Multisearch Techniques for Implementing Data Structures on a Mesh-Connected Computer

(Preliminary Version)

Mikhail J. Atallah*
Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA.

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, NY 14260, USA.

Andrew Rau-Chaplin§
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6.

Jyh-Jong Tsay¶
National Chung Cheng University
Inst. of Comp. Sci. and Inform. Eng.
Chiayi, Taiwan 62107, ROC.

Abstract

The *multisearch* problem consists of efficiently performing $O(n)$ search processes on a data structure modeled as a graph G with n constant-degree nodes. Denote by r the length of the longest search path associated with a search process, and assume that the paths are determined "on-line". In this paper, we solve the multisearch problem in $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer. For most 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 time, $O(\sqrt{n})$. The classes of graphs considered contain most of the important data structures that arise in practice (ranging from simple trees to Kirkpatrick hierarchical search DAGs). Multisearch is a useful abstraction that models many specific problems and can be used to implement parallel data structures on a mesh. Applications include interval trees and the related multiple interval intersection search, as well as hierarchical representations of polyhedra and its many applications (e.g., lines-polyhedron intersection queries, multiple tangent plane determination, intersecting convex polyhedra, and three-dimensional convex hull).

*Research partially supported by the Office of Naval Research under Contracts N00014-84-K-0502 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, and the National Science Foundation under Grant DCR-8451393.

1 Introducion

Given a search structure modeled as a graph G with n constant-degree nodes, and given $O(n)$ search processes on that structure, the *multisearch* problem is that of performing as fast as possible all of the search processes on that structure. The searches need not be processed in any particular order, and can simultaneously be processed in parallel by using, for example, one processor for each. However, the path that a search query will trace in G is *not* known ahead of time, and must instead be determined "on-line": only when a search query is at (say) node v of G can it determine which node of G it should visit next (it does so by comparing its own search key to the information stored at v — the nature of this information and of the comparison performed depend on the specific problem being solved). The multisearch problem is a useful abstraction that can be used to solve many problems (more on this later). It is a challenging problem both for EREW-PRAMs and for networks of processors, since many searches might want to visit a single node of G, creating a "congestion" problem (with the added complication that we cannot even tally ahead of time how much congestion will occur at a node, since we do not know ahead of time the full search paths, only the nodes of G at which they start). When the parallel model used to solve the problem is a network of processors, the graph G is initially stored in the network in the natural way, with each processor containing one node of G and 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 neighbour of node v in G need not be stored in a processor adjacent to the one containing v. Each processor also contains initially (at most) one of the search queries to be processed (in which case that search does not necessarily start at the node of G stored in that processor).

In the EREW-PRAM, the difficulty comes from the "exclusive read" restriction of the model: if k processes were to simultaneously access node v's information, the k processors assigned to these k search processes are, at least apparently, unable to simultaneously access v's information. An elegant way around this problem was given by Paul, Vishkin and Wagener [PVS83] for the case where G is a 2-3 tree (although they assume a linear ordering on the search keys, something which we cannot afford to do here since we also consider applications involving multidimensional search keys for which no linear ordering can be used).

The multisearch problem is even more challenging for networks of processors. In such models, data is not stored in a shared memory, but is distributed over a network and requires considerable time to be permuted to allow different processors access to different data items. Furthermore, similarly to the EREW-PRAM, each memory location can be accessed only by one query process 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 in $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$ 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 most data structures, the search path traversed when answering a query has length $r = O(\log n)$. That is, for these cases our algorithm processes $O(n)$ such queries in asymptotically optimal time, $O(\sqrt{n})$. The classes of graphs considered are listed below. They contain most of the important cases of G that arise in practice (ranging from simple trees to the powerful Kirkpatrick hierarchical search DAG that is so important in both sequential and parallel computational geometry).

As already mentioned, multisearch is a useful abstraction that models many specific problems (and hence can be used to solve them). We shall later in the paper use it to solve the problem of implementing parallel data structures on a mesh-connected computer. Applications include interval trees and the related multiple interval intersection search, as well as hierarchical representations of polyhedra and its myriads of applications including lines-polyhedron intersection queries, multiple tangent plane determination, three-dimensional convex hull[1], and intersecting convex polyhedra. Note that these problems are of considerable importance in robotics and solid modeling, computational geometry, vision, pattern recognition, etc. In addition, multisearching is such a fundamental problem that it probably has many additional applications that we have not yet explored (perhaps in parallel databases and related areas).

The multisearch problem for *hypercube multiprocessors* was studied in [DR90]. That hypercube technique 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. Such an approach is not viable on the *mesh* since, in order to obtain *optimal* mesh algorithms based on multisearch, the time per advancement of all queries by one step needs to be *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 [DR90], and they are also very different from [PVS83]. In very broad terms, our techniques for solving the problem are a judicious combination of the following ideas:

- Partitioning G into pieces and processing some of these in sequence, others in parallel.
- Making many copies of some pieces of G (the "bot-

[1]The 3-D convex hull problem has optimal mesh solutions, recently obtained independently of ours and using different, purely geometric approaches rather than the multisearch method we use [LPJC90, HI90].

tleneck" ones, i.e., those with too many searches trying to go through them), and distributing these copies to various submeshes, each of which then advances some of the "congested" searches. Of course the simple-minded copying strategy of making many copies of G itself, and using one copy for each search, does not work; not only would this take too much time ($O(n)$ time, since we have n searches) but there is not even enough space to store all these copies of G (there is only enough space to store $O(1)$ copies of G, since G has n nodes).

- Mapping some pieces of G into suitably shaped portions of the mesh (not necessarily rectangular submeshes).

Of course, the above-mentioned partitionings, duplications, and mappings cannot be pre-computed, since we do not yet know how the full search paths will develop (in fact the problem of "tracing" the search paths is nontrivial even if we did know them ahead of time). The partitionings/duplications/mappings must instead be done 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 G considered are hierarchical directed acyclic graphs (hierarchical DAGs for short), α-partitionable (directed) graphs, and α-β-partitionable (undirected) graphs. For the exact definitions of the latter two, we refer the reader to Section 4. The first one (hierarchical DAGs) is easy to state in one sentence: the vertex set can be partitioned into levels $L_0, \ldots, L_h$ ($h = O(\log n)$) such that every edge is from some L_i to L_{i+1}, $|L_0| = 1$, and $|L_{i+1}| = \mu|L_i|$, for some $\mu > 1$ (i.e., $|L_i| = \mu^i$). (Our algorithm can also handle the case where the last condition is replaced by $c_1\mu^i \leq |L_i| \leq c_2\mu^i$, for some positive constants c_1 and c_2.) See Figure 1.

The next section contains a more formal definition of the multisearch problem, and of the various terms used in the paper. Sections 3 and 4 contain the main results: our solutions to the multisearch problem for each of the above-mentioned classes of graphs. Sections 5 and 6 use multisearching to solve various problems efficiently on the mesh.

2 Definition of 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 $(v_1, ..., v_h)$ of G defined by a successor function $f : (V \cup start) \times U \rightarrow V$ with $f(start, q) = v_1$, and $v_{i+1} = f(v_i, q)$ 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 $q \in U$, $\{v, f(v,q)\} \in E$. Furthermore, $f(v,q)$ can be computed by one processor, that stores q and v's information, in $O(1)$ time.

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 $path(q) = (v_1, ..., v_h)$ is a process divided into h time steps, $t_1 < t_2 < \ldots < t_h$, such that at time t_i, $1 \leq i \leq h$, query q visits node v_i. We will refer to the change of state between t_i and t_{i+1}, $1 \leq i < h$, as *advancing query* q *one step in its search path.* Recall that we do not assume the search path to be given in advance. Rather, we assume that the search path for each query is constructed *online* during the search by successive applications of the function f.

Of course, 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 $Q = \{q_1, ..., q_m\} \subseteq U$ 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. In particular, at any time t, any node v of G may be visited by an arbitrary number of queries (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, all (or a subset) of the m search queries by one step in their search paths as a *multistep*. A sequence of multisteps such that every search query is advanced $\Omega(\log n)$ steps in its search path, will be referred to as a *log-phase*.

3 A Mesh Solution to the Multisearch Problem for Hierarchical DAGs

Let $G = (V, E)$ be a hierarchical DAG of size n and height h, and let $L_0, \ldots, L_h$ be the levels of G. Note that G has out-degree $O(1)$, $h = O(\log n)$, 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 on a mesh-connected computer of size n in time $O(\sqrt{n})$. The graph G and the set of search queries Q are initially stored in the mesh in the natural way; a precise description of the initial configuration is given in the Appendix. In addition, we assume that every processor storing a node $v \in L_i$ also stores the index i, referred to as *level index* of v in G. Note that the level indices can be easily computed in time $O(\sqrt{n})$ 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 $\log^{(0)} x = \frac{x}{2}$. 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 $0 \leq i \leq \log^*_\mu x$. For the remainder of this section, all logarithms are to 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 the size of B_i, the highest index of a level in B_i, and the number of levels in B_i, respectively. See Figure 4 for an illustration. Note that, $|B_i| = O(\mu^{h-2\log^{(i+1)} h}) = O(\frac{n}{(\log^{(i)} h)^2})$ and $\Delta h_i = O(\log^{(i)} h)$.

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

The general strategy for solving the multisearch problem on G is to solve the multisearch problem for B_0 first, then for B_1, etc., eventually for $B_{\log^* h-1}$, and finally for B^*. Here, the multisearch problem for B_i $[B^*]$ consists of all queries visiting those vertices on their search path that lie in B_i $[B^*]$, assuming that for each query the first of those vertices is known.

Since B^* has $O(1)$ 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 problem for $B_0, \ldots, B_{\log^* h-1}$ (together) in time $O(\sqrt{n})$.

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

Let B_i^1 be the subgraph of G induced by the vertices of G between levels $h_i - \Delta h_i$ and $h_i - 1 - 2\log\Delta h_i$ included, and let B_i^2 be the subgraph induced by the vertices between levels $h_i - 2\log\Delta h_i$ and h_i included. See Figure 5 for an illustration. Note that $|B_i^1| = O(\mu^{h_i - 2\log\Delta h_i}) = O(\frac{|B_i|}{(\Delta h_i)^2})$. On each 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 $\frac{|B_i|}{(\Delta h_i)^2}$, called *$B_i^1$-submeshes*. Note that each 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 each 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 that submesh. This can be easily done in $O(\sqrt{|B_i|})$ time since $|B_i^1| = O(\frac{|B_i|}{(\Delta h_i)^2})$ 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.

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 $O(\sqrt{|B_i|}\log\Delta h_i) = O(\sqrt{|B_i|}\log^{(i+1)} h)$.*

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, then all 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 $0 \leq i \leq \log^* h - 1$. Our algorithm can be easily 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 $i = \log^* h - 1, \ldots, 0$:
   (a) In each B_{i+1}-submesh, Δ, every processor p in the top-left B_i-submesh of Δ sets $label(p) := i$.

   Note: The label of a processor may be overwritten by smaller indices in later iterations. In the next step, in each B_{i+1}-submesh, the processors with $label = i$ will be used to store a copy of B_i. Since, for $j \leq i - 1$, each B_{j+1}-submesh contains one B_j-submesh in its top-left corner whose processors' labels are set to j, the labels of at most $\frac{n}{(\log^{(i)} h)^2}(\frac{\log^{(j+1)} h}{\log^{(j)} h})^2$ processors are changed from i to j. Hence, the number of processors in each B_i-submesh with $label = i$ is at least $\frac{n}{(\log^{(i)} h)^2}(1 - \sum_{j=0}^{i-1}(\frac{\log^{(j+1)} h}{\log^{(j)} h})^2) = \Theta(\frac{n}{(\log^{(i)} h)^2})$.
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 $label = i$. For details, see proof of Theorem 2 in the Appendix.
   (b) $(\frac{\log^{(i)} h}{\log^{(i+1)} h})^2$ 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 $O(1)$ memory per processor) in time $O(\sqrt{n})$.* (For proof see Appendix.)

4 A Mesh Solution to the Multisearch Problem For Partitionable Graphs

In this section, we present mesh solutions to the multisearch problems for α-partitionable graphs and α-β-partitionable graphs. After defining these classes of graphs, we will first introduce a tool referred to as *constrained multisearch* which will be utilized in Sections 4.5 and 4.6.

4.1 Definition of 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 connected components, denoted $\{G_1, \ldots, G_k\}$, for some $k \leq n$.

We define S to be an *δ-splitter* of G, $0 < \delta < 1$, if and only if $|G_i| = |V_i| + |E_i| = O(n^\delta)$, for all $1 \leq i \leq k$. 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-splitting $G(S) = \{G_1, \ldots, G_k\}$ is called *normalized*, if $k = O(n^{1-\delta})$.

4.2 Definition of α-Partitionable (Directed) Graphs

Let $G = (V, E)$ be a directed graph, where the outdegree of any vertex is bounded by some constant. Let $dist_G(v_1, v_2)$ denote the length of the shortest directed path in G connecting two 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 $G(S)$ 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 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 i, j.

Note that, for example, every balanced k-ary search tree with all edges either direct towards the leaves or direct 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 Figure 2.

4.3 Definition of α-β-Partitionable (Undirected) Graphs

Let $G = (V, E)$ be an undirected graph of 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 the 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 $(0 < \alpha, \beta < 1)$. We define that, S_1 and S_2 *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, $0 < \alpha, \beta < 1$, such that S_1 and S_2 have distance $\Omega(\log n)$.

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

4.4 Constrained Multisearch

Let $G = (V, E)$ be a directed or undirected graph. Consider a set $\Psi = \{G_1, ..., G_k\}$ of k edge and vertex disjoint subgraphs of G such that $|G_i| = O(n^\delta)$ and $k = O(n^{1-\delta})$ for some $0 < \delta < 1$. 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 x steps in its search path, such that either $x = \log_2 n$ or the next node to be visited by q is not in G_i. Note that different queries may be advanced by a different number of steps.

In the remainder of this section, we present a procedure *Constrained-Multisearch(Ψ, δ)* which solves the constrained multisearch problem, on a mesh of size n with $O(1)$ memory per processor, in time $O(\sqrt{n})$

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

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 some 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.

1. All queries $q \in Q$ such that $v(q)$ is in some subgraph $G_i \in \Psi$ are *marked*; all others queries are *unmarked*. (Queries whose search paths have already terminated are also unmarked.)
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 $\sqrt{n^\delta} \times \sqrt{n^\delta}$ subsquare of the mesh-connected computer (*δ-submesh*).
5. Every marked query $q \in Q$ with $v(q) \in G_i$ is moved to one of the δ-submeshes storing a copy of G_i, such that each δ-submesh contains at most $O(n^\delta)$ queries.
6. Within each δ-submesh storing a subgraph $G_i \in \Psi$, the following is executed $\log_2 n$ times:
   - (a) For every marked query $q \in Q$, the next node in its search path is determined (by applying the successor function f).
   - (b) Every marked query for which the next node in its search path is not in G_i, is unmarked. (A query whose search path terminates is also unmarked.)
   - (c) Every marked 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 with $O(1)$ memory per processor, in time $O(\sqrt{n})$* (For proof see Appendix.)

4.5 A Mesh Solution to the Multisearch Problem for α-Partitionable Directed Graphs

Let $G = (V, E)$ be a directed and α-partionable graph. Consider a set $Q = \{q_1, ..., q_m\}$ 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 multiserach problem for Q on G in time $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$. Our strategy is to give an algorithm which executes one log-phase of the multisearch in time $(\sqrt{n})$. The entire multisearch algorithm consists of iterating the log-phase algorithm $O(\lceil\frac{r}{\log n}\rceil)$ 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 i,j. 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, every processor stores in addition to a vertex $v \in V$ also an index indicating to which graph in $G(S)$ the vertex v belongs. Note that, for most data structures (e.g., balanced k-ary trees; see Figure 2), the determination of the indices is trivial. 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 4.1. Otherwise, we group the subgraphs H_i and T_i, respectively, such that each resulting subgraph has size $\Theta(n^\alpha)$. This operation is easily performed, on a mesh of size n, in time $O(\sqrt{n})$. Furthermore, the algorithm described in this section does not require that every subgraph in $G(S)$ consists of only one connected component of the graph $(V, E - S)$.

Algorithm 2: Implementation of one log-phase of multisearch on an α-partionable 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 an α-partionable (directed) graph of size n can be performed , on a mesh of size n with $O(1)$ memory per processor, in time $O(\sqrt{n})$.* (For proof see Appendix.)

Therefore, by iterating Algorithm 2 $O(\lceil\frac{r}{\log n}\rceil)$ times, the multisearch problem can be solved for α-partitionable graphs.

Theorem 5 *Let G be an α-partionable (directed) graph of size n and let $Q = \{q_1, ..., q_m\}$ be a set of $m = O(n)$ search queries. The multisearch problem for Q on G can be solved on a mesh of size n (with $O(1)$ memory per processor) in time $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$, where r is the length of the longest search path associated with a query.*

4.6 A Mesh Solution to the Multisearch Problem for α-β-Partitionable Undirected Graphs

Let $G = (V, E)$ be a directed and α-partionable graph. Consider a set $Q = \{q_1, ..., q_m\}$ 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 in $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$ time. As in Section 4.5, we will again give an algorithm to execute one log-phase of the multisearch problem in time $(\sqrt{n})$. The multisearch algorithm consists of iterating the log-phase algorithm $O(\lceil\frac{r}{\log n}\rceil)$ 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, every processor stores in addition to a vertex $v \in V$ also two indices indicating to which graphs in $G(S_1)$ and $G(S_2)$ the vertex v belongs. Note that, for most data structures (e.g., balanced k-ary trees; see Figure 3), the determination of the indices is trivial.

With the same argument as in Section 4.5, 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 $k_2 = O(n^{1-\beta})$.

Algorithm 3: Implementation of one log-phase of multisearch on an α-β-partionable 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 α-β-partionable (undirected) graph of size n can be performed , on a mesh of size n with $O(1)$ memory per processor, in time $O(\sqrt{n})$.* (For proof see Appendix.)

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

Theorem 7 *Let G be an α-β-partitionable (undirected) graph of size n and let $Q = \{q_1, ..., q_m\}$ be a set of $m = O(n)$ search queries. The multisearch problem for Q on G can be solved on a mesh of size n (with $O(1)$ memory per processor) in time $O(\sqrt{n} + r\frac{\sqrt{n}}{\log n})$, where r is the length of the longest search path associated with a query.* (For proof see Appendix.)

5 Applying Multisearch for Hierarchical DAGs: Subdivision Hierarchies, Hierarchical Representations of Polyhedra, and Applications

In [DK87], $O(\log n \log^* n)$ time deterministic and $O(\log n)$ time randomized PRAM algorithms are presented for constructing well known data structures: the subdivision hierarchy for a planar graph (with n nodes) [Kir83] 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 [DK87], once these hierarchies are given, Problems 1-3 listed in Theorem 8 can be solved on the PRAM in time $O(\log n)$.

For the mesh-connected computer, it has been shown in [DSS88] 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 with $O(1)$ memory each. Using Theorem 2, we obtain

Theorem 8 *The following problems can be solved in time $O(\sqrt{n})$ on a mesh of size n with $O(1)$ memory per processor:*

1. *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 tangent to P).
2. *3-d convex polyhedron separation* (Given two convex 3-d polyhedra P and Q of size n each, determine whether there exists a plane which separates P and Q).
3. *Merging 3-d convex hulls.*
4. *Determining the convex hull of n points in 3-space*[1].

6 Applying Multisearch for Partitionable Graphs: Interval Trees and Multiple Interval Intersection Search

Obviously, multisearch for α-partitionable directed graphs can be utilized to obtain optimal parallel mesh implementations for all those data structures based on balanced k-ary search tree (possibly with augmentation), where all queries are moving in the same direction (either from the root to the leaves, or from the leaves to the root).

Multisearch for α-β-partitionable undirected graphs can be applied, e.g., to obtain parallel mesh implementations for data structures based on balanced k-nary search trees (possibly with augmentation) where queries are moving along tree edges in arbitrary directions. Such queries can, e.g., traverse parts of the subtree in inorder.

In the full paper, we explore the following application of our mesh solution to multisearch for α-β-partitionable undirected graphs. Consider a set S of n intervals. The *interval intersection problem* consists of reporting the k intervals in S that intersect a query interval q. The *multiple interval intersection problem* consists of answering, in parallel, m interval intersection queries on S.

References

[AH86] M. J. Atallah and S. Hambrusch. Solving tree problems on a mesh-connected processor array. *Information and Control*, 69:168–186, 1986.

[DK87] N. Dadoun and D. G. Kirkpatrick. Parallel construction of subdivision hierarchies. In *Proceedings of the Third Annual Symposium on Computational Geometry*, pages 205–214, 1987.

[DSS88] F. Dehne, J.-R. Sack, and I. Stojmenovic. A note on determining the 3-dimensional convex hull of a set of points on a mesh of processors. In *Scandinavian Workshop on Algorithm Theory*, pages 154–162, 1988.

[DR90] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8(4):367-375, 1990.

[Ede83a] H. Edelsbrunner. A new approach to rectangle Intersections - Part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[HI90] J. A. Holey and O. H. Ibarra. Triangulation in a Plane and 3-D convex hull on Mesh-Connected Arrays and Hypercubes. Tech. Rep., Univ. of Minnesota, Dept. of Computer Science, 1990.

[Kir83] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983.

[LPJC90] D. T. Lee, F. P. Preparata, C.S. Jeong and A. L. Chow. SIMD Parallel Convex Hull Algorithms, manuscript.

[PVS83] W. Paul, U. Vishkin and H. Wagener. Parallel dictionaries on 2-3 trees. in Proceedings 10th International Colloquium on Automata, Languages, and Programming (ICALP), *LNCS 154*, Springer-Vergerlag, Berlin, 1983, pp. 597-609.

7 Appendix

Details of the Initial Configuration of the Mesh (Before Multisearch):
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. Furthermore, let $Q = \{q_1, ..., q_m\}$ be a set of $m = O(n)$ search queries. G and Q will be represented on the mesh as follows: Every processor stores

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

Note that, the assignments of vertices and queries to processors is not fixed and may change during the course of the algorithms. Every processor p is assumed to have an additional register ***visit(p)***. At any stage of the multisearch algorithms to be presented, a query $q \in Q$ will be said to ***visit*** a node $v \in V$ if the processor p storing the query q also stores, in its register ***visit(p)***, a copy of v.

Proof of Theorem 2:
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 each B_{i+1}-submesh contains a copy of B_i. In Step 3, the multisearch problem for B_i, $i = 0, 1, \ldots, \log^* h - 1$, is solved (in that order). In every loop iteration, within each B_{i+1}-submesh 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 time and space complexity of Algorithm 1 and show that it requires only $O(1)$ space 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 $O(1)$ storage per processor. For Step 2(a), we need to show that in each B_i-submesh there are at least $\Theta(|B_i|)$ 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 $\frac{n}{(\log^{(i)} h)^2}(\frac{\log^{(j+1)} h}{\log^{(j)} h})^2$ processors are changed from i to j. Hence, the number of processors in each B_i-submesh with $label = i$ is at least $\frac{n}{(\log^{(i)} h)^2}(1 - \sum_{j=0}^{i-1}(\frac{\log^{(j+1)} h}{\log^{(j)} h})^2) = \Theta(\frac{n}{(\log^{(i)} h)^2})$. Since $|B_i| = O(\frac{n}{(\log^{(i)} h)^2})$, these processors can store B_i with $O(1)$ storage per processor provided that the B_i's data can be evenly distributed among them. The following is a detailed $O(\sqrt{|B_{i+1}|})$ time implementation of Step 2(a).

1. Every B_i is compressed into top-left B_i-submesh of each B_{i+1}-submesh.
2. Each B_i-submesh is partitioned into four subsquares of equal size.
3. For each subsquare, the number of processors with $label = i$ is determined.
4. The data for B_i is distributed among these subsquares according to the ratio of number of processors with $label = i$.
5. in Steps 2-4 are repeated recursively, in parallel, on each of the four subsquares, until the subsquares are of size $O(1)$.

Summarizing, we obtain that Algorithm 1 requires $O(1)$ storage per processor.

Next, we prove the claimed $O(\sqrt{n})$ 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 $O(\sqrt{n})$. Since B^* contains $O(1)$ levels, the $O(\sqrt{n})$ time complexity of Step 4 is obvious. Since each B_{i+1}-submesh contains a 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 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 $O(\sum_{i=0}^{\log^* h-1} \sqrt{|B_i|} \log \Delta h_i) = O(\sum_{i=0}^{\log^* h-1} \sqrt{n} \frac{\log^{(i+1)} h}{\log^{(i)} h}) = O(\sqrt{n})$. Hence, the time complexity of Algorithm 1 is $O(\sqrt{n})$. □

Proof of Lemma 3:
We first study the correctness of Constrained-Multisearch(Ψ, δ), and then give some implementation details and prove the claimed time complexity. Obviously, every query q either visits the next at most $log_2 n$ nodes in its search path until the next node in its search path is not in the same subgraph $G_i \in \Psi$ that contains $v(q)$, or it will not advance any step in its search path (in case $v(q)$ 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 and total number of subgraphs to be moved match the sises and total number of δ-submeshes available. Item (1) follows from the fact that $\sum_{G_i \in \Psi} \Gamma_{\Psi}^{\delta}(G_i) = O(n^{1-\delta})$, and Item (2) follows from the definition of $\Gamma_{\Psi}^{\delta}(G_i)$ and

the fact that each δ-submesh is of size $O(n^\delta)$. We will now prove the claimed time complexity. Steps 1, 2, 3, and 5 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 of size n^δ. The total number of copies created of subgraphs G_i is $\sum_{G_i \in \Psi} \Gamma^\delta_\Psi(G_i) = O(n^{1-\delta})$. Hence, each such submesh needs to simulate only a constant number of "virtual" δ-submeshes, with each of the "virtual" δ-submeshes storing one copy of a subgraph G_i. Creating the required copies of subgraphs and moving them to the "virtual" δ-submeshes can be implemented by a constant number of standard mesh operations. We finally discuss the time complexity of Steps 5. Note that, each execution of the loop body is executed independently and in parallel on each $O(n^\delta)$ size δ-submesh created in Step 4. Hence, each loop iteration can be implemented in time $O(\sqrt{n^\delta})$, using a standard random access read/write operation on each δ-submesh. Since $0 < \delta < 1$, the total time complexity of Step 5 is $O(\log n \sqrt{n^\delta}) = O(\sqrt{n})$. □

Proof of Lemma 4:
We first study the correctness of Algorithm 2. The basic idea behind the algorithm is that if, in Step 2, a query reaches a vertex at the border of the α-splitter S, the next and all further vertices to be visited are in the same subgraph T_i. These vertices will then be visited in Steps 3 and 4. That is, for every query $q \in Q$, one of three possible cases applies:

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 visits first 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 to be visited within 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 $O(\sqrt{n})$ and requires a mesh of size n with $O(1)$ memory per processor. □

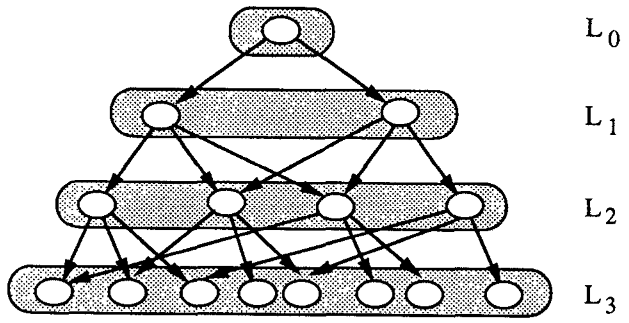

Figure 1. A Hierarchical DAG with $\mu = 2$.

Proof of Lemma 6:
We first study the correctness of Algorithm 3. The basic idea behind the algorithm is that if, in Step 2, a query reaches a vertex at the border of the α-splitter S_1, we will then, in Steps 3 and 4, switch to using the subgraphs defined by the β-splitter S_2. From the definition of α-β-partitionable graphs it follows that such a query can then advance $\Omega(\log n)$ more steps in its search path without visiting a node at the border of S_2; by this time, the log-phase is completed. 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^1_i.
2. All nodes visited by q within the log-phase are in one subgraph W^2_i.
3. Within the log-phase, query q first visits some nodes in one subgraph W^1_i of $G(S_1)$. Once it "leaves" W^1_i, it is sufficient (for the completion of a log-phase) to consider only the subgraph W^2_j 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.

From this, the correctness of Algorithm 3, as well as the claimed time complexity, follow immediately from Lemma 3. □

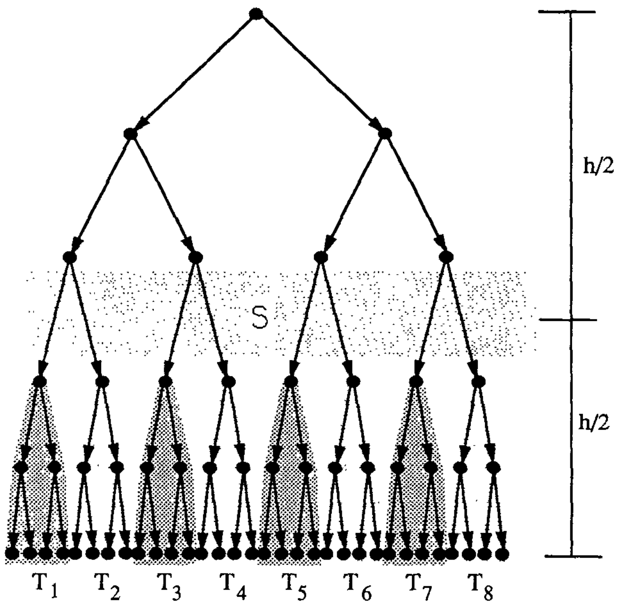

Figure 2. A Directed Balanced Binary Tree And Its α-Splitter ($\alpha = \frac{1}{2}$).

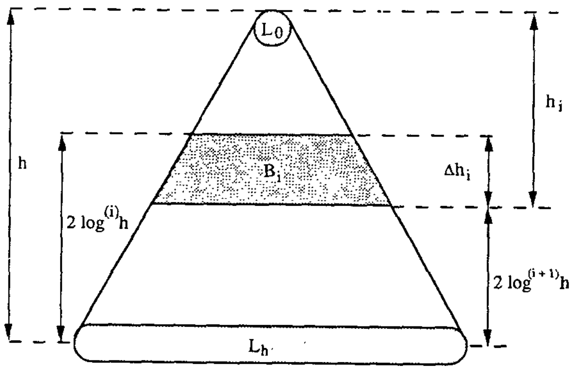

Figure 4. Illustration of the Definition of Subgraphs B_i.

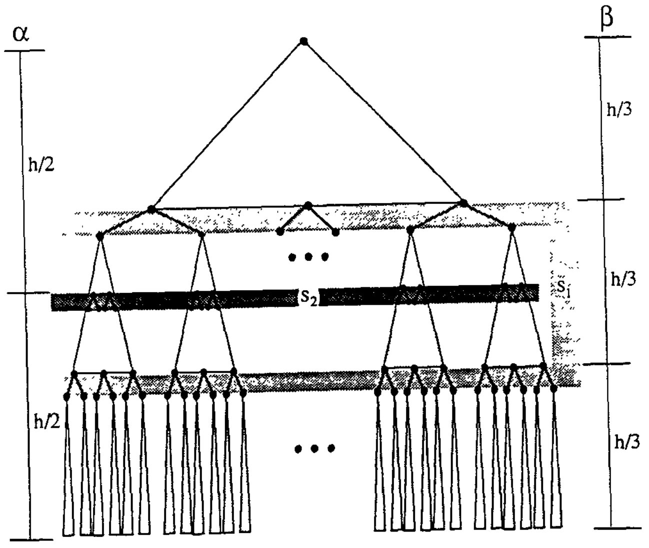

Figure 3. An Undirected Balanced Binary Tree With Its α-Splitter S_1 ($\alpha = \frac{1}{2}$) And β-Splitter S_1 ($\beta = \frac{1}{3}$), Such That S_1 And S_2 Have Distance $\frac{h}{6} = \Omega(\log n)$.

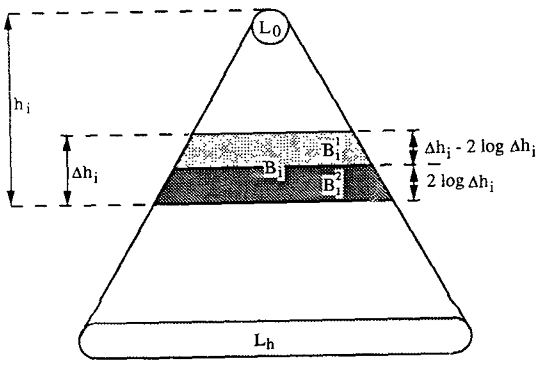

Figure 5. Illustration of the Definition of Subgraphs B_i^1 And B_i^2.