists becomes excessive. PLA* algorithms, on the other hand, can be implemented on distributed-memory machines and have better scalability; hence we focus on them. However, the use of multiple inconsistent *OPEN* lists and a distributed-memory implementation introduce a number of inefficiencies in a PLA* algorithm: (1) *Starvation*: when processors run out of work and idle. (2) *Nonessential work*: when processors expand nonessential nodes; it arises because processors perform local rather than global best-first search. (3) *Memory overhead*: caused by the generation and storage of nonessential nodes. (4) *Duplicated work*: associated with processors pursuing duplicate search spaces and is due to *inter-processor duplicates* (duplicate nodes that arise in different processors), when the search space is a graph.

Load balancing strategies are used to tackle the inefficiencies of starvation and nonessential work (and hence that of memory overhead), while duplicate pruning strategies are required to minimize duplicated work. In [5, 6], we developed the quality equalizing (QE) strategy in which processors utilize load information at neighboring processors to effect quantitative and qualitative load balance via nearest-neighbor work transfers. The type of load information and the manner in which it is used give the QE strategy certain unique properties that enable it to effectively reduce starvation and nonessential work. This paper presents novel duplicate pruning methods that eliminate duplicated work in parallel A* graph-searches. We use these methods in conjunction with the QE strategy to obtain near-linear speedup and high scalability.

Note that for PGA* algorithms, duplicate pruning is straightforward; any duplicate nodes arising during A* search can be readily detected in the global *OPEN* and *CLOSED* lists. Hence duplicate pruning is an issue only in PLA* algorithms. In the next two sections we discuss duplicate pruning schemes for PLA* algorithms—Sec. 5 describes a previous commonly-used duplicate pruning strategy, and Secs. 6 and 7 detail our new duplicate pruning strategies. For termination detection, we have used an optimal spanning-tree based algorithm which can be found in [5, 19].

5 Previous Duplicate Pruning Strategy—Global Hashing of Nodes (GOHA)

The only duplicate pruning technique proposed previously applies a hash function H to a *key* κ derived from the state of a node, to associate with each node a particular processor $H(\kappa)$, which we will call its *owner* processor [8, 20]. No particular hash function is indicated in previous work [8, 20]. Therefore to make a fair comparison between this and other hash-function-based methods to be presented later, we have used the same hash function in all of them. The hash function used is based on a *multiplication method*, which has been observed to hash a random key (node state) to any of the P slots (processors) with almost equal likelihood [3]. Specifically, the hash function is defined by:

$$H(\kappa) = \lfloor P \cdot (\kappa \cdot A - \lfloor \kappa \cdot A \rfloor) \rfloor, \tag{3}$$

where P is the total number of processors, $(\kappa \cdot A - \lfloor \kappa \cdot A \rfloor)$ is the fractional part of $\kappa.A$ and can be considered to be a random number in the range $[0, 1)$, A is a constant[4] in the range $(0, 1)$, and the key κ of a node is defined by a binary vector as:

$$\kappa = (b_{N-1}, b_{N-2}, \ldots, b_1, b_0) \mid b_i = 1 \text{ iff city } i \text{ has been visited, } 0 \leq i < N, \tag{4}$$

where N is the total number of cities in TSP. So whenever a node is expanded in any processor, all the generated children are hashed to their respective owner processors. These children nodes are then inserted into the *OPEN* list of the owner processor after checking to see if a duplicate node already exists in the owner's *OPEN* or *CLOSED* lists. Since the key of a node is derived from its state, all nodes with the same state, i.e., all duplicate nodes, will be hashed to the same processor. This way all duplicate copies of a node arising in different processors are detected and pruned in the owner processor of that node. The PLA* algorithm employing this duplicate pruning strategy will be known as PLA*-GOHA for GlObal HAshing.

There are two major drawbacks in this method: 1) Load balance is determined solely by the hash function. Since the hash function basically seeks to randomize the destination of hashed nodes, the load balancing algorithm is in effect identical to a global random communication strategy [11].

[4] We chose $A = (\sqrt{5} - 1)/2$, the *golden ratio*, since the hash function is known to work well with this value of A [14].

In our previous work on parallel A* we have shown that the random communication strategy is not very effective in achieving a good load balance [5, 6]. 2) Node transmissions for duplicate pruning are global; this increases message latency and the likelihood of hot spots in the network.

In the next two sections we present two different duplicate pruning strategies that remedy the above deficiencies. These strategies differ in their communication patterns and consequently in their scalability. As noted earlier, we have developed the QE load balancing strategy and have demonstrated its superior performance over other well-known methods in [5, 6]. Therefore in both of the following algorithms, we use the QE strategy to achieve load balance.

6 Global Hashing of Nodes and QE (GOHA&QE)

In the algorithm PLA*-GOHA, the task of load balancing was combined with duplicate checking, i.e., a processor receiving a hashed node not only checked for any existing duplicates of the node, but also expanded the node. In this scheme, we decouple the two tasks by allowing processors to participate in load balancing, using the QE strategy, after performing duplicate checking for nodes that have been hashed to them. Therefore when a node u is first generated, it is hashed to its corresponding owner processor. Recall from Sec. 2 that we use a partial expansion scheme in the sequential A* algorithm on each processor. Hence node u may be partially expanded in its owner processor and partially in other processors to which the load balancing algorithm transfers it. 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.

In Fig. 1 we illustrate the way in which pruning and cost updates take place. In Fig. 1(a) node u is shown expanded partially in its owner processor P_2 and partially in another processor P_3 to which it was subsequently donated. So when the duplicate node v of u is generated in processor P_1, it gets hashed to its owner processor P_2. As a result node v is pruned and the cost improvement is propagated to node u and its descendents in P_2 and P_3. Thus the cost of node u in P_2 and P_3 improves by 8 from 68 to 60; other descendents of node u in both processors also receive the same cost improvement. By decoupling the task of duplicate checking from load balancing, we are able to achieve good load balance across processors, while performing complete duplicate pruning. This new algorithm is called PLA*-GOHA&QE for GlObal HAshing plus the QE strategy.

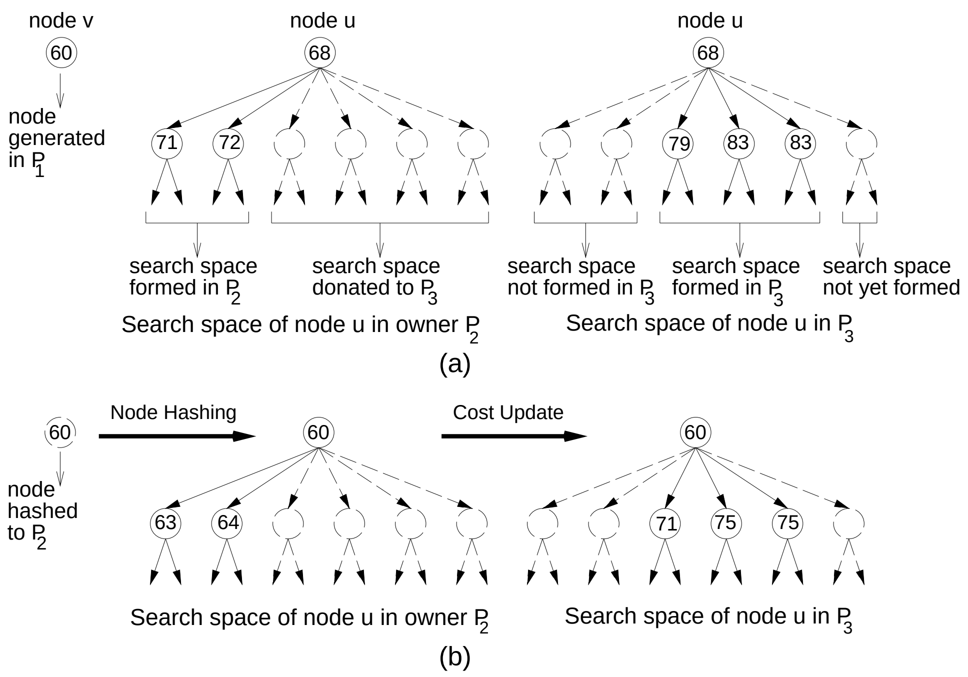

node v
60
node generated in P1
node u
68
71
72
search space formed in P2
search space donated to P3
Search space of node u in owner P2
node u
68
79
83
83
search space not formed in P3
search space formed in P3
search space not yet formed
Search space of node u in P3
(a)
60
node hashed to P2
Node Hashing
60
63
64
Search space of node u in owner P2
Cost Update
60
71
75
75
Search space of node u in P3
(b)

Figure 1: (a) Node u is partially expanded in its owner processor P_2 and another processor P_3. A duplicate node v of u is generated in processor P_1. (b) Node v gets hashed to its owner processor P_2 and is thereby pruned. A cost improvement of 8 is propagated to node u and its descendents in P_2 and P_3.

From Fig. 1 we notice that load balancing entails storing copies of nodes transferred in both donor and acceptor processors to facilitate cost updates. Thus it may seem that GOHA&QE would require more memory than GOHA. However, our experiments show that, because of the significantly larger number of nonessential nodes generated in GOHA due to poorer load balance, its average memory requirement is about 10% more than that of GOHA&QE.[5] The duplicate pruning scheme described next also utilizes the QE load balancing strategy and was found to have a memory requirement similar to that of GOHA&QE.

7 Local Hashing of Nodes and QE (LOHA&QE)

In *global hashing schemes* such as GOHA and GOHA&QE, a node may be hashed to any of the P processors. Therefore there exists a high degree of contention for communication links and router buffers leading to slower message delivery. The situation is further aggravated for lower granularity applications since they correspond to higher hashing rates. To circumvent this problem, we have devised a novel *search-space partitioning scheme* that partitions the search space into disjoint

[5]This data was obtained on a 1024-processor hypercube for 10 44-city TSP instances with uniformly distributed inter-city costs.

components and allocates these components to disjoint processor groups in the target architecture. This ensures that any set of duplicate nodes arise only within a particular processor group. As a result, complete duplicate pruning is achieved with node transmissions either confined to a processor group, or occurring only between neighboring processor groups. The PLA* algorithm employing this duplicate pruning method is referred to as PLA*-LOHA&QE for LOcal HAshing plus the QE strategy. In Sec. 7.1, we first illustrate the basic LOHA&QE scheme by applying it to solve TSP on a hypercube, and then in Secs. 7.2 through 7.5 we generalize it to a large set of COPs and a very broad class of multicomputer architectures called k-ary n-cubes.

7.1 Application to TSP on Hypercube

Define the *level* of a node to be the number of search-space edges between it and the *root* node. Then for our 3-tuple graph formulation of Sec. 3 for TSP, the level of a node is simply $|\text{set of visited cities}| - 1$. Since any set of duplicate nodes have the same set, and hence number, of visited cities, they also belong to the same level. Moreover, when a node at level l is expanded, where $0 \le l < N$, the generated children will all belong to the next higher level $l+1$ since they correspond to an additional city visited. We map adjacent groups or *clusters* of levels to neighboring subcubes in a Hamiltonian cycle of s disjoint subcubes embedded in the hypercube. The subcubes are numbered 0 through $s-1$ in a Gray-code ordering. For a cluster size of c, the ith cluster comprises levels $c \cdot i, c \cdot i + 1, \ldots, c \cdot (i+1) - 2$, called *non-boundary levels*, and level $c \cdot (i+1) - 1$, called *boundary level*, and is mapped to the $(i \bmod s)$th subcube.

For the search space of TSP, the total number of distinct nodes at different levels increases from level 0 to level $N/2$ and then decreases from level $N/2+1$ to level N, with levels l and $(N-l+1)$ having the same number $\frac{N!}{(N-l)!(l-1)!}$ of nodes, where N is the number of cities in TSP. The variation in the number of essential nodes at different levels, the *essential-node distribution*, will be somewhat similar to this and is qualitatively depicted in Fig. 2(a) for $N = 31$. We *fold* the sequence of clusters on the subcube ring f times, where $f \ge 1$ and is referred to as the *folding* of the mapping, so that a subcube receiving a smaller cluster (a cluster with small number of nodes), also gets allocated a bigger cluster, thus resulting in good static load balance across subcubes. Then the number of subcubes s is related to the cluster size c and the folding f by:

$$s = \min\left(P,\ 2^{\left\lceil \log_2\left(\frac{N+1}{(f+1)\cdot c}\right)\right\rceil}\right), \tag{5}$$

where $N+1$ is the total number of levels, and $(f+1)\cdot c$ (or $f \cdot c$) is the number of levels assigned to a subcube.[6] This assignment of clusters to subcubes is illustrated in Fig. 2(a) for $N = 31$, $c = 4$, $f = 1$ and $s = 4$, with the clusters labeled $c_0, \ldots, c_7$, the single fold labeled f_0, and the subcubes labeled $s_0, \ldots, s_3$. Note that this cluster-to-subcube assignment affects only the initial placement of nodes across processors; any load imbalances within and between adjacent subcubes is tackled by the dynamic load balancing algorithm in a manner similar to that in GOHA&QE.

The operation of PLA*-LOHA&QE can now be described as follows. Initially, processor 0, which belongs to subcube 0, has the *root* node, which belongs to level 0 and cluster 0. Subsequently, when nodes belonging to the non-boundary levels of any cluster i assigned to subcube $j = i \bmod s$ are expanded, the generated child nodes, which belong to cluster i and hence subcube j, are hashed to an owner processor H_j in subcube j:

$$H_j(\kappa) = origin_j + \lfloor \frac{P}{s} \cdot (\kappa \cdot A - \lfloor \kappa \cdot A \rfloor) \rfloor \qquad (6)$$

Here $origin_j$ denotes the smallest label of any processor in subcube j and the key κ is again given by Eq. 4. The expansion of boundary-level nodes in cluster i yields child nodes that belong to the next cluster $i + 1$; these child nodes are hashed to their owner processor $H_{(j+1)\bmod s}$ in the next adjacent subcube $(j + 1) \bmod s$ in the Gray-code order. This communication pattern involving intra- and inter-subcube node transmissions is shown in Fig. 2(b). Since any set of duplicate nodes belong to the same level, and hence the same subcube, they are also hashed to the same processor within that subcube, where duplicate detection and pruning takes place. This way complete duplicate pruning is achieved with node transmissions occurring only between neighboring subcubes and within a subcube. Hashing schemes with such localized communication are called *local hashing schemes*.

To preserve the above communication locality feature, we permit nodes in any cluster to move, during load balancing, only within their assigned subcube, with the exception of boundary-level nodes which are allowed to migrate to the next adjacent subcube as well. Without this constraint, the load balancing algorithm may transfer some node u to a processor that is distant from the subcube to which the children of node u belong, thus necessitating non-local hashing for the children of u.

7.2 Generalization Across COPs and Architectures

We now describe the application of LOHA&QE to a variety of COPs on a family of architectures called k-ary n-cube tori. A COP consists in finding a permutation or combination of N variables

[6]The expression $2^{\lceil \log_2 \left(\frac{N+1}{(f+1) \cdot c} \right) \rceil}$ is used instead of $\frac{N+1}{(f+1) \cdot c}$ because s must be a power of 2.

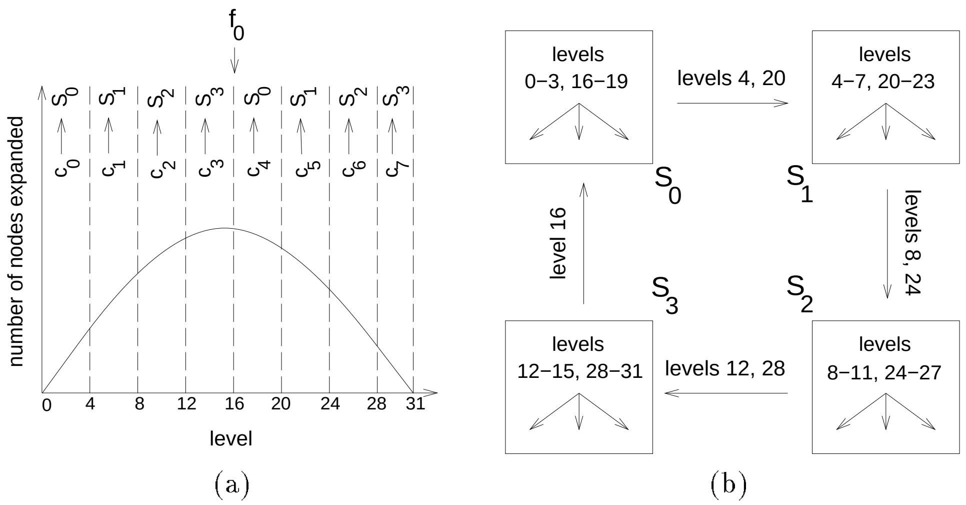


Figure 2: (a) Hypothetical essential-node distribution curve, and the assignment of clusters to subcubes for static load balance, and (b) Intra- and inter-subcube node transmission pattern for $N = 31$, $c = 4$, $f = 1$ and $s = 4$, in LOHA&QE.

that is optimal with respect to some objective function [15]. Its search space has nodes whose states are basically defined by an ordered or unordered subset of the N variables. LOHA&QE is applicable to those COPs whose search-space nodes have states defined by the same variables as those in their parents' state plus one more (or one less) new variable. Besides TSP, this is true of search spaces of many COPs, including integer programming [7], vertex cover, min-cut partitioning and global routing [26]. In such search spaces, every node has a unique level, i.e., the search space is a *levelized graph*. Thus, if the *root* node's state is defined by a single variable, the level of any node is simply one less than the number of variables in its state. Since duplicate nodes by definition have the same state and hence the same number of state variables, they also belong to the same level. Thus LOHA&QE can be applied to such COPs in the same way as to TSP.

The family of k-ary n-cube tori is a very general class of architectures that includes rings ($n = 1$), 2-D tori ($n = 2$), 3-D tori ($n = 3$) and hypercubes ($k = 2$) as special cases [4]. Here, n is referred to as the dimension and k the radix of the architecture. The number of processors $P = k^n$. Every processor 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_1 neighbors along each dimension, where $m_1 = 2$ for $k > 2$, and $m_1 = 1$ for a hypercube ($k = 2$). Figure 3(a) shows how k-ary n-cubes are recursively constructed starting from a single processor. Since the furthest processor along any dimension is at a distance[7] of $k/2$, and since there are n dimensions, the diameter is $n \cdot k/2$. Our

[7]In this paper, we assume k is even for simplicity; the discussion and results obtained are easily extendible to the

discussion below on k-ary n-cube tori can be easily extended to k-ary n-cube meshes, which differ from the former only in that they have no end-around connections in any dimension; the linear array ($n = 1$) and 2-D mesh ($n = 2$) are special cases of the latter architectures. Henceforth, unless otherwise specified, our discussion will pertain to k-ary n-cube tori.

The search-space partitioning scheme is extended to k-ary n-cube tori simply by folding the sequence of clusters on a Hamiltonian cycle $\mathcal{C}$ of disjoint $\hat{k}$-ary $\hat{n}$-cube meshes[8] embedded in the target k-ary n-cube torus, where $\hat{k}$ divides k and $0 \leq \hat{n} \leq n$. To obtain this embedding, a two-step process, illustrated in Figs. 3(b) and (c) for $k = 8, n = 3, \hat{k} = 2$, and $\hat{n} = 2$, is used: (1) First, a linear array of disjoint 2-ary 2-cube meshes is embedded in an 8-ary 2-cube torus. This is done by first partitioning the 8-ary 2-cube torus into a 4-ary 2-cube mesh of 2-ary 2-cube submeshes each of which is assigned a 2-dimensional radix-4 label. The partitioning is done so that processors with labels (j_1, j_0) in the 8-ary 2-cube torus belong to the submesh with label (i_1, i_0), where $2 \cdot i_l \leq j_l < 2 \cdot (i_l + 1), 0 \leq i_l < 4$ and $2 < l \leq 0$. Then, a 2-dimensional radix-4 Gray code (obtained as an extension of reflected binary Gray codes) is used to order the 2-ary 2-cube submeshes in a snake-like fashion to form the required linear array (see Figure 3(b)). Note that consecutive submeshes in this Gray code ordering are physically adjacent. We denote the first mesh in this linear array by A and the last mesh by Z. (2) Next, we form a Hamiltonian cycle $\mathcal{C}'$ of 8-ary 2-cube tori on the target 8-ary 3-cube torus using 1-dimensional radix-8 Gray codes. Now, to obtain the Hamiltonian cycle $\mathcal{C}$ of 2-ary 2-cube meshes, we alternately consider the two Z submeshes and the two A submeshes in adjacent 8-ary 2-cube tori in $\mathcal{C}'$ to be neighbors as shown in Fig. 3(c).

The above two-step embedding procedure may be generalized as follows: (1) First, a linear array of disjoint $\hat{k}$-ary $\hat{n}$-cube submeshes is embedded in a k-ary $\hat{n}$-cube torus by first partitioning the k-ary $\hat{n}$-cube torus into a $\frac{k}{\hat{k}}$-ary $\hat{n}$-cube mesh of these submeshes and assigning each submesh an $\hat{n}$-dimensional radix-$\frac{k}{\hat{k}}$ label. Here each $\hat{k}$-ary $\hat{n}$-cube submesh with label $(i_{\hat{n}-1}, i_{\hat{n}-2}, \ldots, i_0)$ corresponds to processors with labels $(j_{\hat{n}-1}, j_{\hat{n}-2}, \ldots, j_0)$ in the k-ary $\hat{n}$-cube torus, where $\hat{k} \cdot i_l \leq j_l < \hat{k} \cdot (i_l + 1), 0 \leq i_l < \frac{k}{\hat{k}}$ and $\hat{n} < l \leq 0$. Then the submeshes are formed into a linear array by using $\hat{n}$-dimensional radix-$\frac{k}{\hat{k}}$ Gray codes. As before, we denote the first submesh in this linear array by A and the last submesh by Z. (2) Next, a Hamiltonian cycle $\mathcal{C}'$ of k-ary $\hat{n}$-cube tori is formed in the target k-ary n-cube torus using $(n - \hat{n})$-dimensional radix-k Gray codes. The Hamiltonian cycle $\mathcal{C}$ of $\hat{k}$-ary $\hat{n}$-cube submeshes is formed by alternately considering the Z submeshes and A

case when k is odd.

[8]Note that, when $\hat{k} = k$, the $\hat{k}$-ary $\hat{n}$-cubes are tori and not meshes.

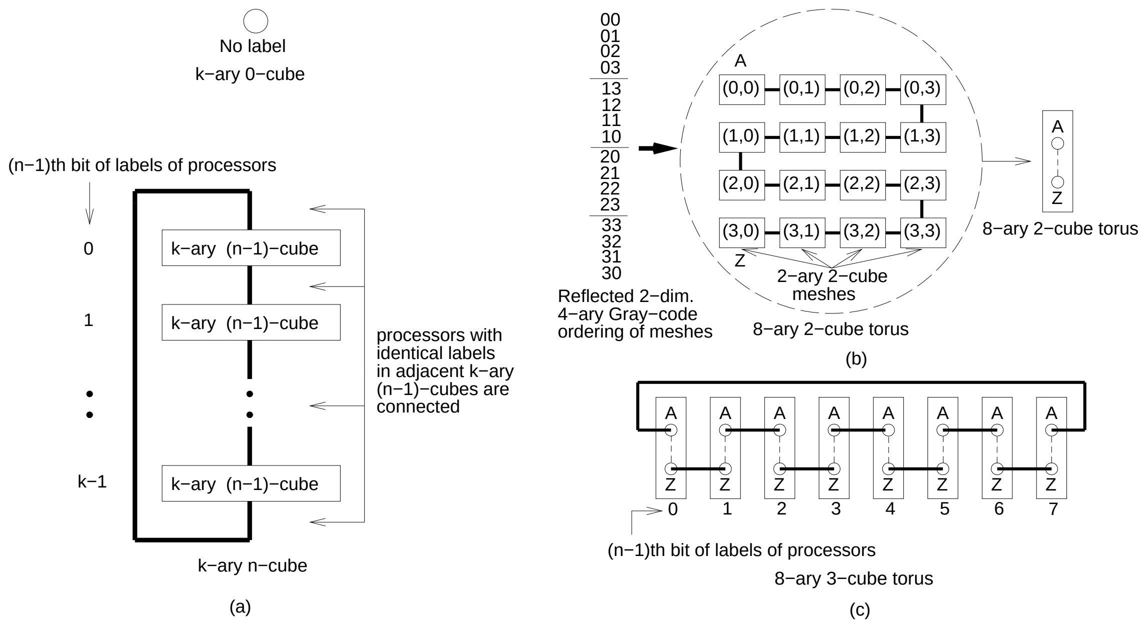


Figure 3: (a) Recursive construction of a k-ary n-cube torus starting with a single processor, where $k \geq 2$ and $n > 0$. (b) Embedding of a linear array of 2-ary 2-cube meshes in an 8-ary 2-cube torus using a reflected 2-dimensional 4-ary Gray code. (c) Embedding of a Hamiltonian cycle of 2-ary 2-cube meshes in an 8-ary 3-cube torus using the embedding of (b), and a Hamiltonian cycle embedding of 8-ary 2-cube tori in an 8-ary 3-cube torus.

submeshes in adjacent k-ary $\hat{n}$-cube tori to be neighbors.[9] The expression corresponding to Eq. 5 for the number of disjoint processor groups s then becomes:

$$s = \frac{k^n}{\hat{k}^{\hat{n}}} = \min\left(P,\ \frac{N+1}{(f+1)\cdot c}\right). \tag{7}$$

All that remains now to be discussed to implement LOHA&QE is to determine appropriate values for the different parameters in the above equation. The choice of cluster size c and folding f is considered in Secs. 7.3 and 7.4, and the choice of number of subcubes s and subcube dimension $\hat{n}$ (and hence radix $\hat{k}$) is taken up in Sec. 8.6. Duplicate pruning is achieved as before in the case of hypercubes, with communication occurring either within or between neighboring $\hat{k}$-ary $\hat{n}$-cube meshes.

7.3 Clustering

We now consider the choice of cluster size by analyzing its effect on the throughput requirement of duplicate pruning for intra- and inter-subcube channels. Let the rate at which nodes are generated

[9]While a linear array of $\hat{k}$-ary $\hat{n}$-cube meshes can always be embedded on a k-ary $\hat{n}$-cube torus, it turns out that only a linear array (and not a Hamiltonian cycle) of k-ary $\hat{n}$-cube tori can be embedded on a k-ary n-cube torus when k is odd. However, this will only marginally impact performance.

to be hashed from any processor be λ. Assume that there are two unidirectional channels between any two neighboring processors, as it is the case on our target machine, the nCUBE2. First, consider intra-subcube channel throughput. Since each processor in a subcube of size $Q = P/s = \hat{k}^{\hat{n}}$ hashes nodes at rate λ, and since each hashed node travels an average distance equal to half the diameter of the subcube, i.e., $m_2 \cdot \hat{n} \cdot \hat{k}/4$ hops, where $m_2 = 2$ when $\hat{k} < k$ (i.e., when each subcube is a mesh) and $m_2 = 1$ when $\hat{k} = k$ (i.e., when each subcube is a torus), the throughput required per channel for a total of approximately $Q \cdot m_1 \cdot \hat{n}$ unidirectional channels within the subcube is $\frac{Q \cdot \lambda \cdot m_2 \cdot \hat{n} \cdot \hat{k}/4}{Q \cdot m_1 \cdot \hat{n}} = \frac{\lambda \cdot m_2 \cdot \hat{k}}{4 \cdot m_1}$, where recall that $m_1 = 2$ for $k > 2$ and $m_1 = 1$ otherwise. When $\hat{k} = k$, each subcube is a torus, and in this case the $\frac{\lambda \cdot m_2 \cdot k}{4 \cdot m_1}$ intra-subcube channel throughput requirement in LOHA&QE is the same as that in a global hashing scheme, since the latter is basically LOHA&QE with just one subcube (i.e., $\hat{k} = k$ and $\hat{n} = n$). Thus the intra-subcube channel throughput requirement in LOHA&QE is either less than or equal to that in global hashing schemes and is independent of the cluster size.

Now to compute the throughput required for channels between subcubes, recall that only one out of every c levels is transmitted to the next subcube in the Gray-code order, and hence these boundary node transmissions are unidirectional. Therefore the rate[10] at which nodes are transmitted to the adjacent subcube is λ/c. Since there are a total of either $\hat{k}^{\hat{n}-1} = Q/\hat{k}$ or Q channels from the ith to the $((i+1) \bmod s)$th subcube, depending upon whether the subcubes lie in the same k-ary $\hat{n}$-cube (for example, two adjacent 2-ary 2-cube meshes in the 0th 8-ary 2-cube of Fig. 3(c)) or separate (but adjacent) k-ary $\hat{n}$-cubes (for example, the Z subcubes in the 0th and 1st 8-ary 2-cubes of Fig. 3(c)), respectively, the maximum throughput required per channel for channels between subcubes is $\frac{Q \cdot (\lambda/c)}{Q/\hat{k}} = \lambda \cdot \hat{k}/c$. When $\hat{k} = k$, there are always Q channels from the ith to the $((i+1) \bmod s)$th subcube and thus the maximum channel throughput requirement becomes λ/c. Hence to avoid a bottleneck in inter-subcube node transmission, the cluster size c should be large enough so that $\lambda \cdot \hat{k}/c$ (or λ/c for $\hat{k} = k$) does not exceed the channel bandwidth.

Since the cluster size can be varied to adjust inter-subcube throughput requirement, the intra-subcube throughput requirement of LOHA&QE determines whether a communication bottleneck will occur for a given channel bandwidth. Recall that this determining throughput requirement of LOHA&QE is the same as those of GOHA and GOHA&QE (in which $\hat{k} = k$), and is proportional to λ and $\hat{k}$. The hashing rate λ is inversely proportional to the node-expansion granularity of an application. Hence for the same channel bandwidth, LOHA&QE can be used in applications with

[10]This assumes that the static assignment of levels to subcubes achieves a reasonably good load balance (which, as we will see next in Sec. 7.4, is a valid assumption), so that the message traffic corresponding to inter-subcube node transmission is directly related to the fraction of all levels that are at the boundary.

node-expansion granularities smaller by a factor of $k/\hat{k}$ than those in which GOHA or GOHA&QE can be used before encountering communication bottlenecks.

Another consideration in the choice of cluster size in LOHA&QE is its effect on startup idling. Recall that in the beginning, there is only the single *root* node which belongs to the 0th cluster. Subsequently, nodes at any cluster become available only after low-cost nodes at the boundary level of the previous cluster have been generated and chosen for expansion. Thus a large cluster size would mean that processors in higher-labeled subcubes, which are assigned higher-labeled clusters, will begin receiving nodes much later than lower-labeled subcubes, leading to increased startup idling. Thus the cluster size should be chosen as small as possible, without having $\lambda \cdot \hat{k}/c$ exceed the channel bandwidth. In our implementation, we used a cluster size of one, since for the node-expansion granularity of TSP, λ is small enough so that this does not cause any router-buffer overflow problems (which will occur when the required throughput exceeds the channel bandwidth).

7.4 Folding

Recall that folding is meant to achieve static load balance across subcubes. However, this may not necessarily be always possible to achieve, even when the essential-node distribution curve is symmetric about the center level as in Fig. 2(a). In fact, for folding to be effective, the essential-node distribution curve should be linear (or approximately linear) as well. For instance, a convex essential-node distribution curve like the one in Fig. 2(a) results in the center subcubes (s_1 and s_2 in Fig. 2(a)) receiving the maximum number of nodes and the extreme subcubes (s_0 and s_3 in Fig. 2(a)) the minimum number of nodes, while a concave essential-node distribution curve has an opposite effect. In any case, increasing the number of folds helps reduce static load imbalance. However from Eq. 5, a large folding also means a small number of subcubes or a large subcube size, which will cause increased contention and hence limit the scalability of the algorithm. Our tests indicate that three folds yield a reasonably good load balance for TSP; the number of extra folds that we use beyond three is fixed by the number of subcubes, since the cluster size was predetermined from other constraints (stated in Sec. 7.3) to be one. In the next section, we will determine analytically how the number of subcubes s should be varied with P to optimize the scalability of LOHA&QE.

To determine the degree of static load imbalance that exists when folding is used, we collected essential-node distribution data for 10 random TSP instances for $N = 44$; the corresponding average essential-node distribution curve is plotted in Fig. 4. Note that the curve peaks at level 16, instead

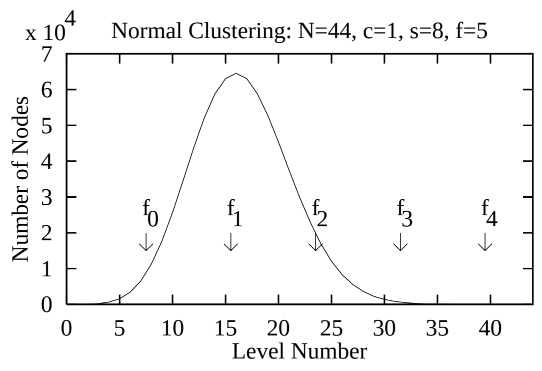


Figure 4: Empirical essential-node distribution curve used to determine static load imbalance under normal clustering.

of at the center (level 22) as in the hypothetical curve of Fig. 2(a). This is because nodes at lower levels are likely to be less expensive than nodes at higher levels. Hence a larger proportion of the total nodes is likely to be essential at lower levels than at higher levels. Let $c = 1$ and $s = 8$ (and hence $f = 5$ from Eq. 5). If w_i denotes the number of essential nodes assigned to subcube i and w_{max} the maximum among all w_i's, then the overhead due to static load imbalance across subcubes is $\sum_{i=0}^{s-1}(w_{max} - w_i)$. It turns out that the average static load imbalance for the data of Fig. 4 is about 0.75% of the total number of essential nodes. The reason for such a low load imbalance is that the essential-node distribution curve is approximately linear and symmetric about level 16 where fold f_1 lies, so that different subcubes receive almost equal number of nodes. Note that the dynamic load balancing algorithm will help further reduce the little load imbalance that remains.

Since it may not always happen that the essential-node distribution curve is nearly as linear and symmetric about a level that is a multiple of s (so that a fold lies at that level) as in Fig. 4, we consider a hypothetical essential-node distribution curve that peaks at level 20 in Fig. 5(a). For the same values of c and s, it turns out that the percentage static load imbalance for this case is about 6.7%. For a more non-linear curve the load imbalance will be higher. To address this problem, we have developed a refinement of the cluster-to-subcube assignment scheme described next that almost completely eliminates static load imbalance irrespective of the shape of the essential node distribution curve.

7.5 Refined Clustering

This scheme uses average essential-node distribution data, which may be obtained either empirically (as in Fig. 4) or analytically. It is clear from the above discussion that if the essential-node distribution curve is piecewise linear and symmetric about the center level, normal clustering as previously discussed will effect perfect static load balance. The refined clustering method emulates this idea by making the distribution of essential nodes across clusters, i.e., the cluster essential-node distribution curve, piecewise linear and symmetric about the center cluster. The key to the refinement consists in devising a search-space partitioning scheme in which cluster boundaries are not necessarily coincident with level boundaries, i.e., it is possible for nodes belonging to the same level to be arbitrarily split up into adjacent clusters for assignment to adjacent subcubes in the Gray-code ordering. This enables one to choose the number of nodes assigned to different clusters, within reasonable limits, in an arbitrary manner.

The refined method is as follows. Suppose we want to split nodes belonging to some level between two consecutive clusters i and j, such that cluster i receives a fraction r of the nodes. Recall from Sec. 5 that for a random node with key κ defined by Eq. 4, the fraction $(\kappa.A - \lfloor \kappa.A \rfloor)$ is a random number in the range $[0, 1)$. Therefore by assigning nodes with $(\kappa.A - \lfloor \kappa.A \rfloor) < r$ to cluster i and the others to cluster j, we will have performed the required split. By appropriately splitting essential nodes in different levels among adjacent clusters, one can make the cluster essential-node distribution curve piecewise linear and symmetric about the center cluster. In Fig. 5(b), we illustrate the cluster essential-node distribution curve corresponding to the essential-node distribution curve of Fig. 5(a). Each linear segment of this curve corresponds to s clusters. The sequence of clusters can now be folded on the ring of subcubes as before, with folds being placed at corners of the cluster distribution curve. Note that nodes belonging to any one level should not be split between more than two clusters, otherwise inter-subcube communication (for the purpose of duplicate pruning) will not necessarily be between adjacent subcubes as it is in Fig. 2(b). We have observed that the shape of essential-node distribution curves for individual instances deviates little from the shape of the average essential-node distribution curve, although they may differ in magnitude. Therefore our refined clustering method based on an average essential-node distribution curve will enable us to achieve near ideal load balance for almost any instance of a problem whose essential-node distribution curve is not known *a priori*.

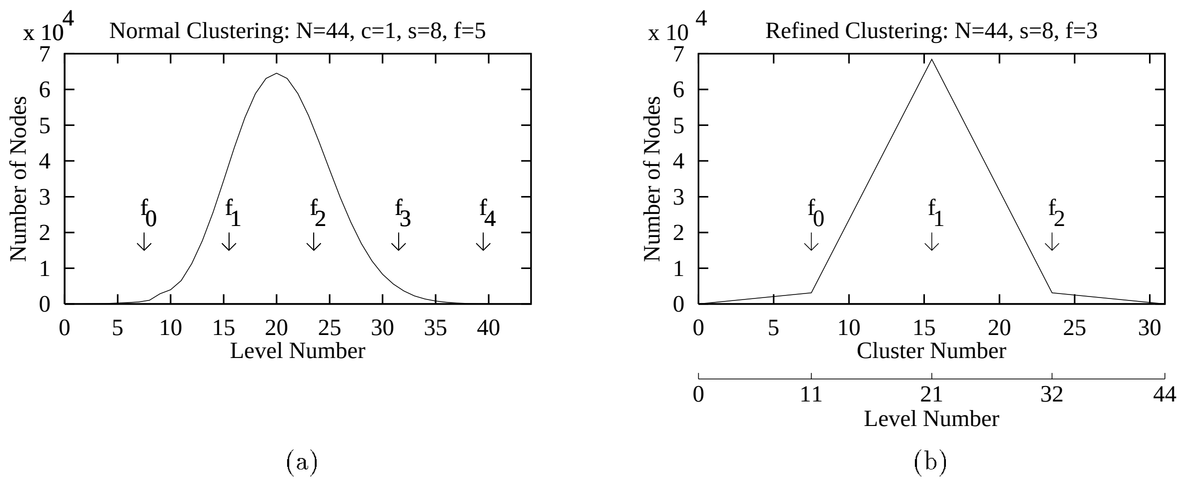


Figure 5: (a) Hypothetical essential-node distribution curve used to determine static load imbalance under normal clustering. (b) Cluster essential-node distribution curve obtained via refined clustering to minimize static load imbalance.

8 Scalability Analysis of GOHA&QE and LOHA&QE

8.1 Preliminaries

In this section, we analyze the scalability of the new duplicate pruning algorithms GOHA&QE and LOHA&QE on P-processor k-ary n-cube tori; the analysis for GOHA&QE also applies to GOHA except for the part accounting for the effect of the QE strategy (Secs. 8.5 and 8.6), which significantly enhances the scalability of GOHA&QE. We focus on k-ary n-cubes because they constitute an important class of architectures—many current distributed-memory machines like the nCUBE2 (hypercube), Intel Delta (hypercube) and Paragon (2-D mesh), and Cray T-3D (3-D torus) belong to the k-ary n-cube family. In order to make a fair comparison between k-ary n-cubes of different dimensions, we will keep the network cost constant. We will use *bisection width*, defined as the minimum number of wires that must be cut to separate the network into two equal halves, as a measure of network cost [4]. The bisection width of a k-ary n-cube with two unidirectional channels between neighboring processors is $B = 2 \cdot m_1 \cdot \eta \cdot P/k$, where η is the channel width, and $m_1 = 2$ for $k > 2$ and $m_1 = 1$ for a hypercube ($k = 2$). We will keep $B = P$ for all k-ary n-cubes, so that $\eta = k/(2 \cdot m_1)$, i.e., we will compare low-dimensional networks having large η with high-dimensional networks having small η. It is assumed that dimension-ordered worm-hole routing is used, as is the case on our target machine, the nCUBE2.

We denote by t_e the average time taken to expand and form a node.[11] Let $t_e = N^y$, where N is

[11]Actually, t_e also includes the average time per node spent by the sequential algorithm in performing tasks such

the number of variables in the COP to be solved.[12]For both the heuristics used in our parallel A* algorithms to solve TSP, viz., our simple heuristic of Sec. 3 and the LMSK heuristic [16], $y = 2$. Note that since each node expansion involves computing the costs of its children, whose number on the average grows as $\Theta(N)$, $t_e = \Omega(N)$ and $y \geq 1$. The *average essential branching factor* b is defined as the number of distinct essential children of an essential node averaged over all such nodes. It has been empirically seen that b increases very slowly with N, and since in our analysis it appears alongside terms that dominate it, we regard it as a constant. The *hashing overhead* per node t_h comprises: (1) The time t_{h1} to compute the hash function, which from Eqs. 3 and 6 involves a constant number of N-bit multiplications. Since each such multiplication can be decomposed into $\left(\frac{N}{M}\right)^2$ M-bit multiplications, which can individually be done in $\Theta(\log M)$ time using an M-bit binary multiplier [9], $t_{h1} = \Theta(\left(\frac{N}{M}\right)^2 \cdot \log M)$. (2) The time t_{h2} to initiate node transmission is proportional to the length of the message. Since message length is $\Theta(N)$ for $\Theta(N)$ state variables in a node, $t_{h2} = \Theta(N)$. Therefore $t_h = \Theta(\left(\frac{N}{M}\right)^2 \cdot \log M)$.

Now in GOHA&QE, nodes are hashed to random destinations in the architecture at a rate:

$$\lambda = \frac{b}{t_e + b \cdot t_h}. \tag{8}$$

For such a communication pattern and for a dimension-ordered worm-hole routing scheme, the latency incurred by an L-bit message is [4]:

$$t_c = \frac{\lambda \cdot D_{avg} \cdot L^2}{\eta^2} \cdot T_c^2 + D_{avg} \cdot T_c + \frac{L}{\eta} \cdot T_c. \tag{9}$$

Here D_{avg} is the average distance traveled by a message and T_c is the flit transmission time over a channel; a *flit* is η bits of message that are transmitted in parallel over a channel of width η. The first term in the above equation represents the delay incurred by the message due to contention, the second term corresponds to the header flit transmission time over D_{avg} channels, and the last term gives the time for the remainder flits to reach the destination processor once the header flit has done so. For GOHA&QE, D_{avg} is the average distance between any two processors in the k-ary n-cube, which is $n \cdot k/4$. Therefore, by using $L = N$ and $\eta = k/(2 \cdot m_1)$, the latency expression in Eq. 9 yields for GOHA&QE:

$$t_c = \frac{\lambda \cdot m_1^2 \cdot n \cdot N^2}{k} \cdot T_c^2 + \frac{n \cdot k}{4} \cdot T_c + \frac{2 \cdot m_1 \cdot N}{k} \cdot T_c. \tag{10}$$

as inserting nodes in *OPEN*, checking for duplicates and cost improvement propagation. Such a general definition of t_e can be used, but does not alter the analysis.

[12]More appropriately, we can use $t_e = a \cdot N^y$, for some constant a. This, however, also does not affect our analysis.

Note that the communication pattern in LOHA&QE is the same as that in GOHA&QE except that the former is confined to $\hat{k}$-ary $\hat{n}$-cube subcubes, so that $D_{avg} = m_2 \cdot \hat{n} \cdot \hat{k}/4$, where recall that $m_2 = 2$ when $\hat{k} < k$ and $m_2 = 1$ when $\hat{k} = k$. Hence for LOHA&QE we obtain:

$$t_c = \frac{\lambda \cdot m_1^2 \cdot m_2 \cdot \hat{n} \cdot \hat{k} \cdot N^2}{k^2} \cdot T_c^2 + \frac{m_2 \cdot \hat{n} \cdot \hat{k}}{4} \cdot T_c + \frac{2 \cdot m_1 \cdot N}{k} \cdot T_c. \tag{11}$$

Thus, message latency in LOHA&QE is less than that in GOHA&QE, except for the case $\hat{n} = n$ and $\hat{k} = k/2$ when it is equal in both.

In our analysis, we assume that all processors expand nodes synchronously and that all node expansions take the same time. This makes the analysis tractable and will give correct performance prediction in order terms. Let T_1 and W denote the amount of work performed by SEL_SEQ_A* in terms of its execution time, and the number of nodes expanded by it, respectively. Since the sequential algorithm expands only essential nodes, T_1 and W are also referred to as the total essential work, and W is actually the total number of essential nodes. We let $W = N^x$; for instance, the average-case complexity of solving TSP is polynomial in N [27]. Thus $T_1 = W \cdot t_e = N^{(x+y)}$. Also, let T_P and I_P denote the parallel algorithm execution time, and the number of parallel node-expansion iterations, respectively, on P processors. Then speedup $S = T_1/T_P = W/I_P$, and efficiency $E = S/P$. Any parallel algorithm must accomplish the essential work T_1 (or W) and may possibly incur other overheads. The total overhead over all processors $T_o = W_o \cdot t_e$ for a parallel algorithm is therefore equal to $P \cdot T_P - T_1 = (P \cdot I_P - W) \cdot t_e$. Hence we have efficiency $E = T_1/(T_1 + T_o) = 1/(1 + T_o/T_1) = 1/(1 + W_o/W)$.

The *temporal isoefficiency function* Φ_T of a parallel algorithm is defined to be the required rate of growth of T_1 with respect to P to keep the efficiency fixed at some value, and is a measure of the scalability of the algorithm [13]. Similarly, the *work isoefficiency function* Φ_W is defined as the required rate of growth of W with respect to P to maintain efficiency at some value. Lower values of Φ_W like $\Theta(P)$ and $\Theta(P \cdot \log^2 P)$ indicate that the algorithm is very scalable, while high values like $\Theta(P^2)$ imply poor scalability. From the expression for efficiency, we notice that T_1 (W) needs to grow as T_o (resp. W_o) to maintain the efficiency at a particular value. In other words, the rate of growth of T_o (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. Note that since $W = N^x$ needs to grow as $\Theta(W_o)$ for isoefficiency, $T_1 = N^{(x+y)} = W^{(1+y/x)}$ will need to grow as $\Theta(W_o^{(1+y/x)})$ to keep efficiency fixed. Thus $\Phi_T = \Phi_W^{(1+y/x)}$. For the heuristic functions used in many applications, including the heuristics used for TSP in this work, $y \ll x$. In such cases,

$\Phi_T \approx \Phi_W$, and load balancing and duplicate pruning at the level of node expansions, as we have done in this work, is sufficient. In fact, trying to parallelize node expansions in such cases may deteriorate performance, since then parallelization overheads may become significant compared to node-expansion computation. But for situations in which y is comparable to x, as is the case when the assignment heuristic is used for TSP [21], node expansions must be parallelized to keep Φ_T comparable to Φ_W and obtain acceptable performance. Note that by parallelizing node expansion, we divide it up into smaller atomic computation units and thus reduce the effective y (read N^y as the computation time of each atomic computation). For brevity, we will obtain expressions for only Φ_W.

In our previous work, we had derived the following result by analyzing the idling overhead incurred by an optimal startup phase because of the limited initial parallelism available in the search problem:

Theorem 1 [6] *A lower bound on the isoefficiency function of any parallel A* algorithm on any P-processor architecture is* $\Phi_W = \Theta(P \cdot \log P)$.

This gives us a lower bound on Φ_W for both GOHA&QE and LOHA&QE. To obtain an upper bound on their Φ_W's, we first establish upper bounds on the isoefficiency functions as determined by the different overheads incurred by them, each considered in isolation. Since the overall isoefficiency function of a parallel algorithm depends upon the total overhead, upper bounds on the isoefficiency functions of GOHA&QE and LOHA&QE will just be the maximum of the isoefficiency functions determined by the individual overheads. All subsequent execution times and overheads denoted by T's are in terms of actual time spent, while those denoted by W's are measured in terms of equivalent node-expansion time units, i.e., in terms of t_e time units.

8.2 Hashing Overhead

First, we consider the effect of hashing overhead on the scalability of our duplicate pruning algorithms. The time spent by a processor in hashing a node is considered to contribute to the total hashing overhead T_o^h over all processors only when the processor has essential nodes to expand. Otherwise, it is included in nonessential work overhead, which is examined later in Sec. 8.5. Thus $T_o^h = W \cdot t_h$. For isoefficiency:

$$T_1 = W \cdot t_e = \Theta(T_o^h) = \Theta(W \cdot t_h) \Rightarrow t_e = \Theta(t_h). \tag{12}$$

Since t_e and t_h both depend on only N, and not on P or other architectural parameters of the machine,[13] Eq. 12 does not yield any isoefficiency function for the algorithms, but tells us only whether they are scalable or not. If $t_e = \Omega(t_h)$, then the algorithms are scalable (assuming hashing overhead to be the only overhead), else they are unscalable. Recall that $t_e = N^y = \Omega(N)$ and $t_h = \Theta(\left(\frac{N}{M}\right)^2 \cdot \log M)$. Therefore applications with low enough granularity (i.e., $y < 2$) will be unscalable. Moreover, for acceptable efficiency, the hashing overhead T_o^h should be low compared to the total essential work T_1.

In the above situations, one of two approaches can be followed to restrict the hashing overhead: (1) The complexity of hash-function computation can be reduced to $\Theta(\log M)$, which is a constant, by using just the least significant word of the key (defined in Eq. 4) in Eqs. 3 and 6. As noted earlier, the other component of hashing overhead, i.e., initiating node transmission, takes $\Theta(N)$ time, and thus $t_h = \Theta(N)$. Note, however, that this approach will adversely affect the randomized load balance across processors obtained via hashing, since then the destination of hashed nodes will be determined by only M of the N state variables (assuming $M < N$). While this will directly degrade the performance of GOHA (since load balance is completely determined by the hash function), the situation can be ameliorated to a large extent in GOHA&QE and LOHA&QE by the QE strategy. (2) We can hash nodes, and hence prune duplicates, at only a subset of the $N+1$ levels, with the size of the subset being large enough for the resultant duplication overhead to be easily compensated by the decrease in hashing overhead. We found empirically that the hashing overhead for TSP is quite low. For example, it is only 2% of the execution time for $N = 44$. Therefore the above approaches for reducing hashing overhead were not needed. The primary reason for such a low hashing overhead is due to our use of the partial expansion scheme of Sec. 2 in the sequential A* algorithm on each processor. For $N = 44$, we found that the use of complete expansion increases the hashing overhead to 30% and doubles the execution time and memory requirement. In the remainder of the analysis, we assume that either one of the above two approaches is adopted, if needed for some application, so that $t_e = \Omega(t_h)$ and the duplicate pruning algorithms are scalable.

8.3 Communication Latency Overhead

Here, we will examine how message latency can cause processors to idle when the hashing rate λ is reasonably high. For simplicity, we will consider just the latency caused by communication for duplicate pruning. Communication required for load balancing is near-neighbor in both GOHA&QE

[13]As long as $P > 1$, $T_o^h = W \cdot t_h$.

and LOHA&QE and can be considered to consume the same fraction of the total channel bandwidth in both cases. We first consider GOHA&QE. Since on the average a message travels $n \cdot k/4$ hops, the per-hop latency incurred by a message is[14] $\delta = 4 \cdot t_c/(n \cdot k)$, where t_c is given by Eq. 10. As the hashing rate λ increases, the rate of arrival μ of nodes at processors also increases, with $\mu = \lambda$ until the arrival rate saturates at some value μ_{sat} and increases no further [4]. To determine μ_{sat}, consider the following scenario under saturation. For simplicity, assume that all nodes travel exactly the average distance of $n \cdot k/4$ hops, and that router buffers are large enough to accommodate the largest messages. Then, after each δ time interval, $m_1 \cdot n$ (the number of neighbors of each processor) nodes will be incident on each processor, of which equal fractions $\frac{m_1 \cdot n}{n \cdot k/4} = 4 \cdot m_1/k$ of nodes will have traveled $1, 2, \ldots, n \cdot k/4$ hops of the total $n \cdot k/4$ hops. Therefore after each δ time interval, $4 \cdot m_1/k$ out of the $m_1 \cdot n$ nodes incident on each processor will have traveled $n \cdot k/4$ hops, and hence will have reached their destination. Thus $\mu_{sat} = 4 \cdot m_1/(k \cdot \delta)$. At these higher saturation-hashing rates, the first term (involving T_c^2) in Eq. 10 dominates the other two, and hence the latter can be neglected. Therefore at saturation:

$$\lambda = \mu_{sat} = \frac{4 \cdot m_1}{k \cdot \delta} = \frac{4 \cdot m_1}{k \cdot \frac{4 \cdot t_c}{n \cdot k}} \Rightarrow \lambda = \mu_{sat} = \frac{1}{N \cdot T_c} \cdot \sqrt{\frac{k}{m_1}}. \tag{13}$$

In saturation, inter-arrival time of nodes at any processor is $1/\mu_{sat}$. Also, the average time to expand and hash a node in a busy processor is b/λ. Therefore from Eq. 8, when the hashing rate is such that

$$\frac{b}{\lambda} \approx \frac{b}{b/N^y} < \frac{1}{\mu_{sat}} = N \cdot T_c \cdot \sqrt{\frac{m_1}{k}} \Rightarrow N^{y-1} < T_c \cdot \sqrt{\frac{m_1}{k}}, \tag{14}$$

every processor idles for $(1/\mu_{sat} - b/\lambda)$ time on the average for each node expanded. Thus the idling overhead due to communication latency in GOHA&QE for a total of W essential nodes is:

$$T_o^{cl} = W \cdot (\frac{1}{\mu_{sat}} - \frac{b}{\lambda}) = O(W \cdot \frac{1}{\mu_{sat}}) = O(W \cdot N \cdot T_c \cdot \sqrt{\frac{m_1}{k}}), \text{ given that } \frac{1}{\mu_{sat}} > \frac{b}{\lambda}. \tag{15}$$

Hence, given that $1/\mu_{sat} > b/\lambda$, we have for isoefficiency:

$$T_1 = W \cdot t_e = W \cdot N^y = \Theta(T_o^{cl}), \Rightarrow W = N^x = O(\left(T_c \cdot \sqrt{\frac{m_1}{k}}\right)^{x/(y-1)}). \tag{16}$$

This leads to the following theorem:

[14]More precisely, $\delta = 4 \cdot t_{c1}/(n \cdot k) + t_{c2}$, where t_{c1} comprises the first two terms and t_{c2} is the last term in Eq. 10, since from Eq. 9 only t_{c1} is a function of distance traveled by a message. However, we will shortly see that at the higher hashing rates we are concerned here, t_{c2} can be neglected.

Theorem 2 *Given that the condition in Eq. 14 is satisfied, the isoefficiency function of* PLA*-GOHA&QE *on a P-processor k-ary n-cube architecture, as determined by communication latency overhead, is $\Phi_W^{cl} = O(\left(\frac{1}{\sqrt{k}}\right)^{x/(y-1)})$. Otherwise, communication latency does not limit the scalability of* PLA*-GOHA&QE.

The idling condition in Eq. 14 implies that idling due to message contention is more likely to arise in low-granularity applications (y small) and higher dimensional architectures, than in coarse-grained applications and lower dimensional architectures.

The effect of communication latency in the case of LOHA&QE can be determined by noting that a similar pattern of communication to that of GOHA&QE exists in each of the individual $\hat{k}$-ary $\hat{n}$-cube subcubes. Thus using Eq. 11, we obtain the idling condition for LOHA&QE as:

$$N^{y-1} < T_c \cdot \frac{\sqrt{m_1 \cdot m_2 \cdot \hat{k}}}{k}. \tag{17}$$

Similarly, we can obtain the isoefficiency condition for LOHA&QE as:

$$W = O(\left(T_c \cdot \frac{\sqrt{m_1 \cdot m_2 \cdot \hat{k}}}{k}\right)^{x/(y-1)}). \tag{18}$$

Thus:

Theorem 3 *Given that the condition in Eq. 17 is satisfied, the isoefficiency function of* PLA*-LOHA&QE *with $\hat{k}$-ary $\hat{n}$-cube subcubes on a P-processor k-ary n-cube architecture, as determined by communication latency overhead, is $\Phi_W^{cl} = O(\left(\frac{\sqrt{\hat{k}}}{k}\right)^{x/(y-1)})$. Otherwise, communication latency does not limit the scalability of* PLA*-LOHA&QE.

Remarks regarding the scalability of different granularity applications, and the suitability of different dimensional architectures can be made similar to that in the case of GOHA&QE. However, other important points to note from Eq. 17 and Theorem 3 are that idling in LOHA&QE will occur at a much lower granularity than in GOHA&QE (compare with Eq. 14) and that LOHA&QE has better scalability than GOHA&QE for any granularity application on any dimension architecture (compare with Theorem 2).

The effectiveness of LOHA&QE over GOHA&QE will be more pronounced in a strictly distributed computing environment of, say, workstation clusters attached to a LAN in which there is much more message contention and latency. In this situation, LOHA&QE can be implemented by regarding each of the different computing units (workstations) as a separate subcube, and then suitably folding the sequence of clusters on them. Also, to reduce inter-computing-unit communications, a reasonably large cluster size should be used. GOHA&QE can be implemented as before with the individual computing units playing the role of processors. Since there will be more inter-computing-unit communications in GOHA&QE than LOHA&QE, average communication latency, and any consequent idling, will be appreciably higher in the former than in the latter.

8.4 Startup Phase Overhead

The isoefficiency lower bound in Theorem 1 was derived assuming hashing overhead and communication latency to be zero [6]. Here we consider the effect of these two factors on startup phase idling. The *startup phase overhead* W_o^{su} is defined to be the sum over all processors of the time spent by each processor idling (or performing nonessential work in case nonessential nodes are received initially by the processor), before obtaining its unique essential node. First, we determine W_o^{su} for GOHA&QE. Initially, say, at time 0, processor 0 has the *root* node of the COP. It expands *root* in t_e time, then computes the destinations to which the b essential children generated will be hashed in another $b \cdot t_h$ time, and finally hashes them. The last node hashed will reach its destination at time $t_e + b \cdot t_h + t_c + b - 1$. This completes one iteration. With successive iterations, more and more processors start receiving nodes as the above process repeats. At any stage, the available nodes are randomly distributed among all P processors. Therefore for a total of r available nodes at some time, the average number of processors with at least one node is $q_r = P \cdot (1 - (1 - 1/P)^r)$. For small values of r relative to P, $q_r \approx r$. We assume for simplicity that $q_r = r$ during the startup phase. Therefore the expansion of r nodes in r different processors yields $b \cdot r$ nodes that lie in as many processors. These $b \cdot r$ nodes are then expanded in the next iteration. Consequently, the time in which all processors receive at least one node, i.e., the startup phase time, $t_{su} = (t_e + b \cdot t_h + t_c + b - 1) \cdot \log_b P$. During startup phase, message contention is small, so the first term in t_c in Eq. 10 can be neglected. Since in the ith stage of node expansion there are b^i nodes that are being expanded in parallel, the total amount of work accomplished in the startup phase $T_{su} = t_e \cdot \sum_{i=0}^{(\log_b P - 1)} b^i = \Theta(t_e \cdot P)$. Now, if we consider startup-phase idling to be the only overhead, then $T_P = t_{su} + (T_1 - T_{su})/P$. Therefore using Eq. 10 and the fact that $W = N^x$, we obtain:

$$\begin{aligned} T_o^{su} &= W_o^{su} \cdot t_e = P \cdot T_P - T_1 = P \cdot t_{su} - T_{su} \\ &= \Theta((t_e + b \cdot t_h + t_c + b - 1) \cdot P \cdot \log P) - \Theta(t_e \cdot P) \end{aligned}$$

$$\begin{aligned}\Rightarrow W_o^{su} &= \Theta(max(1,\frac{t_c}{t_e})\cdot P\cdot\log P) = \Theta(max(1,\frac{(n\cdot k/4)\cdot T_c + (2\cdot m_1\cdot N/k)\cdot T_p}{N^y})\cdot P\cdot\log P)\\ &= \Theta(max(1,\frac{n\cdot k\cdot T_c}{4N^y})\cdot P\cdot\log P) = \Theta(max(1,\frac{n\cdot k\cdot T_c}{4W^{y/x}})\cdot P\cdot\log P). \qquad (19)\end{aligned}$$

For isoefficiency:

$$W = \Theta(W_o^{su}) \Rightarrow W = \Theta(max(P\log P, \left(\frac{n\cdot k\cdot T_c\cdot P\cdot\log P}{4}\right)^{x/(x+y)})). \qquad (20)$$

Hence we obtain:

Theorem 4 *The isoefficiency function of* PLA*-GOHA&QE *on a P-processor k-ary n-cube architecture, as determined by startup phase overhead, is* $\Phi_W^{su} = \Theta(max(P\cdot\log P, (n\cdot k\cdot P\cdot\log P)^{x/(x+y)}))$.

Hence the scalability determined by startup phase overhead improves with increase in the dimension of architectures; since Φ_T, which reflects the actual scalability, is equal to $\Phi_W^{(1+y/x)}$, the scalability of GOHA&QE is impervious to the granularity (i.e., value of y) of applications.

Next, we determine the startup phase overhead for LOHA&QE. Recall from Sec. 7 that we use a cluster size of one to minimize startup-phase idling. Hence the children of nodes expanded in any subcube will belong to the next subcube in the Gray-code order. At the start of the algorithm, subcube 0 has the *root* node. After the first iteration, subcube 1 will have b nodes (the children of the *root* node), and after the second iteration subcube 2 will have b^2 nodes, and so on. Therefore in $(t_e + b\cdot t_h + t_c + b - 1)\cdot\log_b(P/s)$ time all processors in one of the subcubes (subcube $\log_b(P/s)$ mod s) will have received nodes. It will take a further $s-1$ node-expansion stages to ensure that all processors in all subcubes receive nodes. So the startup phase time is $t_{su} = \Theta((t_e + b\cdot t_h + t_c + b - 1)\cdot(\log(P/s)+s))$. The total work completed in the startup phase, as in the case of GOHA&QE, is $T_{su} = \Theta(t_e\cdot P)$. Using Eq. 11 and following a similar line of analysis as for GOHA&QE, we can show that:

$$W_o^{su} = \Theta(max(1,\frac{m_2\cdot\hat{n}\cdot\hat{k}\cdot T_c}{4\cdot W^{y/x}})\cdot P\cdot(\log(P/s)+s)), \qquad (21)$$

and that for isoefficiency:

$$W = \Theta(max(P\cdot\log(P/s), P\cdot s, \left(\frac{m_2\cdot\hat{n}\cdot\hat{k}\cdot T_c\cdot P\cdot\log(P/s)}{4}\right)^{x/(x+y)}, \left(\frac{m_2\cdot\hat{n}\cdot\hat{k}\cdot T_c\cdot P\cdot s}{4}\right)^{x/(x+y)})). \qquad (22)$$

This leads to:

Theorem 5 *The isoefficiency function of* PLA*-LOHA&QE *with s $\hat{k}$-ary $\hat{n}$-cube subcubes on a P-processor k-ary n-cube architecture, as determined by startup phase overhead, is* $\Phi_W^{su} = \Theta(max(P \cdot \log(P/s), P \cdot s, \left(\hat{n} \cdot \hat{k} \cdot P \cdot \log(P/s)\right)^{x/(x+y)}, \left(\hat{n} \cdot \hat{k} \cdot P \cdot s\right)^{x/(x+y)}))$.

When $s < \log P$, scalability of LOHA&QE as determined by startup phase overhead is better than that of GOHA&QE.

8.5 Nonessential Work Overhead

Define *nonessential work overhead* W_o^{ne} to be sum over all processors of the time each processor spends idling (due to nonavailability of work and not due to communication latency, which overhead has been accounted for in Theorems 2 and 3) or performing nonessential work after the startup phase. In our previous work [6], we had derived an upper bound on W_o^{ne} for a PLA* algorithm using the QE strategy for load balance across all P processors as:

$$W_o^{ne} = O(P \cdot k \cdot n^2). \tag{23}$$

Since GOHA&QE is such a PLA* algorithm, Eq. 23 is valid for it as well. Therefore:

Theorem 6 *The isoefficiency function of* PLA*-GOHA&QE *on a P-processor k-ary n-cube architecture, as determined by nonessential work overhead, is* $\Phi_W^{ne} = O(P \cdot k \cdot n^2)$.

It can be easily shown from the above theorem that:

Corollary 1 *The upper bound of* $\Theta(P \cdot k \cdot n^2)$ *for nonessential work incurred by* PLA*-GOHA&QE *on a k-ary n-cube architecture decreases with n until $n = 0.5 \ln P$, when it is minimum, and then increases with n.*

Since $\Theta(P \cdot k \cdot n^2)$ roughly reflects the average-case nonessential work overhead incurred [6], the scalability of GOHA&QE for large machine sizes, assuming nonessential work to be the only overhead, is better on higher than lower dimensional architectures.

Now consider nonessential work in LOHA&QE. In Sec. 7, we saw that the static load imbalance across subcubes in LOHA&QE is negligible. Therefore nonessential work in LOHA&QE principally arises from imperfect dynamic load balance within subcubes. Recall from Sec. 7 that load balancing in LOHA&QE is essentially restricted to subcubes, i.e., each processor communicates for load balance with neighboring processors within its subcube plus a constant number (two) of neighbors belonging to adjacent subcubes. Therefore for s $\hat{k}$-ary $\hat{n}$-cube subcubes of size P/s each we obtain W_o^{ne} from Eq. 23 as:

$$W_o^{ne} = O(s \cdot (P/s) \cdot \hat{k} \cdot \hat{n}^2) = O(P \cdot \hat{k} \cdot \hat{n}^2). \tag{24}$$

This yields:

Theorem 7 *The isoefficiency function of* PLA*-LOHA&QE *with s $\hat{k}$-ary $\hat{n}$-cube subcubes on a P-processor k-ary n-cube architecture, as determined by nonessential work overhead, is $\Phi_W^{ne} = O(P \cdot \hat{k} \cdot \hat{n}^2)$.*

Corollary 2 *The upper bound of $\Theta(P \cdot \hat{k} \cdot \hat{n}^2)$ for nonessential work incurred by* PLA*-LOHA&QE *on a k-ary n-cube architecture, for a given number s of subcubes, decreases with n until $n = 0.5\ln(P/s)$, when it is minimum, and then increases with n.*

Again, similar to the case of GOHA&QE, the optimally scalable dimension for a subcube in LOHA&QE, assuming that the number of subcubes is fixed and that nonessential work is the only overhead, increases with the machine size. Furthermore, we note that Φ_W^{ne} for LOHA&QE is much better than that of GOHA&QE, since in general $\hat{k} < k$ and $\hat{n} < n$.

8.6 Overall Scalability

We have already seen that Theorem 1 gives a lower bound of $\Theta(P \log P)$ on the isoefficiency functions Φ_W^{GOHA} and Φ_W^{LOHA} of GOHA&QE and LOHA&QE, respectively. The upper bound on Φ_W^{GOHA} is $\Theta(max(\Phi_W^{cl}, \Phi_W^{su}, \Phi_W^{ne}))$, where Φ_W^{cl}, Φ_W^{su} and Φ_W^{ne} are as given in Theorems 2, 4 and 6, respectively. After simplification, we obtain:

Theorem 8 *The isoefficiency function of* GOHA&QE *on a P-processor k-ary n-cube architecture is $\Phi_W^{GOHA} = O(P \cdot k \cdot n^2)$.*

As noted earlier, the scalability of GOHA&QE favors higher dimensional architectures for larger machine sizes. This is also evident from the following corollaries of the above theorem for the hypercube ($k = 2$), 3-D torus ($n = 3$) and 2-D mesh ($n = 2$) architectures.

Corollary 3 *The isoefficiency function of* GOHA&QE *on a P-processor hypercube architecture is $\Phi_W^{GOHA} = O(P \log^2 P)$.*

Corollary 4 *The isoefficiency function of* GOHA&QE *on a P-processor* 3-*D torus architecture is $\Phi_W^{GOHA} = O(P^{4/3})$.*

Corollary 5 *The isoefficiency function of* GOHA&QE *on a P-processor 2-D mesh architecture is* $\Phi_W^{GOHA} = O(P^{3/2})$.

Similarly, from Theorems 3, 5 and 7, we obtain, after simplification, the following result:

Theorem 9 *The isoefficiency function of* LOHA&QE *with s $\hat{k}$-ary $\hat{n}$-cube subcubes on a P-processor k-ary n-cube architecture is* $\Phi_W^{LOHA} = O(max(P \cdot s, (\hat{n} \cdot \hat{k} \cdot P \cdot s)^{x/(x+y)}, P \cdot \hat{k} \cdot \hat{n}^2))$.

We should choose $\hat{n}$ and s (and hence $\hat{k}$) so that Φ_W^{LOHA} is minimized. In general, it is not possible to find a closed form expression for the optimal values of these parameters. However, for hypercube ($k = 2$), 3-D torus and 2-D mesh ($n = 2$) architectures, we obtain the following corollaries:

Corollary 6 *The isoefficiency function of* LOHA&QE *with s subcubes on a P-processor hypercube architecture is* $\Phi_W^{LOHA} = O(max(P \cdot s, P \cdot \log^2(P/s))$.

It turns out that the optimal value of s for a hypercube is $\Theta(\log^a P)$, where a increases with P towards an asymptotic value close to 2. For instance, for $P = 1024$, $a \approx 1.44$. Therefore we can say that the best isoefficiency function of LOHA&QE on a hypercube is $O(P \cdot \log^2(\frac{P}{\log^a P}))$, which is better than $O(P \cdot \log^2 P)$ for GOHA&QE (Corollary 3).

Corollary 7 *The isoefficiency function of* LOHA&QE, *when $y \ll x$, on a P-processor 3-D torus architecture is* $\Phi_W^{LOHA} = O(P^{4/3})$.

Thus, when $y \ll x$, which is likely to be true for low to medium granularity applications like the one considered in this paper, isoefficiency upper bound for LOHA&QE on a 3-D torus is the same as that for GOHA&QE (Corollary 4).

Corollary 8 *The isoefficiency function of* LOHA&QE, *when $y \geq x/5$, on a P-processor 3-D torus architecture is* $\Phi_W^{LOHA} = O(P^{5/4})$.

The above isoefficiency upper bound for the case $y \geq x/5$, which is likely to be true for coarse-grained applications, is better than $O(P \cdot \sqrt{P})$ for GOHA&QE (Corollary 4). Furthermore, we have seen that all the different types of overheads considered earlier, except the hashing overhead which is the same in both algorithms, are more in the case of GOHA&QE than LOHA&QE. This favors the performance of LOHA&QE.

Similar remarks apply to the following results for the 2-D mesh architecture.

Corollary 9 *The isoefficiency function of* LOHA&QE*, when $y \ll x$, on a P-processor $2D$-mesh architecture is* $\Phi_W^{LOHA} = O(P \cdot \sqrt{P})$.

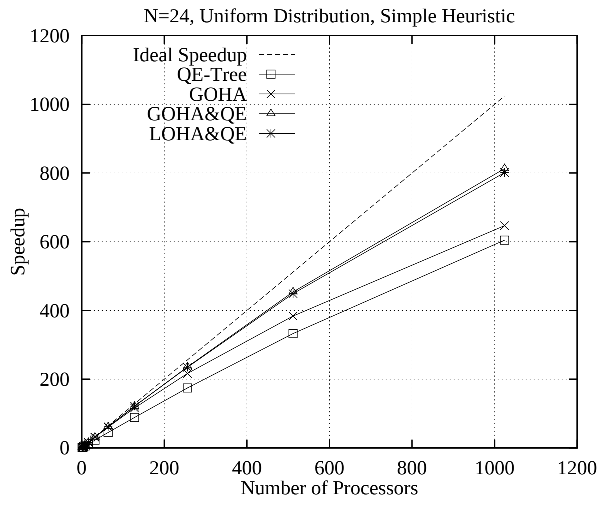
Corollary 10 *The isoefficiency function of* LOHA&QE*, when $y \geq x/4$, on a P-processor $2D$-mesh architecture is* $\Phi_W^{LOHA} = O(P^{4/3})$.

9 Performance Results

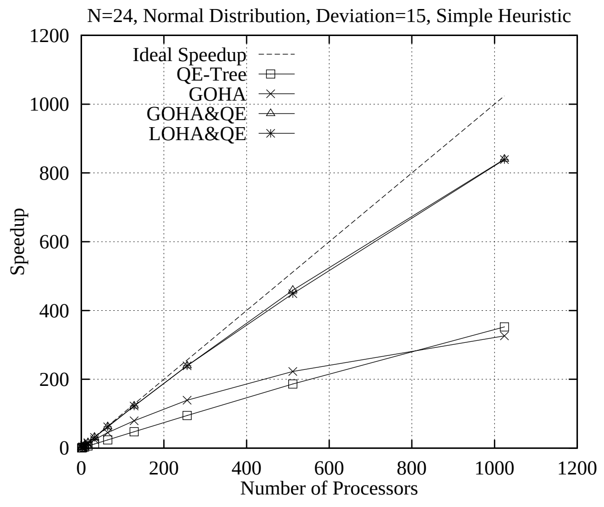
We implemented three PLA* algorithms incorporating either one of GOHA, GOHA&QE and LOHA&QE duplicate pruning strategies on an nCUBE2 multicomputer. These algorithms use the 3-tuple graph-search formulation of Sec. 2 to solve TSP. In order to study the advantage of graph-search over tree-search methods used by other researchers [12, 22], we also implemented a PLA* algorithm referred to as PLA*-QE-Tree. This algorithm uses the first tree-search formulation of TSP described in Sec. 3 in which the state of a node is defined by an ordered list of cities visited; the QE strategy is used to achieve load balance. Two heuristic functions are used in the above algorithms, viz., our simple heuristic of Sec. 3 and the LMSK heuristic [16]. The TSP city graphs chosen were complete, with inter-city distances either uniformly or normally distributed over the interval [1, 100]. Two merits of performance, averaged over ten TSP instances, are used: (1) Average speedup defined as the ratio (average T_1)/(average T_P); and (2) Average isoefficiency function, which is measured as the required rate of growth of essential work (as represented by average T_1 for different values of N) with respect to P to keep the efficiency fixed at some value and is a measure of the scalability of a parallel algorithm on a given architecture (see Sec. 8). Below we present performance results of our duplicate pruning algorithms.

In Fig. 6, we plot the speedup curves for QE-Tree, GOHA, GOHA&QE and LOHA&QE strategies using our simple heuristic.[15] It can be clearly seen from the plots that our pruning algorithms perform with high efficiency for as many as 1024 processors, the maximum number of processors available on the nCUBE2 we used. Specifically, note that both our algorithms perform far superior to those proposed in previous work—for uniformly distributed data, GOHA&QE and LOHA&QE yield a speedup improvement of more than 35% on 1024 processors over a previous method that does not prune any duplicates (QE-Tree), and more than 25% over the previous hashing-only

[15]Some large TSP instances included in the plots of Figs. 6, 7 and 8 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'.



(a)

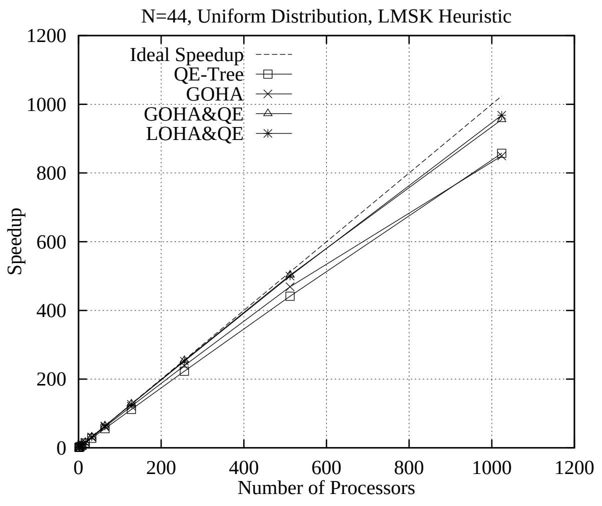


(b)

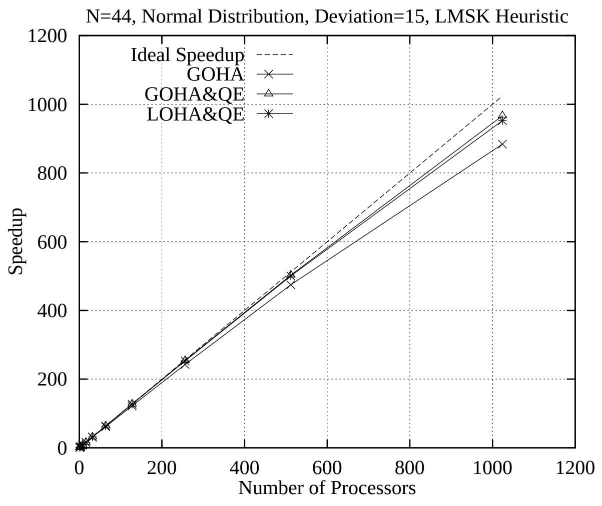
Figure 6: Speedup curves for different PLA* algorithms using the simple heuristic of Sec. 3 and employing different duplicate pruning strategies for (a) uniformly distributed data and (b) normally distributed data.

scheme (GOHA) (Fig. 6(a)). For normally distributed data the corresponding figures are 135% and 155% (Fig. 6(b)); the larger value for the latter means that duplicate pruning using a simple hashing scheme performs worse than an algorithm that does not prune any duplicates. The reason for this apparent anomaly is that the poor load balance dictated by the hash function in the former algorithm, more than offsets any gains from duplicate pruning. We have observed that the number of duplicates arising in A* algorithms increases with decrease in the deviation of the input data distribution, and is more for normally distributed than for uniformly distributed data. This is the reason for the deterioration in performance of PLA*-QE-Tree, when the data distribution changes from uniform to normal. Since in practice data is most often distributed normally, a graph formulation with effective duplicate pruning techniques is critical to attaining good performance in parallel branch-and-bound-type search for solving combinatorial optimization problems.

Next in Fig. 7, we plot the speedup curves for the various duplicate pruning methods using the LMSK heuristic, which is much tighter than the simple heuristic of Sec. 2; the curve for QE-Tree could not be plotted for normally distributed data (Fig. 7(b)) because of memory overflow caused by large amounts of duplicated work. Here again we notice that the speedups of GOHA&QE and LOHA&QE are high and about 13% better on 1024 processors than that of QE-Tree and GOHA for uniformly distributed data (Fig. 7(a)); the corresponding speedup improvement over GOHA for normally distributed data is about 10% (Fig. 7(b)). Also, note from Fig. 7 that we obtain an average speedup of about 970 on 1024 processors using GOHA&QE and LOHA&QE, which represents a very high efficiency of 0.95. The higher speedup values for the various algorithms in Fig. 7 compared to that in Fig. 6 is because the former corresponds to a larger essential search space, thus leading to a more efficient search.



(a)



(b)

Figure 7: Speedup curves for different PLA* algorithms using the LMSK heuristic and employing different duplicate pruning strategies for (a) uniformly distributed data and (b) normally distributed data.

Finally, in Fig. 8 we plot the isoefficiency curves for the various algorithms using our simple heuristic.[16] Although not many data points are available, we notice that the general trends of the isoefficiency functions for GOHA&QE and LOHA&QE are close to the lower bound of $P \cdot \log P$ (Theorem 1) and are much better than that of QE-Tree and GOHA. Also, note that GOHA&QE and LOHA&QE have better isoefficiency functions than the upper bound of $P \cdot \log^2 P$ (Sec. 8.6), thus bearing out our analysis in the previous section. Furthermore, we observe that the isoefficiency curves for PLA*-LOHA&QE and PLA*-GOHA&QE meet each other at 1024 processors. From the slope of these two curves at 1024 processors, it can be extrapolated that the isoefficiency curve of PLA*-LOHA&QE will be better than PLA*-GOHA&QE for larger number of processors. This is because of higher overheads incurred by the latter algorithm as pointed out in Sec. 8. The observed similarity in performance of GOHA&QE and LOHA&QE is due to the fact that $k/\hat{k}$, the ratio of the communication rates for the two methods (see Sec. 7.3), is one for the nCUBE2 hypercube

[16]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 at $P = 1$ with the isoefficiency curve that corresponds to the minimum (among all curves) and maximum (among GOHA&QE and LOHA&QE curves) sequential execution time, respectively, and then letting them grow for larger values of P at the rates $P \cdot \log P$ and $P \cdot \log^2 P$, respectively—this determines the constants associated with the analytical bounds.

system we used. LOHA&QE is likely to outperform GOHA&QE when $k/\hat{k} > 1$, such as on a 2-D or a 3-D mesh system.

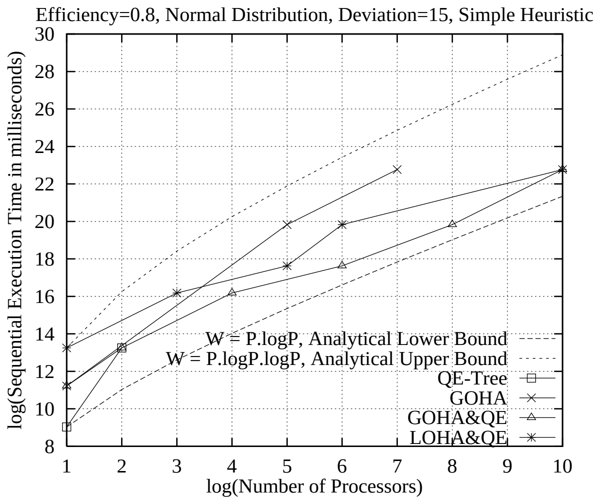


Figure 8: Isoefficiency curves for different PLA* algorithms using the simple heuristic of Sec. 3 and employing different duplicate pruning strategies for normally distributed data.

10 Conclusions

In this paper we have seen that for search spaces that are actually graphs and not simple trees, duplicate pruning is critical to attaining good performance. A graph formulation of the search space is found to be superior to a tree formulation, since the former formulation enables the detection and pruning of duplicates. The only previous scheme for complete pruning, called GOHA, uses a hash function to hash duplicate nodes to the same owner processor, and thereby prunes duplicates. However, it suffers from two major weaknesses: (1) Load balance is unsatisfactory. (2) It involves global communication that makes hot spots very likely. We proposed two different duplicate pruning strategies that address these drawbacks: (1) GOHA&QE decoupled the task of duplicate pruning from load balancing, and achieved good load balance using the QE strategy. (2) LOHA&QE utilized a search-space partitioning scheme that enabled it to perform complete duplicate pruning with only localized communication, thus enhancing the scalability of duplicate pruning. Finally, we analyzed the scalability of our algorithms on the k-ary n-cube family of architectures and obtained lower and upper bounds of $\Theta(P \cdot \log P)$ and $\Theta(P \cdot k \cdot n^2)$, respectively, on their isoefficiency functions. We also showed that our algorithms scale better on higher than lower dimensional architectures up to a certain dimension that increases with P. Performance results for TSP on an nCUBE2 hypercube corroborated our analysis, and revealed the superiority of our strategies over previously known methods—for uniformly distributed inter-city cost data GOHA&QE and LOHA&QE provided speedup improvements of 13-35% on 1024 processors over QE-Tree (that used a tree formulation of the search space), and 13-25% over GOHA; for normally distributed data the corresponding figures were 135% and 10-155%. For TSP (using the simple heuristic of Sec. 3 and the LMSK heuristic, both with $t_e = \Theta(N^2)$) and for a hypercube system, for which $k/\hat{k} = 1$, it turned out that GOHA&QE and LOHA&QE have similar performance. However, our analysis showed that for lower granularity applications and in a distributed computing environment where message latencies are significant or on a network such as a 2-D or a 3-D mesh with $k/\hat{k} > 1$, LOHA&QE will prove to be a better choice because of its localized communication, and smaller message contention and delay. In the future, we intend to apply our duplicate pruning techniques to applications in VLSI CAD, scheduling and integer programming.

References

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

[2] P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. De Sarkar, "Heuristic search in restricted memory," *Artificial Intelligence*, Vol. 41, pp. 197-221, 1989.

[3] T.H. Cormen, C.E. Leiserson and R.L. Rivest, "Introduction to Algorithms," *McGraw Hill*, 1990.

[4] W.J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Trans. on Comp.*, Vol. 39, No. 6, pp. 775-785, June 1990.

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

[6] S. Dutt and N.R. Mahapatra, "Scalable Load Balancing Strategies for Parallel A* Algorithms," *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, pp. 488-505, Sep. 1994.

[7] 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.

[8] 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 Massively Parallel Computation*, pp. 145-149, College Park, MD, Oct. 8-10, 1990.

[9] J.L. Hennessey and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[10] R.C. Holte, C. Drummond and M.B. Perez, "Searching with abstractions: A unifying framework and new high-performance algorithm," *Proc. 10th Canadian Conf. on AI*, pp. 263-270, 1994.

[11] R.M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch-and-bound computation," *Journal of the Association for Computing Machinery*, Vol. 40, No. 3, pp. 765-789, Jul. 1993.

[12] V. Kumar, K. Ramesh and V.N. Rao, "Parallel best-first search of state-space graphs: a summary of results," *Seventh National Conference on Artificial Intelligence (AAAI 88)*, Vol. 1, pp. 122-127, Saint Paul, MN, Aug. 21-26, 1988.

[13] V. Kumar and V.N. Rao, "Load balancing on the hypercube architecture," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Vol. 1, pp. 603-608, Monterey, CA, Mar. 6-8, 1989.

[14] D.E. Knuth, "Sorting and Searching," *The Art of Computer Programming*, Vol.3, Addison-Wesley, 1973.

[15] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[16] J.D. Little, et. al., "An Algorithm for the Traveling Salesman Problem," *Operations Research*, Vol.11, No. 6, pp. 972-989, 1963.

[17] R. Luling and B. Monien, "Load balancing for distributed branch & bound algorithms," *Proc. Sixth International Parallel Processing Symposium*, pp. 543-548, Beverly Hills, CA, Mar. 23-26, 1992.

[18] N.R. Mahapatra and S. Dutt, "New Anticipatory Load Balancing Strategies for Parallel A* Algorithms," *American Mathematical Society's Proc. in the DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 22, pp. 197-232, 1995.

[19] N.R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," submitted to *IEEE Trans. Par. and Distr. Systems*, in second round of review.

[20] 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.

[21] 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.

[22] J. Mohan, "Experience with two parallel programs solving the traveling salesman problem," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 191-193, Bellaire, MI, Aug. 23-26, 1983.

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

[24] 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.

[25] A.K. Sen and A. Bagchi, "Fast recursive formulations for best-first search that allow controlled use of memory," *Proceedings 11th Int'l Joint Conf. on Artificial Intelligence (IJCAI-89)*, pp. 297-302, Detroit, MI, Aug. 2-25, 1989.

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

[27] 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.

[28] B.W. Wah and Y.W.E. Ma, "MANIP - a parallel computer system for implementing branch and bound algorithms," *Proc. 8th Annual Symposium on Computer Architecture*, pp. 239-262, Minneapolis, MN, May 12-14, 1981.

Nihar R. Mahapatra (S'91-M'96) received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Delhi, India, in 1990, and the M.S. and Ph.D. degrees in Electrical Engineering, both with a minor in Computer Science, from the University of Minnesota, Twin Cities, in 1993 and 1996, respectively. He is currently an Assistant Professor in the Department of Electrical & Computer Engineering at the State University of New York at Buffalo. His research interests include parallel and distributed processing, computer architecture and networks, VLSI, and fault tolerance. He is on the program committee of *International Conference on Computer Design*, 1997. He is a member of the IEEE Computer Society and the ACM Special Interest Group on Computer Architecture.

Shantanu Dutt (S'87-M'90) received the B.E. degree in electronics and communication engineering from the M.S. University of Baroda, India, in 1983, the M.Tech. degree in computer engineering from the Indian

Institute of Technology, Kharagpur, India in 1984, and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1990.

In 1984-85, he was a research and development engineer at CMC Ltd., Secunderabad, India. He is currently an assistant professor at the Department of Electrical Engineering, University of Minnesota, Twin Cities.

He was awarded a National Merit Scholarship by the Government of India, a University Fellowship by the M.S. University of Baroda, a Rackham Predoctoral Fellowship by the University of Michigan, and a Research Initiation Award by the National Science Foundation. His current technical interests include CAD for VLSI circuits, parallel and distributed computing, fault-tolerant computing and computer architecture. He has published more than 30 papers in prestigious journals and refereed conferences in all these areas. He received a Best-Paper Award at the *Design Automation Conference*, 1996. On the occasion of the 25th anniversary of the *Fault-Tolerant Computing Symposium (FTCS)* in 1995, one of his papers (published in FTCS'88) in the area of Fault-Tolerant Multicomputers was selected to be among the most influential papers published over the last 25 years in FTCS. He has been a session chair at the *International Conf. on Supercomputing*, 1996, and at the scalability workshop at *IEEE Parallel Processing Symposium*, 1993. He is on the program committee of the *Fault-Tolerant Computing Symp.*, 1997, and was on an NSF panel for selecting CAREER awards. He is a member of the IEEE Computer Society, and the ACM Special Interest Groups on Computer Architecture and Design Automation.