Using PRAM Algorithms on a Uniform-Memory-Access Shared-Memory Architecture

David A. Bader[1], Ajith K. Illendula[2], and Bernard M.E. Moret[3]

[1] Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA. dbader@eece.unm.edu. Supported in part by NSF ITR 00-81404, NSF DEB 99-10123, NSF CAREER 00-93039, and DOE CSRI-14968.
[2] Intel Corporation, Rio Rancho, NM 87124 USA. ajith.k.illendula@intel.com
[3] Department of Computer Science, University of New Mexico, Albuquerque, NM 87131 USA. moret@cs.unm.edu. Supported in part by NSF ITR 00-81404.

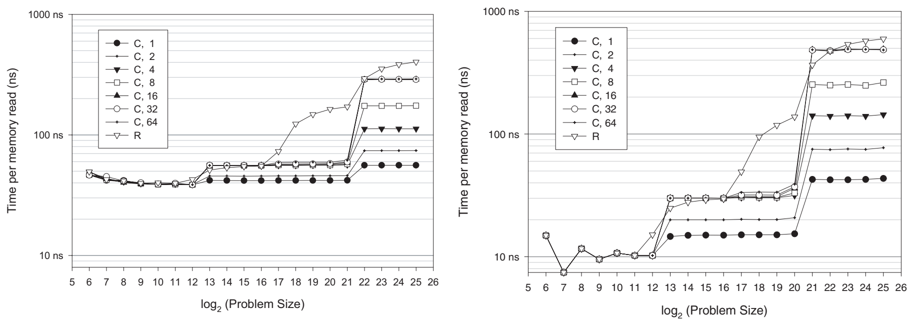
Abstract. The ability to provide uniform shared-memory access to a significant number of processors in a single SMP node brings us much closer to the ideal PRAM parallel computer. In this paper, we develop new techniques for designing a uniform shared-memory algorithm from a PRAM algorithm and present the results of an extensive experimental study demonstrating that the resulting programs scale nearly linearly across a significant range of processors (from 1 to 64) and across the entire range of instance sizes tested. This linear speedup with the number of processors is, to our knowledge, the first ever attained in practice for intricate combinatorial problems The example we present in detail here is a graph decomposition algorithm that also requires the computation of a spanning tree; this problem is not only of interest in its own right, but is representative of a large class of irregular combinatorial problems that have simple and efficient sequential implementations and optimal PRAM algorithms, but have no known efficient parallel implementations. Our results thus offer promise for bridging the gap between the theory and practice of shared-memory parallel algorithms.

1 Introduction

Symmetric multiprocessor (SMP) architectures, in which several processors operate in a true, hardware-based, shared-memory environment and are packaged as a single machine, are becoming commonplace. Indeed, most of the new high-performance computers are clusters of SMPs, with from 2 to 64 processors per node. The ability to provide uniform-memory-access (UMA) shared-memory for a significant number of processors brings us much closer to the ideal parallel computer envisioned over 20 years ago by theoreticians, the *Parallel Random Access Machine (PRAM)* (see [19, 38]) and thus may enable us at long last to take advantage of 20 years of research in PRAM algorithms for various irregular computations. Moreover, as supercomputers increasingly use SMP clusters, SMP computations will play a significant role in supercomputing. For instance, much attention has been devoted lately to OpenMP [32], which provides compiler directives and runtime support to reveal algorithmic concurrency and thus take advantage of the SMP architecture; and to mixed-mode programming, which combines message-passing style between cluster nodes (using MPI [28]) and shared-memory style within each SMP (using OpenMP or POSIX threads [33]).

While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work—synchronization cannot be taken for granted and the number of processors is far smaller than that assumed in PRAM algorithms. The significant feature of SMPs is that they provide much faster access to their shared-memory than an equivalent message-based architecture. Even the largest SMP to date, the 64-processor Sun Enterprise 10000 (E10K), has a memory access time of 600ns (from any processor to any location within its 64GB memory); in contrast, the latency for access to the memory of another processor in a message-based architecture is measured in tens of μs. In other words, message-based architectures are two orders of magnitude slower than the largest SMPs in terms of their worst-case memory access times.

The largest SMP architecture to date, the Sun E10K [7], uses a combination of data crossbar switches, multiple snooping buses, and sophisticated cache handling to achieve UMA across the entire memory. Of course, there remains a large difference between the access time for an element in the local processor cache (around 15ns) and that for an element that must be obtained from memory (at most 600ns)—and that difference increases as the number of processors increases, so that cache-aware implementations are, if anything, even more important on large SMPs than on single workstations. Figure 1 illustrates the memory access behavior of the Sun E10K (right) and its smaller sibling,



Sun Enterprise 4500 Sun Enterprise 10000

Fig. 1. Memory access (read) time using one MHz processor of a Sun E4500 (left) and an E10K (right) as a function of array size for various strides.

the E4500 (left), using a single processor to visit each node in a circular array. We chose patterns of access with a fixed stride, in powers of 2 (labelled `C, stride`), as well as a random access pattern (labelled `R`). The data clearly show the effect of addressing outside the on-chip cache (the first break, at a problem size of 2^{13} words, or 32KB) and then outside the L2 cache (the second break, at a problem size of 2^{21} words, or 8MB). The uniformity of access times was impressive—standard deviations around our reported means are well below 10 percent. Such architectures make it possible to design algorithms targeted specifically at SMPs.

In this paper, we present promising results for writing efficient implementations of PRAM-based parallel algorithms for UMA shared-memory machines. As an example of our methodology, we look at ear decomposition in a graph and show that our implementation achieves near linear speedups, as illustrated in Figure 2. Our main contributions are

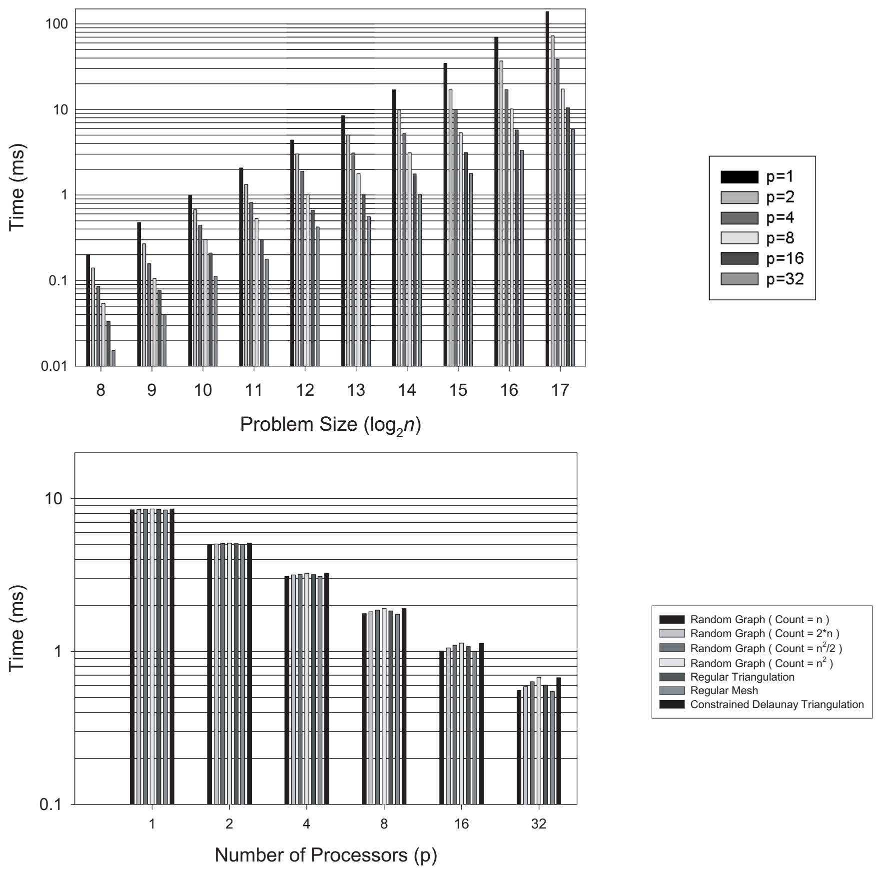


Fig. 2. Running times of ear decomposition on the NPACI Sun E10K with 1 to 32 processors on varying problem sizes (top) and different sparse graph models (bottom).

1. A new methodology for designing practical shared-memory algorithms on UMA shared-memory machines,
2. A fast and scalable shared-memory implementation of ear decomposition (including spanning tree construction) demonstrating the first ever significant—in our case, nearly optimal—parallel speedup for this class of problems.
3. An example of experimental performance analysis for a nontrivial parallel implementation.

2 Related Work

Several groups have conducted experimental studies of graph algorithms on parallel architectures [36, 37, 41, 23]. Their approach to producing a parallel program is similar to ours (especially that of Ramachandran *et al.* [16]), but their test platforms have not provided them with a true, scalable, UMA shared-memory environment or have relied on *ad hoc* hardware [23]. Thus ours is the first study of speedup over a significant range of processors on a commercially available platform.

3 Methodology

3.1 Approach

Our methodology has two principal components: an approach to the conversion of PRAM algorithms into parallel programs for shared-memory machines and a matching complexity model for the prediction of performance. In addition, we make use of the best precepts of algorithm engineering [31] to ensure that our implementations are as efficient as possible.

Converting a PRAM algorithm to a parallel program requires us to address three problems: (i) how to partition the tasks (and data) among the very limited number of processors available; (ii) how to optimize locally as well as globally the use of caches; and (iii) how to minimize the work spent in synchronization (barrier calls). The first two are closely related: good data and task partitioning will ensure good locality; coupling such partitioning with cache-sensitive coding (see [25, 26, 24, 31] for discussions) provides programs that take best advantage of the architecture. Minimizing the work done in synchronization barriers is a fairly simple exercise in program analysis, but turns out to be far more difficult in practice: a tree-based gather-and-broadcast barrier, for instance, will guarantee synchronization of processes at fairly minimal cost (and can often be split when only one processor should remain active), but may not properly synchronize the caches of the various processors on all architectures, while a more onerous barrier that forces the processors caches to be flushed and resynchronized will be portable across all architectures, but unnecessarily expensive on most. We solve the problem by providing both a simple tree-based barrier and a heavy-weight barrier and placing in our libraries architecture-specific information that can replace the heavy barrier with the light-weight one whenever the architecture permits it.

3.2 Complexity model for shared-memory

Various cost models have been proposed for SMPs [1, 2, 3, 42, 5, 15, 17, 35]; we chose the Helman and JáJá model [17] because it gave us the best match between our analyses and our experimental results. Since the number of processors used in our experiments is relatively small (not exceeding 64), contention at the memory location is negligible compared to the contention at the processors. Other models that take into account only the number of reads/writes by a processor or the contention at the memory are thus unsuitable here. Performance on most computer systems dictates the reliance on several levels of memory caches, and thus, cost benefit should be given to an algorithm that optimizes for contiguous memory accesses over non-contiguous access patterns. The Queuing Shared Memory [35, 14] model takes into account both the number of memory accesses and contention at the memory, but does not distinguish between between contiguous versus non-contiguous accesses. In contrast, the complexity model of Helman and JáJá [17] takes into account contention at both the processors and the memory. In the Helman-JáJá model, we measure the overall complexity of an algorithm by the triplet (M_A, M_E, T_C), where M_A is the maximum number of (noncontiguous) 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 work of any of the processors. T_C is measured in standard asymptotic terms, while M_A and 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.

4 Example: Ear Decomposition

4.1 The Problem

Of the many basic PRAM algorithms we have implemented and tested, we chose ear decomposition to present in this study, for three reasons. First, although the speedup observed with our implementation o ear decomposition is no better than what we observed with our other implementations of basic PRAM algorithms, ear decomposition is more complex than such problems as prefix sum, pointer doubling, symmetry breaking, etc. Secondly, ear decomposition is typical of problems that have simple and efficient sequential solutions, have known optimal and efficient PRAM algorithms, but yet have no practical parallel implementation. One of its component tasks is finding a spanning tree, a task that was also part of the original DIMACS challenge on parallel implementation many years ago, in which sequential implementations had proved significantly faster than even massively parallel implementations (using a 65,536-processor CM2). Finally, ear decomposition is interesting in its own right, as it is used in a variety of applications from computational geometry [10, 22, 6, 21, 9], structural engineering [11, 12], to material physics and molecular biology [13].

The efficient parallel solution of many computational problems often requires approaches that depart completely from those used for sequential solutions. In the area of graph algorithms, for instance, depth-first search is the basis of many efficient algorithms, but no efficient PRAM algorithm is known for depth-first search (which is P-complete). To compensate for the lack of such methods, the technique of *ear decomposition*, which does have an efficient PRAM algorithm, is often used [22].

Let $G = (V, E)$ be a connected, undirected graph; set $n = |V|$ and $m = |E|$. An *ear decomposition* of G is a partition of E into an ordered collection of simple paths (called *ears*), $Q_0, Q_1, \ldots, Q_r$, obeying the following properties:

- Each endpoint of Q_i, $i > 0$, is contained in some Q_j, $j < i$.
- No internal vertex of $Q_i, (i > 0)$, is contained in any Q_j, for $j < i$.

Thus a vertex may belong to more than on ear, but an edge is contained in exactly one ear [18]. If the endpoints of the ear do not coincide, then the ear is *open*; otherwise, the ear is *closed*. An *open ear decomposition* is an ear decomposition in which every ear is open. Figure 4 in the appendix illustrates these concepts. Whitney first studied open ear decomposition and showed that a graph has an open ear decomposition if and only if it is biconnected [43]. Lovász showed that the problem of computing an open ear decomposition in parallel is in NC [27]. Ear decomposition has also been used in designing efficient sequential and parallel algorithms for triconnectivity [34] and 4-connectivity [20]. In addition to graph connectivity, ear decomposition has been used in graph embeddings (see [8]).

The sequential algorithm: Ramachandran [34] gave a linear-time algorithm for ear decomposition based on depth-first search. Another sequential algorithm that lends itself to parallelization (see [19, 30, 39, 34]) finds the labels for each edge as follows. First, a spanning tree is found for the graph; the tree is then arbitrarily rooted and each vertex is assigned a level and parent. Each non-tree edge corresponds to a distinct ear, since an arbitrary spanning tree and the non-tree edges form a cycle basis for the input graph. Each non-tree edge is then examined and uniquely labeled using the level of the least common ancestor (LCA) of the edge's endpoints and a unique serial number for that edge. Finally, the tree edges are assigned ear labels by choosing the smallest label of any non-tree edge whose cycle contains it. This algorithm runs in $\mathrm{O}((m + n) \log n)$ time.

4.2 The PRAM Algorithm

The PRAM algorithm for ear decomposition [30, 29] is based on the second sequential algorithm. The first step computes a spanning tree in $\mathrm{O}(\log n)$ time, using $\mathrm{O}(n + m)$ processors. The tree can then be rooted and levels and parents assigned to nodes by using the Euler-tour technique. Labelling the nontree edges uses an LCA algorithm, which runs within the same asymptotic bounds as the first step. Next, the labels of the tree edges can be found as follows. Denote the graph by G, the spanning tree by T, the parent of a vertex v by $p(v)$, and the label of an edge (u, v) by label(u, v). For each vertex v, set $f(v) = \min\{\text{label}(v, u) \mid (v, u) \in G - T\}$ and $g = (x, y) \in T$, where we have $y = p(x)$. label(g) is the minimum value in the subtree rooted at x. These two substeps can be executed in $\mathrm{O}(\log n)$ time. Finally, labelling the ears involves sorting the edges by their labels, which can be done in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n + m)$ processors. Therefore the total running time of this CREW PRAM algorithm is $\mathrm{O}(\log n)$ using $\mathrm{O}(n + m)$ processors. This is not an optimal PRAM algorithm (in the sense of [19]), because the work (processor-time product) is asymptotically greater than the sequential complexity; however, no known optimal parallel approaches are known.

In the spanning tree part of the algorithm, the vertices of the input graph G (held in shared memory) are initially assigned evenly among the processors—although the entire input graph G is of course available to every processor through the shared memory. Let D be the function on the vertex set V defining a pseudoforest (a collection of trees). Initially each vertex is in its own tree, so we set $D(v) = v$ for each $v \in V$. The algorithm manipulates the pseudoforest through two operations.

- **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 $D(r_i) \leftarrow v$ 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 $D(v) \leftarrow D(D(v))$.

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. If all vertices then reside in rooted stars, the algorithm stops. Otherwise pointer jumping is applied at every vertex, reducing the diameter of each tree, and the algorithm loops back to the first step. Figure 5 in the appendix shows the grafting and pointer-jumping operations performed on an example input graph. The ear decomposition algorithm first computes the spanning tree, then labels non-tree edges using independent LCA searches, and finally labels tree edges in parallel.

4.3 SMP Implementation and Analysis

Spanning Tree: Grafting subtrees can be carried out in $\mathrm{O}(m/p)$ time, with two noncontiguous memory accesses to exchange approximately $\frac{n}{p}$ elements; grafting rooted stars onto other trees takes $\mathrm{O}(m/p)$ time; pointer-jumping on all vertices takes $\mathrm{O}(n/p)$ time; and the three steps are repeated $\mathrm{O}(\log n)$ times. Note that the memory is accessed only once even though there are up to $\log n$ iterations. Hence the total running time of the algorithm is

$$T(n,p) = \mathrm{O}\Big(1, \tfrac{n}{p}, ((m+n)/p)\log n\Big). \tag{1}$$

Ear Decomposition: Equation (1) gives us the running time for spanning tree formation. Computing the Euler tour takes time linear in the number of vertices per processor or $\mathrm{O}(n/p)$, with $\frac{n}{p}$ noncontiguous memory accesses to exchange $\frac{n}{p}$ elements. The level and parent of the local vertices can be found in $\mathrm{O}(n/p)$ time with $\frac{n}{p}$ noncontiguous memory accesses. Edge labels (for tree and nontree edges) can be found in $\mathrm{O}(n/p)$ time. The two labeling steps need $\frac{n}{p}$ noncontiguous memory accesses to exchange approximately $\frac{n}{p}$ elements. The total running time of the algorithm is $\mathrm{O}\Big(\frac{n}{p}, \frac{n}{p}, \frac{m+n}{p}\log n\Big)$. Since M_A and M_E are of the same order and since noncontiguous memory accesses are more expensive (due to cache faults), we can rewrite our running time as

$$T(n,p) = \mathrm{O}\Big(\tfrac{n}{p}, \tfrac{m+n}{p}\log n\Big). \tag{2}$$

This expression *decreases* with increasing p; moreover, the algorithm scales down efficiently as well, since we have $T(n,p) = \frac{T^*(n)}{p}$, where $T^*(n)$ is the sequential complexity of ear decomposition, $\mathrm{O}((m+n)\log n)$.

5 Experimental Results

We study the performance of our implementations using a variety of input graphs that represent classes typically seen in high-performance computing applications. Our goals are to confirm that the empirical performance of our algorithms is realistically modeled and to learn what makes a parallel shared-memory implementation efficient.

5.1 Test Data

We generate graphs from seven input classes of planar graphs (2 regular and 5 irregular) that represent a diverse collection of realistic inputs. The first two classes are regular meshes (lattice *RL* and triangular *RT*); the next four classes are planar graphs generated through a simple random process, two very sparse (*GA* and *GB*) and two rather more dense (*GC* and *GD*)—since the graphs are planar, they cannot be dense in the usual sense of the word, but *GD* graphs are generally fully triangulated. The last graph class generates the constrained Delaunay triangulation (*CD*) on a set of random points [40]. For the random graphs *GA*, *GB*, *GC*, and *GD*, the input graph on $n = |V|$ vertices is generated as follows. Random coordinates are picked in the unit square according to a uniform distribution; a Euclidean minimum spanning tree (MST) on the n vertices is formed to ensure that the graph is connected and serves as the initial edge set of the graph. Then for *count* times, two vertices are selected at random and a straight edge is added between them if the new edge does not intersect with any existing edge. Note that a count of zero produces a tree, but that the expected number of edges is generally much less than the count used in the construction, since any crossing edges will be discarded. Table 1 summarizes the seven graph families. Figure 6 in the appendix shows

Table 1. Different classes of input graphs.

Key	Name	Description
RL	Regular Lattice	Regular 4-connected mesh of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ vertices
RT	Regular Triangulation	RL graph with an additional edge connecting a vertex to its down-and-right neighbor, if any
GA	Random Graph A	Planar, very sparse graph with $count = n$
GB	Random Graph B	Planar, very sparse graph with $count = 2n$
GC	Random Graph C	Planar graph with $count = n^2/2$
GD	Random Graph D	Planar graph with $count = n^2$
CD	Constrained Delaunay Triangulation	Constrained Delaunay triangulation of n random points in the unit square

some example graphs with various values of *count*. Note that, while we generate the test input graphs geometrically, only the vertex list and edge set of each graph are used in our experiments.

5.2 Test Platforms

We tested our shared-memory implementation on the NPACI Sun E10K at the San Diego Supercomputing Center. This machine has 64 processors and 64GB of shared memory, with 16KB of on-chip direct-mapped data cache and 8MB of external cache for each processor [7].

Our practical programming environment for SMPs is based upon the SMP Node Library component of SIMPLE [4], 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.

5.3 Experimental Results

Due to space limitations, we present only a few graphs illustrating our results and omit a discussion of our timing methodology. (We used actual execution times, but such are not trivial to obtain reliably for parallel executions; we cross-checked our timings and ran sufficient tests to verify that our measurements did not suffer from any significant variance.) Figure 2 shows the execution time of the SMP algorithms and demonstrates that the practical performance of the SMP approach is nearly invariant with respect to the input graph class.

The analysis for the shared-memory algorithm given in Equation (2) shows that a practical parallel algorithm is possible. We experimented with the SMP implementation on problems ranging in size from 256 to 128,000 vertices on the Sun E10K using $p = 1$ to 32 processors. Clearly, the linear speedup with the number of processors predicted by Equation (2) may not be achievable due to synchronization overhead, serial work, or contention for shared resources. In fact, our experimental results, plotted in Fig-

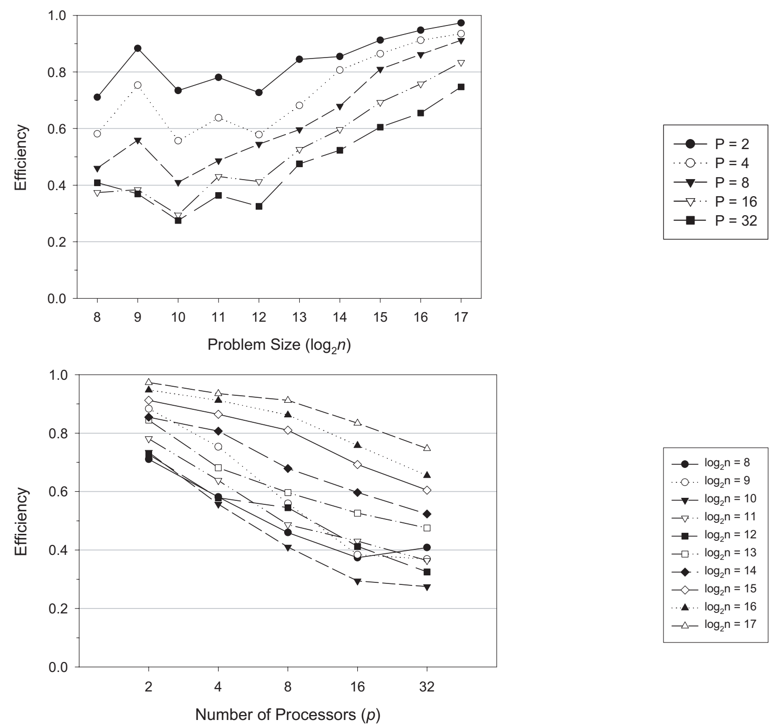


Fig. 3. Efficiency of ear decomposition on the NPACI Sun E10K for a sparse random graph with 256 to 128K vertices with 1 to 32 processors.

ure 2, confirm the cost analysis and provide strong evidence that our shared-memory algorithm achieves nearly linear speedup with the number of processors for each fixed problem size. Figure 3 presents the efficiency of our implementation when compared with the sequential algorithm for ear decomposition. For a fixed problem size, as expected, the efficiency decreases as we add more processors—caused by increasing parallel overheads. Another viewpoint is that for a fixed number of processors, efficiency increases with the problem size. Note that the efficiency results confirm that our implementation scales nearly linearly with the number of processors and that, as expected, larger problem sizes show better scalability.

6 Conclusion

We have implemented and analyzed a PRAM algorithm for the ear decomposition problem. We have shown both theoretically and practically that our shared-memory approach to parallel computation is efficient and scalable on a variety of input classes and problem sizes. In particular, we have demonstrated the first ever near-linear speedup for a nontrivial graph problem. As our example shows, PRAM algorithms that once were of mostly theoretical interest, now provide plausible approaches for real implementations on UMA shared-memory architectures such as the Sun E10K. Our future work includes determining what algorithms can be efficiently parallelized in this manner on these architectures, building a basic library of efficient implementations, and using them to tackle difficult optimization problems in computational biology.

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 (STOC)*, pages 305–314, New York City, 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] B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12:72–109, 1994.

[4] 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.

[5] 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.

[6] M. H. Carvalho, C. L. Lucchesi, and U. S.R. Murty. Ear Decompositions of Matching Covered Graphs. *Combinatorica*, 19(2):151–174, 1999.

[7] A. Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

[8] J. Chen and S. P. Kanchi. Graph Ear Decompositions and Graph Embeddings. *SIAM Journal on Discrete Mathematics*, 12(2):229–242, 1999.

[9] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. LEONARDO: A Software Visualization System. In *Proceedings of the First Workshop on Algorithm Engineering (WAE'97)*, pages 146–155, Venice, Italy, sep 1997.

[10] D. Eppstein. Parallel Recognition of Series Parallel Graphs. *Information & Computation*, 98:41–55, 1992.

[11] D. S. Franzblau. Combinatorial Algorithm for a Lower Bound on Frame Rigidity. *SIAM Journal on Discrete Mathematics*, 8(3):388–400, 1995.

[12] D. S. Franzblau. Ear Decomposition with Bounds on Ear Length. *Information Processing Letters*, 70(5):245–249, 1999.

[13] D. S. Franzblau. Generic Rigidity of Molecular Graphs Via Ear Decomposition. *Discrete Applied Mathematics*, 101(1-3):131–155, 2000.

[14] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *Proceedings of the 9th annual ACM symposium on parallel algorithms and architectures*, pages 72–83, Newport, RI, June 1997.

[15] 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.

[16] B. Grayson, M. Dahlin, and V. Ramachandran. Experimental evaluation of QSM, a simple shared-memory model. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 1–7, San Juan, Puerto Rico, April 1999.

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

[18] L. Ibarra and D. Richards. Efficient Parallel Graph Algorithms Based on Open Ear Decomposition. *Parallel Computing*, 19(8):873–886, 1993.

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

[20] A. Kanevsky and V. Ramachandran. Improved Algorithms for Graph Four-Connectivity. *Journal of Computer and System Sciences*, 42(3):288–306, 1991.

[21] D. J. Kavvadias, G. E. Pantziou, P. G. Spirakis, and C. D. Zaroliagis. Hammock-On-Ears Decomposition: A Technique for the Efficient Parallel Solution of Shortest Paths and Other Problems. *Theoretical Computer Science*, 168(1):121–154, 1996.

[22] 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.

[23] J. Keller, C. W. Keßler, and J. L. Träff. *Practical PRAM Programming*. John Wiley & Sons, 2001.

[24] R. Ladner, J.D. Fix, and A. LaMarca. The cache performance of traversals and random accesses. In *Proc. 10th Ann. ACM/SIAM Symposium on Discrete Algorithms (SODA-99)*, pages 613–622, Baltimore, MD, 1999.

[25] A. LaMarca and R.E. Ladner. The Influence of Caches on the Performance of Heaps. *ACM Journal of Experimental Algorithmics*, 1(4), 1996. `www.jea.acm.org/1996/LaMarcaInfluence/`.

[26] A. LaMarca and R.E. Ladner. The Influence of Caches on the Performance of Heaps. In *Proceedings of the Eighth ACM/SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, LA, 1997.

[27] L. Lovász. Computing Ears and Branchings in Parallel. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS 85)*, pages 464–467, Portland, Oregon, October 1985.

[28] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.

[29] G. L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, UC Berkeley, MSRI, January 1986.

[30] 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.

[31] B.M.E. Moret. Towards a discipline of experimental algorithmics. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2001. To appear. Available at `www.cs.unm.edu/∼moret/dimacs.ps`.

[32] OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. `http://www.openmp.org/`, October 1997.

[33] Portable Applications Standards Committee of the IEEE. *Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API)*, 1996-07-12 edition, 1996. ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1.

[34] V. Ramachandran. Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 275–340. Morgan Kaufman, San Mateo, CA, 1993.

[35] V. Ramachandran. A General-Purpose Shared-Memory Model for Parallel Computation. In M. T. Heath, A. Ranade, and R. S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105, pages 1–18. Springer-Verlag, New York, 1999.

[36] M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 104–113, Cape May, NJ, June 1994.

[37] M. Reid-Miller. List ranking and list scan on the Cray C-90. *Journal of Computer and System Sciences*, 53(3):344–356, December 1996.

[38] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[39] C. Savage and J. JáJá. Fast, Efficient Parallel Algorithms for Some Graph Problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.

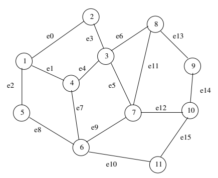
[40] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the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

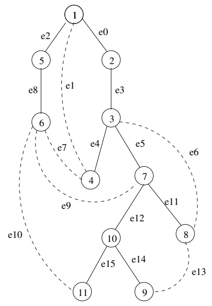
[41] J. Sibeyn. Better trade-offs for parallel list ranking. In *Proceedings of the 9th annual ACM symposium on parallel algorithms and architectures*, pages 221–230, Newport, RI, June 1997.

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

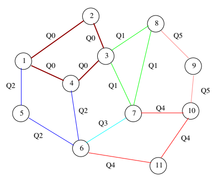
[43] H. Whitney. Non-Separable and Planar Graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932.

Appendix





edge	type	label
e0	non-tree	(0,1)
e1	tree	(0,1)
e2	tree	(0,7)
e3	tree	(0,1)
e4	tree	(0,1)
e5	tree	(2,6)
e6	non-tree	(2,6)
e7	non-tree	(0,7)
e8	tree	(0,7)
e9	non-tree	(0,9)
e10	non-tree	(0,10)
e11	tree	(2,6)
e12	tree	(0,10)
e13	non-tree	(7,13)
e14	tree	(7,13)
e15	tree	(0,10)



the original graph $G = (V, E)$ — *spanning tree of* G — *edge labels* — *ear decomposition of* G

Fig. 4. A graph, a spanning tree for that graph, edge classification and labels, and an ear decomposition for that graph.

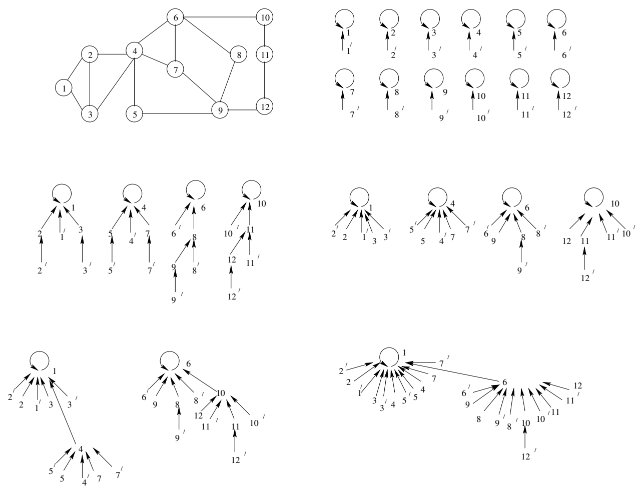
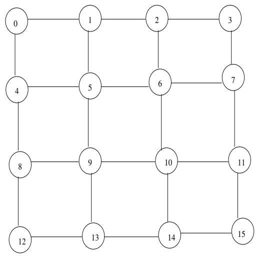
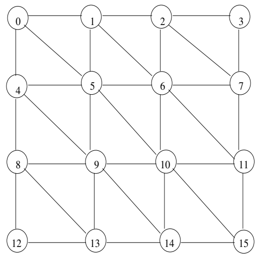


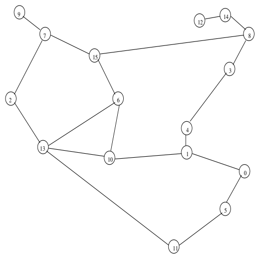
Fig. 5. Grafting and pointer-jumping operations applied to a sample graph (top left) and steps of the algorithm

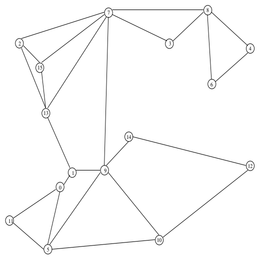




regular lattice **RL**

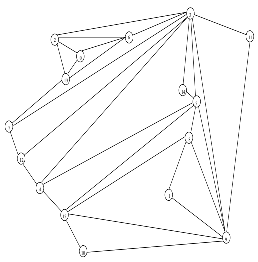
regular triangulation **RT**

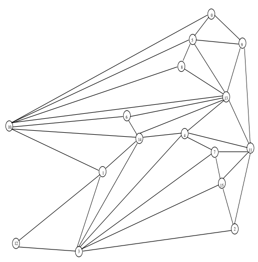




random sparse **GA**
($count = n$)

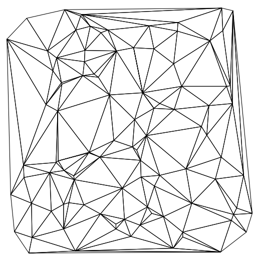
random sparse **GB**
($count = 2n$)

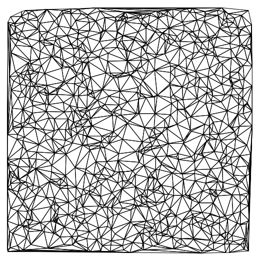




random **GC**
($count = n^2/2$)

random **GD**
($count = n^2$)





constrained Delaunay triangulation **CD**
($count = 100$)

constrained Delaunay triangulation **CD**
($count = 1000$)

Fig. 6. Examples of our seven planar graph families.