DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Implementation of Parallel Graph Algorithms on the MasPar (Preprint)†

TSAN-SHENG HSU, VIJAYA RAMACHANDRAN,

AND NATHANIEL DEAN

July 22, 1993

ABSTRACT. Graphs play an important role in modeling the underlying structure of many real world problems. Over the past couple of decades, efficient sequential algorithms have been developed for several graph problems and have been implemented on sequential machines. The NETPAD system at Bellcore is a general tool for graph manipulations and algorithm design that facilitates such implementations. More recently, several research results on efficient parallel algorithms have been developed, but not much implementation has been done.

We have implemented some of the parallel algorithms for basic graph problems on the massively parallel machine MasPar, and we have interfaced these algorithms with NETPAD. In this paper, we give a description of our implementation together with some performance data. We also describe the interface that we have built between our library of parallel graph algorithms and the NETPAD system.

1. Introduction

This paper summarizes a project we undertook at Bellcore during the summer of 1991 for implementing parallel graph algorithms. In this project, we imple-

1991 *Mathematics Subject Classification.* 68-04; Secondary 05-04, 05C85, 68Q22.

Key words and phrases. parallel algorithms, graph algorithms, implementation, MasPar.

The work reported here was performed while the first two authors were visiting Bellcore.

The first two authors were supported in part by NSF Grants CCR-89-10707 and CCR-90-23059 and Texas Advanced Research Projects Grant 003658480.

This paper is in final form and no version of it will be submitted for publication elsewhere.

†This paper appears in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, volume 15, 1994, pp. 165–198.

mented efficient parallel algorithms for solving several undirected graph problems using the massively parallel computer MasPar [**13**] at Bellcore. In addition, we also built an interface between the graph algorithm design package developed at Bellcore called NETPAD [**3, 14, 15, 16**] and our parallel programs. Our purpose was to experiment and set up programming environments for implementing parallel algorithms on massively parallel computers.

The organization of the paper is as follows. Section 2 gives an introduction to efficient parallel graph algorithms. Section 3 describes the machine architecture of the parallel computer MasPar that we used to implement our parallel algorithms, implementation strategies and an interface that we have built between NETPAD and our parallel programs. Section 4 describes implementation details of our parallel algorithms. Section 5 gives speed-up data of our parallel implementations and analyses of their performance. Finally, Section 6 gives conclusions and directions for further research.

2. Parallel Graph Algorithms

In designing sequential graph algorithms, depth-first search and breadth-first search have been used as basic search strategies for solving various graph problems [**22**]. Unfortunately, at present no efficient parallel algorithms are known for these two search techniques [**7**]. Hence we are unable to obtain efficient parallel algorithms by parallelizing sequential algorithms based on depth-first search or breadth-first search. Instead, an alternative search technique called ear decomposition has proved to be a very useful tool for designing parallel graph algorithms [**7, 6, 10, 17, 20, 19**]. Combined with an efficient parallel routine for finding connected components [**1**] and the Euler tour technique [**24**], we have efficient parallel algorithms for several important graph problems which include various connectivity problems [**6, 17, 19**], st-numbering [**10**], planarity testing and embedding [**20**], finding a strong orientation and finding a minimum cost spanning forest[‡]. Figure 1 illustrates the building blocks for designing parallel graph algorithms using ear decomposition, the Euler tour technique and the routine for finding connected components.

Our parallel implementations followed this approach. We first built a kernel which consists of commonly used routines in parallel graph algorithms. Then we implemented efficient parallel graph algorithms developed on the PRAM model by calling routines in the kernel.

3. Implementation Environment

In this section we describe the environment in which we implemented several efficient PRAM graph algorithms. In Section 3.1, we first describe the architecture of the MasPar machine [**13**]. Section 3.2 discusses the general is-

[‡]In this paper, a spanning forest of a graph G is a maximal subgraph of G (w.r.t. the edges in G) that is a forest.

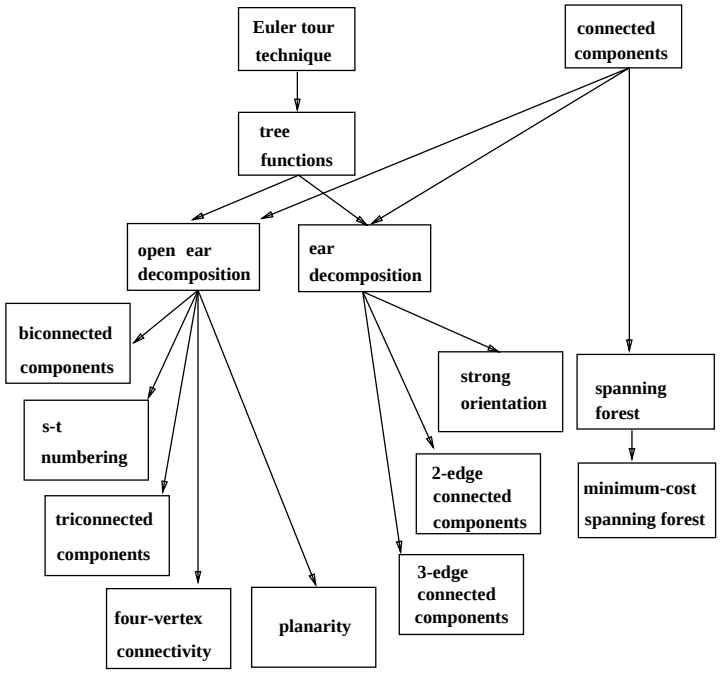

FIGURE 1. Parallel graph algorithms based on algorithms for connected components, the Euler tour technique and ear decomposition.

sues involved in implementing PRAM algorithms on the MasPar. Our parallel implementations are integrated with a graph algorithm design package called NETPAD [**14, 15, 16**]. Section 3.3 describes the NETPAD software and our interface between it and our parallel programs.

3.1. The Parallel Machine MasPar. The MasPar computer [**13**] is a fine-grained massively parallel single-instruction-multiple-data (SIMD) computer. All of its parallel processors synchronously execute the same instruction at the same time. A simplified version of its architecture is shown in Figure 2.

The MasPar has a front end processor running the Unix operating system [**21**] and a Data Parallel Unit (DPU) for execution of parallel programs. The front end machine is a micro-VAX workstation. The DPU consists of an Array Control Unit (ACU) and 16384 Processor Elements (PE's). The ACU is a special purpose processor for controlling the execution of all of the PE's. Programs are stored in one special local memory bank of ACU and broadcast to each PE simultaneously. The architecture of MasPar allows very efficient broadcasting operation from the ACU to all PE's. Since the ACU is about 10 times faster than each individual PE [**13**], the other purpose of the ACU is to perform simple local computations and broadcast the results to all PE's. Since the ACU is a special processor that is designed for load and save operations, it might not be very efficient to do complex arithmetic operations on it. For performing global arithmetic operations, it appears that the preferred method is to perform the computations on the front end processor and have the front end transfer the results back to the ACU.

All of the PE's are organized as a two-dimensional matrix. In Figure 2, $nxproc = 128$ and $nproc = 16384$ for our machine, but these numbers may be different at other installations. Each PE consists of a special processor and a bank of local memory (about 64 K bytes in our system). Upon receiving an instruction from the ACU, the processor will execute the instruction on its own local data. Each PE is connected to its 8 neighbors (toroidal wrapped) in a connection scheme called XNET. All PE's are also connected via a global router. Local data inside each PE can be exchanged through the global router as well as the local XNET. The XNET configuration is faster, but requires all PE's getting their needed data from the same direction. For transmitting a 32-bit data through distance 1, the XNET communication is slightly less than 100 times faster than the time needed to go through the global router [**12**]. During the computation, it might be necessary for the PE's to access the local memory of the ACU (about 1 M bytes). Such I/O requests are sequentialized and carried out one at a time. This process is very time-consuming.

The MasPar system provides a fast way of transferring data between the local memory of the front end and the local memory of the ACU. It also provides a fast way of transferring data between the local memory of the front end and the local memory banks of the PE's. In order to perform the latter transfer, the set

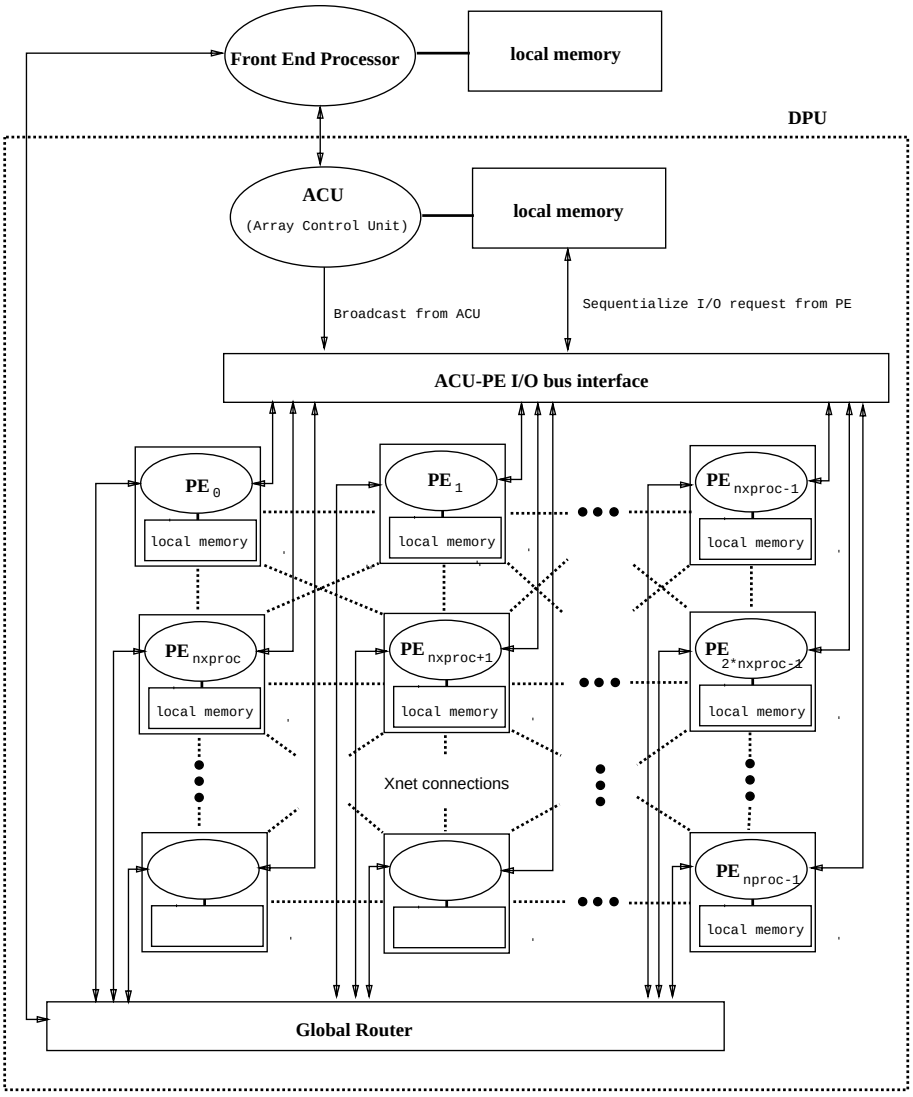

FIGURE 2. System architecture for the MasPar computer.

of PE's to receive data from the front end must form a rectangular block. More details are described in Section 3.3.

We used the MasPar Parallel Language (MPL) [**11, 12**] to implement our algorithms. MPL is an extension of the C language described in [**8**]. In addition to all of the standard C language features, it allows the user program a set of PE's to execute the same instruction on their own local data. MPL leaves the responsibility of processor allocation to the programmer. During the execution of the MPL program, the programmer must specify the set of PE's that are to execute the current instruction. MPL also allows the user to instruct the ACU to do local computations. MPL takes two kinds of input files. The first is a program file with a suffix ".c" which indicates that it is a pure C language program. MPL compiles this program and generates executable code that can be run on the front end. The second file is a program file with a suffix ".m" which indicates that it is a C language program that includes extended features for doing operations on the PE's. MPL compiles the program and generates executable code that can be run on the DPU. In the ".m" program, one can allocate local data on each PE by adding the keyword *plural* to a C data declaration statement. Variables declared without the *plural* keyword are allocated on the ACU. Any expression that involves a plural variable is computed on each active PE. An expression that contains no plural variables is computed on the ACU.

Using the MPL programming language, each PE can perform operations on 32-bit words and also on 64-bit words. (A 64-bit word can be declared by using the MPL command *long long*.)

3.2. Implementation Strategies for PRAM Algorithms. In this section, we describe the PRAM model and the strategies we used to map algorithms designed on a PRAM onto the MasPar architecture.

PRAM Models

A PRAM machine [**7**] consists of a pool of random access machines (RAM) and a global memory. Each random access machine has a processor with a reasonable set of instructions and a local memory. Each RAM has an unique ID numbered from 1 to the number of processors in the PRAM machine. During each step of the computation, each processor synchronously executes the same instruction, but with possibly different operands. During each time cycle, a process can read a global memory cell, perform some local computations on its local data and write data into a global memory cell. A schematic diagram of a PRAM is shown in Figure 3.

The PRAM model assumes that each instruction takes constant time no matter how many processors want to access the global memory. Depending on the type of global memory access allowed, the PRAM model can be further classified into the following models: EREW PRAM, CREW PRAM and CRCW PRAM. The EREW (exclusive read exclusive write) PRAM model requires that no two processors read or write into the same global memory location at any

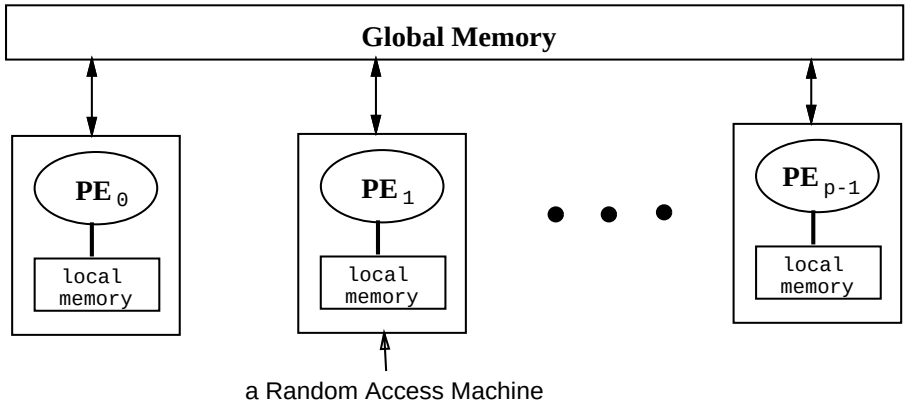

FIGURE 3. Schematic diagram for the PRAM model.

given time. The CREW (concurrent read exclusive write) PRAM model allows concurrent access to the same global memory location by more than one processor for reading, but disallows more than one processor write into the same memory location at the same time. CRCW (concurrent read concurrent write) PRAM models allow different processors to read and write into the same global memory location at the same time. In the case of writing different data into the same global memory location at a given time, we must define the result of the concurrent write. The COMMON CRCW model requires data that are written into a common global memory location by different processors to be the same at any given time. In the PRIORITY CRCW model, if several processors try to write different data in the same global memory location at any given time, the data sent by the processor with the least ID is written into the memory location. There are various other CRCW models. For details, see [**7**].

Mapping of the PRAM Model onto the MasPar Architecture

We map part of the local memory in each PE and the local memory of the ACU onto the PRAM global memory. The major difference between the PRAM model and the MasPar architecture is on the issue of global memory access. The MasPar allows constant time broadcasting of a constant amount of local memory contents from the ACU to all PE's. However, it takes $O(P)$ time for P PE's to access the local memory of the ACU. Hence it is not efficient to map the global memory in the PRAM model to the local memory of the ACU. In addition to the problem of efficient memory access, the size of the local memory of the ACU is not large enough to put all the global data we need. Instead, we partition the local memory bank of each PE into two halves. One of them, which we call *the global data bank of each PE*, is mapped onto part of the global memory bank and the other half, which is called *the local data bank of each PE*, is used for storing local data for local computations. The entire local memory of the ACU will be part of the global memory of the PRAM model. When implementing a

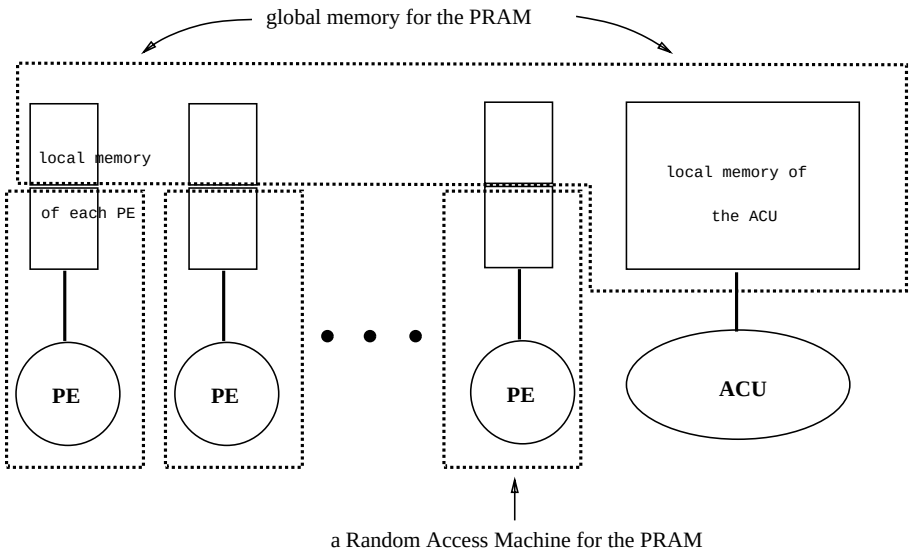

FIGURE 4. Mapping from the MasPar architecture to the PRAM model.

PRAM algorithm on the MasPar architecture, we put information that is most frequently used by a certain RAM into the global data bank of that particular PE. We put common data used by all the PE's into the local memory bank of the ACU and arrange for the ACU to broadcast the needed data to all PE's. We illustrate the mapping in Figure 4.

Mapping Efficient PRAM Algorithms for Graph Problems

Since the MPL programming language requires that the user takes care of the processor allocation problem, we use the simple strategy of allocating one PE for a node and an edge in our implementation. Hence our implementation can only handle graphs of size less than or equal to the number of PE's in the MasPar. We place data generated for each node or each edge into the global data bank of the PE that is in charge of the node or the edge. Global variables (for example, the total number of nodes and the total number of edges) are put into the local memory of the ACU. Each PE can access its global data bank efficiently under the mapping; however, global memory accesses to global data banks of other PE's will require going through the global router to get the data. In Section 3.1, we mentioned that a faster way of getting data from the other PE is by going through the XNET configuration. We can use the XNET configuration if PE_i wants to read the global data bank of PE_{i+c}, for all processor elements PE_i, where c is a constant. Some of the global memory accesses required by the PRAM algorithm fall into this category as in the case when each PE wants to obtain the data from the PE with an ID that is one greater than itself. In our implementation, we always try to take advantage of the XNET configuration whenever possible.

List Ranking on the MasPar

One problem that we often face in mapping PRAM graph algorithms onto the MasPar architecture comes from the fact that we usually define the graph using the edge list data structure (an edge list of an undirected graph G is a list of all the edges in G). The PRAM algorithms often link all the edges or all the nodes in a special ordered linked list and perform list processing computations such as list ranking [**7**] (a *list ranking* on a linked list requires each element in the list to compute the suffix sum of all the elements in front of it; the sum could be any associative operator). The list ranking problem on a list of length n can be solved by a sequence of $O(\log n)$ global memory accesses on a PRAM. These global memory accesses can be implemented only as requests to the global router, since elements in a list are not structurally allocated such that we can use the XNET configuration.

An operation on an array of elements called *prefix sums* [**9**] (or *scan* [**2, 11**]) is to compute the prefix sum of all the elements before each array element; the prefix sum operator can be any associative operator. This computation is similar to list ranking, except that the input is in an array rather than a linked list. The prefix sums problem can be solved by a sequence of $O(\log n)$ global memory access on a PRAM. In implementing the PRAM prefix sums algorithm on the MasPar, if we put the ith element of the array into the ith PE, then global memory accesses can be structured in a way that we can make use of the XNET connection. There is already such a routine called *scan* which is implemented in MasPar system library [**11**]. The scan operation is very fast compared to the list ranking algorithm we implemented. Note that a linked list can be converted into an array by first performing a list ranking computation and then rearranging the list elements into an array using a global memory write. Since the list ranking operation is one of the most commonly used subroutines in PRAM undirected graph algorithms, we convert a linked list to an array if several list ranking operations are to be performed on the list. Figure 5 compares the relative speed-up of the list ranking program and the scan (prefix sums) operation.

Concurrent Global Memory Access on the MasPar

Some of the PRAM algorithms that we have implemented are based on concurrent read and/or concurrent write PRAM models. If a PRAM global memory concurrent read request is sent to the global router, the global router will automatically satisfy all concurrent read requests. To understand the behavior of the concurrent read operation executed by the global router, we implemented a routine to transform concurrent read requests to exclusive read requests using the standard simulation algorithm (see, e.g., [**7**]). We found that the performance of our parallel algorithms when using the global router and using our simulated concurrent read routines is very similar.

For concurrent write, the global router does not allow more than one PE to write into the global memory bank of any particular PE. It also does not allow any concurrent write into the local memory bank of the ACU. However, the

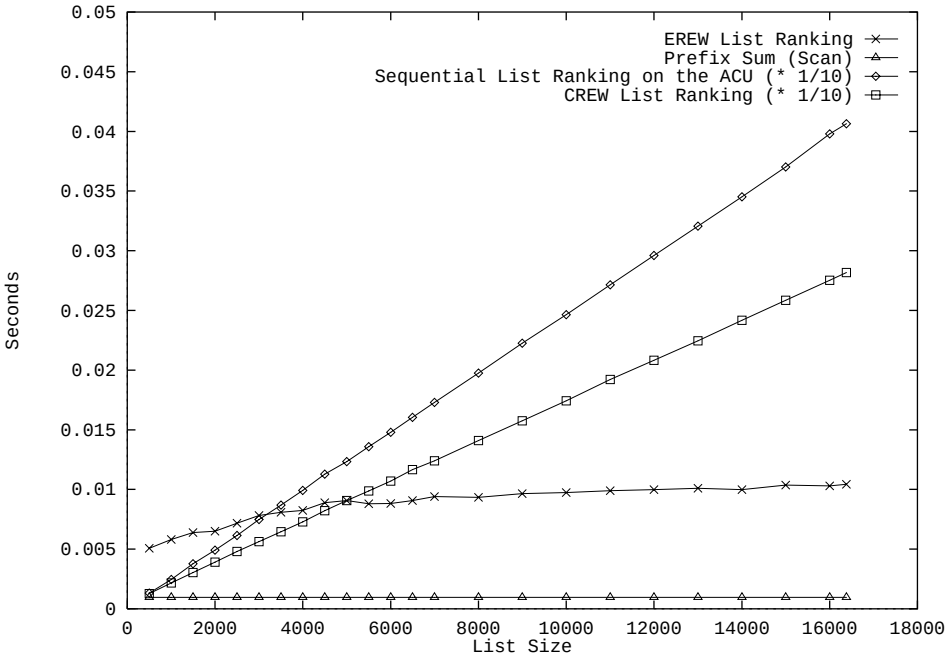

FIGURE 5. Relative performances of sequential list ranking on the ACU, parallel CREW list ranking, parallel EREW list ranking and the prefix sums (scan) operation. In order to fit all data on the same chart, the data for sequential list ranking and CREW list ranking has been divided by 10. The ACU is about 10 times more powerful than each individual PE. Thus on an input list of size 16384, prefix sums on an array gains a speed-up factor of about 4000, while EREW list ranking has a speed-up factor of about 380. The prefix sums operation is more than 10 times faster than the EREW list ranking.

system library routine *sendwith* [**11**] can be used to implement concurrent write. The *sendwith* routine allows each PE to send data to the global memory bank of any PE. If there is more than one PE trying to send data to any one PE at a given time, all of these data are collected and an associative operator specified by the routine is performed to compute the result which is then given to the destination PE. For example, a *sendwithMax32(item,destination)* command tells each active PE to send the 32-bit variable *item* in its own local memory to the PE with the ID equal to its local variable *destination*. If there are several PE's trying to write into any one PE, the result is the maximum value of all variables sent. The *sendwith* operation uses a sorting routine, several non-conflict (EREW) global memory accesses and a scan operation. The *sendwith* operation can be performed in $O(\log P)$ time on a EREW PRAM model with P processors.

In our implementation of PRIORITY CRCW algorithms, we used the global router to satisfy concurrent read requests and the *sendwith* operations to implement concurrent write requests.

3.3. NETPAD Interface. The NETPAD software [**14, 15, 16**] is a general tool for graph manipulations and graph algorithm design. It uses the X-window system [**18**], and one can edit graphs and display them easily. The NETPAD system also provides a rich set of basic graph manipulation operations. By calling these operations in one's own program (preferably written in C [**8**]), users can implement their own graph algorithms easily.

NETPAD was used in the following ways to support the design of our parallel graph algorithm packages. It provided routines for generating test graphs. We also used it as a standard interface to input graphs generated by other packages. Most important of all, we used it to display the outputs of our parallel graph algorithms. Since NETPAD has been designed primarily for supporting sequential computations, it was necessary to build an interface between it and the parallel programs that we implemented on the MasPar. We describe the interface in the following paragraphs. Figure 6 gives a schematic diagram of this interface.

For each MasPar routine that we implemented, we wrote a NETPAD external program in C language (with a suffix ".c" in its filename) and included it in one of the NETPAD pop-out menus. While running NETPAD software in the front end, this external program would be invoked through the NETPAD system. The NETPAD system uses the Unix *fork* system call to create another process in the front end to run the external program. The graph in the current window is saved into a file and the name of that file is passed to the forked process. The external program first uses NETPAD system routines to retrieve the input graph from the input file. NETPAD system routines are also used to collect the edge list for the graph and store it in an array inside the local memory bank of the front end. (The external program that we wrote can also be executed independently from the NETPAD system. As long as we have a file that describes the input graph using the NETPAD format, we can run the external program under the

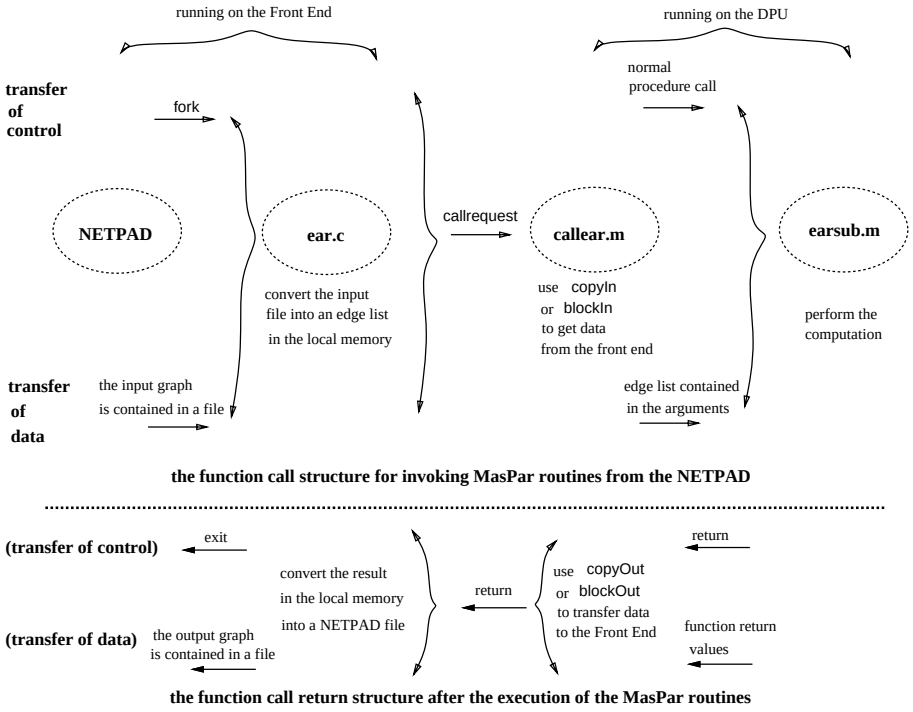

FIGURE 6. Interface for calling the MasPar routines for executing the ear decomposition algorithm from NETPAD.

Unix system without going through the NETPAD system.)

After the external program collects the edge list, we can request a MasPar MPL program (with a suffix ".m" in its filename) to be executed in the DPU using a special MasPar system call named *callrequest*. An interfacing MPL program must be written for each MasPar routine that we want to use. This routine is invoked by the external program running on the front end by using *callrequest* with arguments describing memory addresses (of the front end) that will be used by the DPU. The MPL program first uses the system routine *copyIn* to copy the contents of any consecutive block of memory in the front end to the local memory of the ACU. Then it uses the system routine *blockIn* to copy the contents of any consecutive block of memory in the front end to each PE. Routine *blockIn* takes two sets of parameters. The first set describes the starting location of the front end memory to be copied and the size of each memory cell. The second set of parameters describes the rectangular block of PE's that are to receive data. The routine puts the ith cell into the ith PE within the block. The MPL routine then calls the MasPar library that we have implemented to perform the actual computations for the parallel graph algorithms.

After completing the MasPar computations, the MPL interfacing program uses the system routine *copyOut* to copy any data in the local memory of the ACU to the local memory of the front end. A similar routine *blockOut* transfers data from a rectangular block of PE's to a consecutive block of memory locations in the front end. After the termination of the MPL interfacing routine, the external program running on the front end takes the output graph in its local memory and writes the output graph into a file using the NETPAD format. The external program exits and sends a signal to the NETPAD system. The NETPAD system gets the output file and displays the graph on the drawing window.

In Figure 6, we illustrate this interface by using an example of calling a MasPar routine to perform ear decomposition [**19**] from NETPAD.

4. Our Implementation of Parallel Graph Algorithms

In this section, we describe the parallel graph algorithms we have implemented. In Section 4.1, we describe the data structures used in implementing the algorithms. Then we give the structure of the library of programs we have implemented with a brief description of techniques used for fine tuning each program in Section 4.2.

4.1. Data Structures.

Array and Linked List

We map a global memory array used in a PRAM algorithm onto the MasPar by putting the ith element of the array into the ith PE. We map a linear linked list used in a PRAM algorithm by putting each element in the list into a different PE. Pointers in PRAM are replaced by the ID's of PE's.

Tree

We represent an edge in an undirected tree by two directed edges of opposite directions. A tree is represented by a list of directed edges. In implementing the tree data structure on the MasPar, we put one directed edge in one PE with the requirement that the set of edges that are incoming to the same vertex have to be allocated on a consecutive segment of PE's. Using this representation, we can use the XNET configuration to perform interprocess communications needed for computing an Euler tour on a tree. Since computing an Euler tour is one of the most commonly used routines for performing tree-based parallel computations, we save time by using this type of mapping.

Undirected Graph

A general undirected graph is also represented by a list of edges. Each edge has two copies with the two end points interchanged. On the MasPar, we put an edge on a PE with the requirement that the two copies of the edge have to be located on adjacent PE's.

Let the input graph contain n vertices and m edges. We number vertices from 1 to n. Since we have to take care of processor allocation when using the MPL programming language, we use the following method to allocate processors. The PE with ID equal to i takes care of computations for the ith vertex in the PRAM algorithm. Although the MasPar numbers PE's from 0, we do not use the PE with the ID 0 for convenience of programming. Each edge has two copies which have consecutive ID's. The PE with an ID equal to i takes care of computations for the ith edge in the PRAM algorithm. Under this scheme, our current implementation can handle graphs with $P-1$ vertices and $\lfloor \frac{P-1}{2} \rfloor$ edges where P is the number of PE's on the MasPar.

4.2. The Parallel Graph Algorithm Library. To build our parallel graph algorithm library, we first wrote a kernel that includes all of the commonly used subroutines for designing parallel graph algorithms. Then we built our graph application programs by calling routines in the kernel. The structure of the whole library is shown in Figure 7. We first describe routines in the kernel.

Routines in the Kernel

All of these routines are based on PRAM algorithms that run in $O(\log n)$ time using n processors for an input of size n. These are not optimal algorithms, but they are within an $O(\log n)$ factor of optimality, and they are very simple. Our implementations were only for input of size less than 16384, so the overall optimality was not a serious problem.

(i) **List ranking.** We implemented EREW PRAM list ranking routines that compute the rank of each element in a list, where each element in the list is stored in a different PE with a pointer that points to the ID of the PE that stores the next element in the list. These routines require $O(\log n)$ global memory accesses on a list of n elements.

(ii) **Rotation.** These routines rotate the data stored in PE with ID i to

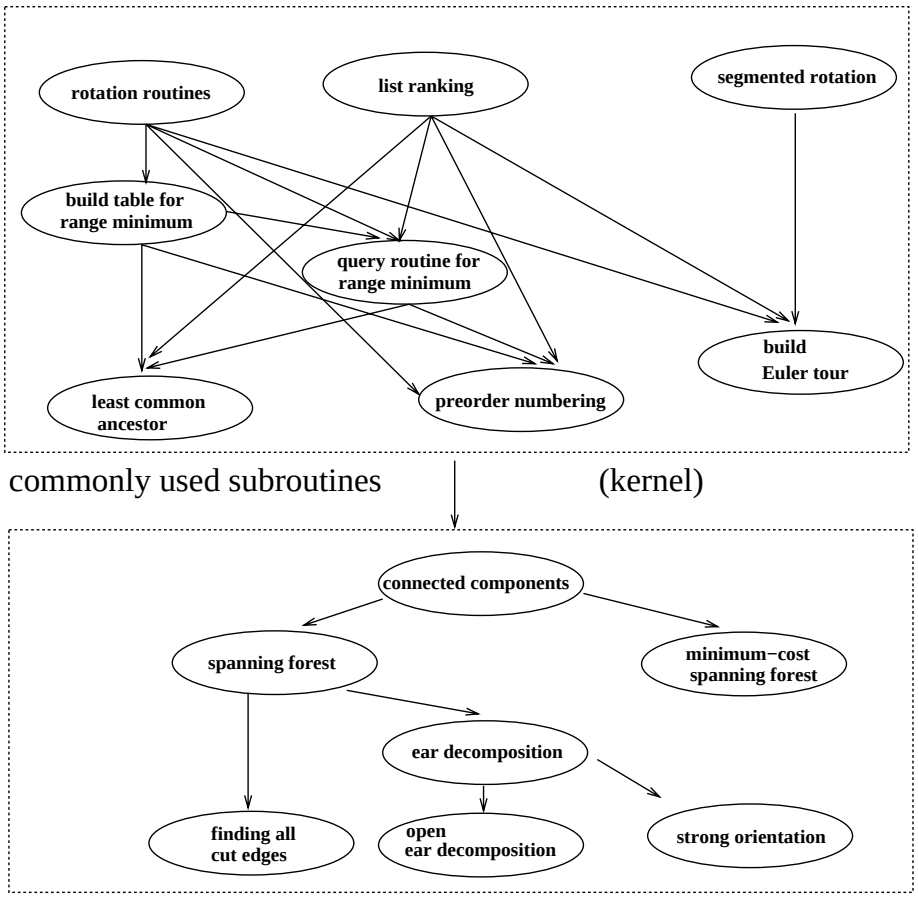

FIGURE 7. The structure of the routines we built for the parallel graph algorithm library: The kernel of the library will be used by the application routines. An arrow from one node to another node means the routine at the tail of the arrow (upper) will be used by the routine at the head of the arrow (lower).

the PE with ID $(i + d) \bmod P$, where d is a constant and P is the number of PE's in the system. The rotation routines make use of the mesh-connected XNET connection and are faster than implementing the same functions using the global router. These routines require 2 XNET communications.

(iii) **Segmented rotation.** We store data in each PE and partition PE's into sequences of consecutive segments. These routines rotate the data stored in each PE within each segment. Data within each segment are rotated in a way similar to the rotation routines described in (ii). These routines use a constant number of XNET communications and non-conflict (EREW) global memory accesses.

(iv) **Range minimum.** Let v be a local variable stored in each PE. We want to build a table such that given a starting ID s_i and an ending ID e_i in each PE, we can compute the minimum value of all v's from PE s_i to PE e_i using only one global memory access. For this, we implemented the $O(\log n)$-time $O(n)$-processor PRAM algorithm in [**24**] for solving the range minimum problem on n elements. In this algorithm, the n elements are stored in an array A. We are required to build a two-dimensional array $W[i, j]$, such that $W[i, j] = \min_{k=0}^{2^j-1} A[i + k]$, for all $0 \le j \le \lceil \log n \rceil$ and $1 \le i \le n-2^j+1$. We construct the two-dimensional array by storing all elements of $\{W[i, j] \mid 0 \le j \le \lceil \log n \rceil\}$ in the ith PE. Thus each PE has a one-dimensional local array. Because each global memory access is very structured in the algorithm for building the table, we can use XNET communications to implement it. Let k_i be the least integer such that 2^{k_i} is greater than or equal to $\frac{e_i - s_i + 1}{2}$. To process each query, a PE with ID i reads $W[s_i, k_i]$ and $W[e_i - 2^{k_i} + 1, k_i]$ through the global router and returns the minimum of the two values.

(v) **Euler tour construction** [**24**]. Given a tree represented by an adjacency list, we store each edge in the adjacency list in a different PE. Edges that are adjacent to a vertex are stored in consecutive PE's. By replacing each undirected edge between two vertices with two directed edges of opposite directions, the algorithm returns an Euler tour for the input tree by building a linear linked list. The routine uses the segmented rotation routines, list ranking routines, rotation routines and one global memory access.

(vi) **Preorder numbering.** Given a tree, we assign a consecutive preorder numbering for its vertices starting from 1. To implement this, we have to use list ranking and a constant number of global memory accesses.

(vii) **Least common ancestor.** This routine finds the preorder number of the least common ancestor for any given pairs of vertices in a rooted tree. Each PE stores two vertex ID's of the tree. For each PE, the routine returns the vertex which is the least common ancestor of the two vertices in the input rooted tree. The routine uses the range minimum queries

and a constant number of global memory accesses.

Graph Application Routines

We now describe the graph algorithms we implemented using the above kernel.

(i) **Connected components.** We implemented the CRCW PRAM algorithm described in [**1**]. The PRAM algorithm runs in $O(\log n)$ time using $O(n+m)$ processors on a graph with n vertices and m edges. The concurrent read operation was implemented by using the global router. The concurrent write operation needed in the algorithm was implemented by the *sendwith* operation. After the execution of the algorithm, the ith PE gets a number indicating the connected component containing the ith vertex. The component number is the vertex number of the vertex with the least vertex number in the connected component.

(ii) **Spanning forest.** We modified the algorithm in [**1**] for finding connected components to find a spanning forest of the input graph. The original algorithm partitions the set of vertices into a set of disjoint sets such that vertices in each set are in the same connected component. Initially, the algorithm puts a vertex in each set. During the execution, the algorithm merges two sets of vertices if they are in the same connected component. Our program selects an edge connecting a vertex in one set to a vertex in the other set while merging these two disjoint vertex sets.

(iii) **Minimum cost spanning forest.** Given the input graph with an integer weight assigned on each edge, we modify the algorithm in [**1**] for finding connected components to find a minimum cost spanning forest for the input graph. This algorithm also partitions the graph into disjoint sets of vertices. In addition, for each current set of vertices, we compute a minimum-cost edge with exactly one end point in the set using the *sendwith* operation. This edge determines which other set of vertices is to be merged with its set. Once the merger is completed, the edge that caused the merging is marked as one of the edges in the minimum cost spanning forest.

(iv) **Ear decomposition of a two-edge connected undirected graph.** We implemented the PRAM parallel algorithm in [**19**] for finding an ear decomposition by calling the MasPar system sorting routine, routines in the kernel and the routine for finding a spanning forest.

(v) **Strong orientation of a two-edge connected undirected graph.** We first find an ear decomposition for the input graph. Then we direct the edges of each ear so that each ear forms a directed path or a directed cycle. Observe that the ear decomposition algorithm first finds a rooted spanning tree T. The edges in an ear are of the form (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k), (v_k, u_r), (u_r, u_{r-1}), (u_{r-1}, u_{r-2}), $\ldots$, (u_2, u_1), where (v_i, v_{i+1}) is a tree edge and v_i is the parent of v_{i+1} in T, for $1 \leq i < k$; (u_{i+1}, u_i) is a tree edge and u_{i+1} is the parent of u_i in T, for $1 < i \leq r$; (v_k, u_r) is a non-tree edge. Thus we direct every non-tree edge (u, v) from u to v where u has a smaller ID than that of v. Then

we assign directions to tree edges in such a way that the edges in an ear form a directed path or directed cycle and the first two ears together form a directed cycle.

(vi) **Cut edges.** We first find a rooted spanning tree T for the input graph G. (The current version of the program requires G to be connected.) A cut edge is a tree edge (u, v), where u is the parent of v and there is no non-tree edge (x, y) in G such that either x or y is a descendant of v or equal to v and the least common ancestor of x and y is an ancestor of u or equal to u. This can be implemented by using the Euler tour technique and the range minimum queries.

5. Performance Data

In this section, we present performance data for our parallel programs. In Section 5.1 we describe the sequential algorithms that we implemented corresponding to the parallel graph algorithms that we implemented. Then in Section 5.2, we describe the way we tested and made measurements on the running time of the sequential programs and their parallel counterparts. We obtained their CPU running time and computed the speed-up factor between our parallel programs and sequential programs. This data is presented and analyzed in Section 5.3.

5.1. Sequential Algorithms. After we implemented the parallel graph algorithm library, we also implemented the following sequential graph algorithms using NETPAD.

(i) A recursive version of depth-first search for finding connected components.

(ii) A routine for finding all cut edges in the graph based on the recursive version of depth-first search [**22**].

(iii) A routine for finding a strong orientation based on the recursive version of depth-first search [**22**].

(iv) A routine for finding an ear decomposition based on the recursive version of depth-first search [**19**].

(v) Kruskal's algorithm [**23**] for finding a minimum cost spanning forest.

All but the last of the above five routines are based on depth-first search with some special book keeping. The routine for finding an ear decomposition also needs a linear time bucket sort routine. For depth-first search, finding cut edges and finding a strong orientation, the running time is linear in the size of the graph (with a very small constant factor).

The routine for finding an ear decomposition is linear time, but has a slightly larger constant factor because of the usage of the bucket sort routine. Kruskal's algorithm for finding a minimum cost spanning forest runs in $O(n + m \log n)$ time on an input graph with n vertices and m edges. Although faster algorithms are known for this problem [**23**], we implemented Kruskal's algorithm for its simplicity.

5.2. Testing Scheme. Except for the routine for finding an ear decomposition, the other four sequential programs were run on SPARC II workstations. The program for finding an ear decomposition was run on SPARC I workstations. We wrote a random graph generator using NETPAD and generated random graphs for various input sizes.

We tested our programs on graphs with more edges than vertices. To generate a random graph with n vertices and m edges, we first generated an empty graph with n vertices. Then we added one edge at a time with each edge being chosen with uniform probability until all m edges were generated. To generate a connected test graph with n vertices and m edges, we first generated an empty graph with n vertices. Then we constructed a spanning tree by adding edges to connect different connected components with each edge being chosen with uniform probability over all candidate edges. Finally, we randomly added edges until the graph contains m edges. For testing the algorithm for finding a minimum spanning forest, we assigned a random integer cost ranging from 0 to 999 to each edge in the graph with each number being equally likely to be chosen (with repetition).

We generated a two-edge connected graph with n vertices and m edges by first generating an empty graph with n vertices. We then chose a random length k, $3 \leq k \leq n$, and k isolated vertices in random. We randomly permuted these k vertices and constructed a simple cycle by adding an edge between every two adjacent vertices in the random permutation and by adding an edge between the first and the last vertices in the permutation. After that, we added non-trivial ears of random lengths to connect all isolated vertices. To add a non-trivial ear, we chose a random length l, $1 \leq l \leq x$, where x is the number of remaining isolated vertices. Then we randomly picked l isolated vertices and two non-isolated vertices u and v (with repetition if $l > 1$). We randomly permuted the l isolated vertices and constructed a simple path by adding an edge between every two adjacent vertices in the random permutation. We added an edge between u and the first vertex in the above permutation and another edge between v and the last vertex in the above permutation. After connecting all isolated vertices, we randomly added edges until all m edges are generated.

Let n and m be the number of vertices and the number of edges in the input graph, respectively. Given any input size (the number of vertices), we generated graphs with three different densities: sparse graphs with $m = \frac{3}{2}n$; intermediate-dense graphs with $m = n^{1.5}$; dense graphs with $m = \frac{n^2}{4}$. For each input size on each density, we generated 4 different random graphs. On each input graph, each program was run 10 times. (We ran each program 10 times on the same input data in order to even out any fluctuation in the Unix routines we used for measuring CPU running time.) The CPU time used in each run was collected. (We only measured the part of the CPU time used for graph computations. The overhead for input/output and for using the NETPAD system was not included.) We then calculated the average CPU time used by the sequential programs for

each input size on each density.

The same set of testing graphs was then fed into our parallel programs. On each input graph, we ran the parallel program 10 times. The CPU time used for graph computation was collected for each run. We then calculated the average CPU time used by the parallel programs for each input size on each density.

5.3. Analysis. For each graph problem that we solved sequentially and in parallel, we have plotted the relative CPU time used by both programs on sparse graphs, intermediate-density graphs and dense graphs. The results are shown in Figure 8 for finding connected components; Figure 10 for finding a minimum cost spanning forest; Figure 12 for finding all cut edges; Figure 14 for finding a strong orientation on a two-edge connected graph; Figure 16 for finding an ear decomposition on a two-edge connected graph. Although each PE is much slower than the SPARC workstation, we found that in the case of finding a minimum cost spanning forest and finding an ear decomposition, parallel programs in fact run faster in real time compared to sequential programs. For example, the routine for finding an ear decomposition on the MasPar is about 3 times faster (in real CPU time) on the largest test graph we have than the one that runs on the SPARC I workstation.

To compute the relative speed-up between our parallel programs running on the MasPar and the sequential programs running on SPARC workstations, we need to have the ratio of the computation speed of a MasPar PE to that of a SPARC workstation. In [**13**] it is stated that each PE is about 10 times slower than the MasPar ACU. We tested programs running on the MasPar ACU and the MasPar front end. Test data show the current MasPar front end, which is a micro-VAX workstation, is about 2 to 3 times faster than the MasPar ACU. We then ran our sequential programs on the MasPar front end. Test data show that the front end is at least 10 times slower than the same programs running on the SPARC II workstation. In some tests, it is more than 15 times slower. Our tests also show that the SPARC I workstation is about 1.75 times slower than the SPARC II workstation. (This figure is confirmed by data in [**5**].) Thus the SPARC II workstation is at least 200 times faster than each MasPar PE and the SPARC I workstation is at least 114 times faster than each MasPar PE.

We rescaled the CPU time used by sequential programs running on SPARC II and SPARC I according to the above figures and computed the speed-up of the parallel programs running on the MasPar relative to the sequential programs on the SPARC workstation. All of our parallel programs run in $O(\log^2 m)$ time using $2m$ processors on an input graph of m edges. Thus the theoretical speed-up for our parallel programs using P PE's is $\Theta(\frac{P}{\log^2 P})$ for all problems, except for the one for finding a minimum cost spanning forest. The theoretical speed-up for finding a minimum cost spanning forest is $\Theta(\frac{P}{\log P})$ using P PE's since the sequential algorithm runs in $O(n + m \log n)$ time. We plotted the relative speed-up for each problem with its theoretical speed-up curve. In plotting each

theoretical speed-up curve, we used a constant multiplicative factor that best approximated the experimental data. The results are shown in Figure 9 for finding connected components; Figure 11 for finding a minimum cost spanning forest; Figure 13 for finding all cut edges; Figure 15 for finding a strong orientation on a two-edge connected graph; Figure 17 for finding an ear decomposition on a two-edge connected graph.

We found that our parallel program for finding an ear decomposition has better speed-up than the rest of the programs, while our parallel routine for finding a minimum cost spanning forest has the least speed-up. The other three problems have a speed-up factor that is very close to $\frac{P}{\log^2 P}$[‡] (i.e. the constant multiplicative factor is close to 1). This shows that PRAM graph algorithms for connected components, cut edges, strong orientation and ear decomposition can be implemented and run with their expected asymptotic time bound with a small constant multiplicative factor.

In terms of actually writing code, we wrote about 3000 lines of MPL code for our library of parallel programs which includes the kernel and the graph algorithms. We used about 1600 lines of C code for our sequential programs with the help of NETPAD library routines. Without the help of NETPAD to take care of the general graph data structures and other commonly used graph operations, the size of our sequential programs would have been larger. (In our parallel programs, we only use NETPAD for input and output.) Thus the code for our parallel algorithms was not much larger than that for the sequential algorithms.

6. Conclusion and Future Work

We have implemented several efficient PRAM graph algorithms on the MasPar. We have developed an interface between the graph manipulation package NETPAD and our parallel programs. We have also written sequential programs for solving the same graph problems and studied the relative speed-up of the parallel programs over the sequential ones. The performance presented in this paper is further analyzed in [**4**] where least-squares-fit curves are obtained for each set of data.

We note a few observations.

- **PRAM based graph algorithms can be implemented efficiently and easily.** The PRAM model has proven to be a very good theoretical model for designing parallel algorithms. By developing a general mapping strategy between the PRAM model and the target machine hardware architecture, we can make use of results developed on the PRAM model. Our experience with the MasPar shows that we can achieve reasonable speed-up by this approach. The whole process of programming and debugging is easy and fast. (All of the work reported here was

[‡]We use $\log P$ to represent $\log_2 P$ in this paper.

accomplished within a period of 11 weeks.)

- **Global routing bottleneck.** The current global router on the MasPar is very slow compared to the XNET configuration. (It is 100 times slower than the XNET for transferring a 32-bit data to each PE [**12**].) Although we use the XNET configuration when we can in our implementations, our parallel graph algorithms often need to use the global router for performing list computations. The performance of our parallel programs would be significantly improved if the global router could be made to run faster when routing large data sets.
- **NETPAD is a useful tool for designing graph algorithms.** NETPAD takes care of the input of test graphs and the output of results. NETPAD also provides an interface for generating test data from other programs. Its graphic display capability provides a good tool for debugging. In the design of the sequential programs, NETPAD also provides library routines for maintaining graph data structures. Our parallel programs used NETPAD for the input of test graphs and the output of results.

There are several avenues for future work. We list some of them:

- **NETPAD for designing parallel graph algorithms.** The original goal of NETPAD was for supporting the design of sequential graph algorithms. Although we used NETPAD to design our parallel graph algorithms, we did not get as much support from NETPAD as we did in designing our sequential programs. Graph data structures for parallel computations should be supported by NETPAD. Better visualization routines should be added for viewing large graphs. Although we have built an interface between our parallel programs and NETPAD, the interface is very immature. Further work should be done to use the current animation capabilities of NETPAD in the parallel environment and to provide better tools for animating parallel graph algorithms.
- **Further fine tuning of our parallel programs.** With a better understanding of the MPL language and the MasPar architecture, we could fine tune our programs to run faster. Some of the things that could be done include better utilization of registers in each PE, using faster I/O interface between the PE's and the MasPar file system and finding the trade-off between computing operations on the ACU and on the PE.
- **Processor allocation for large input size.** Our current implementation does not support the use of virtual processors. Our programs can only handle input graphs of size less than or equal to the number of physical processors. An obvious avenue for further work is to extend our programs so that they could run on inputs of any size. Because the MPL programming language requires explicit allocation of PE's, we need to modify our code to handle this. We also need to implement optimal PRAM algorithms to obtain the best speed-up results when using

virtual processors. (These issues are discussed in [**4**].)

- **Further extension of the parallel graph algorithm library.** Since we have implemented most of the commonly used routines for implementing PRAM undirected graph algorithms, we expect that it will be fairly easy to implement other graph algorithms; for example, the routines for finding an open ear decomposition, biconnectivity, 3-edge connectivity, triconnectivity and planarity [**19, 20**], since we have already implemented most of the basic subroutines for these problems.

Acknowledgment. We would like to thank Monika Mevenkamp for her help in developing the interface between the parallel programs on the MasPar and NETPAD. We also thank Peter Winkler for helpful discussions on how to generate two-edge connected graphs and Clyde Monma for his support of this project. We thank Michael B. Carter for providing a set of routines for measuring CPU time used by programs run on the MasPar.

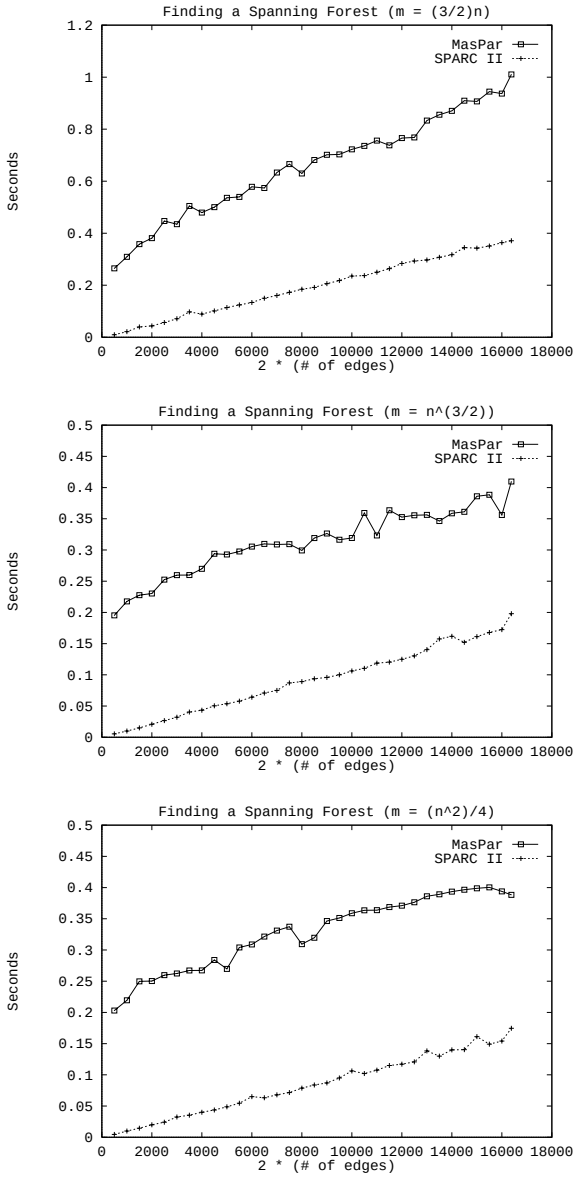

FIGURE 8. Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding connected components.

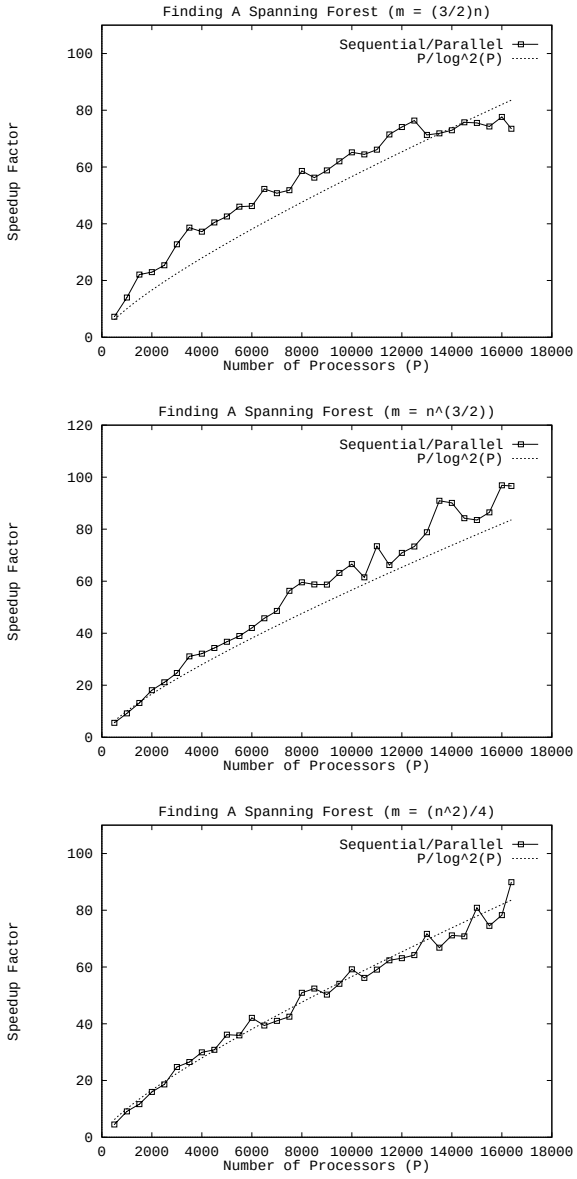

FIGURE 9. Speed-up data for finding connected components. Note that the constant multiplicative factor used in plotting each theoretical speed-up curve is 1.

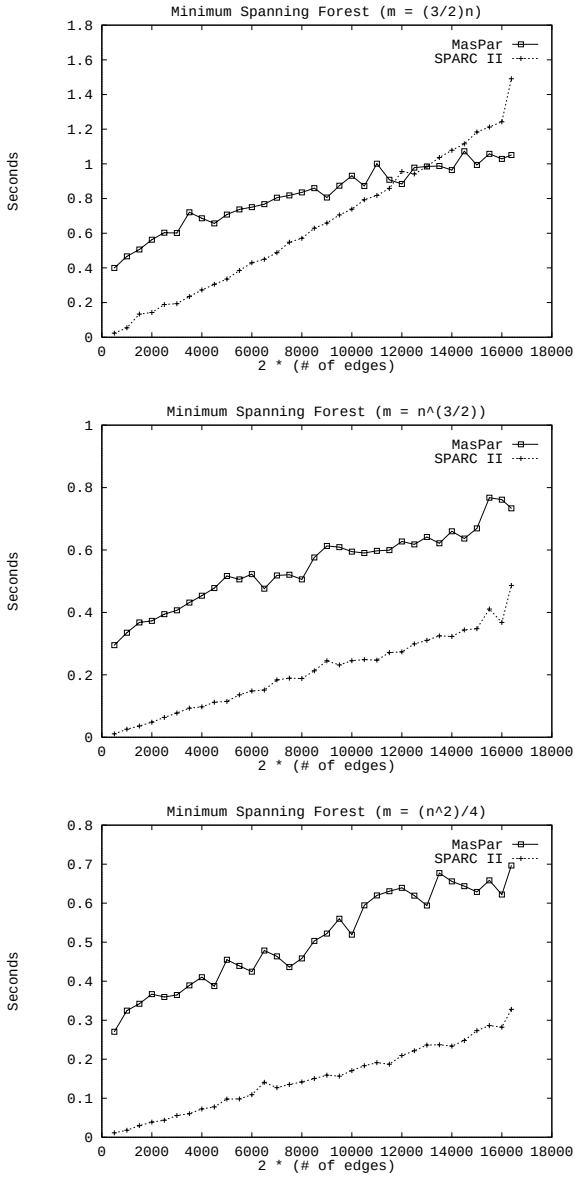

FIGURE 10. Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding a minimum spanning forest.

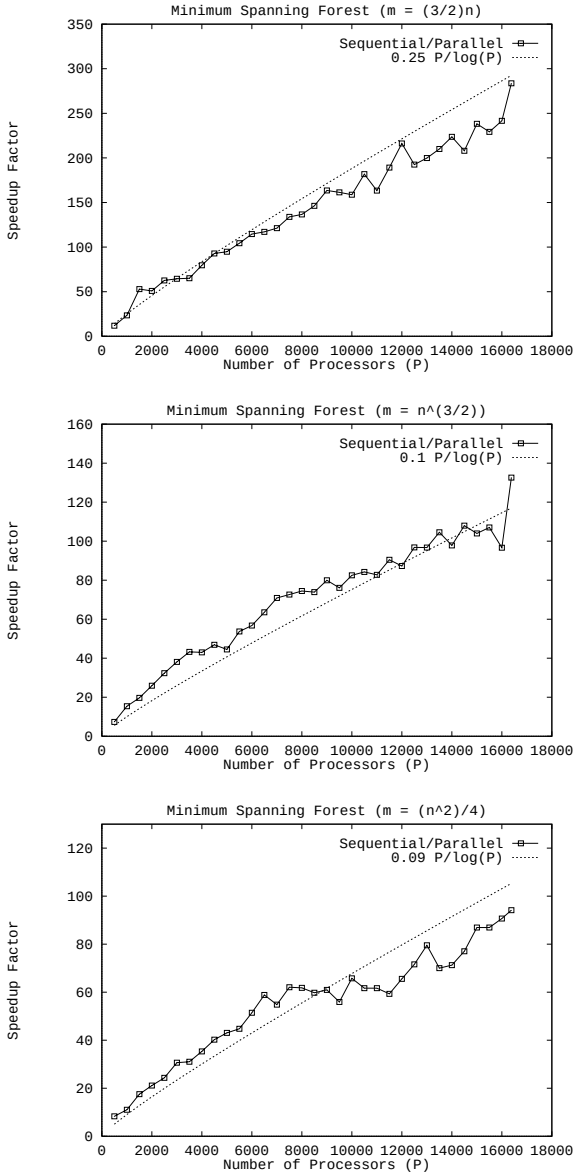

FIGURE 11. Speed-up data for finding a minimum spanning forest. Note that the constant multiplicative factors used in plotting the theoretical speed-up curves are 0.25, 0.1 and 0.09, respectively (from the top to the bottom), for the above three sets of data.

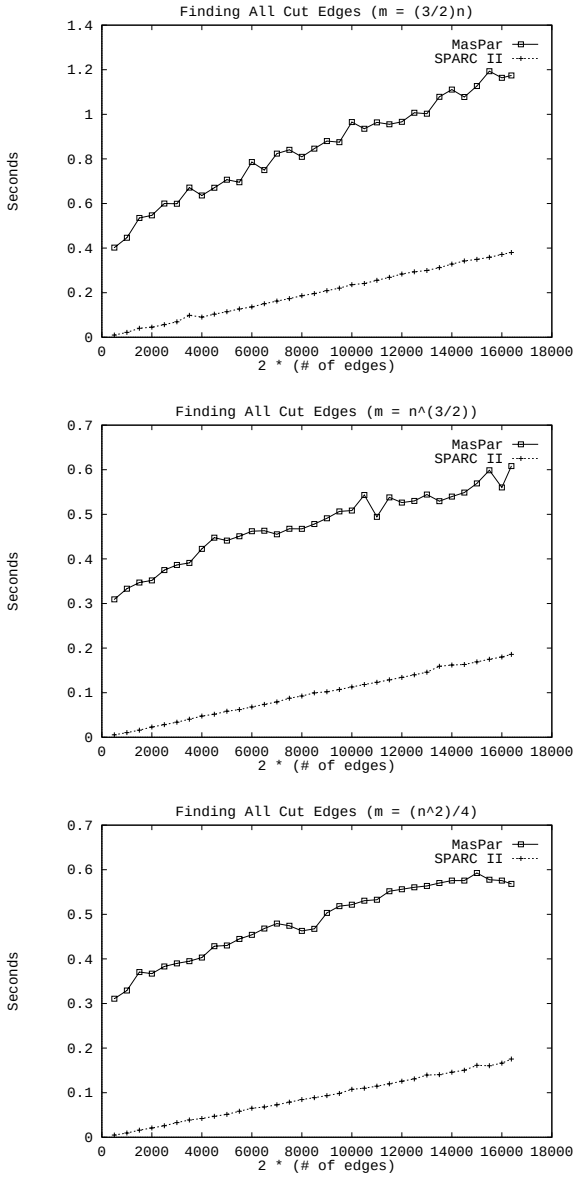

FIGURE 12. Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding all cut edges.

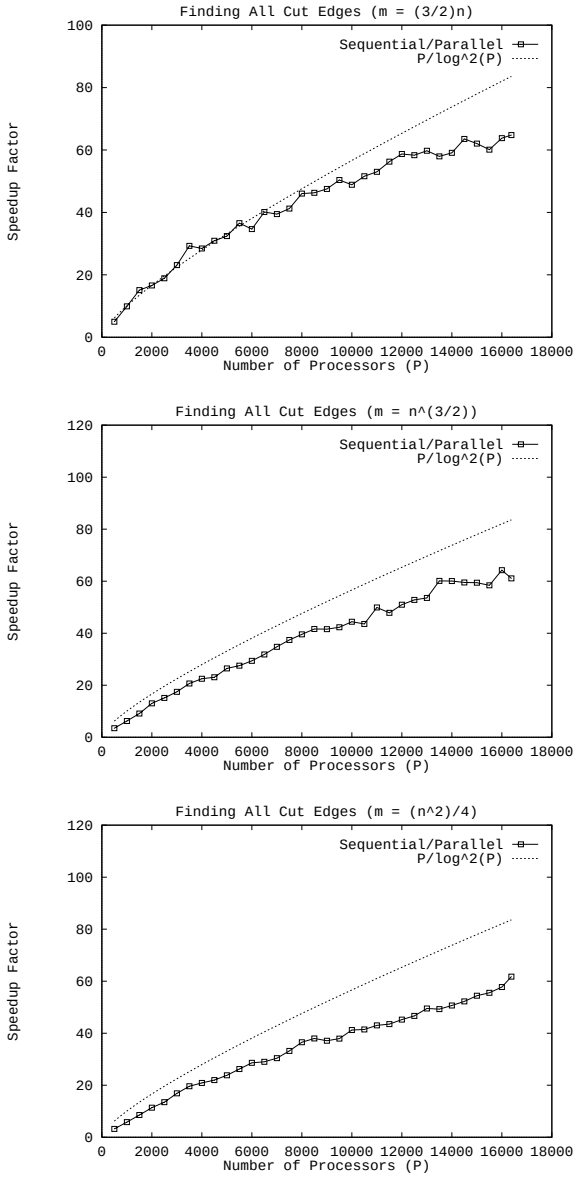

FIGURE 13. Speed-up data for finding all cut edges. Note that the constant multiplicative factor used in plotting each theoretical speed-up curve is 1.

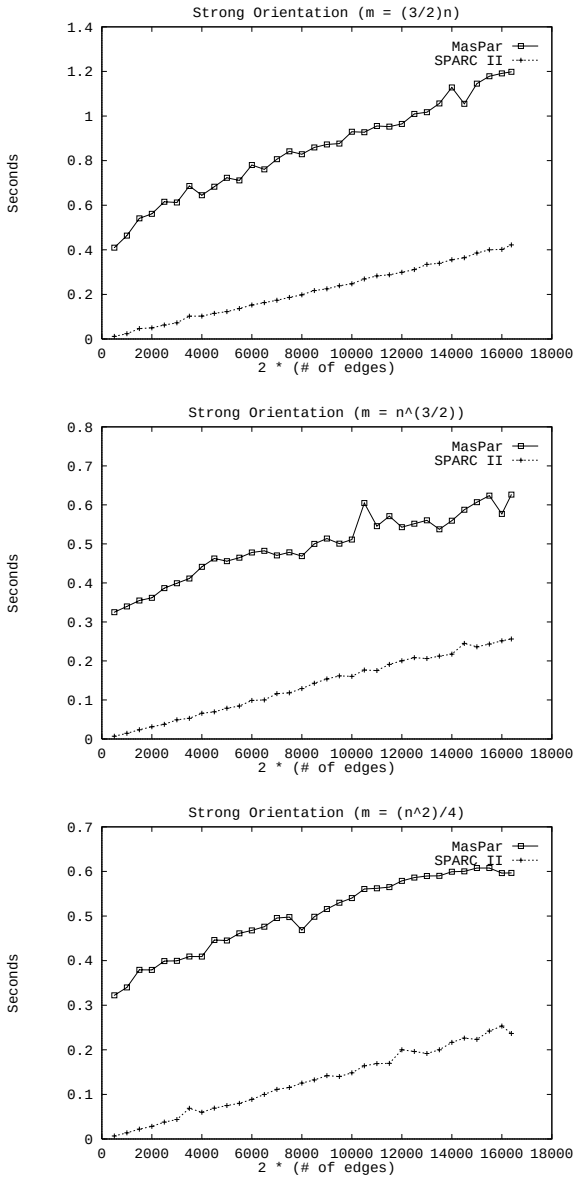

FIGURE 14. Relative performances of the sequential program on a SPARC II workstation and the parallel program on the MasPar for finding a strong orientation on a two-edge connected graph.

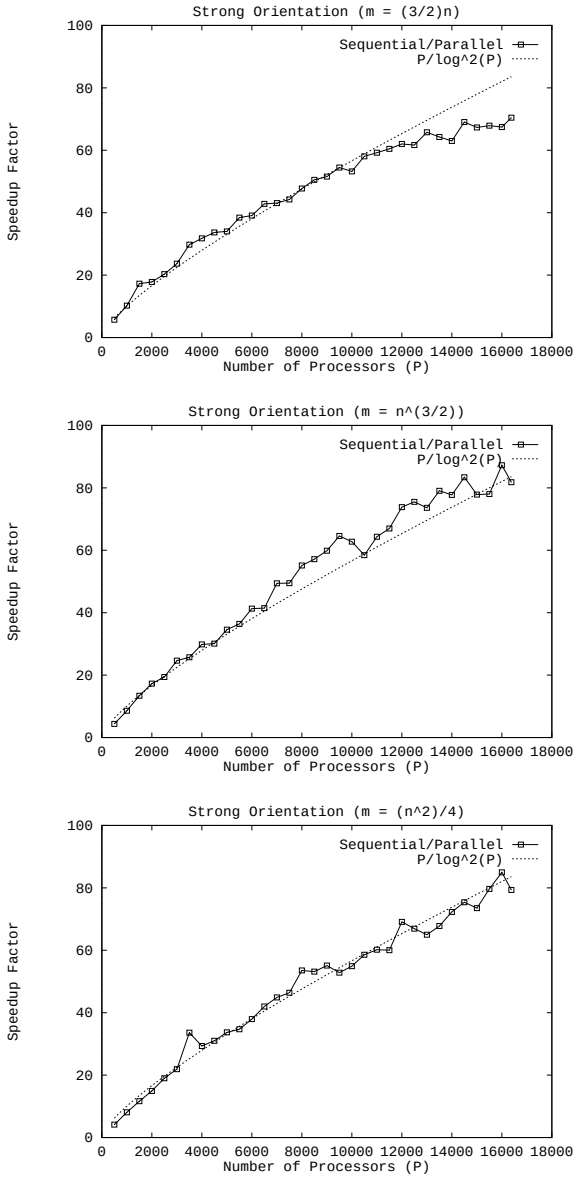

FIGURE 15. Speed-up data for finding a strong orientation on a two-edge connected graph. Note that the constant multiplicative factor used in plotting each theoretical speed-up curve is 1.

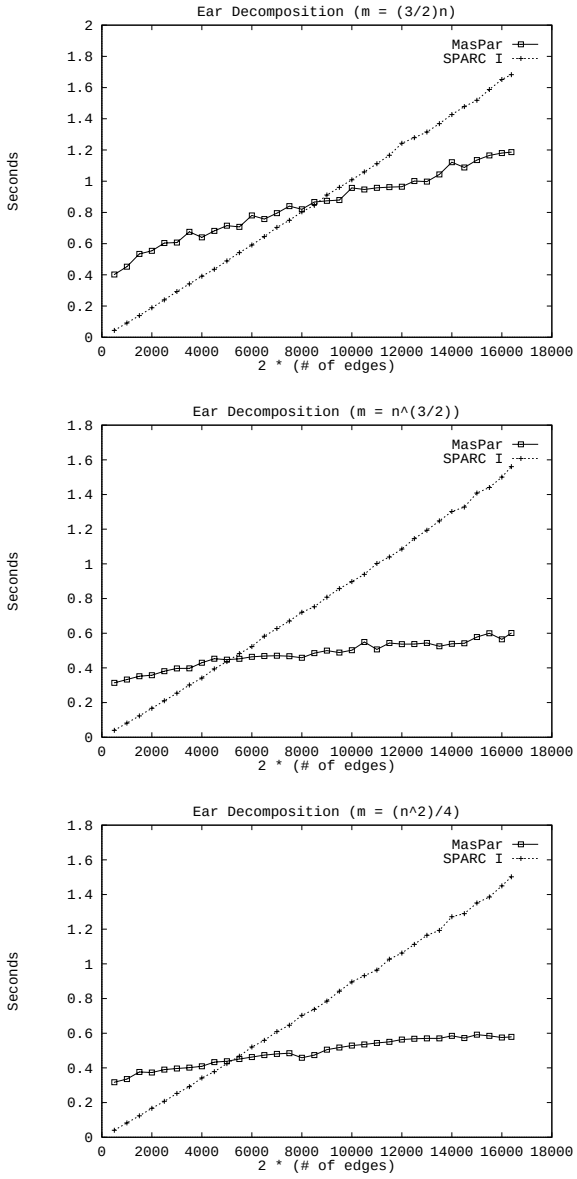

FIGURE 16. Relative performances of the sequential program on a SPARC I workstation and the parallel program on the MasPar for finding an ear decomposition on a two-edge connected graph.

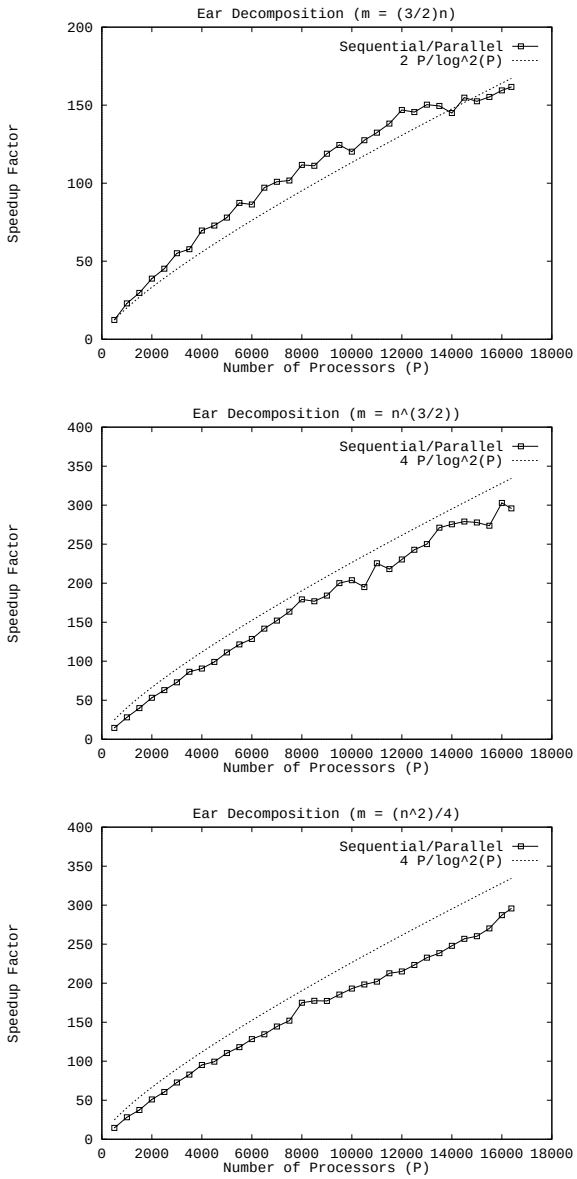

FIGURE 17. Speed-up data for finding an ear decomposition on a two-edge connected graph. Note that the constant multiplicative factors used in plotting the theoretical speed-up curves are 2, 4 and 4, respectively (from the top to the bottom), for the above three sets of data.

References

1. B. Awerbuch and Y. Shiloach, *New connectivity and MSF algorithms for shuffle-exchange network and PRAM*, IEEE Tran. on Computers (1987), 1258–1263.
2. G. E. Blelloch, *Scan primitives and parallel vector models*, Ph.D. thesis, M.I.T., October 1989.
3. N. Dean, M. Mevenkamp, and C. L. Monma, *NETPAD: An interface graphics system for network modeling and optimization*, Proc. Computer Science & Operations Research: New Developments in their Interfaces, Pergamon Press, 1992, pp. 231–243.
4. T.-s. Hsu, V. Ramachandran, and N. Dean, *Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing*, Proc. 9th International Parallel Processing Symp., 1995, pp. 106–112.
5. IEEE Computer, *New products column*, January 1991, pp. 113–115.
6. A. Kanevsky and V. Ramachandran, *Improved algorithms for graph four-connectivity*, J. Comp. System Sci. **42** (1991), 288–306.
7. R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), North Holland, 1990, pp. 869–941.
8. B. W. Kernighan and D. M. Ritchie, *The C programming language*, Prentice Hall, Englewood Cliffs, NJ, 1978.
9. R. E. Ladner and M. J. Fischer, *Parallel prefix computation*, J. ACM **27** (1980), 831–838.
10. Y. Maon, B. Schieber, and U. Vishkin, *Parallel ear decomposition search (EDS) and st-numbering in graphs*, Theoret. Comput. Sci. (1986), 277–298.
11. MasPar Computer Co., *MasPar parallel application language (MPL) reference manual*, version 2.0 ed., March 1991.
12. MasPar Computer Co., *MasPar parallel application language (MPL) user guide*, version 2.0 ed., March 1991.
13. MasPar Computer Co., *MasPar system overview*, version 2.0 ed., March 1991.
14. M. Mevenkamp, *NETPAD programmer's guide*, Bellcore, August 1991.
15. ———, *NETPAD reference guide*, Bellcore, August 1991.
16. M. Mevenkamp, N. Dean, and C. L. Monma, *NETPAD user's guide*, Bellcore, August 1991.
17. G. L. Miller and V. Ramachandran, *A new triconnectivity algorithm and its applications*, Combinatorica **12** (1992), 53–76.
18. A. Nye, *X window system user's guide*, O'Reilly & Associates, Inc., 1988.
19. V. Ramachandran, *Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity*, Synthesis of Parallel Algorithms (J. H. Reif, ed.), Morgan-Kaufmann, 1993, pp. 275–340.
20. V. Ramachandran and J. Reif, *Planarity testing in parallel*, Jour. Comput. and Sys. Sci. **49** (1994), no. 3, 517–561, Special Issue for *FOCS '89*.
21. D. M. Ritchie and K. Thompson, *The Unix timesharing system*, Communications of the ACM **17** (1974), 365–375.
22. R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), 146–160.
23. ———, *Data structures and network algorithms*, SIAM Press, Philadelphia, PA, 1983.
24. R. E. Tarjan and U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput. **14** (1985), 862–874.

Department of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
E-mail address: tshsu@cs.utexas.edu

Department of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
E-mail address: vlr@cs.utexas.edu

Room 2M-389, 445 South Street, Bellcore, Morristown, NJ 07960
E-mail address: nate@bellcore.com