University of New Mexico
UNM Digital Repository

Electrical & Computer Engineering Technical Reports

Engineering Publications

5-1-2000

An Experimental Comparison of Parallel Algorithms for Ear Decomposition of Graphs using Two Leading Paradigms

D.A. Bader

A.K. Illendula

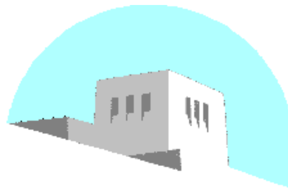
Follow this and additional works at: https://digitalrepository.unm.edu/ece_rpts

Recommended Citation

Bader, D.A. and A.K. Illendula. "An Experimental Comparison of Parallel Algorithms for Ear Decomposition of Graphs using Two Leading Paradigms." (2000). https://digitalrepository.unm.edu/ece_rpts/2

This Technical Report is brought to you for free and open access by the Engineering Publications at UNM Digital Repository. It has been accepted for inclusion in Electrical & Computer Engineering Technical Reports by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



SCHOOL OF ENGINEERING
UNIVERSITY OF NEW MEXICO

An Experimental Comparison of Parallel Algorithms for Ear Decomposition of Graphs using Two Leading Paradigms

David A. Bader [1] Ajith K. Illendula[2]

UNM Technical Report: EECE-TR-00-002

Report Date: May 1, 2000

[1]Supported in part by NSF Grant DEB 99-10123 and Department of Energy Sandia-University New Assistant Professorship Program (SUNAPP) Award # AX-3006. e-mail: dbader@eece.unm.edu

[2]Supported by a UNM Albuquerque High Performance Computing Center Research Assistantship. e-mail: ajith@eece.unm.edu

Abstract

Graph decomposition is an important and necessary operation when solving graph-theoretic problems on parallel computers, for instance, from computational fluid dynamics, mechanics, and astrophysics. In this paper, we design and analyze message-passing and shared-memory parallel algorithms that efficiently decompose a planar graph into a number of ears, known as the Ear Decomposition. This decomposition provides a general framework for solving graph problems efficiently in parallel. Our study includes both theoretical analysis and confirmation of the complexity cost using two leading parallel programming paradigms, namely, message-passing (MPI) and shared-memory (SMP) implementations. A catalog of both regular and irregular input graphs are provided to benchmark these algorithms in our empirical study.

Keywords

Parallel Algorithms, Symmetric Multiprocessors (SMP), Shared Memory, Message Passing Interface (MPI), Experimental Parallel Algorithmics.

1 Introduction

The efficient parallel solution of many computational problems often requires the invention and use of original, novel approaches radically different from those used to solve the same problem sequentially. Graph-traversal methods are useful mainly because they induce a decomposition of the graph into a structured set of simple components. Depth-first search and breadth-first search are two such methods that have been found to be effective in handling many graph-theoretic problems. However, no efficient parallel implementations of these two methods are known. To compensate for the lack of such methods, the technique of *ear decomposition*, which does have an efficient parallel implementation is often used [21]. Let $G = (V, E)$ be an undirected graph with no cycles or multiple edges, and let $n = \mid V \mid$ and $m = \mid E \mid$. Throughout this paper, every path and cycle is simple. We will also assume that the graph G is connected. An *ear decomposition* of G is a partition of the graph's edge set into an ordered collection of paths $Q_0, Q_1, \dots, Q_r$, called *ears*, such that following properties hold:

- Q_0 is a cycle called the *root ear*.
- Each endpoint of $Q_i, (i > 0)$, is contained in any Q_j, for $j < i$.
- No internal vertex of $Q_i, (i > 0)$, is contained in any Q_j, for $j < i$.

If the endpoints of the ear do not coincide, then the ear is *open*; otherwise, the ear is *closed*. An *open ear decomposition* of a graph G is an ear decomposition of G in which every ear $Q_i, (i > 0)$, is open. Some observations, which are true for both open and closed ear decomposition, follow from the definitions:

- An edge is contained in exactly one ear.
- A vertex is contained in one or more ears.
- A vertex is an internal vertex of exactly one ear, where we consider the root ear's internal vertices to be all its vertices.[18]

Whitney first studied open ear decomposition and showed that a graph G has an open ear decomposition if and only if G is biconnected [32]. Lovász showed that the problem of computing an open ear decomposition in parallel is in NC [23, 10].

The main results in this paper are

1. Practical parallel algorithms for Ear Decomposition on distributed- and shared-memory parallel machines,
2. A fast and scalable shared-memory implementation whose performance is best known to the authors, and
3. A methodology for empirically testing parallel graph algorithms using a collection of regular and irregular planar input graphs.

The organization of the paper is as follows.We will first give a sequential algorithm for ear decomposition in Section 2. Section 3 discusses parallel approaches for this problem using the idealistic Parallel Random Access Machine (PRAM) model, and two more realistic paradigms, for message-passing and shared-memory architectures. We experimentally test our algorithms on a variety of inputs, described in Section 4, and offer some conclusions on our results.

2 Sequential Algorithm

In this section, we give a well-known sequential algorithm for ear decomposition [19, 24, 27, 25]. There are five major steps in this algorithm. First, a spanning tree is found for the graph, a root vertex is arbitrarily chosen, and each vertex is then assigned a level and parent using the spanning tree. Next, the non-tree edges are examined and uniquely labeled. Finally the tree edges are assigned ear labels. Fig. 1 shows an example of ear decomposition on a graph with eight vertices. For a graph $G = (V,E)$, Alg. 1 outlines the sequential approach.

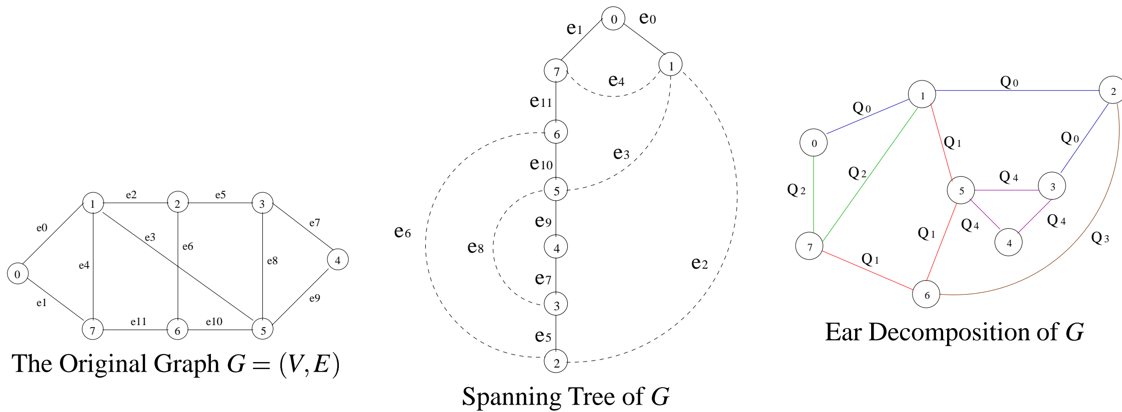


Figure 1: On the left, we show the original graph $G = (V,E)$, and the spanning tree of the graph G. On the right is the ear decomposition of G. It is not *open ear decomposition* because Q_4 is a cycle.

The spanning tree T of a graph G in Step (1) can be found in $O(m+n)$ time. Step (2) can be done using the Euler tour technique in $O(n)$ time. In Step (3), LCA(e) refers to the least common ancestor of an edge e. Step (3) can be performed within the asymptotic bounds of Step (1). The label of tree edges can be found in $O(m)$ time. Steps (5) and (6) can be done in $O((m+n)\log n)$ time. Therefore the complexity of the sequential algorithm is $O((m+n)\log n)$, where the number of vertices $n = |V|$ and the number of edges $m = |E|$.

3 Parallel Algorithms

In Section 3.1 we first discuss a parallel algorithm for *ear decomposition* using the PRAM model [19]. Because of idealistic assumptions in the PRAM model, we find that the PRAM approach does not readily adapt to real parallel architectures. In Sections 3.2 and 3.3, we design more realistic parallel algorithms for the two leading parallel

Step (1): Find a spanning tree T of G.

Step (2): Root T at an arbitrary vertex r, and compute level(v) and p(v), for each vertex $v \neq r$, where level(v) and p(v) are the level and parent of v respectively.

Step (3): For each nontree edge e=(u,v), compute LCA(e) = LCA(u,v) and level(e) = level(LCA(e)). Set label(e) = (level(e),s(e)), where s(e) is the serial number of e.

Step (4): For each tree edge g, compute label(g).

Step (5): For each tree edge e, set Q_e=$\{e\} \cup \{g \in \mathrm{T} | \mathrm{label}(g) = \mathrm{label}(e)\}$.

Step (6): Sort the Q_e's by label(e).

Algorithm 1: Sequential Algorithm for Ear Decomposition.

computing paradigms, namely message-passing and shared-memory. In our parallel algorithms, we assume, without loss of generality, that the number of processors is a power of two and that number of processors evenly divides the number n of vertices in the input graph.

3.1 PRAM Algorithm

The PRAM algorithm for ear decomposition [24] uses an approach to that of the sequential algorithm but applies concurrent operations whenever possible. We next analyze the complexity bounds of the algorithm using the Concurrent-Read Exclusive-Write (CREW) PRAM Model [19].

Step (1) requires a spanning tree algorithm, which can be found in time O($\log n$), using O($n+m$) processors. The tree can be rooted and levels and parents found in Step (2) by using the Euler-tour technique after the input representation has been modified to support this technique. Step (3) requires the LCA algorithm and can be performed within the same asymptotic bounds of Step (1).

The labels in Step (4) can be found as follows. For each vertex v, let $f(v)$= min$\{label(v,u) \mid (v,u)$ in $G-T\}$. Let $g=(x,y) \in T$, where $y=p(x)$. Then, $label(g)$ is the minimum w value in the subtree rooted at x. These two substeps can be executed in O($\log n$) time. Step (5) involves sorting the edges by their labels. This sort can be performed in O($\log n$) time using O($n+m$) processors. Therefore the total running time of the PRAM algorithm is O($\log n$) using O($n+m$) processors. This is an optimal PRAM algorithm since the work (processor-time product) is asymptotically equivalent to the best-known sequential complexity. The PRAM model provides the ideal parallel algorithm for ear decomposition.

Our goal is to design practical, parallel algorithms for ear decomposition that executes efficiently on current high-performance computers. The PRAM model makes several assumptions that may limit the performance of our algorithm on today's computers. For example, the PRAM model assumes that we have O($n+m$) processors (rather than tens or hundreds), and does not charge on algorithm for communication overhead.

In the next two sections, we address these limitations and give ear decomposition algorithms for the message-passing (Section 3.2) and shared-memory (Section 3.3) paradigms.

3.2 Message-Passing Approach

In this section we discuss the message-passing complexity model and present the spanning tree and the ear decomposition algorithms for message-passing.

Complexity Model for Message-Passing

We use a simple model to analyze the performance of our message-passing parallel algorithms. Our model is based on the fact that current distributed-memory hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, where we allow computation and communication to overlap. We account for communication costs as follows.

Assuming no congestion, the transfer of a block consisting of m contiguous words between two processors takes $(\tau + \sigma m)$ time, where τ is the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. Note that the bandwidth per processor is inversely proportional to σ. We assume that the bisection bandwidth is sufficiently high to support block permutation routing amongst the p processors at the rate of $\frac{1}{\sigma}$. In particular, for any subset of q processors, a block permutation amongst the q processors takes $(\tau + \sigma m)$ time, where m is the size of the largest block. Such cost (which is an overestimate) can be justified by previous work [20, 5].

Using this cost model, we can evaluate the communication time $T_{comm}(n, p)$ of an algorithm as a function of the input size n, the number of processors p, and the parameters τ and σ. The coefficient of τ gives the total number of times collective communication primitives are used, and the coefficient of σ gives the maximum total amount of data exchanged between a processor and the remaining processors.

This communication model is close to a number of similar models (e.g. [13, 30, 3]) that have recently appeared in the literature and seems to be well-suited for designing parallel algorithms on current high performance platforms.

We define the computation time $T_{comp}(n, p)$ as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp}(n, p) + T_{comm}(n, p)$ involves a tradeoff between $T_{comm}(n, p)$ and $T_{comp}(n, p)$. Our aim is to develop parallel algorithms that achieve $T_{comp}(n, p) = O\left(\frac{T_{seq}}{p}\right)$ such that $T_{comm}(n, p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but other optimization criteria are possible. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

Spanning Tree

Before describing the message-passing approach for ear decomposition, we will describe the parallel algorithm for the formation of spanning tree. Initially the vertices of the input graph are partitioned evenly among the processors. We find a forest of spanning trees on each processor concurrently and merge the processor subgraphs together to form a single, global spanning tree. The global spanning tree is then broadcast to each processor.

The spanning tree algorithm for message-passing (Alg. 2) is explained in detail here. Each processor runs a depth-first search algorithm and forms a forest of trees (Step (1)). We call each tree a *supervertex*, which has a unique connected component label across the machine. Each vertex in the supervertex holds the information of its parent and the connected component label to which it belongs. Next, we combine the supervertices into a single spanning tree by iteratively merging the forests on pairs of processors. The straightforward greedy approaches of randomly pairing processors or regular pairwise merging are known to perform poorly [26]. Thus, we use the following heuristic for pairwise merging. A matrix M is formed (Step (2)) whose entries are such that $M[i][j]$ represents the number of edges spanning from supervertices on processor p_i to those on processor p_j for $0 \le i, j \le p-1$ (and the diagonal $M[i][i] = \infty$). After the matrix M is formed, we find the maximum weight perfect matching among the processors (Step (3)). The objective of the maximum weight perfect matching is to reduce trees into supervertices as rapidly as possible. Also, it is known that maximum weight perfect matching involves

Step (1): On each processor, run the depth-first search algorithm to find the supervertices.

Step (2): Form the matrix M, where $M[i][j]$ represents the number of edges spanning between supervertices on processor P_i with that of processor P_j.

Step (3): Run the maximum weight perfect matching algorithm and determine the pairs of processors to be merged.

Step (4): Merge the processor pairs determined in Step (3), reducing the number of active processors.

Step (5): Repeat from Step (2) until a single processor is active.

Step (6): Broadcast the spanning tree to each processor.

Algorithm 2: Spanning Tree Algorithm for the Message-Passing Paradigm.

the selection of pairs of processors such that the sum of the entries $M[i][j]$ for all i and j such that processors P_i and P_j are paired together, is maximum [12]. Using this technique, pairs of processors are merged (Step (4)) during each of the $\log p$ iterations until there is only a single processor left, holding the entire spanning tree. Finally in Step (6) this spanning tree is broadcast to each processor.

The analysis for this message-passing approach is as follows. In Step (1), depth-first search can be performed concurrently on each precessor's n/p vertices and takes $O(n/p)$ time. As previously described, steps (2), (3), and (4) are repeated for $\log p$ iterations. Let k be the step number, with $0 \leq k < \log p$. In Step (2), we compute one row of M on each active processor, and use a Regular all-to-all broadcast communication to collect the entire matrix M on each processor. Computing each row is linear in the number of vertices held by each active processor and takes $O((2^k)n/p)$. The broadcast takes $O(\log p(\tau + p\sigma))$ communication time. In Step (3) each processor solves the maximum weight perfect matching in $O((\frac{p}{2^k})^3)$ for $\frac{p}{2^k}$ elements [29, 22], using the primal dual approach. (Note that our implementation uses the Blossom4 library [12] that takes $O((\frac{p}{2^k})^4)$). This step involves only computation because each processor has an identical copy of the matrix M. In Step (4) we merge the subgraphs held by the matched processor pairs, so a message of length $(2^k)n/p$ is sent from one processor to its partner. Thus, Step (4) takes a total of $O(\sum_{k=0}^{\log p-1}(\tau + 2^k \frac{n}{p}\sigma)) = O(n\sigma + \tau \log p)$ communication time. The broadcast in Step (6) takes a time of $O(\log p(\tau + n\sigma))$ because the spanning tree is of size $O(n)$. Hence, the total computational complexity $T_{comp}(n,p) = O(n + p^3)$ and the communication complexity $T_{comm}(n,p) = O(\log p(\tau + n\sigma))$. Therefore, the total running time for the spanning tree algorithm is

$$T(n,p) = O(n + p^3 + \log p(\tau + n\sigma)). \tag{1}$$

Notice that we do not expect this approach to perform well since, the computation and communication complexities do not scale with the number of processors or problem size.

Ear Decomposition

The parallel algorithm for ear decomposition of graph $G = (V,E)$ on a message-passing platform is given in Alg. 3. We hold a copy of the entire input graph G on each processor, but initially assign the vertices evenly among the processors.

The analysis of the message-passing ear decomposition algorithm is as follows. The time to construct the spanning tree in Step (1) is given in Eq. (1) in the previous section and takes $O(n + p^3)$ for computation and $O(\log p(\tau + n\sigma))$ for the merging and broadcast communication. The Euler-tour takes linear time in the number of vertices assigned to each processor, and thus, Step (2) takes $O(n/p)$. In Step (3) each processor sends its tour to every other processors, so it takes $O(p\tau + n\sigma)$ using a regular all-to-all broadcast. In parallel and using

Step (1): Run Alg. 2 to find the spanning tree of G.

Step (2): Each processor finds the Euler tour of its assigned vertices.

Step (3): The p subtours are broadcast to each processor.

Step (4): In parallel, find the level and parent of each vertex using the Euler tour.

Step (5): In parallel determine the label of each nontree edge.

Step (6): Each processor finds the label of $\frac{n}{p}$ tree edges. (Note that processor $p-1$ finds the labels of $\frac{n}{p}-1$ tree edges)

Step (7): Sort the Q_e's by label(e), where for every nontree edge e, Q_e=$\{$e$\}\cup\{$g$\in$T$|$label(g)=label(e)$\}$.

Algorithm 3: Parallel Algorithm for Ear Decomposition using Message-Passing.

only local operations, determining the level for each vertex takes O(n/p), labeling the non-tree edges takes O($m\log n$), and labeling the tree edges takes O(n/p). Assuming $n \gg p^3$, the total running time of the algorithm is T(n,p)=O($m\log n + n + p\tau + n\log p\sigma$). For connected input graphs, we can assume that $m \gg n$, so the term $m\log n$ dominates n. Sorting the ears in Step (7) can be done using parallel sorting algorithms [6, 15, 17]. Because parallel sorting is well-studied and is applied here only to sort the final output of ears, we do not include it in our complexity analysis or empirical tests. Hence the total running time of the parallel algorithm is

$$T(n,p) = O(m\log n + p\tau + n\sigma\log p). \quad (2)$$

Thus, as with the spanning tree, we do not expect ear decomposition to be suitable for message-passing since the time complexity does not scale with problem or machine size.

3.3 Shared-Memory Approach

In the shared-memory model, processors may communicate information by coordinated accesses to shared memory. While our shared-memory algorithm for ear decomposition is similar to the message-passing approach in that it finds a spanning tree and uses the Euler-tour technique, in fact, the underlying algorithms are quiet different. For instance, our algorithms benefit from shared-memory capabilities, elimination of significant communication steps, and faster shared-memory primitive routines. The advantage of shared-memory over message-passing is that information can be accessed concurrently by all the processors by reading a shared buffer in memory instead of broadcasting messages through a data network.

In this section we first describe the shared-memory complexity model, then present the spanning tree and ear decomposition algorithms for the shared-memory paradigm.

Complexity Model for Shared-Memory

A symmetric multiprocessor (SMP) architecture primarily consists of several powerful, identical processors and a main (shared) memory with multiple levels of memory hierarchy (e.g., L1 on-chip cache, L2 external cache, and main memory). The processors can access the main memory typically using shared buses or an interconnection network.

We use the SMP complexity model proposed by Helman and JáJá [16] to analyze our shared algorithms. The complexity models used in [1, 2, 4, 31] focus on the relative cost of accessing different levels of memory. On the other hand, a number of shared-memory models [9, 14] have focused instead on the contention caused by

multiple processors competing to access main memory. In the SMP complexity model, we measure the overall complexity of the algorithm by the triplet (M_A, M_E, T_C), where M_A is the maximum number of accesses made by any processor to main memory, M_E is the maximum amount of data exchanged by any processor with main memory, and T_C is an upper bound on the local computational complexity of any of the processors.

The term M_A is simply a measure of the number of non-contiguous main memory accesses, where each such access may involve an arbitrary-sized contiguous blocks of data. T_C is represented by the customary asymptotic notation and M_A, M_E are represented as approximations of the actual values. In practice, it is often possible to focus on either M_A or M_E when examining the cost of algorithms.

Spanning Tree

The vertices of the input graph G (held in shared memory) are initially assigned evenly among the processors, and the entire input graph G is available to all the processors through the shared memory. The spanning tree algorithm is motivated by the PRAM approach [19]. Before describing the algorithm, we first define some helpful operations. Let D be the function on the vertex set V defining a pseudoforest. A pseudoforest is defined as a collection of trees. Initially each vertex is in its own tree, and we set $D(v) = v$, for each $v \in V$. There are two basic operations performed on the pseudoforest which are defined next.

- **Grafting:** Let T_i and T_j be two distinct trees in the pseudoforest defined by D. Given the root r_i of T_i and a vertex v of T_j, the operation $D(r_i) = v$ is called *grafting* T_i *onto* T_j.
- **Pointer jumping:** Given a vertex v in a tree T, the *pointer jumping* operation applied to v sets $D(v) = D(D(v))$.

Step (1): Perform a grafting operation of trees onto smaller vertices of other trees, for all $(i, j) \in E$, if $D(i) = D(D(i))$ and $D(j) < D(i)$ then set $D(D(i)) = D(j)$.

Step (2): Graft rooted stars onto other trees if possible. For all $(i, j) \in E$, if i belongs to a star and $D(j) \neq D(i)$ then set $D(D(i)) = D(j)$.

Step (3): If all the vertices are in rooted stars, then exit. Otherwise, perform the pointer jumping opertion on each vertex v. For all i, set $D(i) = D(D(i))$.

Algorithm 4: Spanning Tree Algorithm for Shared-Memory.

The spanning tree algorithm for shared-memory (Alg. 4) is as follows. Initially each vertex is in a rooted star. The first step will be several grafting operations of the same tree. The next step attempts to graft the rooted stars onto other trees if possible. In the last step if all the vertices are in rooted stars then exit. If not then the pointer jumping operation is done, and the height of the tree is reduced. The same process is repeated from the first step until all the vertices are in rooted stars. Fig. 2 shows the grafting and pointer jumping operations performed on an example input graph.

Step (1) involves the grafting of trees which can be done in O(m/p) time, with two non-contiguous memory accesses to exchange approximately $\frac{n}{p}$ elements. In Step (2) the rooted stars are grafted onto other trees in O(m/p) time. Step (3) uses pointer jumping on all vertices, so it takes a time of O(n/p). The three steps are repeated for $\log n$ times. Note that the memory is accessed only once even though there are $\log n$ iterations. Hence the total running time of the algorithm is

$$T(n,p) = O(1, \frac{n}{p}, ((m+n)/p)\log n). \qquad (3)$$

The first term (M_A) in the triplet is the maximum number of accesses made by any processor to main memory. The second term (M_E) represents the maximum amount of data exchanged by any processor with main memory. The last term (T_C) represents the upper bound on the local computational complexity of any of the processors.

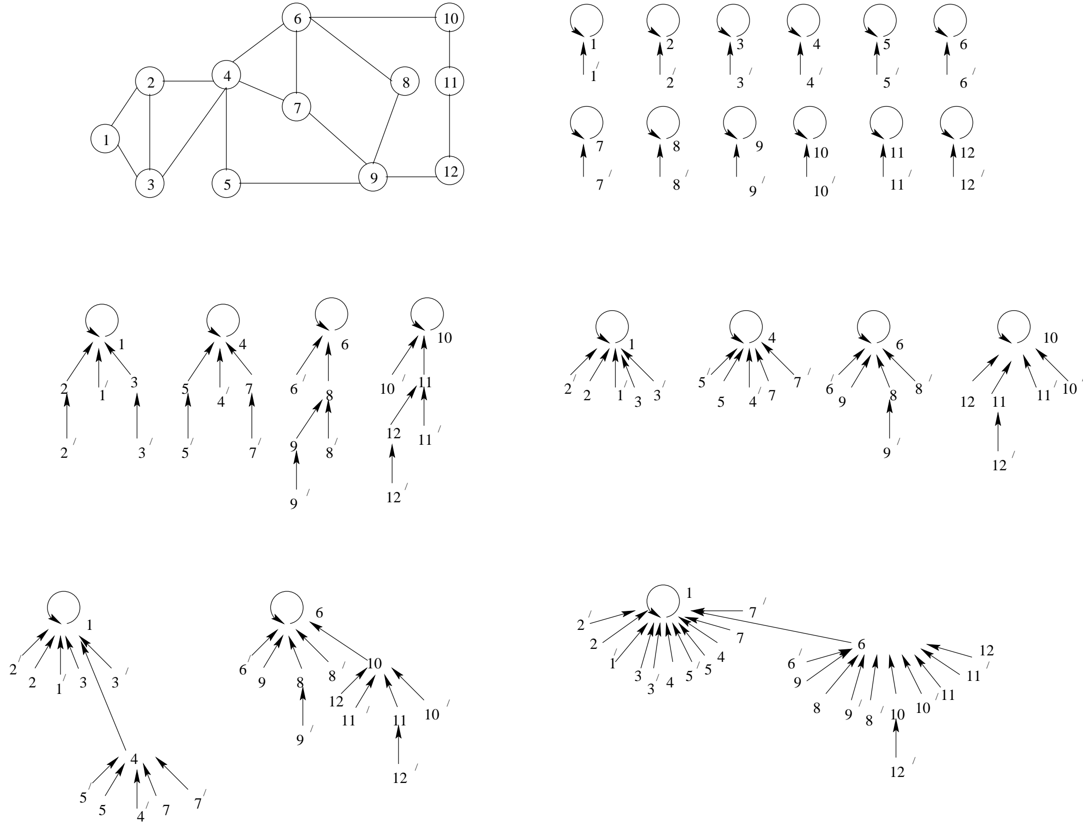


Figure 2: An example of the grafting and the pointer jumping operations applied on an input graph (top left), and steps of the algorithm (top right, middle left and right, and bottom left and right).

Ear Decomposition

Step (1): Run Algorithm 4 to find the spanning tree of G.

Step (2): Each processor finds the Euler tour with the assigned vertices.

Step (3): Find the level and parent of each assigned vertex.

Step (4): For each nontree edges, determine its label.

Step (5): Every processor finds the label of $\frac{n}{p}$ tree edges. (except processor $p-1$ which finds label of $\frac{n}{p}-1$ tree edges)

Step (6): Sort the Q_e's by label(e), where for every nontree edge e, Q_e={e}∪{g∈T|label(g)= label(e)}.

Algorithm 5: Parallel Algorithm for Ear Decomposition on a Shared-Memory Machine.

Our new shared-memory algorithm for the ear decomposition is also motivated by the PRAM algorithm. Given a graph $G=(V,E)$ in shared-memory, we can find its ear decomposition in parallel using Alg. 5. From Eq. (3), spanning tree formation in Step (1) takes $O(1, \frac{n}{p}, \frac{m+n}{p}\log n)$. Computation of the Euler tour in Step (2) takes linear time in the number of vertices per processor and hence can be done in time $O(n/p)$ with $\frac{n}{p}$ non-contiguous memory accesses to exchange $\frac{n}{p}$ elements. The level and parent of the local vertices can be found

in $O(n/p)$ time with $\frac{n}{p}$ non-contiguous memory accesses. The label of nontree edges can be found in $O(n/p)$. The label of tree edges can be computed in time $O(n/p)$. Both Step (3) and Step (4) need $\frac{n}{p}$ non-contiguous memory accesses to exchange approximately $\frac{n}{p}$ elements. The ears in Step (6) can be sorted using parallel sorting algorithms [16]. The total running time of the algorithm is $O(\frac{n}{p}, \frac{n}{p}, \frac{m+n}{p}\log n)$. Since both M_A and M_E are of the same order we can consider any one of them. As M_A is the relatively expensive than the M_E, the total complexity of the algorithm is

$$T(n,p) = O(\frac{n}{p}, \frac{m+n}{p}\log n). \tag{4}$$

Notice that our shared-memory complexity $O(\frac{m+n}{p}\log n)$ is optimal and scales efficiently since $T(n,p) = \frac{T^*(n)}{p}$, where $T^*(n)$ is the sequential complexity of ear decomposition (given previously as $O((m+n)\log n)$.

One of the differences between the shared-memory and the message-passing approaches is the algorithm for the spanning tree formation. The shared-memory approach uses grafting and pointer jumping while the message-passing algorithm uses iterative, pairwise merging to form the spanning tree. The other improvement is the elimination of communication steps in the shared-memory algorithm. For example, notice that Step (3) of Alg. 3 for broadcasting subtours is completely eliminated in the shared-memory algorithm. From our analysis, we expect the running time of the shared-memory algorithm to scale well both with problem size and with the number of processors.

4 Experimental Results

As previously described, there is an increasing gap between theoretically optimal parallel algorithms and those that are efficient in practice. Here we attempt to experimentally validate the performance of our algorithms using a large collection of input graphs that represent classes typically seen in a variety of high-performance computing applications. Our goal is to confirm that the empirical performance of our algorithms is realistically modeled, and to discover efficient and fast techniques for parallel graph algorithms such as the ear decomposition problem. In Section 4.1 we give our collection of input graph generators, Section 4.2 describes the test platforms, and Section 4.3 presents the empirical performance of our algorithms on these machines and analyzes these results.

4.1 Test Planar Input Graphs

In this section, we describe both the regular and irregular planar graph classes (see Table I) that are used in our experimental study. We generate graphs from seven test input classes (2 regular, 5 irregular) that represent a diverse collection of possible inputs. These graphs can be easily generated for arbitrary number of vertices. The classes are defined as follows. The first two classes are regular meshes (lattice RL and triangular RT). The next four classes, irregular graphs, are randomly generated sparse (GA and GB) and dense (GC and GD) planar graphs (as described in the next paragraph). The last graph class generates the constrained Delaunay triangulation (CD) on a set of random points [28].

For the random graphs GA, GB, GC, and GD, the input graph on $n = |V|$ vertices is generated as follows. A pair of uniformly random numbers between 0 and 1 are chosen and used as a two-dimensional coordinate for a vertex in the unit square. This process of generating vertices is repeated n times. A minimum-spanning tree (MST) on the n vertices using Euclidean distances is formed to ensure that the graph is connected. The graph's edge set is initially set to the edges in the MST. Then for *count* times, two vertices are selected at random and a straight edge is added between them (and to the edge set) if the new edge does not intersect with any existing edge. Note that while we generate the test input graphs geometrically, the ear decomposition is more general and uses only the vertex list and edge set for each graph.

Key	Name	Description
RL	**Regular Lattice**	A simple mesh (of $\lfloor\sqrt{n}\rfloor \times \lceil\sqrt{n}\rceil$ vertices) with each vertex connecting to each of its four neighbours, if each exists.
RT	**Regular Triangulation**	A regular lattice with an additional edge connecting each vertex to the vertex to its lower right hand, if it exists.
GA	**Random Graph**	A sparse randomly-generated planar graph with the value of $count = n$.
GB	**Random Graph**	Random planar, sparse graph with $count = 2n$.
GC	**Random Graph**	Random planar, dense graph with $count = n^2/2$.
GD	**Random Graph**	Random planar, dense graph with $count = n^2$.
CD	**Constrained Delaunay Triangulation**	Constrained Delaunay triangulation of a set of n randomly-placed points in the unit square.

Table I: Different classes of input graphs.

Notice that in the random planar graphs (GA, GB, GC and GD), the value of *count* is directly proportional to the density of the graph. For example, a count of zero produces an acyclic planar graph with n vertices and $n-1$ edges. A value of $count \gg n$ produces a dense planar graph. Fig. 3 shows some examples graphs with various values of *count*.

4.2 Experimental Platforms

We test our message-passing approach on the UNM/Alliance Roadrunner Linux Supercluster, high-performance computing platform at the Albuquerque High Performance Computer (AHPCC). Roadrunner contains 64 SMP nodes (for a total of 128 processors) interconnected by Myrinet, a high-performance gigabit switched network. Each node in the cluster contains dual 450 MHz Intel Pentium II processors with 512MB RAM and 6.4GB hard disk. Our MPI implementation uses the Portland Group C compiler, MPICH-GM version 1.1.2.4 (the Myricom extension of MPICH for Myrinet), Myrinet GM drivers 1.0.4, Redhat Linux 5.2, and the Linux SMP kernel 2.2.10. In our experiments, we are using a single processor per node [8].

Our shared-memory implementation uses the NPACI Sun HPC 10000 Uniform Memory Access (UMA) shared-memory parallel machine at the San Diego Supercomputing Center. The Sun HPC 10000 has 64 UltraSPARC II processors and 64GB of shared memory. Each 400 MHz UltraSparc II processor has 16KB of on-chip direct-mapped data cache and 8MB of external cache. The Sun HPC 10000 runs the Solaris 7 operating system and uses Sun Workshop 6.0 compilers [11].

Our practical programming environment for SMPs is based upon the SMP Node Library component of SIMPLE [7], which provides a portable framework for describing SMP algorithms using the single-program multiple-data (SPMD) program style. This framework is a software layer built from POSIX threads that allows the user to use either already developed SMP primitives or direct thread primitives. We have been continually developing and improving this library over the past several years and have found it to be portable and efficient on a variety of operating systems (e.g., Sun Solaris, Compaq/Digital UNIX, IBM AIX, SGI IRIX, HP-UX, Linux). The SMP Node Library contains a number of SMP node algorithms for barrier synchronization, broadcasting the location of a shared buffer, replication of a data buffer, reduction, and memory management for shared-buffer allocation and release. In addition to these functions, we have control mechanisms for contextualization (executing a statement on only a subset of processors), and a *parallel do* that schedules n independent work statements implicitly to p processors as evenly as possible.

4.3 Empirical Results

The graphs in Fig. 4-7 examine the performance of our ear decomposition algorithms using a variety of problem sizes and machines. Specifically, in Fig. 4 and 5 we plot the execution time of the ear decomposition using MPI and SMP, respectively. In Fig. 6 we compare the two parallel programming paradigms for similar inputs. In Fig. 7 we focus on the shared-memory approach and demonstrate that the practical performance is nearly invariant to the input graph class.

Fig. 4 plots the running time of the message-passing ear decomposition algorithm for a range of problem and machines sizes. We ran experiments with the MPI implementation on problems ranging in size from 256 to 8192 vertices on the Roadrunner Linux SuperCluster using $p = 1$ to 32 processing nodes. Notice that for a fixed problem size, the running time grew as we used more processors to find the ears of the same input graph. This performance characteristic matches our predicted cost given in Eq. (2). Thus, message-passing is not an efficient paradigm for this graph-based problem due to the significant communication overheads in our algorithm.

On the other hand, the theoretic analysis for the shared-memory ear decomposition algorithm given in Eq. (4) shows that a practical parallel algorithm is possible. We experimented with the SMP implementation on problems ranging in size from 256 to 128K vertices on the Sun HPC 10000 using $p = 1$ to 32 shared-memory processors. Recall that for a fixed problem size, we predicted that the shared-memory approach should scale linearly with the number of processors. Clearly, a linear speedup with the number of processors may not be achievable due to synchronization overhead, serial work, or contention for shared resources. In fact, our empirical results plotted in Fig. 5 confirm the cost analysis and provide strong evidence that with the shared-memory paradigm, our algorithm achieves almost linear speedup with the number of processors for a fixed problem size. Furthermore, it is clear that our shared-memory approach is practical and scalable in terms of both the problem and machine size.

In Fig. 6 we plot a direct comparison of the two leading parallel paradigms (MPI and SMP) that we used to implement the ear decomposition problem. In this graph we selected the problem size of 8192 vertices and compare the execution times of the message-passing and shared-memory approaches. In addition, the Sun HPC platform is able to run message-passing codes (using MPI from the Sun ClusterTools package), and thus we have included this performance result as well in our comparison. Note that on a single processor ($p = 1$) the algorithms run in approximately the same time. However, as the number of processors is increased, we see performance degradation in the message-passing approach on both the Linux SuperCluster and the Sun HPC platforms. In striking comparison, our shared-memory approach scales very well using the same input.

Finally in Fig. 7 we show empirical data that the running time of our shared-memory ear decomposition algorithm depends on problem and machine size, and is invariant to the *class* of the input graph.

5 Discussion

In conclusion, we have designed and analyzed fast and practical parallel algorithms for the ear decomposition. We have shown both theoretically and practically that our shared-memory implementation of the decomposition is efficient and scalable on a variety of input classes and problem sizes, and in contrast to the distributed-memory MPI approach, has almost linear speedup on current SMP architectures.

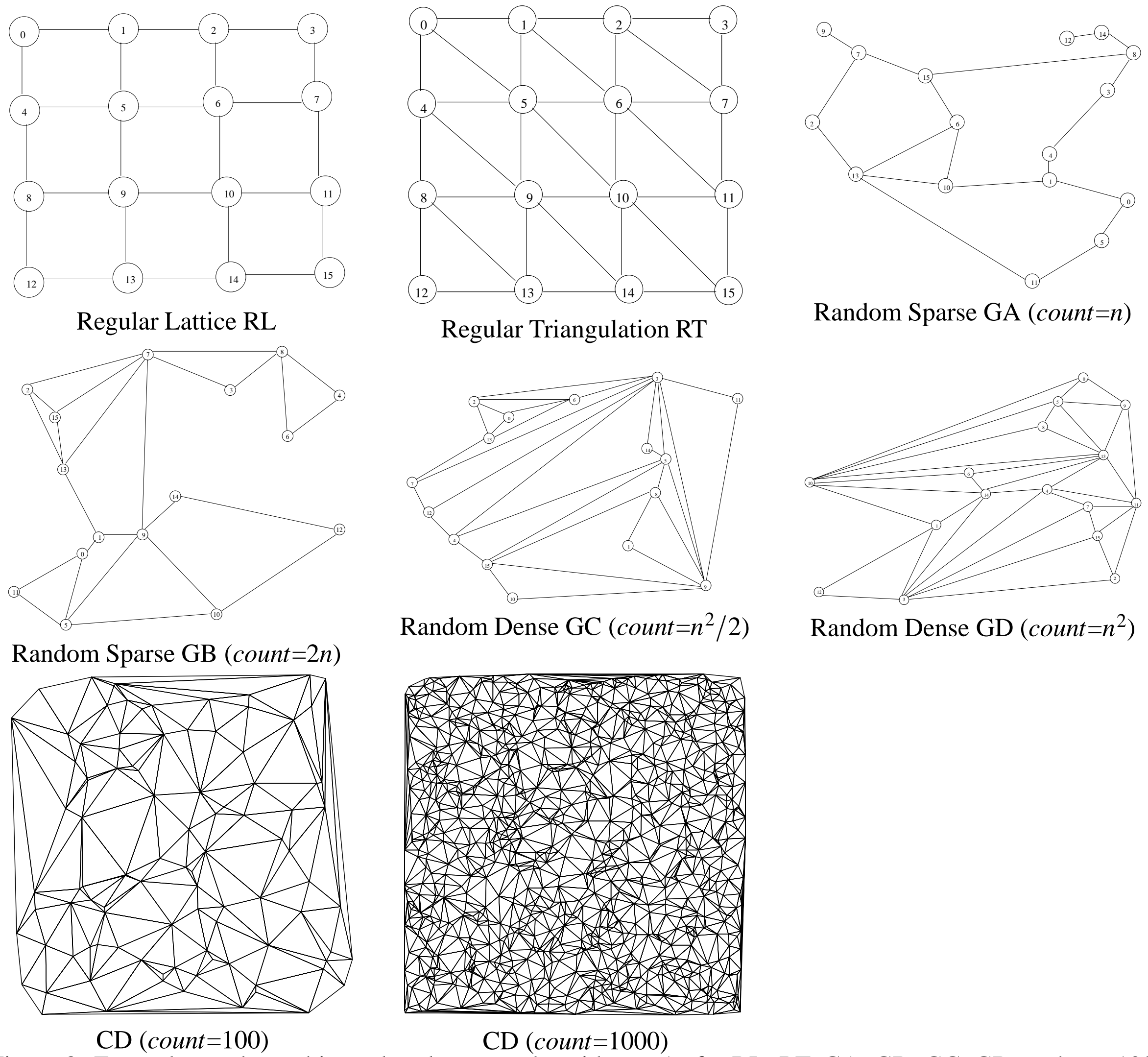


Figure 3: Example regular and irregular planar graphs with $n = 16$ for RL, RT, GA, GB, GC, GD, and $n = 100$ and 1000 for CD. From the top left are a 4 x 4 regular lattice and a regular triangulation. The next four graphs are irregular planar graphs generated randomly with $count = n, 2n, \frac{n^2}{2}$ and n^2. The last two graphs are the constrained Delaunay triangulation of a set of $n = 100$ and $n = 1000$ points, respectively.

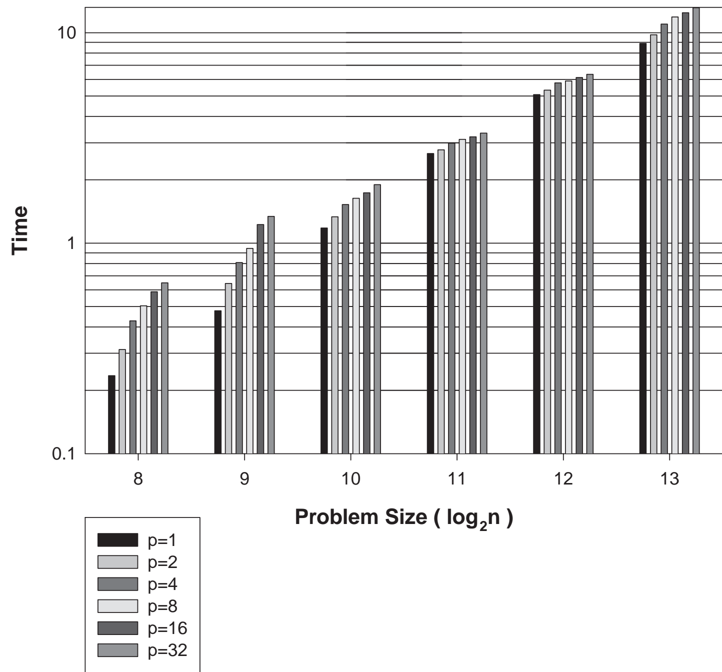


Figure 4: Execution time (in seconds) of the message-passing ear decomposition algorithm on the Roadrunner Linux SuperCluster for problem sizes of 256 to 8K vertices on varying machine sizes.

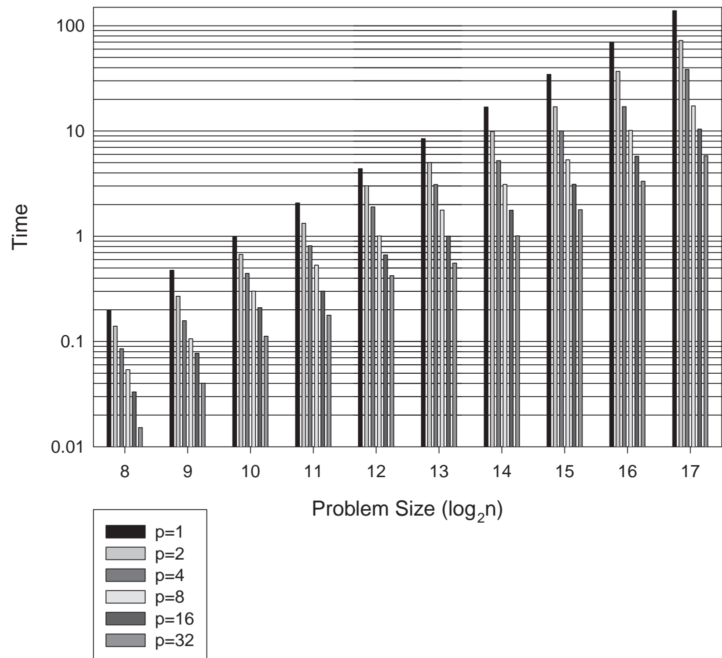


Figure 5: Execution time (in seconds) of the shared-memory ear decomposition algorithm on the NPACI Sun HPC 10000 for problem sizes of 256 to 128K vertices on varying machine size.

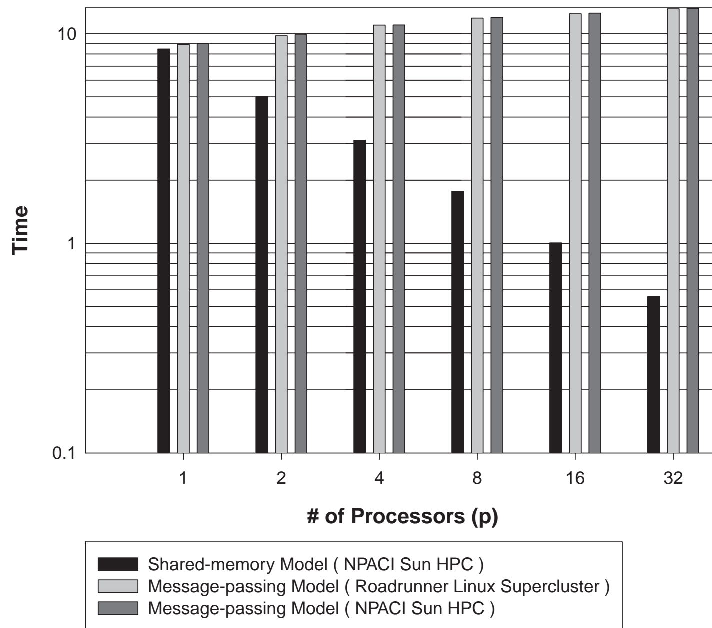


Figure 6: Comparision of the running times (in seconds) for ear decomposition on different architectures and paradigms for a fixed problem size (8192 vertices).

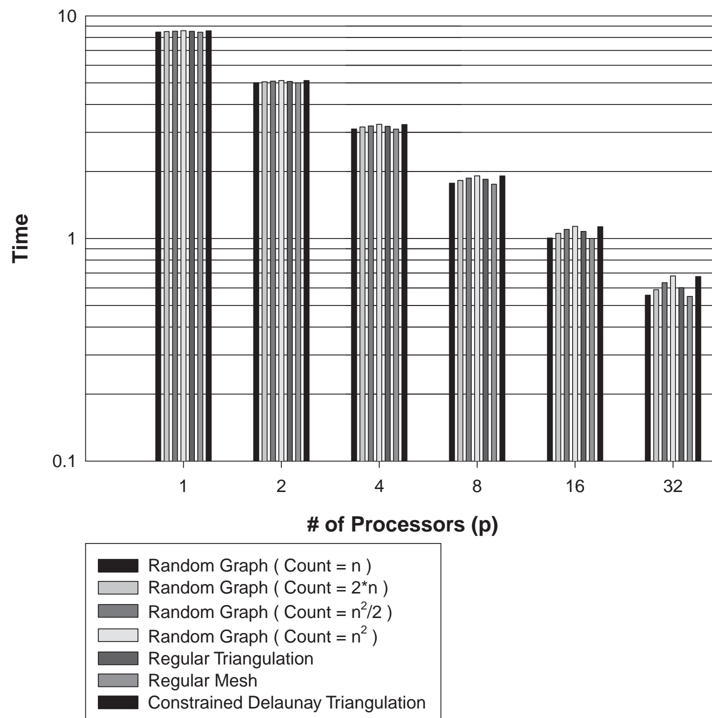


Figure 7: Comparision of the running times (in seconds) for ear decomposition with different input graphs for a fixed problem size (8192 nodes) on the NPACI Sun HPC 10000 shared-memory machine.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A Model for Hierarchical Memory. *In Proceedings of the 19th Annual ACM Symposium of Theory of Computing, New York City*, pages 305–314, May 1987.

[2] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31:1116–1127, 1988.

[3] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[4] B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12:72–109, 1994.

[5] D.A. Bader. *On the Design and Analysis of Practical Parallel Algorithms for Combinatorial Problems with Applications to Image Processing*. PhD thesis, University of Maryland, College Park, Department of Electrical Engineering, April 1996.

[6] D.A. Bader, D.R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1(3):1–42, 1996. `http://www.jea.acm.org/1996/BaderPersonalized/`.

[7] D.A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[8] D.A. Bader, A.B. Maccabe, J.R. Mastaler, J.K. McIver III, and P.A. Kovatch. Design and Analysis of the Alliance/University of New Mexico Roadrunner Linux SMP Supercluster. *IEEE Computer Society International Workshop on Cluster Computing*, pages 9–18, December 1999.

[9] G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, 1997.

[10] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP. In *24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 pp. 390-400, Bologna, Italy, 1997. Springer Verlag Lecture Notes in Computer Science.

[11] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, January/February 1998.

[12] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *Informations Journal on Computing*, 11:138–148, 1999.

[13] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[14] P.B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM Journal on Computing*, 28(2):733–769, 1998.

[15] D.R. Helman, D.A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm With an Experimental Study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[16] D.R. Helman and J. JáJá. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. *Algorithm Engineering and Experimentation (ALENEX'99)*, pages 37–56, Baltimore, MD, January 1999.

[17] D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 3(4), 1997. `http://www.jea.acm.org/1998/HelmanSorting/`.

[18] L. Ibarra and D. Richards. Efficient parallel graph algorithms based on open ear decomposition. *Mathematical Foundations of Computer Sciences*, 711:506–516, August/September 1993.

[19] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[20] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):830–840, 1996.

[21] A. Kazmierczak and S. Radhakrishnan. An Optimal Distributed Ear Decomposition Algorithm with Applications to Biconnectivity and Outerplanarity Testing. *IEEE Transactions on Parallel and Distributed Systems*, 11(1):110–118, 2000.

[22] E.L Lawler. *Combinatorial Optimization : Networks and Matroids*. General Publishing company, Ltd., 1976.

[23] L. Lovász. Computing ears and branchings in parallel. *In Proc. 26th Annual IEEE Symp. on Foundations of Computer Science, Portland, Oregon*, pages 464–467, 1985.

[24] Y. Moan, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

[25] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufman, San Mateo, CA, 1991.

[26] E.M. Reingold and R.E. Tarjan. On a Greedy Heuristic for Complete Matching. *SIAM Journal of Computing*, 10(4):676–681, 1981.

[27] C. Savage and J.JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Computing*, 10(4):682–691, 1981.

[28] J.R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In M.C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[29] K. Steiglitz and C.H. Papadimitriou. *Combinatorial Optimization : Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[30] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[31] J. Vitter and E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12:110–147, 1994.

[32] H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932.