JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 20, 1–13 (1994)

Multisearch Techniques: Parallel Data Structures on Mesh-Connected Computers*

MIKHAIL J. ATALLAH†

Department of Computer Science, Purdue University, West Lafayette, Indiana 47907

FRANK DEHNE‡

School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

RUSS MILLER§

Department of Computer Science, State University of New York at Buffalo, Buffalo, New York 14260

ANDREW RAU-CHAPLIN¶

School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

AND

JYH-JONG TSAY||

National Chung Cheng University, Institute of Computer Science and Information Engineering, Chiayi, Taiwan 62107, Republic of China

The *multisearch problem* is defined as follows. Given a data structure D modeled as a graph with n constant-degree nodes, perform $O(n)$ searches on D. Let r be the length of the longest search path associated with a search process, and assume that the paths are determined "on-line." That is, the search paths may overlap arbitrarily. In this paper, we solve the multisearch problem for certain classes of graphs in $O(\sqrt{n} + r(\sqrt{n}/\log n))$ time on a $\sqrt{n} \times \sqrt{n}$ n mesh-connected computer. For many data structures, the search path traversed when answering one search query has length $r = O(\log n)$. For these cases, our algorithm processes $O(n)$ such queries in asymptotically optimal $\Theta(\sqrt{n})$ time. The classes of graphs we consider contain many of the important data structures that arise in practice, ranging from simple trees to Kirkpatrick hierarchical search DAGs. Multisearch is a useful abstraction that can be used to implement parallel versions of standard sequential data structures on a mesh. As example applications, we consider a variety of parallel on-line tree traversals, as well as hierarchical representations of polyhedra and its myriad of applications (line–polyhedron intersection queries, multiple tangent plane determination, intersecting convex polyhedra, and three-dimensional convex hull).

1. INTRODUCTION

Let D be a data structure modeled as a graph G with n constant-degree nodes. The *multisearch problem* consists of performing $O(n)$ searches on D, where the searches need not be processed in any particular order. Further, the searches may be simultaneously processed in parallel by using, for example, one processor per search. However, the path that an individual search will trace in G is *not* known ahead of time, and must instead be determined "on-line." That is, only when a search query is at node v of G can it determine which node of G it should visit next. (This is accomplished by comparing the search key to the information stored at v. It should be noted that the nature of the information stored at the nodes, as well as the nature of the comparison that is performed at every node, depends on the specific problem being solved.) It is important to note that the paths of

* A preliminary version of this work appeared in the *Proc. 1991 ACM Symposium on Parallel Algorithms and Architectures* (pp. 204–214).

† Research partially supported by the Office of Naval Research under Contracts N00014-84-K-0592 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

‡ Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

§ Research partially supported by the National Science Foundation under Grant IRI-8800514.

¶ Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

|| Research partially supported by the Office of Naval Research under Contract N00014-84-K-0502, the Air Force Office of the Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the ROC National Science Council under Contract NSC80-0408-E194-13.

the search queries can overlap arbitrarily. That is, at any time, any node of G may be visited by an arbitrary number of search queries.

Multisearch is a useful abstraction that can be used to implement parallel versions of standard sequential data structures on a mesh. The multisearch problem is a challenging problem both for EREW-PRAMs and for networks of processors. This is due to the fact that many search queries might want to visit a single node of G, creating a "congestion" problem. In fact, this problem of congestion can be complicated by the fact that we cannot even tally ahead of time the amount of congestion that will occur at a node, since we do not know ahead of time the full paths of the search queries, only the nodes of G at which the queries start. On the PRAM, the graph G is stored in the shared memory in the standard way. When the parallel model used to solve the problem is a network of processors, the graph G is initially stored in the network such that each processor contains one node of G, as well as that node's adjacency list. It is important to keep in mind that the computational network's topology is *not* the same as the search structure G, so that a neighbor of node v in G need not be stored in a processor adjacent to the one containing v.[1] Initially, the $O(n)$ search queries are arbitrarily distributed one per processor.

In the EREW-PRAM, the difficulty of providing an efficient solution to the multisearch problem comes from the "exclusive read" restriction of the model. A very elegant way around this restriction was given by Paul *et al.* [26] for the case where G is a 2-3 tree. However, it should be noted that they assume a linear ordering on the search keys. We cannot afford to make this assumption since we consider applications involving multidimensional search keys for which no linear ordering can be used.

The multisearch problem appears to be even more challenging for networks of processors than it is for the EREW-PRAM, due to the fact that the data structure is distributed over a network. Furthermore, similar to the EREW-PRAM, each memory location can be accessed only by a constant number of search queries at a time since a processor containing, say, node v's information would be unable to simultaneously store more than a constant number of search queries.

The main contribution of this paper is in solving the multisearch problem for certain classes of graphs in $\Theta(\sqrt{n} + r(\sqrt{n}/\log n))$ time on a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer, where r is the length of the longest search path associated with a query. Note that for many data structures the search path traversed when answering a query has length $r = O(\log n)$. For this situation, our algorithm processes $O(n)$ search queries in asymptotically optimal $\Theta(\sqrt{n})$ time.

The classes of graphs considered include many important data structures that arise in practice, ranging from simple trees to the powerful Kirkpatrick hierarchical search DAGs that are so important to solving problems in computational geometry. We will show how to exploit our multisearch algorithm to efficiently implement parallel on-line tree traversals as well as to traverse hierarchical representations of polyhedra. The latter yield solutions to problems including line-polyhedron intersection queries, multiple tangent plane determination, three-dimensional convex hull[2], and intersection of convex polyhedra. Notice that these problems are of considerable importance in robotics, solid modeling, computational geometry, vision, and pattern recognition, to name a few.

We believe that the multisearch problem is such a fundamental problem that we expect it to have additional applications (e.g., in parallel databases and related areas).

The multisearch problem for *hypercube multiprocessors* was studied in [8]. The hypercube technique presented in [8] was based on the idea of moving the search queries synchronously through G, and required time proportional to the diameter of the network to move all queries to the next nodes' in their search paths. Unfortunately, such an approach is not viable on the *mesh*, since in order to obtain an *optimal* mesh algorithm to solve the multisearch problem, the time per advancement of all queries by one step needs to be $O(\sqrt{n}/\log n)$, which is *less* than the diameter of the network. The techniques we use to solve the multisearch problem for the mesh are very different from those used in [8], and they are also very different from those used in [26].

In very broad terms, our techniques for solving the multisearch problem are a judicious combination of the following ideas.

- Partition G into pieces, some of which are processed sequentially, while others are processed in parallel.
- Create multiple copies of those pieces of G for which too many searches need access, and distribute the copies to disjoint submeshes, each of which is responsible for advancing a manageable subset of the "congested" searches. It should be noted that the straightforward strategy of making multiple copies of G, and using one copy for each search, does not work. This is due to the

[1] Note that due to the congestion problem, even an efficient embedding of the graph G into the network will not lead to an efficient multisearch algorithm.

[2] The 3-d convex hull problem has optimal mesh solutions recently obtained [20, 16] independently of ours and using very different, purely geometric approaches, rather than the multisearch method we use.

fact that it would not only take too much time to create the $O(n)$ copies, but there is not enough space to store all of these copies of G. In fact, there is only enough space to store $\Theta(1)$ copies of G, since G has n nodes.

- Map some pieces of G into suitably shaped portions of the mesh, which are not necessarily rectangular submeshes.

Of course, the parameters needed to efficiently perform these partitioning, duplication, and mapping strategies cannot be precomputed, since the full search paths are computed on-line. Therefore, these parameters must also be determined on-line, as the searches advance through G. The above description is necessarily an over simplification, and only a careful look at the details can reveal the exact interplay between the above ideas, as well as the exact nature of each.

The classes of graphs considered in this paper include hierarchical directed acyclic graphs (i.e., hierarchical DAGs) and partitionable graphs, which contain many important data structures that arise in practice.

Hierarchical DAGs consist of a vertex set that can be partitioned into $h = O(\log n)$ levels, $L_0, \ldots, L_h$, such that every edge is from some L_i to L_{i+1}, $|L_0| = 1$, and $c_1\mu^i \leq |L_i| \leq c_2\mu^i$, for some $\mu > 1$ and positive constants c_1 and c_2. An important member of this class of graphs is the Kirkpatrick subdivision hierarchies [19]. Once an optimal mesh implementation of multisearch for these graphs is obtained, new optimal mesh algorithms for numerous geometric problems follow immediately.

Partitionable graphs will be defined in detail later, but it should be noted that an important member of this class of graphs is the balanced k-ary tree. For partitionable graphs, we consider the multisearch problem for both the *undirected* and the *directed* case. For tree data structures, the directed partitionable graphs model tree algorithms for which search queries move along tree edges only in one direction, either from the root towards the leaves, or from the leaves towards the root. Many standard tree searches are of this type. Undirected partitionable graphs model tree algorithms for which search queries are permitted to move within the tree in an arbitrary manner. Such cases arise when queries are traversing parts of a tree, for example, in order. Note that other instances of the multisearch problem for search trees have been further studied in [31].

The next section contains a more formal definition of the multisearch problem, and of the various terms used in the paper. Sections 3.1 and 3.2 contain the main results: our solutions to the multisearch problem for each of the above-mentioned classes of graphs. Section 4 illustrates the use of multisearch to solve various problems efficiently on the mesh.

2. DEFINITIONS

In this section we will define the model of computation, the multisearch problem, and the classes of graphs for which we will present efficient multisearch algorithms in Section 3.

2.1. The Mesh-Connected Computer

The *mesh-connected computer (mesh) of size n* is a SIMD machine with n simple *processors* arranged in a square lattice. To simplify the exposition, it is assumed that $n = 4^c$, for some integer c. For all $i, j \in [0, \ldots, n^{1/2} - 1]$, let $P_{i,j}$ represent the processor in row i and column j. Processor $P_{i,j}$ is connected via bidirectional unit-time communication links to its four *neighbors*, $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$, and $P_{i,j+1}$, assuming they exist. Each processor has a fixed number of $\Theta(\log n)$ bit words of memory (registers), and can perform standard arithmetic and Boolean operations on the contents of these registers in unit time. Each processor can also send or receive a word of data to or from one of its neighbors in unit time.

The communication diameter of a mesh of size n is $\Theta(\sqrt{n})$, as can be seen by examining the distance between processors in opposite corners of the mesh. This means that if a processor in one corner of the mesh needs data from a processor in another corner of the mesh at some time during an algorithm, then a lower bound on the running time of the algorithm is $\Omega(\sqrt{n})$. It is easy to see that, because of the communication diameter, the problems in this paper have time complexities $\Omega(\sqrt{n})$.

In this paper, we will frequently use $\Theta(\sqrt{n})$ time *standard mesh operations* such as sorting, random access read, random access write, compression, parallel prefix, and list ranking [4, 23, 24, 25, 29].

2.2. The Multisearch Problem

Let $G = (V, E)$ be a directed or undirected graph of size $n = |V| + |E|$, where the out-degree or degree, respectively, of any vertex is bounded by some constant. Let U be a universe of possible *search queries* on G. Define the *search path* of a query $q \in U$, denoted $path(q)$, to be a sequence of h vertices $(v_1, \ldots, v_h)$ of G defined by a successor function f: $(V \cup start) \times U \rightarrow V$ as

- $f(start, q) = v_1$, and
- $f(v_i, q) = v_{i+1}$ for $i = 1, \ldots, h - 1$.

The function f has the following properties.

- If G is directed, then for every vertex $v \in V$ and query $q \in U$, $(v, f(v, q)) \in E$.
- If G is undirected, then for every vertex $v \in V$ and query $q \in U$, $\{v, f(v, q)\} \in E$.

• $f(v, q)$ can be computed in $\Theta(1)$ time by a single processor that contains the information pertinent to q and v.

We say that a query $q \in U$ *visits* a node $v \in V$ at time t if and only if, at time t, the mesh is in a state where there exists a processor which contains a description of both the query q and the node v. (Note that this definition implies that many queries can simultaneously visit node v, if each such query uses a different copy of v's information.) The *search process* for a search query q with search path $path(q) = (v_1, \ldots, v_h)$ is a process divided into h time steps, $t_1 < t_2 < \cdots < t_h$, such that at time t_i, $1 \le i \le h$, query q visits node v_i. We will refer to the change of state between t_i and t_{i+1}, $1 \le i < h$, as *advancing query q one step in its search path*. It is important to note that we do not assume the search path to be given in advance. In fact, we assume that the search path for each query is constructed *on-line* during the search by successive applications of the function f.

Note that for a *directed* graph, a query can be advanced along an edge only in the indicated direction, whereas for *undirected* graphs a query can advance along an edge in both directions.

Given a set $Q = \{q_1, \ldots, q_m\} \subseteq U$ of m search queries, where $m = O(n)$, then the *multisearch problem* for Q on G consists of executing (in parallel) all m search processes induced by the m search queries. It is important to note that the m search processes can overlap arbitrarily. That is, at any time t, any node of G may be visited by an arbitrary number of queries, which may, in fact, be at very different time steps in their respective search paths (of course each such query would be using a different copy of v's information).

We will refer to the process of advancing, in parallel, a subset of the m search queries by one step in their respective search paths as a *multistep*. Notice that we do not require all queries to be advanced synchronously. We will refer to a sequence of multisteps which has the property that every search query is advanced $\Omega(\log n)$ steps in its respective search path, as a *log-phase*.

A convenient way of visualizing the multisearch process is by associating a *pebble* with each query. Initially, the pebble associated with query q is placed on the first node in $path(q)$. During the multisearch process, the m pebbles move in parallel along edges of G, each pebble according to its respective search path. Each node of the graph may be visited, at any time, by an arbitrary number of pebbles. Notice that if G is undirected, then pebbles can move freely along edges of the graph, while if G is directed, then pebbles can only move in the proper direction of an edge. Note that, pebbles may move with different and possibly changing speeds.

For the remainder of this paper, we will assume that G is connected (by a "connected" directed graph we mean that the undirected version of that graph is connected). For graphs with several connected components, the multisearch algorithms described in Sections 3.1 and 3.2 can be easily applied independently and in parallel to each connected component, such that the overall time complexity remains unchanged.

2.3. Hierarchical DAGs

Let $G = (V, E)$ be a directed acyclic graph with vertex set V, edge set E, and size $n = |V| + |E|$, where the outdegree of any vertex is bounded by some constant. The graph G is called a *hierarchical DAG of size n and height h* if and only if V can be partitioned into $h + 1$ subsets $L_0, \ldots, L_h$ such that

1. $h = O(\log n)$,
2. $|L_0| = 1$,
3. There exists a constant $\mu > 1$ such that, for all $i \in \{0, \ldots, h - 1\}$, $|L_{i+1}| = \mu |L_i|$.
4. For every directed edge $(v, w) \in E$, there exists an $i \in \{0, \ldots, h - 1\}$ such that $v \in L_i$ and $w \in L_{i+1}$.

See Fig. 1 for an illustration. The subsets $L_0, \ldots, L_h$ are called the *levels* of G. For a node $v \in L_i$, the index i is called the *level index* of v. Notice that Requirement 3 implies that $|L_i| = \mu^i$. This requirement is introduced to simplify the exposition of our algorithm in Section 3.1. However, our algorithms can be easily adapted to the case $c_1 \mu^i \le |L_i| \le c_2 \mu^i$, for some positive constants c_1 and c_2. It should be noted that subdivision hierarchies, as described in [19], are hierarchical DAGs.

2.4. Partitionable Graphs

2.4.1. δ-Splitters. Let $G = (V, E)$ be a (directed or undirected) graph with vertex set V, edge set E, and size $n = |V| + |E|$. Let $S \subset E$. Then $(V, E - S)$ is a graph with vertex set V and edge set $E - S$ that consists of a set of $k \le n$ connected components, denoted $\{G_1, \ldots, G_k\}$.

We define S to be a *δ-splitter* of G, $0 < d < 1$, if and only if $|G_i| = |V_i| + |E_i| = O(n^\delta)$, for all $1 \le i \le k$. Given a δ-splitter S, we will refer to $G(S) = \{G_1, \ldots, G_k\}$ as a *δ-splitting of G*.

A vertex $v \in V$ is defined to be *at the border* of a δ-splitter S if and only if v is a vertex of an edge $e \in S$. A δ-

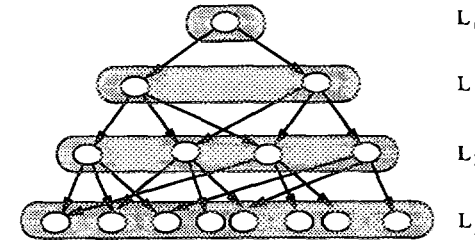

FIG. 1. A hierarchical DAG with $\mu = 2$.

splitting $G(S) = \{G_1, \ldots, G_k\}$ is called *normalized*, if $k = O(n^{1-\delta})$.

2.4.2. α-Partitionable (directed) graphs. Let $G = (V, E)$ be a directed graph with vertex set V, edge set E, and size $n = |V| + |E|$, where the out-degree of any vertex is bounded by some constant. Let $dist_G(v_1, v_2)$ denote the length of a shortest directed path in G connecting vertices v_1 and v_2. We define G to be *α-partitionable* if and only if G has an α-splitter S, $0 < \alpha < 1$, such that $G(S) = \{G_1, \ldots, G_k\}$ can be partitioned into two sets of graphs, $\{H_1, \ldots, H_{k_1}\}$ and $\{T_1, \ldots, T_{k_2}\}$, such that for every directed edge $(v_1, v_2) \in S$, $v_1 \in H_i$ and $v_2 \in T_j$, for some i, j.

Note that, for example, every balanced k-ary search tree with all edges either directed towards the leaves or directed towards the root (i.e., all search queries can only move in one direction, either from the root towards the leaves, or from the leaves towards the root) is α-partitionable; see Fig. 2.

2.4.3. α-β-Partitionable (undirected) graphs. Let $G = (V, E)$ be an undirected graph with vertex set V, edge set E, and size $n = |V| + |E|$, where the degree of any vertex is bounded by some constant. For two vertices $v_1, v_2 \in V$, let $dist_G(v_1, v_2)$ denote the length of a shortest (undirected) path in G connecting v_1 and v_2.

Let S_1 and S_2 be an α-splitter and a β-splitter, respectively, of G. We define S_1 and S_2 to *have distance k* if and only if $k = \min\{dist_G(v_1, v_2)$: v_1 is at the border of S_1 and v_2 is at the border of $S_2\}$.

G is called *α-β-partitionable* if and only if G has an α-splitter S_1 and a β-splitter S_2, such that S_1 and S_2 have distance $\Omega(\log n)$.

Note that, for example, every undirected balanced k-ary search tree (i.e., search queries can move within the tree in arbitrary direction) is α-β-partitionable; see Fig. 3.

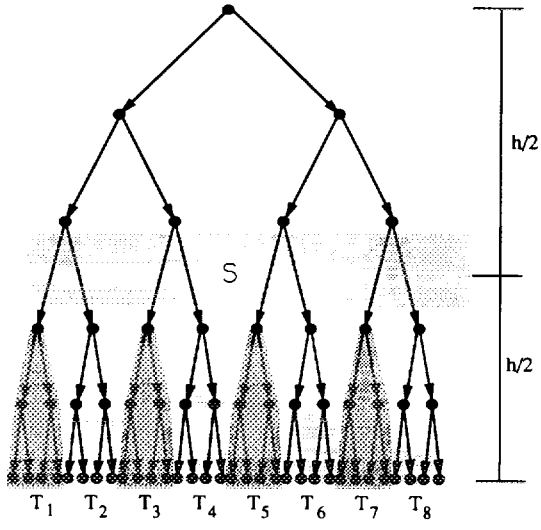

FIG. 2. A directed balanced binary tree and its α-splitter ($\alpha = \frac{1}{2}$).

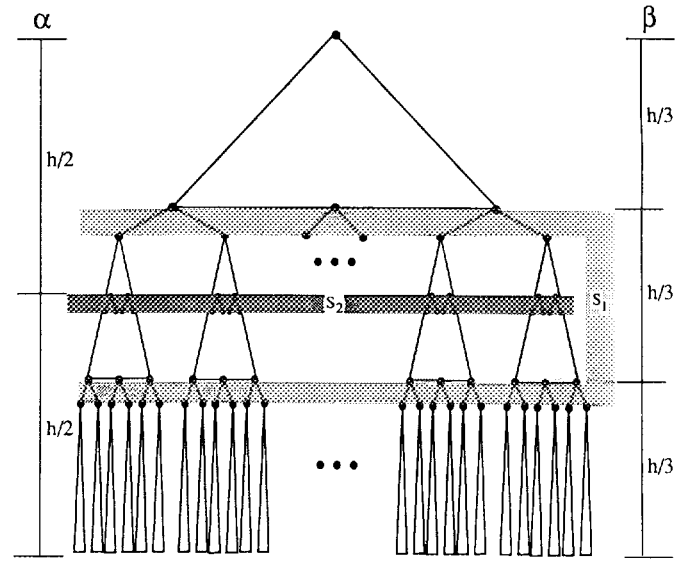

FIG. 3. A undirected balanced binary tree with its α-splitter S_1 ($\alpha = \frac{1}{2}$) and β-splitter S_2 ($\beta = \frac{1}{3}$), such that S_1 and S_2 have distance $h/6 = \Omega(\log n)$.

3. MESH SOLUTIONS TO THE MULTISEARCH PROBLEM

In this section, we present mesh solutions to the multisearch problem for hierarchical DAGs, α-partitionable graphs, and α-β-partitionable graphs.

First, we define some notation that will be used throughout this section. Define $G = (V, E)$ to be a graph with vertex set V, edge set E, and size $n = |V| + |E|$. In each subsection, we will specify whether the graph is directed or undirected. For directed graphs, we assume that the out-degree of every vertex is bounded by some constant, and for undirected graphs, we assume that the degree of every vertex is bounded by some constant. Finally, we define $Q = \{q_1, \ldots, q_m\}$ to be a set of $m = O(n)$ search queries.

We now discuss the manner in which G and Q will be represented on the mesh. Every processor will initially store

- one arbitrary vertex $v \in V$,
- the addresses of all processors storing a vertex $w \in V$, such that $(v, w) \in E$ (recall that G has out-degree $\Theta(1)$), and
- one arbitrary query $q \in Q$.

During an algorithm, no processor will store information associated with more than $\Theta(1)$ items of V nor more than $\Theta(1)$ items of Q. Notice that the assignment of vertices and queries to processors may change during the course of the algorithms. In addition, we assume that every processor p has a register *visit*(p), where at any stage of a multisearch algorithm, a query $q \in Q$ will be said to *visit*

a node $v \in V$ if processor p is responsible for query q and stores a copy of v in *visit*(p).

3.1. *The Multisearch Problem for Hierarchical DAGs*

Let $G = (V, e)$ be a hierarchical DAG of size n and height h. Let $L_0, \ldots, L_h$ be the levels of G. Recall that this implies G has out-degree $\Theta(1)$, $h = O(\log n)$, and $|L_i| = \mu^i$, for some $\mu > 1$.

Consider a set $Q = \{q_1, \ldots, q_n\}$ of n search queries. Due to the structure of the hierarchical DAG, a search path for a query q has length $r \leq h + 1$ and consists of r vertices in consecutive levels $L_i, \ldots, L_{i+r-1}$, for some $i \in \{0, \ldots, h - r + 1\}$. We will henceforth assume, w.l.o.g., that each query has a search path of length $h + 1$.

In this section we show how to solve the multisearch problem for G and Q on a mesh-connected computer of size n in time $\Theta(\sqrt{n})$. The initial configuration of the machine is as given at the beginning of Section 3. In addition, we assume that every processor storing a node v also stores the level index of v in G. Note that the level indices can be easily computed in time $\Theta(\sqrt{n})$ by successively identifying the vertices in each level L_i, starting with level L_h, and compressing after each step the remaining levels into a subsquare of processors.

For $i \geq 1$, we will use $\log^{(i)}$ to denote the function obtained by applying the log function i times, i.e., $\log^{(1)} x = \log x$ and $\log^{(i)} x = \log \log^{(i-1)} x$. For convenience, we define $\log^{(0)} x = x/2$. Note that there exists a constant c such that $\mu^y \geq y^2$ for any $y \geq c$. For any $x \geq \mu^c$, we define $\log^*_\mu x = \max\{i | \log^{(i)}_\mu x \geq c\}$. Hence, $\log^{(i)}_\mu x \geq (\log^{(i+1)}_\mu x)^2$ for $0 \leq i \leq \log^*_\mu x - 1$. For the remainder of this section, all logarithms are taken to be the base μ.

Let $B_i = (V_i, E_i)$, $0 \leq i \leq \log^* h - 1$, be the subgraph of G induced by the vertices of G between levels $h - 2\log^{(i)} h$ and $h - 1 - 2\log^{(i+1)} h$, inclusive. We will use $|B_i|$, $h_i = h - 1 - 2\log^{(i+1)} h$, and Δh_i, to refer to the size of B_i, the highest index of a level in B_i, and the number of levels in B_i, respectively. See Fig. 4 for an illustration. Notice that $|B_i| = \Theta(\mu^{h-2\log^{(i+1)}h}) = \Theta(n/(\log^{(i)}h)^2)$ and $\Delta h_i = \Theta(\log^{(i)}h)$.

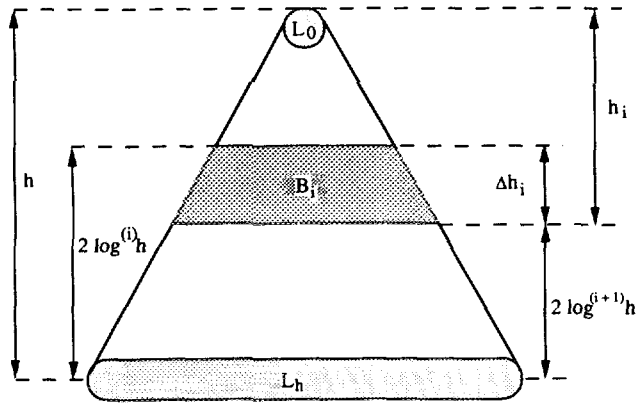

FIG. 4. Illustration of the definition of subgraphs B_i.

Let B^* be the subgraph induced by the vertices between levels $h - 2\log^{(\log^* h - 1)} h$ and h, inclusive. Notice that B^* consists of $\Theta(1)$ levels.

The general strategy for solving the multisearch problem on G is to solve the multisearch problem for B_0, then for B_1, and so on, until we solve the problem for $B_{\log^* h - 1}$, and finally for B^*. That is, we first consider those queries which originate in B_0, and process them until they either terminate or wish to leave B_0. Next, we process those queries that wanted to leave B_0 (for B_1), as well as those queries which originate in B_1, and process them until either they terminate or wish to leave B_1 (for B_2). This process continues until all queries terminate that need to be processed by B^*.

Since B^* has $\Theta(1)$ levels, the multisearch problem for B^* can be easily solved in time $O(\sqrt{n})$. What remains to be shown is how to solve the multisearch problems for $B_0, \ldots, B_{\log^* h - 1}$ in total time $O(\sqrt{n})$.

Consider the partitioning of the entire mesh-connected computer into $\log^{(i)} h \times \log^{(i)} h$ submeshes of $(\sqrt{n}/\log^{(i)} h) \times (\sqrt{n}/\log^{(i)} h)$ processors. Such a partitioning will be called a *B_i-partitioning*, and each submesh will be called a *B_i-submesh*. Notice that each *B_i-submesh*. Notice that each B_i-submesh can store a copy of the subgraph B_i. Further, notice that every B_{i+1}-submesh, Δ, contains several B_i-submeshes. We will refer to the top-left B_i-submesh as the *top-left B_i-submesh of Δ*.

LEMMA 1. *Consider a B_i-partitioning of the mesh-connected computer, $0 \leq i \leq \log^* h - 1$, and assume that every B_i-submesh stores a copy of B_i. Then the multisearch problem for B_i can be solved in time $O(\sqrt{|B_i|} \log \Delta h_i) = O(\sqrt{|B_i|} \log^{(i+1)} h)$.*

Proof. Let B_i^1 be the subgraph of G induced by the vertices of G between levels $h_i - \Delta h_i$ and $h_i - 1 - 2 \log \Delta h_i$, inclusive, and let B_i^2 be the subgraph induced by the vertices between levels $h_i - 2 \log \Delta h_i$ and h_i, inclusive. See Fig. 5 for an illustration. Notice that $|B_i^1| = O(\mu^{h_i - 2\log \Delta h_i}) = O(|B_i|/(\Delta h_i)^2)$. On every B_i-submesh in parallel, we will solve the multisearch problem for B_i for those queries stored in that submesh. We next describe our solution for one B_i-submesh. The solution consists of two phases. In Phase 1, every query visits the vertices on its search path that lie in B_i^1; in Phase 2 the queries will visit the vertices on their search path that lie in B_i^2. For Phase 1, the B_i-submesh is partitioned into $\Delta h_i \times \Delta h_i$ submeshes of size $|B_i|/(\Delta h_i)^2$, called *B_i^1-submeshes*. Notice that every B_i^1-submesh can store a copy of B_i^1. In time $O(\sqrt{|B_i|})$, we can identify B_i^1 from B_i and duplicate B_i^1 such that every B_i^1-submesh contains a copy of B_i^1. Each B_i^1-submesh then (independently and in parallel) solves the multisearch problem for B_i^1 for those queries stored in

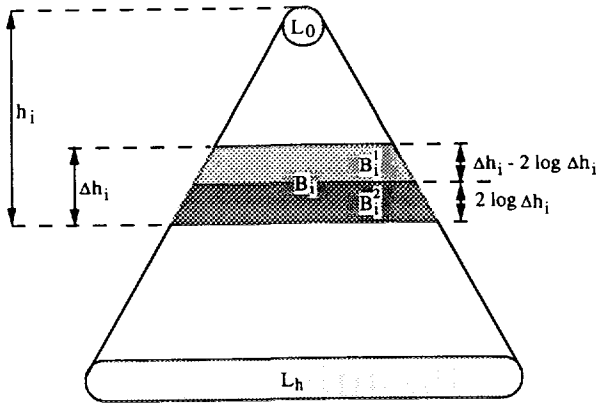

FIG. 5. Illustration of the definition of subgraphs B_i^1 and B_i^2.

that submesh. This can be easily done in $O(\sqrt{|B_i|})$ time since $|B_i^1| = O(|B_i|/(\Delta h_i)^2)$ and B_i^1 consists of $O(\Delta h_i)$ levels. For Phase 2, the search process is advanced level by level. Since B_i^2 consists of $O(\log \Delta h_i)$ levels, Phase 2 can be executed in $O(\sqrt{|B_i|} \log \Delta h_i)$ time. Thus, the time complexity of the above process is $O(\sqrt{|B_i|} \log \Delta h_i)$. ■

Obviously, if every B_i-submesh stores a copy of B_i then we need $O(\log^* n)$ memory per processor. Our strategy will be to distribute the subgraphs B_i over the mesh in such a way that, when the multisearch problem for B_i needs to be solved, all of the required copies of B_i can be created in time $O(\sqrt{|B_{i+1}|})$. From this, we obtain a $O(\sqrt{n})$ time solution to the multisearch problem for G.

To simplify the presentation, we assume $\log^{(i)} h$ is divisible by $\log^{(i+1)} h$, for $0 \le i \le \log^* h - 1$. Our algorithm can easily be modified to handle the general case. Let $B_{\log^* h}$-*submesh* denote the entire mesh.

ALGORITHM 1. An algorithm for solving the multisearch problem for a hierarchical DAG G.

1. A register *label*(p) is allocated at every processor p, and the following is executed for $i = \log^* h - 1, \ldots, 0$:

• In each B_{i+1}-submesh, Δ, every processor p in the top-left B_i-submesh of Δ sets *label*(p) := i.

Notice that the label of a processor may be overwritten by smaller indices in later iterations.

2. For $i = \log^* h - 1, \ldots, 0$, on each B_{i+1}-submesh the following is executed independently and in parallel:

(a) The subgraph B_i is identified and its data is distributed evenly among the processors with *label* = i.

(b) $(\log^{(i)} h/\log^{(i+1)} h)^2$ copies of the union of $B_0, \ldots, B_{i-1}$ are created and one copy is moved to each B_i-submesh.

Note that, after this step, each $B_{(i+1)}$-submesh stores a copy of B_i using the processors with *label* = i.

3. For $i = 0, \ldots, \log^* h - 1$, on each B_{i+1}-submesh the following is executed independently and in parallel:

(a) B_i is duplicated such that each B_i-submesh stores a copy of B_i.

(b) For each B_i-submesh, the multisearch problem for B_i with respect to those queries stored in that submesh is solved as indicated by Lemma 1.

4. Finally, the multisearch problem for B^* is solved.

THEOREM 2. *Let G be a hierarchical DAG of size n and let $Q = \{q_1, \ldots, q_m\}$ be a set of $m = O(n)$ search queries. Then the multisearch problem for Q on G can be solved on a mesh of size n (with $\Theta(1)$ memory per processor) in $\Theta(\sqrt{n})$ time.*

Proof. We first study the correctness of Algorithm 1, and then give some implementation details and prove the claimed time complexity and space requirement. In Steps 1 and 2, each B_i, for $0 \le i \le \log^* h - 1$, is duplicated such that every B_{i+1}-submesh contains one copy of B_i. In Step 3, the multisearch problem is solved sequentially for $B_0, B_1, \ldots, B_{\log^* h - 1}$. Notice that within every B_{i+1}-submesh, $0 \le i \le \log^* h - 1$, the graph B_i is copied into every B_i-submesh, such that Lemma 1 can be applied to solve the multisearch problem for B_i. Finally, in Step 4, the multisearch problem for B^* is solved. Thus, the multisearch problem for G is solved.

Next, we analyze the space complexity of Algorithm 1, showing that only $\Theta(1)$ space is required per processor. This is obvious for Steps 1, 3 and 4; a potential problem lies in the duplication scheme in Step 2. For Step 2(b) we observe that $\sum_{j=0}^{i-1} |B_j| = O(|B_i|)$ and, hence, it requires only $\Theta(1)$ storage per processor. For Step 2(a), we need to show that in each B_i-submesh there are $\Omega(|B_i|)$ processors with *label* = i. Note that for $j \le i - 1$, each B_{j+1}-submesh contains one B_j-submesh in its top-left corner whose processors' labels are set to j (see Step 1). That is, in Step 1, the labels of at most $(n/(\log^{(i)} h)^2)\ (\log(j+1)\ h/\log^{(j)} h)^2$ processors are changed from i to j. Hence, the number of processors in each B_i-submesh with *label* = i is $\Omega(n/(\log^{(i)} h)^2)\ (1 - \sum_{j=0}^{i-1} (\log(j+1)\ h/\log^{(j)} h)^2)) = \Omega(n/(\log^{(i)} h)^2)$. Since $|B_i| = O(n/(\log^{(i)} h)^2)$, these processors can store B_i with $\Theta(1)$ storage per processor provided that the B_i's data can be evenly distributed among them. This can be achieved in $O(\sqrt{n})$ time, using a combination of the standard mesh operations. Summarizing, we have shown that Algorithm 1 requires $\Theta(1)$ storage per processor.

Next, we prove the time complexity of Algorithm 1. Since $\sum_{i=0}^{\log^* h - 1} \sqrt{|B_i|} = O(\sqrt{n})$ and $O(\sum_{i=0}^{\log^* h - 1} \sqrt{|B_{i+1}|}) = O(\sqrt{n})$, the time complexity of Steps 1 and 2 is $\Theta(\sqrt{n})$. Since B^* consists of $\Theta(1)$ levels, the $\Theta(\sqrt{n})$ time complexity of Step 4 is obvious. Since each B_{i+1}-submesh contains one copy of B_i, the total time complexity for Step 3a (over all iterations) is $O(\sum_{i=0}^{\log^* h - 1} \sqrt{|B_{i+1}|}) =$

$O(\sqrt{n})$. From Lemma 1 it follows that for each $i = 0, \ldots, \log^* h - 1$, the time complexity of Step 3b is $O(\sqrt{|B_i|} \log \Delta h_i)$. Thus, the total time for all iterations of Step 3b is $O(\sum_{i=0}^{\log^* h - 1} \sqrt{|B_i|} \log \Delta h_i) = O(\sum_{i=0}^{\log^* h - 1} \sqrt{n}\, (\log^{(i+1)} h / \log^{(i)} h)) = O(\sqrt{n})$. Hence, the time complexity of Algorithm 1 is $\Theta(\sqrt{n})$. ■

3.2. The Multisearch Problem For Partitionable Graphs

In this section, we present mesh solutions to the multisearch problem for α-partitionable graphs and α-β-partitionable graphs. We will first introduce a tool referred to as *constrained-multisearch*, which will be utilized in Sections 3.2.2 and 3.2.3.

3.2.1. Constrained-Multisearch. Let $G = (V, E)$ be a directed or undirected graph. Consider a set $\Psi = \{G_1, \ldots, G_k\}$ of k edge and vertex disjoint subgraphs of G such that $|G_i| = O(n^\delta)$ and $k = O(n^{1-\delta})$, for some $0 < \delta < 1$. It is important to note that we do not assume that the union of the subgraphs in Ψ contains all vertices of G.

Consider any stage of the multisearch for Q on G, and let $v(q) \in path(q)$ denote the node currently visited by query $q \in Q$.

The *constrained-multisearch problem* with respect to Ψ consists of advancing, for every $G_i \in \Psi$, every search query q with $v(q) \in G_i$ by $\log_2 n$ steps in its search path, unless the next node to be visited by q is not in G_i. Notice that the queries may be advanced by a nonuniform number of steps.

The remainder of this section focuses on procedure *Constrained-Multisearch*(Ψ, δ), which solves the constrained multisearch problem on a mesh of size n in $\Theta(\sqrt{n})$ time.

For every $G_i = (V_i, E_i) \in \Psi$, we define

$$\Gamma_\Psi^\delta(G_i) = \left\lceil \frac{|\{q \in Q: v(q) \in V_i\}|}{n^\delta} \right\rceil.$$

PROPERTY 1.

$$\sum_{G_i \in \Psi} \Gamma_\Psi^\delta(G_i) = O(n^{1-\delta}).$$

Proof. A trivial consequence of the fact that $|Q| = m = O(n)$. ■

We now present our mesh algorithm for solving the constrained-multisearch problem with respect to Ψ.

PROCEDURE *Constrained-Multisearch* (Ψ, δ). Implementation of constrained multisearch with respect to Ψ.

Initial configuration. A stage of the multisearch for Q on G, where every $q \in Q$ currently visits node $v(q) \in path(q)$. Furthermore, every processor storing a vertex $v \in V$, also stores an index indicating to which $G_i \in \Psi$ the vertex v belongs, if any.

Implementation.

1. All queries $q \in Q$, such that $v(q)$ is in some subgraph $G_i \in \Psi$, are marked *active*; all other queries are marked *inactive*. (Queries whose search paths have already terminated are also marked inactive.)
2. For every $G_i \in \Psi$, the value of $\Gamma_\Psi^\delta(G_i)$ is computed.
3. If

$$\sum_{G_i \in \Psi} \Gamma_\Psi^\delta(G_i) = 0$$

then **EXIT**.

4. For each $G_i \in \Psi$, $\Gamma_\Psi^\delta(G_i)$ copies of G_i are created. Each copy is placed in a $\sqrt{n^\delta} \times \sqrt{n^\delta}$ size subsquare of the mesh-connected computer. That is, a *submesh* of size $\sqrt{n^\delta} \times \sqrt{n^\delta}$.
5. Every active query $q \in Q$, with $v(q) \in G_i$, is moved to one of the submeshes storing a copy of G_i. This movement is coordinated so that each submesh containing a copy of G_i will receive $O(n^\delta)$ queries.
6. Within every submesh storing a subgraph $G_i \in \Psi$, the following is executed $\log_2 n$ times.
   (a) For every active query $q \in Q$, the next node in its search path is determined (by applying the successor function f).
   (b) Every active query for which the next node in its search path is not in G_i, is marked inactive. (A query whose search path terminates is also marked inactive.)
   (c) Every active query visits the next node in its search path.
7. Discard the copies of the subgraphs $G_i \in \Psi$ created in Step 4.

LEMMA 3. *The constrained-multisearch problem with respect to Ψ can be solved on a mesh of size n in $\Theta(\sqrt{n})$ time.*

Proof. We first study the correctness of *Constrained-Multisearch*(Ψ, δ), then give some implementation details, and finally prove the time complexity. Obviously, every query q either

- visits the next $\log_2 n$ nodes in its search path,
- visits the next N nodes in its search path, where $N < \log_2 n$, until the next node to be visited is no longer in the same subgraph $G_i \in \Psi$ that contains $v(q)$, or
- does not advance any steps in its search path, for the case where $v(q)$ is not in any $G_i \in \Psi$.

The crucial step for proving the correctness of the procedure is to show that (1) the total size of the copies of subgraphs G_i created in Step 4 is $O(n)$, and (2) in Step 5, the sizes the total number of queries to be moved match

the sizes and total number of submeshes available. Item (1) follows from Property 1, and Item (2) follows from the definition of $\Gamma_{\psi}^{\delta}(G_i)$ and the fact that each submesh is of size $O(n^{\delta})$.

We will now prove the claimed time complexity. Steps 1, 2, 3, and 7 can be easily implemented in time $O(\sqrt{n})$ by applying a constant number of standard mesh operations. For Step 4, the mesh is subdivided into a grid of $\sqrt{n^{1-\delta}} \times \sqrt{n^{1-\delta}}$ submeshes, each of size n^{δ}. The total number of copies created of every subgraph G_i is $O(n^{1-\delta})$ (Property 1). Hence, every submesh needs to simulate only a constant number of "virtual" submeshes, where each "virtual" submesh stores just one copy of some subgraph $G_i \in \psi$. Creating the required copies of subgraphs and moving them to the "virtual" submeshes can be implemented by a constant number of standard mesh operations. Step 5 is implemented analogously. Finally, we discuss the time complexity of Step 6. Notice that each execution of the loop body is executed independently and in parallel on every submesh of size $O(n^{\delta})$ created in Step 4. Therefore, by using standard random access read and write operations within every submesh, each iteration of the loop can be implemented in $O(\sqrt{n^{\delta}})$ time, which implies a total of $O(\log n\sqrt{n^{\delta}})$ time for Step 6 (since there are $\log_2 n$ iterations). Since $0 < \delta < 1$, the total time complexity of Step 6 is $O(\log n\sqrt{n^{\delta}}) = O(\sqrt{n})$. ■

3.2.2. The multisearch problem for directed α-partitionable graphs. Let $G = (V, E)$ be a directed α-partitionable graph. Let $Q = \{q_1, \ldots, q_m\}$ be a set of $m = O(n)$ search queries, and let r denote the length of the longest search path associated with a query $q \in Q$. In this section, we present an algorithm to solve the multisearch problem for Q on G in $O(\sqrt{n} + r(\sqrt{n}/\log n))$ time. Our strategy is to give an algorithm which executes one log-phase of multisearch in $(\sqrt{n})$ time. The entire multisearch algorithm consists of iterating the log-phase algorithm $O(\lceil r/\log n \rceil)$ times.

Let $G(S) = \{H_1, \ldots, H^{k_1}, T_1, \ldots, T_{k_2}\}$ be an α-splitting of G such that for every edge $(v_1, v_2) \in S$ (directed from v_1 to v_2), $v_1 \in H_i$ and $v_2 \in T_j$, for some $1 \le i \le k_1$, $1 \le j \le k_2$. Recall that this implies $0 < \alpha < 1$, $|H_i| = O(n^{\alpha})$, and $|T_i| = O(n^{\alpha})$.

We assume that the α-splitter S is known a priori. That is, initially the processor that stores vertex $v \in V$ also stores an index indicating the graph in $G(S)$ to which v belongs. We can also assume, without loss of generality, that $G(S)$ is normalized. That is, we can assume that $k = k_1 + k_2 = O(n^{1-\alpha})$; see Section 2.4.1. Otherwise, we group the subgraphs $H_i(T_i)$ such that each resulting subgraph has size $\Theta(n^{\alpha})$. This operation is easily performed on a mesh of size n in $O(\sqrt{n})$ time. Furthermore, the algorithm described in this section does not require that every subgraph in $G(S)$ consist of only one connected component of the graph $(V, E - S)$.

Before presenting our mesh algorithm for one log-phase of the multisearch problem for Q on G, we observe some properties of α-partitionable graphs.

PROPERTY 2. *Let $G(S) = \{H_1, \ldots, H_{k_1}, T_1, \ldots, T_{k_2}\}$ be an α-splitting of G. Then the following hold.*

- *A query q that has a node of a subgraph H_i in its search path does not visit any node of another subgraph H_j, $i \neq j$.*
- *Once a query q has visited a node in a subgraph T_i, all subsequent nodes visited by q will be in the same subgraph T_i.*

Proof. The proof follows from the fact that edges of an α-partitionable graph are either directed from some H_i to some T_j, or have both endpoints in the same subgraph H_i or T_i. ■

ALGORITHM 2. Implementation of one log-phase of multisearch on a directed α-partitionable graph.

1. If this is the first log-phase, then every query $q \in Q$ visits the first node in its search path; otherwise, every $q \in Q$ visits the next node in its search path.
2. *Constrained-Multisearch* $(\{H_1, \ldots, H_{k_1}, T_1, \ldots, T_{k_2}\}, \alpha)$.
3. Every $q \in Q$ visits the next node in its search path.
4. *Constrained-Multisearch* $(\{H_1, \ldots, H_{k_1}, T_1, \ldots, T_{k_2}\}, \alpha)$.

LEMMA 4. *One log-phase of multisearch on a directed α-partitionable graph of size n can be performed in $\Theta(\sqrt{n})$ time on a mesh of size n.*

Proof. We first consider the correctness of Algorithm 2. The algorithm is based on the following. Initially, every query starts at the first node in its search path, which is in some H_i, $1 \le i \le k_1$, or T_j, $1 \le j \le k_2$. Using *Constrained-Multisearch*, every query is advanced until it visits either its $\log_2 n$ successors, or needs to visit a node that is not in its initial subgraph, at which point it stops. Next, every query is advanced one node and then *Constrained-Multisearch* is performed again. So, by 2 applications of *Constrained-Multisearch*, every query will be advanced at least $\log_2 n$ nodes. (Note, if there are fewer than $\log_2 n$ nodes in a given search path, then that query will terminate at the appropriate time.) Property 2, it follows that for every query $q \in Q$, one of the following cases must apply:

1. All nodes visited by q within the log-phase are in one subgraph H_i.
2. All nodes visited by q within the log-phase are in one subgraph T_i.
3. Within the log-phase, query q first visits only nodes within one subgraph H_i, and once it "leaves" H_i it will only visit nodes in one subgraph T_j.

For those queries to which either Case 1 or Case 2 applies, all nodes visited on the search path during the log-phase are visited during Steps 1 and 2; see Lemma 3. Let q be a query to which Case 3 applies, and let $(v_1, \ldots, v_x, v_{x+1}, \ldots, v_y)$ be the sequence of nodes to be visited within the log-phase, where $v_1, \ldots, v_x$ are in some subgraph H_i, and $v_{x+1}, \ldots, v_y$ are in some subgraph T_j. It follows from Lemma 3 that $v_1, \ldots, v_x$ are visited during Steps 1 and 2, and that $v_{x+1}, \ldots, v_y$ are visited during Steps 3 and 4.

From Lemma 3 it also follows that Algorithm 2 has time complexity $\Theta(\sqrt{n})$ and requires only $\Theta(1)$ memory per processor. ■

Therefore, by iterating Algorithm 2 $O([r/\log n])$ times, the multisearch problem can be solved for α-partitionable graphs.

THEOREM 5. *Let G be a directed α-partitionable graph of size n, and let $Q = \{q_1, \ldots, q_m\}$ be a set of $m = O(n)$ search queries. Then the multisearch problem for Q on G can be solved in $O(\sqrt{n} + r(\sqrt{n}/\log n))$ time on a mesh of size n, where r is the length of the longest search path associated with a query $q \in Q$.* ■

3.2.3. The multisearch problem for undirected α-β-partitionable graphs. Let $G = (V, E)$ be an (undirected) α-β-partitionable graph. Let $Q = \{q_1, \ldots, q_m\}$ be the set of $m = O(n)$ search queries, and let r denote the length of the longest search path associated with a query $q \in Q$. In this section, we present an algorithm to solve the multisearch problem for Q on G in $O(\sqrt{n} + r(\sqrt{n}/\log n))$ time. As in Section 3.2.2, we will again give an algorithm to execute one log-phase of the multisearch problem in $\Theta(\sqrt{n})$ time. The multisearch algorithm will consist of iterating this log-phase algorithm $O([r/\log n])$ times.

Let S_1 and S_2 be an α-splitter and a β-splitter, respectively, of G such that S_1 and S_2 have distance $\Omega(\log n)$. We assume that S_1 and S_2 are known a priori. That is, initially the processor that stores vertex $v \in V$ also stores an index indicating the graph $G(S_1)$ to which v belongs, and an index indicating the graph $G(S_2)$ to which v belongs.

With the same argument as in Section 3.2.2, we also assume that $G(S_1)$ and $G(S_2)$ are normalized. Let $G(S_1) = \{W_1^1, \ldots, W_{k_1}^1\}$ and $G(S_2) = \{W_1^2, \ldots, W_{k_2}^2\}$. Recall that $0 < \alpha < 1$, $0 < \beta < 1$, $|W_i^1| = O(n^\alpha)$, $|W_i^2| = O(n^\beta)$, $k_1 = O(n^{1-\alpha})$, and $k_2 = O(n^{1-\beta})$.

We first state a property of α-β-partitionable graphs that will be used in the algorithm.

PROPERTY 3. *Let S_1 and S_2 be an α-splitter, and β-splitter, respectively, of G, such that S_1 and S_2 have distance $\Omega(\log n)$. Then, if at any stage of the multisearch, a query $q \in Q$ visits a node v at the border of S_1, it can advance $\Omega(\log n)$ more steps in its search path without visiting a node v' at the border of S_2.*

Proof. The proof follows immediately from the definition of α-β-partitionable graphs. ■

ALGORITHM 3. Implementation of one log-phase of multisearch on an α-β-partitionable graph.

1. If this is the first log-phase, then every query $q \in Q$ visits the first node in its search path; otherwise, every $q \in Q$ visits the next node in its search path.
2. *Constrained-Multisearch* $(\{W_1^1, \ldots, W_{k_1}^1\}, \alpha)$.
3. Every $q \in Q$ visits the next node in its search path.
4. *Constrained-Multisearch* $(\{W_1^2, \ldots, W_{k_2}^2\}, \beta)$.

LEMMA 6. *One log-phase of multisearch on an (undirected) α-β-partitionable graph of size n can be performed in $\Theta(\sqrt{n})$ time on a mesh of size n.*

Proof. We first consider the correctness of Algorithm 3. The algorithm is based on the following. Initially, every query starts at the first node in its search path. Using *Constrained-Multisearch* on $G(S_1)$, every query is advanced until it visits either its $\log_2 n$ successors, or needs to visit a node that is not in its initial subgraph, at which point it stops. Next, every query is advanced one node and then *Constrained-Multisearch* is performed again, but this time with respect to $G(S_2)$. Notice that by performing the second application of *Constrained-Multisearch* with respect to $G(S_2)$, every query that had reached a border of $G(S_2)$ will be able to advance $\Omega(\log n)$ more steps in its search path without visiting another node at the border of S_2; by this time, the log-phase is completed. Therefore, by two applications of *Constrained-Multisearch*, every query will be advanced at least $\log_2 n$ nodes. (Note, if there are fewer than $\log_2 n$ nodes in a given search path, then that query will terminate at the appropriate time.) That is, for every query $q \in Q$, one of the following cases applies:

1. All nodes visited by q within the log-phase are in one subgraph W_i^1.
2. All nodes visited by q within the log-phase are in one subgraph W_i^2.
3. Within the log-phase, query q first visits some nodes in one subgraph W_i^1 of $G(S_1)$. Once it "leaves" W_i^1, it is sufficient (for the completion of a log-phase) to consider only the subgraph W_j^2 of $G(S_2)$ visited at that point in time, and let the query continue on its search path until it reaches a vertex at the border of S_2.

The correctness of Algorithm 3, as well as the time and space complexity, follow immediately from Lemma 3. ■

Therefore, by iterating Algorithm 3 $O(\lceil r/\log n \rceil)$ times, the multisearch problem can be solved for α-β-partitionable graphs.

THEOREM 7. *Let G be an (undirected) α-β-partitionable graph of size n, and let $Q = \{q_1, \ldots, q_m\}$ be a set of $m = O(n)$ search queries. Then the multisearch problem for Q on G can be solved in $O(\sqrt{n} + r(\sqrt{n}/\log n))$ time on a mesh of size n, where r is the length of the longest search path associated with a query.* ■

4. IMPLEMENTING PARALLEL DATA STRUCTURES ON A MESH-CONNECTED COMPUTER

In this section, we illustrate the use of the multisearch techniques presented in Section 3. The multisearch technique for partitionable graphs, described in Section 3.2, can be immediately applied to parallelize standard query processes on balanced search trees. When processing many such queries independently and in parallel, the query paths may overlap arbitrarily. Of particular interest are on-line processes where the paths taken by the queries can not be computed a priori. Such cases occur, for example, when no global order exists for the set of queries and data.

We give two simple illustrations of possible applications. Consider a set S of n nonintersecting line segments spanning a vertical slab. Each query consists of a point within the slab, for which the two segments determining the region containing that point must be computed. The obvious sequential solution is to build a balanced binary tree for the line segments and answer queries by a straight forward tree search. Using our multisearch technique, a set Q of n such queries can be processed in time $O(\sqrt{n})$ on a mesh of size n. Note that, there exists no total ordering on the set $Q \cup S$.

Now, consider the problem of determining the "best" common ancestor of a pair of nodes in a tree. Such a problem occurs, e.g., in clustering [17]: given a hierarchical agglomerative clustering scheme, determine for two data elements the "best" cluster (e.g., the cluster with closest cluster center) containing both elements. The obvious sequential solution to the general "best" common ancestor problem, in a tree of size n, is to visit the path of all common ancestors in the tree while maintaining the current best element. Using our multisearch technique, a set of n such queries can be processed in time $O(\sqrt{n})$ on a mesh of size n.

The multisearch techniques for multiple on-line overlapping queries on partitionable graphs also supports cases where queries may change directions independently. For example, in a tree, queries may move both upwards and downwards during the search. Possible applications include cases where each query performs an inorder traversal of a certain subtree [7].

An interesting application of multisearch techniques for hierarchical DAGs (Section 3.1) are mesh implementations of Kirkpatrick's subdivision hierarchies. In [6], $O(\log n \log^* n)$ time deterministic and $O(\log n)$ time randomized PRAM algorithms are presented for constructing two well known data structures, namely, the subdivision hierarchy for a planar graph (with n nodes) and the hierarchical representation for a convex polyhedron (with n vertices). Both are hierarchical DAGs of size $O(n)$ with triangles and triangular faces, respectively, associated with their vertices. As stated in [6], once these hierarchies are given, the following problems can be solved in time $O(\log n)$ on the PRAM.

- Multiple planar point location: Given a planar graph G of size n, and n points in the plane, determine for each point p the face of G containing p.
- Multiple line–polyhedron queries: Given a 3-d convex polyhedron P of size n, and n lines in 3-space, determine for each line l whether it intersects P and, if not, determine the two planes through l that are tangent to P.
- 3-d convex polyhedron separation: Given two convex 3-d polyhedra P and Q, each of size n, determine whether or not there exists a plane which separates P and Q.
- Merging 3-d convex hulls: Given two-separated convex 3-d polyhedra P and Q, construct the convex hull of the union of P and Q.

The first two problems can be solved in $O(\log n)$ time for a single query on a sequential machine [19, 10]. Therefore, for the CREW PRAM, both problems can be solved in $O(\log n)$ time by assigning one processor to each query and performing the sequential algorithm concurrently for all processors. The third problem can be reduced to a linear number of independent line-polyhedron queries [6, 11]. The major step in solving the fourth problem consists of determining for each vertex/edge/face of P and Q, whether it is a vertex/edge/face, respectively, of the convex hull of the union of P and Q. With this information, the hulls can be merged by a fixed number of parallel prefix operations. As presented first in [1], with corrected versions in [9] and [3], each edge of P can locally determine whether or not it is in the convex hull based on the result of its line-polyhedron query with respect to Q. Hence, the problem of merging 3-d convex hulls reduces to $2n$ line-polyhedron queries.

For the mesh-connected computer, it has been shown in [9] that the subdivision hierarchy for a planar graph (with n nodes), as well as the hierarchical representation for a convex polyhedron (with n vertices), can be constructed in time $O(\sqrt{n})$ using $O(n)$ processors. Using Theorem 2, we obtain

THEOREM 8. *The following problems can be solved in time* $\Theta(\sqrt{n})$ *on a mesh of size* n:

1. *Multiple planar point location.*[3]
2. *Multiple line-polyhedron queries.*
3. *3-d convex polyhedron separation.*
4. *Merging 3-d convex hulls; determining the convex hull of* n *points in 3-space.*[4] ■

5. CONCLUSION

In this paper, we have considered the *multisearch problem* for $O(n)$ search queries on a data structure modeled as a graph G with n constant-degree nodes. We have presented a $\Theta(\sqrt{n} + r\ (\sqrt{n}/\log n))$ time algorithm for performing, in parallel, $O(n)$ searches on a shared data structure stored in a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer. The main problem for the mesh, in comparison to other networks like the hypercube, is that in order to obtain optimal algorithms from multisearch, the time per advancement of all queries by one step in their search paths must be $O(\sqrt{n}/\log n)$. That is, it must be less than the diameter of the network. The algorithms presented here show how to overcome this problem.

To illustrate the use of the multisearch techniques, we considered parallel online traversals of trees and hierarchical representations of polyhedra. The parallel mesh implementation of the latter one yields optimal mesh algorithms for multiple line-polyhedron intersection queries, multiple tangent plane determination, intersecting convex polyhedra, and computation of the three-dimensional convex hull. We believe that the multisearch problem is such a fundamental problem that we expect it to have many additional applications (e.g., in parallel databases and related areas).

ACKNOWLEDGMENTS

The authors are grateful to the referees for their helpful comments. Useful conversations with Professors Susanne Hambrusch and Rao Kosaraju are also gratefully acknowledged.

REFERENCES

1. Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., and Yap, C. Parallel computational geometry. *Algorithmica* **3,** 3 (1988), 293–327.
2. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
3. Amato, N., and Preparata, F. P. The parallel 3D convex-hull problem revisited. Tech. Rep. UILU-ENG-90-2251, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Nov. 1990.
4. Atallah, M. J., and Hambrusch, S. Solving tree problems on a mesh-connected processor array. *Information and Control* **69,** (1986), 168–186.
5. Chazelle, B., An optimal algorithm for intersecting three-dimensional convex polyhedra. *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science.* 1989, pp. 586–591.
6. Dadoun, N., and Kirkpatrick, D. G. Parallel construction of subdivision hierarchies. *Proc. Third Annual Symposium on Computational Geometry.* 1987, pp. 205–214.
7. Dehne, F., Ferreira, A., and Rau-Chaplin, A. A massively parallel knowledge-base server using a hypercube multiprocessor. *Proc. IEEE International Conference on Tools for Artificial Intelligence.* Washington, DC, 1990, pp. 660–666.
8. Dehne, F., and Rau-Chaplin, A. Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry. *J. Parallel Distrib. Comput.* **8,** 4 (1990), 367–375.
9. Dehne, F., Sack, J.-R., and Stojmenovic, I. A note on determining the 3-dimensional convex hull of a set of points on a mesh of processors. *Scandinavian Workshop on Algorithm Theory* 1988, pp. 154–162.
10. Dobkin, D. P. and Kirkpatrick, D. G. Fast detection of polyhedral intersection. *Proc. International Colloquium on Automata, Languages, and Programming.* 1982, pp. 154–165.
11. Dobkin, D. P., and Kirkpatrick, D. G. A linear time algorithm for determining the separation of convex polyhedra. *J. Algorithms* **6,** (1985), 381–392.
12. Edelsbrunner, H. A new approach to rectangle Interseactions—Part I. *Internat. J. Comput. Math.* **13,** (1983), 209–219.
13. Edelsbrunner, H. A new approach to rectangle Interseactions—Part II. *Internat. J. Comput. Math.* **13,** (1983), 221–229.
14. Edelsbrunner, H. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.
15. He, X., and Miller, R. Optimal mesh algorithms for maximal independent subset and 5-coloring. Tech. Rep. 90-27, Department of Computer Science, SUNY-Buffalo, NY, Oct. 1990.
16. Holey, J. A., and Ibarra, O. H. Triangulation in a plane and 3-D convex hull on mesh-connected arrays and hypercubes. Tech. Rep., University of Minnesota, Department of Computer Science, 1990.
17. Jain, A. K., and Dubes, R. C. *Algorithms for Clustering Data.* Prentice Hall Advanced Reference series, USA, 1988.
18. Jeong, C. S., and Lee, D. T. Parallel geometric algorithms on a mesh connected computer. *Algorithmica* **5,** 2 (1990), 155–178.
19. Kirkpatrick, D. G. Optimal search in planar subdivisions. *SIAM J. Comput.* **12** 1 (1983), 28–35.
20. Lee, D. T., Preparata, F. P., Jeong, C. S., and Chow, A. L. SIMD parallel convex huil algorithms, Northwestern University Tech. Rep. AC-91-02, Mar. 1991.
21. Miller, R., and Stout, Q. F., Efficient parallel convex hull algorithms. *IEEE Trans. Comput.* **37,** 12 (December 1988), 1605–1618.
22. Miller, R., and Stout, Q. F. *Parallel Algorithms for Regular Architectures.* MIT Press, 1991.
23. Miller, R., and Stout, Q. F. Mesh computer algorithms for computational geometry. *IEEE Trans. Comput.* (January 1989).
24. Nassimi, D., and Sahni, S. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput.* **C-27,** 1 (January 1979), 2–7.

[3] A $\Theta(\sqrt{n})$ time mesh algorithm was first presented in [18]. The problem is listed here only to show that, within the multisearch framework, a $\Theta(\sqrt{n})$ time algorithm is now obvious.

[4] Other optimal mesh solutions have recently been obtained [20, 16] independently of ours and using very different, purely geometric approaches, rather than the multisearch method we use.

25. Nassimi, D., and Sahni, S. Data broadcasting in SIMD computers. *IEEE Trans. Comput.* **C-30,** 2 (February 1981), 101–107.
26. Paul, W., Vishkin, U., and Wagener, H. Parallel dictionaries on 2–3 trees. *Proc. 10th International Colloquium on Automata, Languages, and Programming (ICALP), LNCS 154.* Springer-Verlag, Berlin, 1983, pp. 597–609.
27. Preparata, F. P., and Shamos, M. I. *Computational Geometry.* Springer-Verlag, Berlin, 1985.
28. Samet, H. The quadtree and related hierarchical data structures. *Computing Survey* **16,** 2 (June 1984), 187–260.
29. Thompson, C. D., and Kung, H. T. Sorting on a mesh-connected parallel computer. *Commun ACM* **20,** 4 (April 1977), 263–271.
30. Tsay, J. J. Techniques for solving geometric problems on mesh-connected computers. Ph.D. thesis, Department of Computer Science, Purdue University, 1990.
31. Tsay, J. J. Searching tree structures on a mesh of processors. *Third Annual ACM-SIAM Symposium on Discrete Algorithms, 1992,* to appear.

MIKHAIL J. ATALLAH received the B.E. in electrical Engineering from the American University, Beirut, Lebanon, in 1975 and the M.S.E. and Ph.D. in Electrical Engineering and Computer Science from Johns Hopkins University, Baltimore, in 1980 and 1982, respectively. In August 1982 he joined Purdue University, West Lafayette, where he is currently Professor of Computer Science. In 1985 he received a Presidential Young Investigator award from the National Science Foundation. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry. Dr. Atallah is a member of the Association for Computing Machinery and the Society for Industrial and Applied Mathematics and a senior member of the Institute of Electrical and Electronics Engineers. He serves on the editorial boards of the journals *Computational Geometry: Theory & Applications, Information Processing Letters, International Journal on Computational Geometry & Applications, Methods of Logic in Computer Science, Parallel Processing Letters, SIAM Journal on Computing,* and is Guest Editor for a Special Issue of *Algorithmica* on Computational Geometry. He has also served on conference program committees and state and federal panels.

FRANK DEHNE was born in Hannover (W. Germany) in 1960. He received his M.C.S. degree (Dipl. Inform.) from the Technical University of Aachen (W. Germany) in 1983 and his Ph.D. (Dr. rer. nat.) from the University of Wurzburg (W. Germany) in 1986. He is currently an Associate Professor of Computer Science at Carleton University in Ottawa. His research interests include computational geometry, data structures, and parallel and VLSI algorithms. Since 1989, Dr. Dehne serves as Managing Editor for *Information Processing Letters* and, Editor for *Parallel Algorithms and Applications,* Guest Editor for *Algorithmica,* and as program committee member for several international conferences including the *7th ACM Symposium on Computational Geometry.*

RUSS MILLER was introduced to computer science in the public schools of Huntington Station, New York. He received a B.S. (1980), M.A. (1982), and Ph.D. (1985) in computer science/mathematics from the Department of Mathematical Sciences, State University of New York at Binghamton. In the Fall of 1985, he joined the Department of Computer Science at the State University of New York at Buffalo, where he is currently an Associate Professor. Dr. Miller is spending the 1992–1993 academic year on sabbatical as a Senior Research Scientist at the Medical Foundation of Buffalo Research Institute. He is also a consultant for Thinking Machines Corporation, and for three years served as Associate Director for the Graduate Group in Advanced Scientific Computing at the State University of New York at Buffalo. His primary research interests are parallel algorithms, parallel computing, parallel architectures, and computational crystallography. Dr. Miller is a member of the IEEE (including the Computer Society), ACM, SIAM, and Phi Beta Kappa. He is also on the editorial board of *Parallel Processing Letters.*

ANDREW RAU-CHAPLIN received a B.A. in Computer Science from the York University, Toronto, Canada, in 1986 and a M.C.S. in Computer Science from Carleton University, Ottawa, Canada, in 1989. Between September 1986 and September 1988 he was a member of Scientific Services at Bell Northern Research where he was the principal designer of a real-time expert system for the monitoring and control of packet switching networks. He is currently on a Bell Northern Research Post Graduate Scholarship completing a Ph.D. in computer science at Carleton University. Mr. Rau-Chaplin is a member of the Association for Computing Machinery His research interests include the design and analysis of algorithms and parallel computation from both a practical and theoretical perspective.

JYH-JONG TSAY received the B.E. in Computer Science and Information Engineering from National Taiwan University, Taipei, Taiwan, Republic of China, in 1982 and the M.S. and Ph.D. in Computer Science from Purdue University, West Lafayette, in 1988 and 1990, respectively. In August 1990 he joined National Chung Cheng University, Chiayi, Taiwan, Republic of China, where he is currently an assistant professor of Computer Science and Information Engineering. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry.

Received December 13, 1991; accepted June 3, 1992