Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1990

Multisearch Techniques of Hierarchical DAGs on Mesh-Connected Computers, with Applications

Jyh-Jong Tsay

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Report Number:
90-1008

Tsay, Jyh-Jong and Atallah, Mikhail J., "Multisearch Techniques of Hierarchical DAGs on Mesh-Connected Computers, with Applications" (1990). *Department of Computer Science Technical Reports.* Paper 10.
https://docs.lib.purdue.edu/cstech/10

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

MULTISEARCH TECHNIQUES OF HIERARCHICAL DAGS ON MESH-CONNECTED COMPUTERS, WITH APPLICATIONS

Jyh-Jong Tsay
Mikhail J. Atallah

CSD-TR-1008
August 1990

Multisearch Techniques of Hierarchical DAGs on Mesh-Connected Computers, with Applications

Jyh-Jong Tsay*
National Chung Cheng University
Institute of Computer Science
and Information Engineering
Chiayi, Taiwan 62107, ROC.

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

Abstract

We present a technique for optimally performing n searches of an n-vertex hierarchical DAG on Mesh-Connected Computers. As immediate consequences of this, we obtain the first optimal mesh algorithms for the problems of computing the convex hull of a set of 3-dimensional points, and of computing the intersection of two 3-dimensional convex polyhedra. The best previously known bounds for these problems were a factor of $\log n$ away from optimality.

*This author's research was 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.

†This author's research was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Reserch under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

1 Introduction

Data structures, which have been used as a fundamental technique in the design of efficient sequential algorithms, have also proved to be very useful in efficient parallel algorithm design. In general, there are two phases to using a data structure in an n-processor machine: the building phase and the searching phase. The searching phase often follows immediately if the data structure has been properly defined and built. This is frequently true for parallel models in which a data structure can be stored in a common memory and each processor can access any memory cell in one time unit. However, in network models, the searching can be the bottleneck because a data structure has to be stored distributively in the processors, and an access of a data element in another processor might take more time than building the data structure. In fact, on an n-processor Mesh-Connected Computer (MCC), there were situations in which a data structure of n nodes could be built optimally; however, optimal solutions to performing n searches simultaneously were elusive [DSS88]. Thus, in network models, the issue of optimally performing multiple searches can be as challenging as that of optimally building the data structure.

In this paper, we study the multisearch problem on a class of hierarchical directed acyclic graphs (DAGs) (which will be defined later) on a $\sqrt{n} \times \sqrt{n}$ MCC. We show that n searches of an n-vertex hierarchical DAG can all be performed in parallel in optimal $O(\sqrt{n})$ time. As immediate consequences of this, we obtain the first optimal $O(\sqrt{n})$ time MCC algorithms for the three dimensional convex hull and the convex polyhedra intersection problems, settling an open problem mentioned in [AW88] and [MS88b]. The algorithms easily generalize to higher dimensional MCCs, running in $O(n^{1/d})$ time on a $n^{1/d} \times n^{1/d} \times \ldots \times n^{1/d}$ MCC. The previously known best algorithms ran in $O(\sqrt{n}\log n)$ time [DSS88] (their time bounds became $O(\sqrt{n})$ if each processor has $O(\log n)$ local storage). The multisearch technique of hierarchical DAGs presented in this paper will probably have applications to other problems as well. The formulation of the multisearch problem is an abstraction of the searching problems in [DK87, DK82, Kir83].

The rest of this paper is organized as follows. Section 2 gives preliminaries, Section 3 gives the algorithms for the multisearch problem, Section 4 gives applications, and Section 5 concludes.

For $i \geq 1$, we will use $\log^{(i)}$ to denote the function which applies the log function i times,

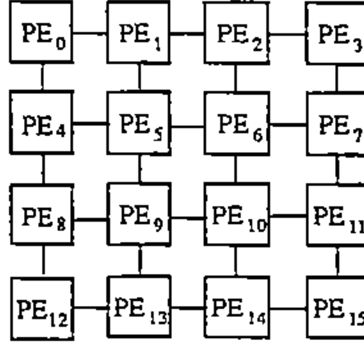

Figure 1: A 4×4 Mesh-Connected Computer

i.e. $\log^{(1)} n = \log n$ and $\log^{(i)} n = \log \log^{(i-1)} n$. For convenience, we define $\log^{(0)} n = n/2$. For any $n \geq 16$, $\log^* n = \max\{i | \log^{(i)} \geq 4\}$ (hence $\log^{(i)} n = 2^{\log^{(i+1)} n} \geq (\log^{(i+1)} n)^2$, for $1 \leq i \leq \log^* n - 1$).

2 Preliminaries

In this section, we review the MCC model and some basic MCC operations, then we define a class of hierarchical DAGs and the multisearch problem.

2.1 The MCC model and basic MCC operations

A $\sqrt{n} \times \sqrt{n}$ MCC is a parallel machine in which the n processors are arranged as in a 2-dimensional array (see Figure 1) and operate synchronously in SIMD mode (i.e., at any time step, all of the n processors are executing the same instruction, although the details of that instruction can depend on the processor's ID). The processor at location (i, j) is connected to the processor at location (i', j') iff $|i - i'| + |j - j'| = 1$. At each time step, a processor can either perform a local operation or communicate with one of its neighbors. The communication links between processors are bidirectional. Each processor has a unique ID from 0 to $n - 1$ and has only $O(1)$ storage registers, each of which can store $O(\log n)$ bits (hence it can store its own ID).

A $\sqrt{n} \times \sqrt{n}$ MCC has a diameter $\Theta(\sqrt{n})$ (i.e., the maximum of the distances between any two processors is $\Theta(\sqrt{n})$). Hence $\Omega(\sqrt{n})$ is a clear lower bound for any nontrivial

computation in this model. We next review some basic MCC operations which will be used later in our algorithms. All of the operations can be done in $(\sqrt{n})$ time [MS89, NS79, NS81, TK77] and easily generalize to higher dimensional MCCs, running in $O(n^{1/d})$ time on a $n^{1/d} \times n^{1/d} \times \ldots \times n^{1/d}$ MCC.

1. *Sorting* takes as input an array of n records, one in each processor, and rearranges them into nondecreasing order of a specified key, i.e. the record in processor i has key values no larger than that of the record in processor $i+1$.

2. *Random access read* takes as input an array $R = r_0, r_1, \ldots, r_{n-1}$ of n records whose key values are pairwise distinct, and an array $A = a_0, a_2, \ldots, a_{n-1}$ of n (not necessarily distinct) key values, where r_i and a_i are in processor i, and sends the record r_j whose key value equals a_i to processor i. If there is no record with key value a_i, processor i receives a null message.

3. *Concentration* moves m $(m \leq n)$ selected records, which are arbitrarily distributed in m of the n processors, into the first m processors, i.e those with IDs equal to 0, 1, $\ldots$, $m-1$.

4. *Compression* moves m $(m \leq n)$ selected records, which are arbitrarily distributed in m of the n processors, into the top-left $\sqrt{m} \times \sqrt{m}$ submesh (see Figure 2). In our algorithm, the problem size is usually reduced by a constant factor before further recursion. We usually use compression to compress the small problem into the top-left submesh to reduce the diameter (from $\sqrt{n}$ to $\sqrt{m}$) before we recursively solve the subproblem. The information which will be used in the conquer stage (but not needed in the recursive call) is stored in the remaining processors as described next.

5. *Uncompression* distributes $n-m$ selected records, which are arbitrarily distributed in $n-m$ of the n processors, to the processors which are *not* in the top-left $\sqrt{m} \times \sqrt{m}$ submesh.

6. *Prefix operations* take as input an array $A = a_0, a_1, \ldots, a_{n-1}$, where a_i is in processor i, and an associative binary operator $\oplus$, and compute an array $B = b_0, b_1, \ldots, b_{n-1}$ where $b_i = a_0 \oplus a_1 \oplus \ldots \oplus a_i$. The value of b_i is stored in processor i. *Segmented prefix operations* partition the array into clusters and perform the prefix operation on each cluster independently. A special type of prefix operations is the *segmented*

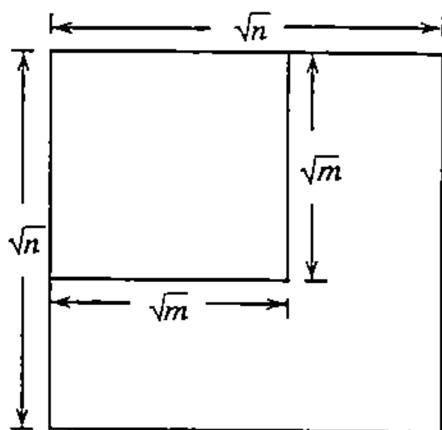

Figure 2: Compression

broadcasting operation in which the value of the first element of each cluster is copied to the remaining elements in the cluster.

In the rest of the paper, we usually want to partition and MCC into square submesh of m processors and duplicate a set S of m selected records such that each square submesh of m processors contains a copy of S (see Figure 3). The duplication of S can be done in $O(\sqrt{n})$ time as follows: (i) compress the m selected records into the $\sqrt{m} \times \sqrt{m}$ square submesh, (ii) for each of the first $m^{1/4}$ rows of the MCC, partition that row into $m^{1/4}$ blocks each consisting of $m^{1/4}$ contiguous processors, and duplicate the subset of the $m^{1/4}$ records stored in (the first $m^{1/4}$ processors of) that row such that each block contains a copy of that subset, (iii) for each column of the MCC, duplicate the $m^{1/4}$ records stored in (the first $m^{1/4}$ processors of) that column as in (ii). Step (i) can be done in $O(\sqrt{n})$ time, using the compression operation, and Step (ii) and (iii) can be done in $O(\sqrt{n})$ time, using basic row and column operations. When $\sqrt{n}$ is not divisible by $\sqrt{m}$, the partitioning will introduce "leftover pieces". These leftover pieces will be combined with their nearest square submeshes, and each new combined submesh will contain only one copy of S (note that the diameter of the new combined submesh is no more than $2\sqrt{m}$).

Lemma 2.1 *Given a set S of $m < n$ elements, S can be duplicated such that each square submesh of m processors contains a copy of S in $O(\sqrt{n})$ time.*

We next define the class of hierarchical DAGs and the multisearch problem.

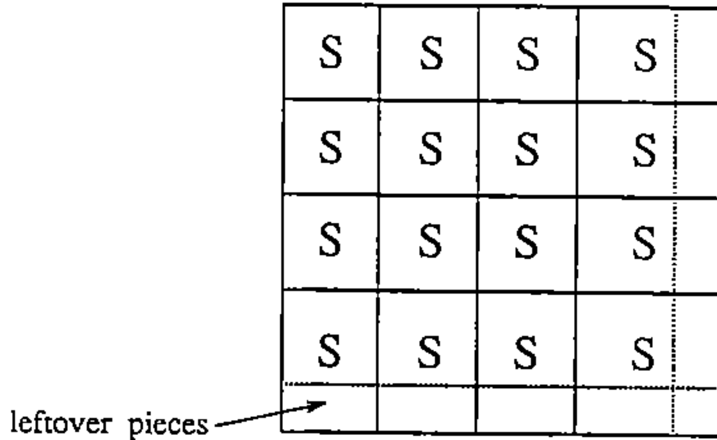

Figure 3: Partition the MCC into square submeshes. The "leftover pieces" are combined with their nearet square submeshes

2.2 Hierarchical DAGs

A directed graph $D = (V, A)$ consists of a set of vertices V and a set of arcs $A \subseteq V \times V$. Each arc of D connects a vertex u (called its *tail*) to another vertex v (called its *head*). A DAG is a directed graph containing no directed cycle. To represent D, we will associate each vertex v with an adjacency list which contains the heads of the arcs emanating from v.

An n-vertex DAG $D = (V, A)$ is said to be *hierarchical* if its vertices are partitioned into levels $L_0, L_1, \ldots, L_h$ where $h = O(\log n)$, $L_0 = \{v_0\}$ and $|L_{i+1}| = \beta|L_i|$ for some constant $\beta > 1$ (hence $|L_i| = \beta^i$), each arc of D connects a vertex in L_i to a vertex in L_{i+1} for some $i \in \{1, 2, \ldots, h\}$, each vertex has no more than d outgoing arcs for some constant $d \geq \beta$. A vertex v is said to have *level number* i if $v \in L_i$. The restriction that $|L_i| = \beta^i$ is made to simplify the presentation, in fact, $c_1\beta^i \leq |L_i| \leq c_2\beta^i$, for some constants c_1 and c_2, is sufficient to establish the results we seek. We will use h, β and d as above throughout the rest of this paper. When we say that "D is stored in an MCC", we mean that each vertex of D is stored in one processor of the $\sqrt{n} \times \sqrt{n}$ MCC, and the processor containing it also contains its adjacency list (of length no more than $d = O(1)$).

Lemma 2.2 *Given an n-vertex hierarchical DAG $D = (V, A)$ on a $\sqrt{n} \times \sqrt{n}$ MCC, we can compute all vertices' level numbers in $O(\sqrt{n})$ time.*

Proof: Let $D' = (V', E')$ be the subgraph of D induced by vertices in $V - L_h$. We obtain

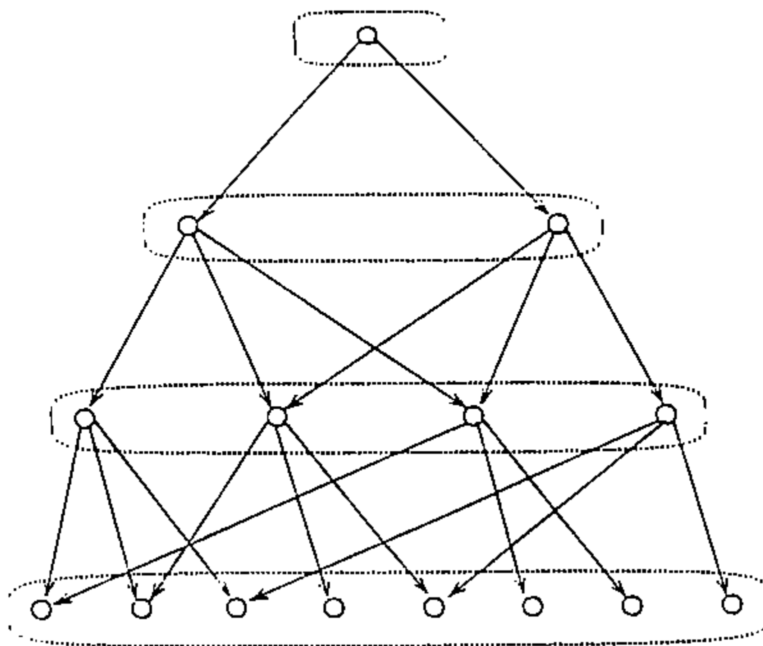

Figure 4: A Hierarchical DAG of $\beta = 2$, $d = 3$ and $h = 3$

D' from D by removing all the vertices which have no outgoing arcs and the arcs incident to them. We then compress D' in the top-left $\sqrt{\lceil V' \rceil} \times \sqrt{\lceil V' \rceil}$ submesh and uncompress $D - D'$ in the remaining processors, and then we recursively compute the level number for each vertex of D', using the top-left $\sqrt{\lceil V' \rceil} \times \sqrt{\lceil V' \rceil}$ submesh. Once the level numbers of vertices in V' are known, the level number of vertices in $V - V'$ is one plus the maximum level number in V'. If we exclude the recursive call, the other parts (including compression, uncompression and prefix operations) of the above computation can clearly be done in $O(\sqrt{n})$ time. Since $|V'| = \sum_{i=0}^{h-1} |L_i| = \sum_{i=0}^{h-1} \beta^i \leq n/\beta$, the time complexity, $T(n)$, of the above procedure satisfies the recurrence $T(n) \leq T(n/\beta) + c\sqrt{n}$, where c is some constant. Since $\beta > 1$, this recurrence implies that $T(n) = O(\sqrt{n})$. □

From now on, it is implicitly assumed that every vertex of D knows its level number.

2.3 The multisearch problem

We will view a hierarchical DAG $D = (V, A)$ as a search structure. That is, D is used to guide the computation of the answers to queries. A query thus corresponds to a path from L_0 to L_h. Instances of hierarchical DAGs are the subdivision hierarchies of Kirkpatrick [Kir83] and (overlapped) quad-tree [Sam84] which have wide applications in computational geometry and computer vision, respectively.

Let U denote the set of possible search queries. A *search path* for a query q is a directed path $P(q) = (v_0, v_1, \ldots, v_h)$, where $v_i \in L_i$, is defined by a successor function $succ_q : V \times U \rightarrow V$ such that $v_{i+1} = succ_q(v_i)$. Note that $(v_i, succ_q(v_i))$ is an arc of D. A search path $P(q)$ is not given as input, instead, it has to be generated "on the fly" with v_{i+1} generated only after reaching v_i for all $0 \leq i \leq h-1$. We will assume that, for any $q \in U$ and $v \in P(q)$, $succ_q(v)$ can be computed in $O(1)$ time. The assumption is reasonable because each vertex has only a constant number of outgoing arcs. Given a set of n queries $q_1, q_2, \ldots, q_n$, the multisearch problem is then to trace the $P(q_i)$s, starting from level 0 and ending with level h. Here by "tracing" a path $P(q_i) = (v_0, v_1, \ldots, v_h)$, we mean *touching* the vertices along it in the order of v_0, v_1, ..., v_h (vertex v_j is *touched* by $P(q_i)$ if v_j and q_i are held in the same processor). In the applications of this graph multisearch problem, the computation of the answer to a query q occurs as the vertices of its search path are touched in the correct sequence, that is, as query q "travels" from v_0 to v_1 to v_2 ...to v_h. The difficulty is in how to do the n queries simultaneously (it is trivial to do just one query).

In this paper, we will show that this multisearch problem can be solved in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC, and give its applications. We will first present an algorithm which runs in $O(\sqrt{n})$ time but uses $O(\log^* n)$ storage per processor. The number of storage registers used in each processor will then be further reduced to $O(1)$ to obtain an optimal algorithm.

We will assume, in what follows, that the vertices of D are evenly distributed in the $\sqrt{n} \times \sqrt{n}$ MCC, i.e. that each of the n processors contains a vertex of D (together with its adjacency list). By Lemma 2.2, we also assume that each vertex already knows its level number. Since $|A| \leq \beta|V|$, we will use $|D|$ to denote the size of D (, i.e. the number of its vertices).

3 Multisearch on hierarchical DAGs

In this section, we present an optimal MCC algorithm for the multisearch problem on hierarchical DAGs. We will first present an algorithm that runs in $O(\sqrt{n})$ time but uses $O(\log^* h)$ storage per processor. Later in this section, the storage per processor will be further reduced to $O(1)$ to obtain an optimal algorithm.

3.1 A preliminary algorithm

In this subsection, we present an algorithm for performing n queries on a hierarchical DAG, which runs in $O(\sqrt{n})$ time and uses $O(\log^* h)$ storage per processor on a $\sqrt{n} \times \sqrt{n}$ MCC. We first review an algorithm of [DSS88] that achieves the $O(\sqrt{n})$ time performance but uses $O(\log n)$ stroage per processor, and then show that $O(\log^* n)$ storage per processor is sufficient to achieve the $O(\sqrt{n})$ time performance. Since each application of the successor function takes $O(1)$ time, the search can be advanced from level i to $i+1$ in $O(\sqrt{n})$ time, using the random access read operations, provided that the search is at level i.

Lemma 3.1 *The search can be advanced from level i to level $i+1$ in $O(\sqrt{n})$ time, provided that the search is at level i.*

The algorithm of [DSS88] is sketched in the following lemma.

Lemma 3.2 *[DSS88] Given an n-vertex hierarchical DAG $D = (V, A)$ and n queries on D, the multisearch problem can be solved in $O(\sqrt{n})$ time with $O(\log n)$ storage per processor.*

Proof: For simplicity, let us assume that that $\sqrt{|L_i|}$ is divisible by $\sqrt{|L_{i-1}|}$, for $1 \leq i \leq \log^* h$. The modification to handle the general case will be explained later. Let S_i be the subgraph of D induced by its vertices between level 0 and i inclusive. The algorithm of [DSS88] consists of two phases: the duplication phase and the searching phase. In the duplication phase, each L_i is duplicated such that each square submesh of $|L_i|$ processors contains a copy of L_i. In the searching phase, the search is advanced from level i to $i+1$ in $O(\sqrt{|L_i|})$ time, using each square submesh of $|L_i|$ processors for $|L_i|$ of the n queries (Lemma 3.1). This is possible because, after the duplication phase, each square submesh of $|L_i|$ processors contains a copy of L_i. Since $|L_i| = O(\beta^i)$ and β is a constant, the total time of the searching phase is $O(\sum_{i=0}^{h} \sqrt{|L_i|}) = O(\sqrt{n})$ time. We next explain the duplication phase.

In the duplication phase, the L_is are duplicated in the order of $L_h, L_{h-1}, \ldots, L_0$. Each L_i is duplicated in $O(\sqrt{|L_i|})$ time, using each square submesh of $|L_i|$ processors. In order to do this, we need to maitain that, before L_i is duplicated, each square submesh of $|L_i|$ processors contains a copy of S_i. This is possible because the number of vertices in S_i is $O(\sum_{j=0}^{i-1} |L_j|) = O(|L_i|)$ since $|L_j| = \Theta(\beta^j)$ and β is a constant. Thus the duplication phase is done as follows. For $i = \log^* h, \log^* h - 1, \ldots, 0$, we do the following three steps: (i) partition the MCC into square submeshes of $|L_i|$ processors, (ii) in each such submesh, obtain a copy of L_i from the copy of S_i stored in that submesh, (iii) in each such submesh, obtain a copy of S_{i-1} from S_i, and duplicate S_{i-1} such that each square subsubmesh (i.e. the submesh of that submesh) of $|L_{i-1}|$ processors contains a copy of S_{i-1}. The initial consition of the above process is that each square submesh of $|L_h|$ processors contains a copy of D. This condition can be set up in $O(\sqrt{n})$ time since $|D| = O(|L_h|)$. Each iteration of (ii) can be done in $O(\sqrt{|L_i|})$ time since each vertex knows its level number. Each iteration of (iii) can be done in $O(\sqrt{|L_i|})$ time as in Lemma 2.1. The total time of (i)–(iii) thus is $O(\sum_{i=0}^{h} \sqrt{|L+i|}) = O(\sqrt{n})$. The number of storage used in each processor is $O(h) = O(\log n)$ since each processor conains a vertex of each L_i for $1 \leq i \leq h$.

When $\sqrt{|L_i|}$ is not divisible by $\sqrt{|L_{i-1}|}$, we need to handle the following two problems in the searching phase: (i) leftover pieces might appear while we partition the MCC (resp., a submesh) into submeshes (resp., subsubmeshes) to duplicate the L_is (resp., S_is), (ii) the partitioning (of a square submesh of $|L_i|$ processors) to duplicate S_i might not match the partitioning (of the MCC) to duplicate L_i. The leftover pieces can be handled as previously

explained in Lemma 2.1. Since, in the layouts of the partitioning (mentioned in problem (ii)), each square of one partitioning overlapps no more than four squares of the other partitioning, problem (ii) can be handled by increase of only a constant factor of the total time complexity. □

With a different duplication scheme in the duplication phase, and also a faster constant storage per processor algorithm in the searching phase, we show that the number of storage per processor can be reduced to $O(\log^* n)$ while maintaining the $O(\sqrt{n})$ time performance. We next outline the algorithm which runs in $O(\sqrt{n})$ time with $O(\log^* h)$ storage per processor. The details are given in the lemmas which follow.

Let $B_i = (V_i, A_i)$ be the subgraph of D induced by the vertices of D between level $h - 2\log^{(i)} h$ and $h - 1 - 2\log^{(i+1)} h$ inclusive. In the algorithm which runs $O(\sqrt{n})$ time with $O(\log^* h)$ storage per processor, D is partitioned into $\log^* h$ partitions, $B_0, B_1, \ldots, B_{\log^* h)}$i, and each B_i is duplicated such that each square submesh of $|B_i|$ processors contains a copy of B_i. We the do the search on B_0, then $B_1, \ldots$ then $B_{\log^* h}$ in the searching phase, using the algorithm of Lemma 3.5. It is clear that the number of storage used in each processor is $O(\log^* h) = O(\log^* n)$. We next show that the duplication and the searching phases can be done in $O(\sqrt{n})$ time.

Let B_i be defined as above and $R_i = D - \cup_{j=i}^{\log^* h} B_j$. The following lemma is used in the analysis of the algorithms.

Lemma 3.3 *1. B_i has $O(\log^{(i)} h)$ levels and $|B_i| = O(\frac{n}{(\log^{(i)} h)^2})$.*

2. R_i has $O(\frac{n}{(\log^{(i)} h)^2})$ vertices.

3. $\sum_{i=1}^{\log^* h} \sqrt{|B_i|} = O(\sqrt{n})$.

4. $\sum_{i=1}^{\log^* h} \sqrt{|B_i|} \log^{(i+1)} h = O(\sqrt{n})$.

Proof: direct consequences of simple combinatorial calculations and the definitions of hierarchical DAGs, B_is, and R_is. □

The details of the duplication phases are given in the following lemma.

Lemma 3.4 *Given an n-vertex hierarchical DAG D, in $O(\sqrt{n})$ time, the B_is can be duplicated such that each square submesh of $|B_i|$ processors contains a copy of B_i.*

Proof: The initial condition of the following process is that each square submesh of $|B_{\log^* h}|$ processors contains a copy of D. This initial condition can be set up in $O(\sqrt{n})$ time with $O(1)$ storage per processor since $|B_{\log^* h}| = O(n)$. Let $R_{1+\log^* h} = D$. The duplication is done in $O(\sqrt{n})$ time as follows. For $i = \log^* h, \log^* h - 1, \ldots, 0$, partition the MCC into square submesh of $|B_i|$ processors, and on each such square submesh, do the following: (i) obtain B_i from R_{i+1}, (ii) obtain R_i from R_{i+1} and duplicate R_i such that each square submesh of $|B_{i-1}|$ processors contain a copy of R_i. The purpose of (ii) is to ensure that (i) can be done independently on each sauqre submesh of the MCC and hence is done in $O(\sqrt{|B_i|})$ time. Since R_{i+1} contains $O(|B_{i-1}|)$ vertices, each iteration of (ii) takes $O(|B_i|)$ time and increases the number of storage used in each processor by $O(1)$. Thus, the total time of the above process is $O(\sqrt{n} + \sum_{i=1}^{\log^* h} |B_i|) = O(\sqrt{n})$ (by Lemma 3.3), its storage complexity of each processor is $O(\log^* h) = O(\log * n)$. □

The searching phase is based on the routine described in the following lemma.

Let h_1 and h_2 be two level numbers ($0 \le h_1 < h_2 \le h$), and m be the number of vertices between level h_1 and h_2 inclusive (i.e. $m = \sum_{i=h_1}^{h_2} |L_i|$).

Lemma 3.5 *Given the m-vertex subgraph of D induced by its vertices between level h_1 and h_2 inclusive, and m queries on D on a $\sqrt{m} \times \sqrt{m}$ MCC, the search can be advanced from level h_1 to level h_2 in $O(\sqrt{m}\log(h_2 - h_1))$ time with $O(1)$ storage per processor.*

Proof: Let $D' = (V', A')$ be the subgraph of D induced by the vertices between level h_1 and $h_2 - 2\log(h_2 - h_1)$ inclusive. Note that $|V'| = O(m/(h_2 - h_1)^2)$. The algorithm consists of the following four steps: (i) identify D' from D, (ii) duplicate D' such that each square submesh of $|V'|$ processors contains a copy of D' (as in Lemma 2.1), (iii) partition the MCC into square submeshes of $|V'|$ processor and use each square submesh processors to advance $|V'|$ of the m queries from level h_1 to level $h_2 - 2\log(h_2 - h_1)$, and (iv) for the remaining levels, advance the search level by level. Step (i) is trivial to do in $O(\sqrt{n})$ time since each vertex knows its level number. Step (ii) can be done in $O(\sqrt{n})$ time by Lemma 2.1. Since each application of the successor function takes $O(1)$ time, the search can be advanced from level i to level $i+1$ in $O(\sqrt{m})$ time, using the random access read. Because of this, Step (ii) can be done in $O(\sqrt{|V'|}(h_2 - h_1)) = O(\sqrt{m})$ time since $|V'| = O(m/(h_2 - h_1)^2)$, and Step (iii) can be done in $O(\sqrt{m}\log(h_2 - h_1))$ time. □

Lemma 3.6 *Given B_is such that each square submesh of $|B_i|$ processors contain a copy of B_i for $i = 0, 1, \ldots, \log^* h$, and n queries on D, the search can be done in $O(\sqrt{n})$ time.*

proof: We do the search on B_0 then B_1 then ... then $B_{\log^* h}$. Since each square submesh of $|B_i|$ processors contain a copy of B_i, we can partition the MCC into square submeshes of $|B_i|$ processors and use each such square submesh to do the search for $|B_i|$ of the n queries. Since B_i has $O(\log^{(i)} h)$ levels, the search on B_i can be done in $O(\sqrt{|B_i|}\log^{(i+1)} h)$ time, using the algorithm described in by Lemma 3.5. Thus, the time complexity $T(n)$ of the search satisfies the inequality $T(n) \leq c\sum_{i=1}^{\log^* h} \sqrt{|B_i|}\log^{(i+1)} h$, where c is some constant. By Lemma 3.3), $T(n) = O(\sqrt{n})$. □

We therefore have the following theorem.

Theorem 3.1 *Given an n-vertex hierarchical DAG D of h levels and n queries on D, the multisearch problem can be solved in $O(\sqrt{n})$ time with $O(\log^* h)$ storage per processor.*

Proof: clear from Lemma 3.4 and Lemma 3.6.

The same idea can be used to obtain an algorithm that runs in $O(\sqrt{n}\log^* h)$ time and $O(1)$ storage per processor.

Corollary 3.1 *Given an n-vertex hierarchical level graph D of h levels and a set of n quries on D, the multisearch problem can be solved on a $\sqrt{n} \times \sqrt{n}$ MCC in $O(\sqrt{n}\log^* h)$ time with $O(1)$ storage per processor.*

Proof: To obtain an algorithm that achieves $O(\sqrt{n}\log^* h)$ time and $O(1)$ storage per processor, we interleave the duplication phase and the searching phase. That is, B_{i+1} is duplicated after the search on B_i is completed. This is done as follows. For $i = 0, 1, \ldots, \log^* h$, do the following two steps: (i) duplicate B_i such that each square submesh of $|B_i|$ processors contains a copy of B_i, (ii) partition the MCC into square submesh of $B_i|$ processors, and use each such submesh to do the search on B_i for $|B_i$ of the n processors. It is clear that the total time of (i) is $O(\sqrt{n}\log^* h)$ since it is iterated $\log^* h$ times and each iteration can be done in $O(\sqrt{n})$ time. The total time of (ii) is $O(\sqrt{n})$ time as in Lemma 3.6. □

3.2 Reducing the storage

In this subsection, we will show how to reduce the $O(\log^* h)$ storage bound of theorem 3.1 to $O(1)$ and hence obtain an MCC algorithm whose time and storage bounds are both

optimal. We will develop a new duplication scheme to reduce the total number of storage to $O(n)$, and a new distribution scheme to store the data evenly among the processors and still preserve the required locality (locality requires that each B_i be stored contiguously, i.e. in a submesh).

3.2.1 Reducing the total storage

To reduce the $O(n \log^* h)$ total storage to $O(n)$, we decrease the duplications of each B_i. The idea is as follows. Let $c > 1$ be a constant such that $|B_i| < \frac{cn}{(\log^{(i)} h)^2}$. Note that such a constant c exists since $|B_i| = O(\frac{n}{(\log^{(i)} h)^2})$ (recall that B_i is the subgraph of D induced by the vertices in $\cup_{j=h-2\log^{(i)} h}^{h-2\log^{(i+1)} h} L_j$ and $|L_j| = \beta^j$). Instead of partitioning the $\sqrt{n} \times \sqrt{n}$ MCC into square submeshes of size $|B_i|$ each, each of which contains a copy of B_i, we now partition the MCC into square submeshes of size $\frac{cn}{(\log^{(i+1)} n)^2}$ each, each of which contains a copy of B_i. Before we do the search on B_i in the searching phase, we duplicate B_i within each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors in $O(\frac{\sqrt{n}}{\log^{(i+1)} h})$ time, to ensure that each square submesh of $|B_i|$ processors contains a copy of B_i. We next show that the total number of the storage of the above procedure is $O(n)$. In the next subsection, we will show how to avoid congestion such that each processor uses only $O(1)$ storage.

Let $f(k) = 2^{f(k-1)}$ and $f(1) = 2$. The following two inequalities will be used in the analysis.

$$\sum_{k=1}^{\infty} 1/f(k) \leq 1 \tag{1}$$

$$\sum_{k=1}^{\infty} (1/f(k))^2 \leq 1/2 \tag{2}$$

Lemma 3.7 *The multisearch problem can be solved in $O(\sqrt{n})$ time with $O(n)$ total storage.*

Proof: Let $S(n)$ be the total number of storage used in the above process. Since each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors contains exactly one copy of B_i, there are $\frac{1}{c}(\log^{(i+1)} h)^2$ copies of B_i stored in the MCC. The total storage, $S(n)$, of the above process then satisfies the following inequality

$$\begin{aligned} S(n) &\leq c_1 \sum_{i=0}^{\log^* h-1} \frac{n}{(\log^{(i)} h)^2}(\log^{(i+1)} h)^2) \\ &\leq c_1 n \sum_{i=0}^{\log^* h-1} (\frac{\log^{(i+1)} h}{\log^{(i)} h})^2, \end{aligned}$$

where c_1 is some constant. Since $\log^{(i)} h = 2^{\log^{(i+1)} h}$ and $\log^{(i+1)} h \geq 4$, for $0 \leq i \leq \log^* h - 1$, we have $\log^{(i)} h \geq (\log^{(i+1)} h)^2$. Hence,

$$\begin{aligned} \sum_{i=0}^{\log^* h-1} \left(\frac{\log^{(i+1)} h}{\log^{(i)} h}\right)^2 &\leq \sum_{i=0}^{\log^* h-1} \left(\frac{1}{\log^{(i+1)} h}\right)^2 \\ &\leq \sum_{i=1}^{\infty} \left(\frac{1}{f(i)}\right)^2 \\ &\leq 1/2, \text{ by inequality (2).} \end{aligned}$$

Thus, $S(n) \leq \frac{c_1}{2} n$. We next show that the extra time to do the local duplications is $O(\sqrt{n})$.

Let $T(n)$ be the total extra time to do the local duplications. Since each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors contains a copy of B_i, we can duplicate B_i within each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors in $O(\frac{\sqrt{cn}}{\log^{(i+1)} h})$ time. Then,

$$\begin{aligned} T(n) &\leq c_2 \sum_{i=0}^{\log^* h-1} \frac{\sqrt{n}}{\log^{(i+1)} h} \\ &\leq c_2 \sqrt{n} \sum_{i=0}^{\log^* h-1} \frac{1}{\log^{(i+1)} h} \\ &\leq c_2 \sqrt{n} \sum_{i=1}^{\infty} \frac{1}{f(k)} \\ &\leq c_2 \sqrt{n}, \text{ by inequality (1)} \end{aligned}$$

where c_2 is some constant.

We therefore reduce the total storage to $O(n)$ and still maintain the $O(\sqrt{n})$ time performance. □

The previous lemma guarantees that the total storage requirements are not too much, but it does not guarantee that "congestion" does not locally occur at a processor, causing it to need too much storage. We next show how to store these B_is such that the number of storage per processor is reduced to $O(1)$ and the time performance is still $O(\sqrt{n})$.

3.2.2 Avoiding congestion and preserving locality

Since each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors might be allocated to store one copy of B_{i+1}, and $(\frac{\log^{(j+1)} h}{\log^{(i+1)} h})^2$ copies of B_j for $j = 0, 1, \ldots, i$, it is not clear how to distribute the data to achieve the following two requirements.

1. The locality is preserved well enough to maintain the $O(\sqrt{n})$ time performance (recall that by "locality", we mean that each B_i is in a square submesh of $O(|B_i|)$ processors).

2. Congestion is avoided, and hence the number of storage used in each processor is $O(1)$.

We adopt the following distribution scheme to achieve both 1 and 2.

Rule 1. When a submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors is allocated to hold a copy of $B_i = (V_i, A_i)$. B_i is stored in that submesh's top-left square portion ("subsubmesh") of $|B_i|$ processors.

Rule 2. When a submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors is allocated to hold a copy of B_{i+1}, B_{i+1} is stored in the processors of that submesh which are not allocated to hold any B_j for all $j \leq i$.

Rule 1 is adopted to ensure the "locality" needed for maintaining the $O(\sqrt{n})$ time performance. and Rule 2 is adopted to ensure even distribution and hence reduce the storage registers per processor to $O(1)$. We next prove that the above distribution scheme does indeed reduce the number of storage registers needed per processor. while maintaining the required locality. In the next subsection. we will give an $O(\sqrt{n})$ implementation to realize Rule 1 and Rule 2.

Lemma 3.8 *The above distribution scheme stores the B_is such that*

1. *each square submesh (of the MCC) of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors contains a copy of B_i,*
2. *each copy of B_i is stored in the top-left square subsubmesh (of a square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$) of $\frac{cn}{(\log^{(i)} h)^2}$ processors, and*
3. *the number of storage used in each processor is $O(1)$.*

Proof: By Rule 1. it is clear that claims 1 and 2 are true. We next show that Rule 2 does reduce the storage per processor to $O(1)$.

Consider a square submesh M of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors. Let r denote the number of processors of M which are allocated to hold some B_j, for $0 \leq j \leq i$. Since each pair of adjacent copies of B_j are stored $\frac{\sqrt{cn}}{\log^{(j+1)} h}$ apart (both horizontally and vertically), there are at most $(\frac{\log^{(j+1)} h}{\log^{(i+1)} h})^2$ submeshes of $\frac{cn}{(\log^{(j)} h)^2}$ processors of M that are allocated to hold B_j. Therefore,

$$\begin{aligned} r &\leq \textstyle\sum_{j=0}^{i} \frac{cn}{(\log^{(j)} h)^2}\left(\frac{\log^{(j+1)} h}{\log^{(i+1)} h}\right)^2 \\ &\leq \frac{cn}{(\log^{(i+1)} h)^2} \textstyle\sum_{j=0}^{i}\left(\frac{\log^{(j+1)} h}{\log^{(j)} h}\right)^2 \\ &\leq \frac{cn}{(\log^{(i+1)} h)^2} \textstyle\sum_{j=0}^{i}\left(\frac{1}{\log^{(j+1)} h}\right)^2, \text{ since } \log^{(j)} h \geq (\log^{(j+1)} h)^2 \\ &\leq \frac{1}{2}\frac{cn}{(\log^{(i+1)} h)^2}. \end{aligned}$$

This gives us that, when M is allocated to hold a copy of B_{i+1}, there are $\frac{cn}{2(\log^{(i+1)} h)^2}$ processors of M available, under Rule 1 and Rule 2. Since $B_{i+1} \le \frac{cn}{(\log^{(i+1)} h)^2}$, the number of storage registers used in each processor is indeed constant. □

We next give an $O(\sqrt{n})$ implementation to realize Rule 1 and Rule 2.

3.3 The implementation

The implementation to realize the allocation and distribution schemes of the last two subsections consists of two phases: the allocation phase and the distribution phase. In the allocation phase, the processors allocated to store a copy of B_i are "marked" with i, for $i = 0, 1 \ldots, \log^* h - 1$. Each B_i is then duplicated and its copies are distributed to their allocated processors in the distribution phase.

Let c be a constant such that $|B_i| \le \frac{cn}{(\log^{(i)} h)^2}$ (such a constant exists since $|B_i| = O(\frac{n}{(\log^{(i)} h)^2})$).

Algorithm SPARSE-DUPLICATION

1. For $i = \log^* h - 1, \log^* h - 2, \ldots, 0$, "mark" with index i the top-left $\frac{cn}{(\log^{(i)} h)^2}$ processors of every square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors.

2. For $i = \log^* h - 1, \log^* h - 2, \ldots, 0$, do the following (a)–(b):

   (a) In parallel, for each square submesh of $\frac{cn}{(\log^{(i)} h)^2}$ processors, obtain a copy of B_i from R_{i+1} and store that copy in the processors which are "marked" with index i (recall that B_i is the subgraph of D induced by the vertices in $\cup_{j=h-2\log^{(i)} h}^{h-2\log^{(i+1)} h} L_j$ and $R_i = V - \cup_{j=i}^{\log^* h - 1} B_i$). If there is no processor marked i, then discard that copy of B_i.

   (b) In parallel, for each square submesh of $\frac{cn}{(\log^{(i)} h)^2}$ processors, obtain R_i from R_{i+1} and duplicate R_i such that each square submesh of $\frac{cn}{(\log^{(i-1)} h)^2}$ processors holds a copy of R_i, if $i \ge 1$.

End of Algorithm SPARSE-DUPLICATION

Lemma 3.9 *Given an n-vertex hierarchical DAG D of h levels on a $\sqrt{n} \times \sqrt{n}$ MCC, we can duplicate $\frac{1}{c}(\log^{(i+1)} h)^2$ copies of B_i, for all $0 \le i \le \log^* h - 1$, and store each of B_i in*

the top-left square submesh of $\frac{cn}{(\log^{(i)} h)^2}$ *of each square submesh of* $\frac{cn}{(\log^{(i+1)} h)^2}$ *processors in* $O(\sqrt{n})$ *time with* $O(1)$ *storage per processor.*

Proof: It is clear that Step 1 realizes Rule 1 since the processors marked with i are in the top-left square submesh of $\frac{cn}{(\log^{(i)} h)^2}$ processors of each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors. Step 2 duplicates and distributes each copy of B_i in its allocated processors. By Lemma 3.7, the storage registers used in each processor would be $O(1)$ if Step 2(a) stores each copy of B_i evenly in its allocated processors. The time complexity of Step 1 is $O(\sum_{i=0}^{\log^* h} \frac{\sqrt{n}}{\log^{(i)} h}) = O(\sqrt{n})$. Similarly, the total time of substep 2(b) is $O(\sqrt{n})$. We next explain how to implement Step 2(a) in $O(\sqrt{n})$ time to store each copy of B_i evenly in its allocated processors.

Step 2(a) can be implemented in $O(\sqrt{n})$ time as follows.

(i) For each of the four quadrants of the submesh, count the number of processors allocated to hold B_i.

(ii) Partition B_i into $B_{i,1}$, $B_{i,2}$, $B_{i,3}$ and $B_{i,4}$ whose sizes are proportional to the counts obtained in (i) and send $B_{i,j}$ to the (j)th quadrant of the submesh for $j = 1, 2, 3, 4$.

(iii) In parallel, recursively repeat the process in each quadrant of the submesh (to distribute $B_{i,j}$ in the (j)th quadrant for $j = 1, 2, 3, 4$).

It is clear that the above process takes $O(\frac{\sqrt{n}}{\log^{(i)} h})$ time to store B_i evenly in its allocated processors. The total time of Step 2(a) is thus $O(\sum_{i=0}^{\log^* h} \frac{\sqrt{n}}{\log^{(i)} h}) = O(\sqrt{n})$. This completes the proof of this lemma. □

Since each square submesh of $\frac{cn}{(\log^{(i+1)} h)^2}$ processors contain a copy of B_i after the above data duplication and distribution phases, we can duplicate B_i such that each square submesh of $\frac{n}{(\log^{(i)} h)^2}$ processors holds a copy of B_i in $O(\frac{\sqrt{cn}}{\log^{(i+1)} h})$ time, and then advance the search from level $h - 2\log^{(i)} h$ to level $h - 2\log^{(i+1)} h$ in $O(\frac{\sqrt{n}}{(\log^{(i)} h)} \log^{(i+1)} h)$ time, using each square submesh of $\frac{n}{(\log^{(i)} h)^2}$ processor for $\frac{n}{(\log^{(i)} h)^2}$ of the n queries. This gives us an $O(\sqrt{n})$ time and $O(1)$ storage registers per processor algorithm for the multisearch problem.

Theorem 3.2 *Given an n-vertex hierarchical DAG D and n queries on D, the multisearch problem can be solved in $O(\sqrt{n})$ time with $O(1)$ storage per processor on a $\sqrt{n} \times \sqrt{n}$ MCC.*

Proof: We sketch how to modify our algorithm for arbitrary n. The following two minor problems might occur.

1. $\sqrt{n}$ might not be divisible by $\frac{cn}{(\log^{(i)} h)^2}$ (and hence introduce "leftover pieces" while partitioning the mesh into submeshes).

2. A square submesh of $\frac{cn}{(\log^{(i)} h)^2}$ might not be total inside any square submesh of $\frac{cn}{\log^{(i+1)} h)^2}$ processors (hence the duplication of R_i (in Step 2(b) of algorithm SPARSE-DUPLICATION) has to cross the boundaries of submeshes).

They can be handled as explained in Lemma 2.1 and Lemma 3.2. □

4 Applications of the multisearch technique

The multisearch problem discussed in Section 3 is an abstraction of the search problems arising in many applications, and the class of hierarchical DAGs includes several important DAGs, such as, for example, the complete d-ary tree, the overlapped quadtree [Sam84], and the subdivision hierarchies of Kirkpatrick [Kir83]. In this section, we apply the multisearch technique presented in the Section 3 to obtaining the first optimal MCC algorithms for the 3-dimensional versions of the following geometric problems: (i) the lines-polyhedron intersection problem, in which we are given a convex polyhedron P and n lines, and we are asked to to compute the intersections of each line with P, (ii) the multiple tangent determination problem, in which we are given a convex polyhedron P and n lines, and we are asked to compute the plane tangent to P through each line, (iii) the convex hull problem, in which we are given a set of n points, and we are asked to compute their convex hull, and (iv) the intersection of convex polyhedra problem, in which we are given two convex polyhedra, and we are asked to compute their intersection. The above problems have been discussed in [ACG+88, Cho80, DK82, DK87, DSS88] on various parallel models. We review some of them. In [DK82], an $O(n \log n)$ time sequential algorithm for problem (i) was given, which implies an $O(\log n)$ time n-processor PRAM algorithm. That algorithm was modified in [DK87] to obtain an $O(\log n)$ time n-processor PRAM algorithm for the multiple tangent planes determination problem, which is then used to complete the "conquer" stage of the algorithm of [DK87] for the three dimensional convex hull problem. In [DSS88], Dehne *et. al.* presented an $O(\sqrt{n} \log n)$ time algorithm for the three dimensional convex hull problem, based on the idea of [DK87]. The polyhedra intersection problem can be easily translated to the three dimensional convex hull problem by solving the lines-polyhedra intersection problem first. It is clear from the algorithms of [DK82], [DK87], and [DSS88] that the major

tasks of their algorithms are to solve instances of the multisearch problem, and this is the only bottleneck of the MCC algorithm of [DSS88]. Therefore, it is not surprising that our optimal solution to the multisearch problem leads to optimal algorithms for the above four problems. For the sake of completeness, we review below the algorithms of [DK82], [DK87], and [DSS88]. The search DAGs implied in their algorithms are instances of hierarchical DAGs, and are defined from the hierarchical representation of convex polyhedra introduced by Dobkin and Kirkpatrick [DK82, DK85], which we review next.

4.1 Hierarchical representations of polyhedra

The algorithms of [DK82] and [DK87] for the lines-polyhedron intersection problem and the multiple tangent planes determination problem, respectively, make use of the hierarchical representation of convex polyhedra introduced by Dobkin and Kirkpatrick, which we review next (for the full details, we refer the reader to [DK82, DK85]).

Let P be an n-vertex convex polyhedron with vertex set $V(P)$, edge set $E(P)$ and face set $F(P)$. A hierarchical representation of P is a sequence of convex polyhedra $P_1, P_2, \ldots, P_h$, which refines P progressively. $P_1 = P$, P_h has no more than 15 vertices, and P_{i+1} is obtained from P_i by removing a set of low-degree independent (i.e. pairwise nonadjacent in P_i) vertices. (The degree of a vertex is the number of edges incident to it.) Since the graph of the vertices and edges of a convex polyhedron P_i is an embedding of a planar graph, at least a fixed fraction of its vertices is pairwise independent and has degree less than or equal to 6, and hence a fixed fraction of its vertices is of low-degree and pairwise independent, say, $c|V(P_i)|$ of them for some constant $0 < c < 1$ (where $V(P_i)$ is the vertex set of P_i). The number of vertices in P_{i+1} is then guaranteed to be a fixed fraction of that of P_i (specifically, $|V(P_{i+1})| = (1-c)|V(P_i)|$).

More formally, a sequence of convex polyhedra, $H(P) = P_1, P_2, \ldots, P_h$, is said to be a *hierarchical representation* of P if

1. $P_1 = P$ and $|V(P_h)| \leq 15$;

2. $V(P_{i+1}) \subset V(P_i)$ and $|V(P_{i+1})| = \alpha|V(P_i)|$, for some constant $\alpha < 1$;

3. the vertices of $V(P_i) - V(P_{i+1})$ are of degree less than or equal to 6 and form an independent set (i.e. are pairwise non-adjacent) in P_i.

Lemma 4.1 *[DSS88] Given an n-vertex convex polyhedron P. a hierarchical representation $H(P) = P_1, P_2, \ldots, P_h$ of P can be constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC.*

Proof: Dehne *et al* [DSS88] implemented the algorithm of Dadoun and Kirkpatrick [DK87] to construct a hierarchical representation of P in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC, using the list ranking algorithm of Atallah and Hambrusch [AH86]. □

4.2 Defining the search DAG

In this subsection, we review the search DAG that was implied in [DK82] and in [DK87].

Let $H(P) = P_1, P_2, \ldots, P_h$ be a hierarchical representation of P. We say that $v \in V(P_i) - V(P_{i+1})$ is a *relevant vertex* of an edge $e \in E(P_{i+1})$ if e is entirely visible from v, assuming P_{i+1} is the only opaque object in the space. Intuitively, v is a relevant vertex of an edge e if e is in the "base" of the "cone" with apex v which is cut from P_i if v were the only vertex removed from P_i (a polyhedron is a *cone* with *apex* v if all the other vertices are adjacent to v, and its *base* is the faces not containing v). Note that each edge of P_{i+1} is visible from at most two vertices of $V(P_i) - V(P_{i+1})$ since $V(P_{i+1}) \subsetneq V(P_i)$, and all the vertices in $V(P_i) - V(P_{i+1})$ are pairwise nonadjacent in P_i. One of the search DAGs $D(H(P))$ implied in the algorithms of [DK82] and [DK87] is defined as follows: (i) the vertices of $D(H(P))$ are partitioned into $h+1$ levels L_0, L_1, $\ldots$, L_h, (ii) $L_0 = v_0$ (v_0 is some distinguished node), and each node of L_i corresponds to an edge of P_{h-i} for $0 \le i \le h-1$, (iii) an edge $e_1 \in P_i$ is connected to an edge $e_2 \in P_{i-1}$, i.e. (e_1, e_2) is an arc of $D(H(P))$, if and only if $e_1 = e_2$ or e_2 is incident to a relevant vertex of e_1.

Lemma 4.2 *The $D(H(P))$ defined as above is a hierarchical DAG, and can be constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC.*

Proof: Since $|P_{i+1}| = \alpha|P_i|$ $(0 < \alpha < 1)$ and $|P_h| \le 15$, there exists two constants c_1 and c_2 such that $c_1\beta^i \le |E(P_i)| \le c_2\beta^i$ $(\beta = 1/\alpha)$ because the P_is are convex. Since each vertex of $V(P_i) - V(P_{i+1})$ has degree no more than 6 and each edge of $V(P_{i+1})$ has no more than 2 relevant vertices, each edge of P_{i+1} will be connected to no more than 12 edges of P_i. $D(H(P))$ can be easily constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC while $H(P)$ is being constructed. □

The successor function used to guide the search on $D(H(P))$ will be defined later, when we review the algorithms of [DK82] and [DK87].

4.3 Lines-polyhedron intersections

Given an n-vertex convex polyhedron P and n lines l_1, l_2, ..., l_n, the lines-polyhedron intersection problem is to compute the intersections $I_{l_i}(P)$ of l_i and P, for all $1 \leq i \leq n$. An $O(n \log n)$ time sequential algorithm for the lines-polyhedron problem was given by Dobkin and Kirkpatrick in [DK82], which implies an $O(\log n)$ time n-processor PRAM algorithm. In this subsection, we review their algorithm and explain how to implement their algorithm in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC, using the multisearch algorithm presented in Section 3. The idea of their algorithm is as follows.

Let $H(P) = P_1, P_2, \ldots, P_h$ be a hierarchical representation of P. To compute $I_{l_i}(P)$ for $1 \leq i \leq n$, the algorithm of [DK82] computes the sequence $I_{l_i}(P_h)$, $I_{l_i}(P_{h-1})$, ..., $I_{l_i}(P_1)$. It consists of two phases: the detection phase and the determination phase. In the detection phase, the lines which intersect P are identified, and in the determination phase, the intersections are computed for those lines. The details are reviewed next.

We first review the algorithm of Dobkin and Kirkpatrick [DK82] for the identification of the l_is which intersect P. Let l be one of l_1, l_2, ..., l_n, and let τ be some chosen plane perpendicular to l. To check if l intersects P, it suffices to check if $l' = l \cap \tau$ is inside the projection P' of P on τ along direction l. (note that l' is a point and P' is a convex polygon on τ). Let P_i' be the projection of P_i on τ. To check if l' is inside P', the algorithm of [DK82] checks if l' is in P_i' for $i = h, h-1, \ldots, 1$, in that order. Their algorithm is based on the fact that the sequence P_1', P_2', ..., P_h' is a refinement of P' [DK82]. In other words, $P_1' = P'$, $V(P_{i+1}') \subseteq V(P_i')'$, $V(P_i') - V(P_{i+1}')$ forms an independent set, and $|V(P_h')| \leq 15$.

For any edge e of P_i', define the *growing region* $g(e, P_i')$ of e w.r.t. P_i' to be the triangle bounded by three lines, one of them through e and each of the other two through each of the two edges adjacent to e in P_i'. Their algorithm is outlined as follows.

1. Project P_h on τ and decide if l' is in P_h'. If l' is in P_h', then conclude that l intersects P. Otherwise, identify an edge of P_h' whose growing region contains l'. If no such edge exists, then conclude that l does not intersect P. Otherwise, let e be the edge so identified and proceed to Step 2 below.

2. For $i = h-1, h-2, \ldots, 1$ or until the conclusion is reached, repeat the following steps: (a) explore the portion of P_i' which are in $g(e, P_{i+1})$, (b) if the projection on τ of that portion contains l', then conclude that l intersects P, (c) if l' is not in the projection

of that portion, then identify the edge of P'_i whose growing region contains l', (d) if no such edge of P'_i exists, then conclude that l does not intersect P; otherwise, let e be such an edge and repeat (a)–(d).

Correctness of the above procedure follows from the fact that $P' \subseteq P'_i \cup (\cup_{e \in P'_i} g(e, P'_i))$ for any $1 \leq i \leq h$, and that for any two distinct edges e_1, e_2 of P'_i, $g(e_1, P'_i)$ and $g(e_2, P'_i)$ are disjoint except possibly at the vertices of P'_i. Step 1 can be done in $O(1)$ time since $|V(P_h)| \leq 15$ (hence $|V(P'_h)| \leq 15$). Since $V(P'_{i+1}) \subseteq V(P'_i)$ and $V(P'_i) - V(P'_{i+1})$ forms an independent set, no more than one vertex of P'_i is in $g(e, P'_{i+1})$ for any $e \in V(P'_{i+1})$, and if such a vertex exists, it must be the projection on τ of a relevant vertex of e. Since each edge has no more than two relevant vertices and each relevant vertex is of constant degree, each iteration of Steps 2(a)–2(d) can be done in $O(1)$ time. Since (e_1, e_2) is an arc of $D(H(P))$ defined in Subsection 4.2 if and only if $e_1 = e_2$ or e_2 is incident to a relevant vertex of e_1, $D(H(P))$ is the search DAG of the above process. The successor $succ_l$ function identifies the edge in the next level whose growing region contains l'. We also associate each node of $D(H(P))$ with its relevant vertices (note that each node of $D(H(P))$ corresponds to an edge in $H(P)$).

We thus have the following lemma.

Lemma 4.3 *Given an n-vertex convex polyhedron P and n lines $l_1, l_2, \ldots, l_n$ on a $\sqrt{n} \times \sqrt{n}$ MCC, we can decide if $I_{l_i}(P) = \emptyset$ for all $1 \leq i \leq n$ in $O(\sqrt{n})$ time.*

We next explain how to actually compute the intersections. We will compute the sequence $I_l(P_h)$, $I_l(P_{h-1})$, ..., $I_l(P_1)$ as follows.

Suppose P_i is the first one of the sequence P_h, P_{h-1}, ..., P_1 that intersects l (i.e. $I_l(P_i) \neq \emptyset$, and $I_l(P_j) = \emptyset$, for $j > i$). The subsequence $I_l(P_h)$, $I_l(P_{h-1})$, ..., $I_l(P_i)$ can be computed while detecting the intersection. Note that $I_l(P_i)$ is the intersection of l with the portion of P_i that is explored in Step 2(a). Since that portion is of size $O(1)$, $I_l(P_i)$ can be computed in $O(1)$ time. Once we have $I_l(P_i)$, we can compute the sequence $I_l(P_{i-1})$, $I_l(P_{i-2})$, ..., $I_l(P_1)$ (with $I_l(P_{j-1})$ from $I_l(P_j)$ in $O(1)$ time for $1 \leq j \leq i$). We next explain how to compute $I_l(P_{i-1})$ from $I_l(P_i)$ in $O(1)$ time. Similarly, we can compute $I_l(P_{j-1})$ from $I_l P(j)$ in $O(1)$ time for any $j < i$. The DAG and successor function to guide the search process of the computation of $I_l(P)$ will be clear from the following' discussion. The idea

is to explore the portion of P_{i-1} from the faces of P_i that intersect l, in order to compute $I_l(P_{i-1})$.

Given $I_l(P_i) \neq \emptyset$, we can compute $I_l(P_{i-1})$ in $O(1)$ time as follows. Let u be one of the two elements of $I_l(P_i)$. We compute an element of $I_l(P_{i-1})$ from u as follows. For simplicity, assume u is in the face f of P_i and is not on any edge of P_i. We say that a vertex $v \in V(P_{i-1})$ is a *relevant vertex* of a face $f \in F(P_i)$ if and only if f is entirely visible from v when P_i is the only opaque object in the space. Note that each face has no more than one relevant vertex since $V(P_i) \subseteq V(P_{i-1})$ and $V(P_{i-1}) - V(P_i)$ forms an independent set. Let v be the relevant vertex of f (if no such vertex exists, then $u \in I_l(P_{i-1})$). One element of $I_l(P_{i-1})$ belongs to the intersection of l with the "cone" of P_{i-1} whose apex is v. Since v is of constant degree, that element can be computed in $O(1)$ time. When u is on an edge e and is not a vertex of P_i, we examine the faces of P_{i-1} incident to relevant vertices of e to obtain one element of $I_l(P_{i-1})$ in $O(1)$ time. When u is a vertex of P_i and $|I_l(P_i)| = 1$, we do the following: (i) choose a point that is on a face of $P_{i-1} - CH(V(P_{i-1} - \{u\})$, which does not contain u (note that $P_{i-1} - CH(V(P_{i-1} - \{u\})$ is a "cone" with apex u), and let h be the line defined by that point and u, (ii) choose a plane τ which is perpendicular to h (note that the projections on τ along direction h of the incident edges of u in P_{i-1} preserve their adjacent relationship around u in P_{i-1}), (iii) examine relevant vertices of the faces or edges of P_{i-1} whose projections on τ along direction h intersect the projection of l on τ along direction h (there are no more than two such faces or edges). The correctness of (i)-(iii) follows from the fact that the P_is form a hierarchical representation of P.

The search DAG of the above search process is defined as follows. Each node of L_i corresponds to and edge or a face of P_{h-i+1}, and (v_1, v_2) is an arc if and only if (i) $v_1 = v_2$ or (ii) v_2 incident to some relevant vertex of v_1. It is clear that the DAG so defined is hierarchical and can be obtained in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC. The successor function identifies the next faces or edges (no more than two) whose relevant vertices are examined to explore $O(1)$-sized portion of the next convex polyhedra. Note that the number of paths traversed by each query line l may be more than one (which violates the assumption in the formulation of the multisearch problem). This can be easily handled in our algorithm.

We therefore have the following lemma.

Lemma 4.4 *Given an n-vertex convex polyhedron P and n lines $l_1, l_2, \ldots, l_n$, we can compute $I_{l_i}(P)$ for all $1 \leq i \leq n$ in $O(\sqrt{n})$ time.*

We next explain how to compute the tangent plane $T_{l_i}(P)$ of P through l_i for all $1 \le i \le n$ in $O(\sqrt{n})$ time.

4.4 Multiple tangent planes determination

Given an n-vertex polyhedron P and n lines $l_1, l_2, \ldots, l_n$, in this subsection, we show that the tangent plane $T_{l_i}(P)$ of P through l_i for all $1 \le i \le n$ can be computed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC. In [DK87], Dadoun and Kirkpatrick presented an $O(\log n)$ time n-processor PRAM algorithm for this problem. In what follows, we will review their algorithm and show that it can be implemented on a $\sqrt{n} \times \sqrt{n}$ MCC in $O(\sqrt{n})$ time by using the multisearching result of Section 3. Since the intersections of all l_i and P can be detected in $O(\sqrt{n})$ time (see Lemma 4.3), we assume that none of the l_is intersects P. For simplicity, we assume that no four vertices of P are coplanar, and no vertex of P is on any l_i.

Let l be one of the given l_is. Let $H(P) = P_1, P_2, \ldots, P_h$ be a hierarchical representation of P. In order to compute $T_l(P)$, the algorithm of [DK87] computes the sequence $T_l(P_h)$, $T_l(P_{h-1})$, $\ldots$, $T_l(P_1)$ as follows.

1. Compute $T_l(P_h)$. For each $t \in T_l(P_h)$, (a) choose a plane τ perpendicular to t, (b) project on τ the edges of P_h incident to the vertex of P_h supporting t (note that those projections are in some half plane of τ, since P_h is convex), (c) identify the two edges whose projections form the largest angle less than π among all the projections obtained in (b). Let $b(t)$ be the two edges so identified in (c).

2. For $i = h-1, h-2, \ldots, 1$, for each $t \in T_l(P_{i+1})$ do the following steps to obtain an element of $T_l(P_i)$: (a) examine the relevant vertices of edges in $b(t)$ to see if t intersects P_i, (b) if t intersects P_i, then obtain a tangent plane t' of $T_l(P_i)$ from t and compute $b(t')$, (c) if t doesn't intersect P_i, conclude that $t \in T_l(P_i)$, and update $b(t)$.

Clearly, Step 1 can be done in $O(1)$ time since $|V(P_h)| \le 15$. Correctness of Step 2 follows from the following facts: (i) for any vertex v of P_i such that v and P_{i+1} are in different half spaces defined by t, v is a relevant vertex of an edge of $b(t)$, (ii) if such v exists, then v is unique since $V(P_i) - V(P_{i-1})$ forms an independent set, (iii) if such v does not exist, then $b(v)$ can be updated by examining the edges of P_i incident to relevant vertices of edges of $b(v)$. Since $|b(t)| = 2$, each edge has no more than two relevant vertices and each relevant vertex is of constant degree, Step 2 can be done in $O(1)$ time. It is clear

that the search DAG is the DAG $D(H(P))$ defined in Subsection 4.2. Note that the portion of $D(H(P))$ traversed by the query l is not a path, but rarher is a DAG which has no more than four vertices from each level of $D(H(P))$. However, this can be handled by our multisearch algorithm since what matters to our algorithm is that the search process for each query has the following properties: (i) it moves from one level L_i to the next level $i+1$, (ii) the vetices reached by the search at any level are on the adjacency lists of those reached at the previous level, and (iii) the number of vertices reached at any level is constant. These conditions (i)–(iii) on a query's search process are easily seen to be the only requirements for the algorithm of Section 3 to work (i.e., it is not necessary for a query's search process to trace a path).

Lemma 4.5 *Given n lines $l_1, l_2, \ldots, l_n$, and an n-vertex convex polyhedron P, $T_{l_i}(P)$ for all $1 \leq i \leq n$ can be computed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC, where $T_{l_i}(P)$ is the tangent planes of P through l_i ($T_{l_i}(P) = \emptyset$ if no such tangent planes exist).*

Based on the above lemma, we next present an optimal MCC algorithm for the three dimensional convex hull problem.

4.5 Three dimensional convex hull

The algorithm outlined below is similar to that of Dadoun and Kirkpatrick [DK87] and that of Dehne *et al* [DSS88]. Where we differ from [DK87] and [DSS88] is in the details of Steps 3 and 4 which are given in Lemma 4.5. For simplicity, we assume that the given n points are in general position, i.e. no four distinct points are co-planar. Under this assumption, all the faces of the convex hull are triangular. The algorithm can easily be modified to handle the degenerate case.

Algorithm 3D-CONVEX-HULL

Input: A set $S = \{v_1, v_2, \ldots, v_n\}$ of n points in three dimensions where each point is in one of the $\sqrt{n} \times \sqrt{n}$ processors.

Output: The convex hull $CH(S)$ of points in S. $CH(S)$ is specified by its vertices set $V(CH(S))$, edge set $E(CH(S))$, and face set $F(CH(S))$.

1. Partition S into S_1, S_2, S_3 and S_4 of size $n/4$ each by horizontal (i.e. parallel to the x-y plane) cut-planes in top-to-bottom order (i.e. the points in S_i have larger z-coordinate than those in S_{i+1}).

2. Recursively compute the convex hulls $CH(S_1)$, $CH(S_2)$, $CH(S_3)$ and $CH(S_4)$ in parallel, using one quadrant of the $\sqrt{n} \times \sqrt{n}$ processors for each S_i.

3. Combine $CH(S_1)$ and $CH(S_2)$ (resp., $CH(S_3)$ and $CH(S_4)$) to obtain $CH(S_1 \cup S_2)$ (resp., $CH(S_3 \cup S_4)$) in parallel, using one half of the $\sqrt{n} \times \sqrt{n}$ processors.

4. Combine $CH(S_1 \cup S_2)$ and $CH(S_3 \cup S_4)$ to obtain $CH(S)$.

End of 3D-CONVEX-HULL

We do Step 1 and move the points in each S_i to the ith quadrant of the $\sqrt{n} \times \sqrt{n}$ processors in $O(\sqrt{n})$ time by sorting. If we could obtain the hull of the union of two $O(n)$-size convex polyhedra in $O(\sqrt{n})$ time, then the time complexity $T(n)$ of algorithm 3D-CONVEX-HULL would satisfy the following recurrence:

$$T(n) \leq T(n/4) + c_1\sqrt{n} + c_2\sqrt{n}$$
$$T(4) \leq c_3,$$

where c_1, c_2 and c_3 are constants. This would imply that $T(n) = O(\sqrt{n})$. Thus it suffices to show how to compute the convex hull of the union of two linearly separable convex polyhedra in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC.

For a convex polyhedron P and a line (or a line segment) l, let $T_l(P)$ denote the two planes that are tangent to P and contain l ($T_l(P) = \emptyset$ if no such tangent planes exist, i.e. if l intersects P). To compute $CH(S_1 \cup S_2)$ from $CH(S_1)$ and $CH(S_2)$, we do the following two steps.

1. Compute $T_e(CH(S_2))$ for each $e \in CH(S_1)$ and also $T_e(CH(S_1))$ for each $e \in CH(S_2)$.

2. Use the tangent planes obtained in Step 1 as well as $CH(S_1)$ and $CH(S_2)$ to compute $CH(S_1 \cup S_2)$.

In Lemma 4.5, it has been shown that Step 1 can be done in $O(\sqrt{n})$ time. We next review a lemma of [DSS88] which shows that Step 2 can be done in $O(\sqrt{n})$ time.

Lemma 4.6 *[DSS88] Given $T_e(CH(S_2))$ for each $e \in CH(S_1)$ and $T_e(CH(S_1))$ for each $e \in CH(S_2)$, we can compute $CH(S_1 \cup S_2)$ in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC, provided that $CH(S_1)$ and $CH(S_2)$ are linearly separable.*

Proof: A proof was sketched in [DSS88], and we review it briefly. Since $CH(S_1)$ and $CH(S_2)$ are linearly separable, each face of $CH(S_1 \cup S_2)$ shares at least one edge with either $CH(S_1)$ or $CH(S_2)$. To construct $CH(S_1 \cup S_2)$, it suffices to examine every edge of $CH(S_1)$ and $CH(S_2)$ and, if it is an edge of $CH(S_1 \cup S_2)$, determine the faces of $CH(S_1 \cup S_2)$ incident to it. Consider an edge e of $CH(S_1)$ or $CH(S_2)$. WLOG, assume e is an edge of $CH(S_1)$. If $T_e(CH(S_2)) = \emptyset$, the line l containing e then intersects $CH(S_2)$ and hence e is not in $CH(S_1 \cup S_2)$. Otherwise, $T_e(CH(S_2))$ contains two planes t_1 and t_2 each supported by a vertex of $CH(S_2)$. Let v_1 (resp., v_2) be the vertex of $CH(S_2)$ supporting t_1 (resp., t_2). Let f_1 (resp., f_2) be the face formed by e and v_1 (resp., v_2), and let f_3 and f_4 be the two faces of $CH(S_1)$ to which e is incident. We observe that e is an edge of $CH(S_1 \cup S_2)$ if and only if the four faces f_1, f_2, f_3 and f_4 are in the same half space defined by some plane through e. Furthermore, if e is an edge of $CH(S_1 \cup S_2)$, the two faces of $CH(S_1 \cup S_2)$ to which e is incident are those two of f_1, f_2, f_3 and f_4 that form the largest angle less than π. The entire process takes $O(1)$ random access reads and $O(1)$ local computations. □

We therefore have the following theorem.

Theorem 4.1 *The convex hull $CH(S)$ of a set S of n points in three dimensions can be computed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC.*

Proof: Immediate from Lemma 4.6, 4.4, 4.5, and algorithm 3D-CONVEX-HULL.

4.6 Convex polyhedra intersection

It is easy to see that the intersection of two convex polyhedra P_1 and P_2 is the convex hull of the set of points that contains all the vertices of P_1 (resp., P_2) inside P_2 (resp., P_1) and all the intersections between the edges of P_1 (resp., P_2) and the surface of P_2 (resp., P_1). Based on this, we give the following algorithm for computing the intersection of P_1 and P_2 in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$.

Algorithm CONVEX-POLYHEDRA-INTERSECTION

Input: Two n-vertex convex polyhedra P_1 and P_2. P_1 and P_2 are distributed in the $\sqrt{n} \times \sqrt{n}$ MCC. All of the faces of P_1 and P_2 are triangulated.

Output: The intersection $P_1 \cap P_2$ of P_1 and P_2. All of the faces of $P_1 \cap P_2$ are triangulated.

1. For each $e \in E(P_1)$ (resp., $e \in E(P_2)$), find the intersections of e and P_2 (resp., P_1), if any. Let I_1 (resp., I_2) be the set of intersection points so obtained.

2. Identify the set V_1' (resp., V_2') of vertices of P_1 (resp., P_2), which are inside P_2 (resp., P_1), using the intersection information obtained in Step 1.

3. Compute the Convex Hull $CH(I \cup V_1' \cup V_2')$, using algorithm 3D-CONVEX-HULL. $CH(I_1 \cup I_2 \cup V_1' \cup V_2')$ is the intersection of P_1 and P_2.

End of algorithm CONVEX-POLYHEDRA-INTERSECTION

Step 1 can be done in $O(\sqrt{n})$ time as in Lemma 4.4. Step 2 is straightforward, and Step 3 can be done in $O(\sqrt{n})$ time as in Theorem 4.1.

We therefore have the following theorem.

Theorem 4.2 *Given two convex polyhedra P_1 and P_2, we can compute $P_1 \cap P_2$ in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ MCC.*

5 Conclusion

In this paper, we consider the multisearch problem on a class of hierarchical DAGs and present an optimal MCC algorithm for it, which leads to the frist optimal MCC algorithms for the 3-dimensional convex hull and convex polyhedra intersection problems, settling an open problem posed in [AW88] and in [MS88b]. We believe our multisearch technique would have applications to other problems as well. Although our algorithms are described for a $\sqrt{n} \times \sqrt{n}$ MCC, all of them generalize to higher dimensional MCCs.

Acknowledgements. A useful conversation with each of Professors Frank Dehne, Susanne Hambrusch, and Nou Dadoun is gratefully acknowledged.

References

[ACG$^+$88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, pages 293–327, 1988.

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

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[AW88] A. Aggarwal and J. Wein. Computational geometry. MIT Lab for Computer Science Research Seminar Series Lecture Notes 18.409, MIT, 1988.

[Cha89] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 586–591, 1989.

[Cho80] A. Chow. *Parallel Algorithms for Geometric Problems.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[DK82] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. In *ICALP*, pages 154–165, 1982.

[DK85] D. P. Dobkin and D. G. Kirkpatrick. A linear time algorithm for determining the separation of convex polyhedra. *Journal of Algorithms*, 6:381–392, 1985.

[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 mesh of processors. In *SWAT*, pages 154–162, 1988.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[JL90] C. S. Jeong and D. T. Lee. Parallel geometric algorithms on a mesh connected computer. *Algorithmica*, 1990.

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

[MS88a] R. Miller and Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37(12):1605–1618, December 1988.

[MS88b] R. Miller and Q. F. Stout. *Parallel Algorithms for Regular Architectures*. MIT Press, 1988.

[MS89] R. Miller and Q. F. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, January 1989.

[NS79] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-27(1):2–7, January 1979.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–107, February 1981.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *Computing Survey*, 16(2), June 1984.

[TK77] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, April 1977.