

Implementation of Parallel Graph Algorithms on a Massively Parallel SIMD Computer with Virtual Processing

Tsan-sheng Hsu^{*†} & *Vijaya Ramachandran*^{*}

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Nathaniel Dean

Combinatorics and Optimization Research
Bell Communications Research
Morristown, NJ 07960

July 22, 1993

Abstract

We describe our implementation of several PRAM graph algorithms on the massively parallel computer MasPar MP-1 with 16,384 processors. Our implementation incorporated virtual processing and we present extensive test data.

In a previous project [13], we reported the implementation of a set of parallel graph algorithms with the constraint that the maximum input size was restricted to be no more than the physical number of processors on the MasPar. The MasPar language MPL that we used for our code does not support virtual processing. In this paper, we describe a method of simulating virtual processors on the MasPar. We re-coded and fine-tuned our earlier parallel graph algorithms to incorporate the usage of virtual processors. Under the current implementation scheme, there is no limit on the number of virtual processors that one can use in the program as long as there is enough main memory to store all the data required during the computation. We also give two general optimization techniques to speed up our computation.

We tested our code with virtual processing on test graphs with various edge densities. We also compared the performance data for our parallel code with the performance data of sequential code for these problems. We found that the extra overhead for simulating virtual processors is moderate and the performance of our code tracks theoretical predictions quite well, although real-time speed-ups are quite small since the MasPar processors are rather slow. In addition, our parallel code using virtual processing runs on much larger size inputs than our sequential code.

Key words and phrases: parallel algorithms, graph algorithms, implementation, virtual processing, MasPar.

^{*}Supported by NSF Grant CCR-90-23059 and Texas Advanced Research Projects Grant 003658480.

[†]Also supported by an IBM graduate fellowship.

1 Introduction

This paper describes an on-going project for implementing parallel graph algorithms on the massively parallel machine MasPar MP-1. There has been a fair amount of prior work on implementing parallel algorithms on massively parallel machines [1, 5, 9, 10, 11, 25, 29] since the completion of the first phase of our project reported in [13]. However, most of this work has been targeted towards solving problems that are highly structured and are not very difficult to scale up. The focus of our work is on solving graph-theoretical problems for which the algorithms require large amounts of non-oblivious memory accesses.

In [13], we reported the implementation of several parallel graph algorithms on the MasPar MP-1 using the parallel language MPL [19, 20] which is an extension of the C language. The MPL language provides a very efficient way of using the MasPar with the drawback of requiring the specification of the physical organization of the processors used in the program. Our implementation in [13] used an edge list data structure to store the input graph. An undirected edge (u, v) was stored twice as one directed edge from u to v and another directed edge from v to u . Each of the two copies of an undirected edge was stored in one processor along with a node. As a result, we could only handle the case when the input graph has no more than $nproc$ nodes and $\frac{nproc}{2}$ edges where $nproc$ is the maximum number of processors that we can use in the system. The machine that we used, the MasPar MP-1, had $nproc = 16,384$ processors. In the current paper, we report the second phase of this work, which consisted of implementing these algorithms to handle inputs of size greater than 16,384. A major advantage in using massively parallel machines with virtual processing is that we can solve problems on large-sized inputs that cannot be handled by conventional sequential machines.

Several parallel machines offer the convenience of using virtual processors in their high-level programming languages. For example, the Connection Machine [18] offers the support of using virtual processors with the assistance of the hardware and microcodes in the C* programming language. The parallel Fortran language used in the MasPar also supports the usage of virtual processors. However, these supports for using virtual processors come with the penalty of having a very large overhead. Programs that want to achieve a high percentage of machine utilization are either coded in low level programming language (e.g. the Paris language in the Connection Machine [5]) or coded in a language that does not support virtual processing (e.g. the MPL language in the MasPar [13, 29]). We have used the latter approach in the current work.

Our results are reported in the following sections which are organized as follows. Section 2 gives our implementation strategy and the programming environment of the MasPar MP-1. Section 3 gives a high-level description of our implementation. Section 4 describes our strategy in mapping the PRAM model and in mapping virtual processors onto the MP-

1. Section 5 describes the implementation details of our parallel graph algorithms library. Section 6 gives performance analysis. Finally, Section 7 gives the conclusion and possible future work.

2 Preliminaries

2.1 Implementation Strategy

Several strategies can be used to implement parallel algorithms on a parallel computer. One possible strategy is to implement different algorithms for different architectures. Since parallel machines are widely diverse in their architectures, one can take advantage of the special properties offered by an architecture and fine-tune the algorithms to run well on a particular machine. For parallel algorithms using this approach, see [17, 30]. However, this time-consuming process must be carried out each time a new architecture arrives. This approach may be useful for some of the very important subroutines used in the machine (e.g. sorting [5, 29]). However, for complicated combinatorial problems, reinventing different algorithms for different architectures tends not to be a feasible solution. As the problems get more complicated, it takes longer time to derive efficient algorithms. Further, we feel that this is not a very good strategy as one often discovers that the fundamental algorithmic techniques underlying the parallel algorithms for most problems are independent of the particular parallel machine being used. Thus one should study and utilize these basic techniques to assist the implementation of parallel algorithms.

In view of the above, a natural strategy is to use parallel algorithms developed on an abstract parallel machine model. Several abstract models that are closely related to real parallel machine architectures have been proposed [4, 8, 12, 33]. Instead of using a new model, we have performed a direct implementation of parallel algorithms based on the popular PRAM model [14, 15, 32]. Although the PRAM is an idealized theoretical model that does not capture the real cost of performing inter-processor communications on the MasPar, we believe that it provides a good abstract model for developing parallel algorithms. Parallel algorithms developed on the PRAM model are often very modular in structure (or have parallel primitives). Problems are solved by calling these parallel primitives. For solving undirected graph problems, a set of parallel primitives required for constructing an ear decomposition has proved to be very useful [31, 37]. Our parallel implementation follows 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.

Our experience with implementing PRAM graph algorithms on the MASPAR MP-1 as reported in this paper and in [13] supports our viewpoint that efficient PRAM algorithms are adaptable to run on real machines. The basic primitives should be fine-tuned for the real

machine, but the overall structure of a complex PRAM algorithm can be mapped directly on to the real machine.

2.2 Programming Environment

The MasPar computer [21] 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 1. For details, see [3].

Each physical processor (called a PE) is a 4-bit CPU with 64 kilobytes of main memory and a unique ID ranging from 0 to 16,383. Through the emulation of microcode, each PE can perform operations on 8-bit, 16-bit, 32-bit, and 64-bit data. There are two types of inter-processor communication available. First, PE's are organized as a two-dimensional wrap-around mesh. Each PE can communicate with its 8 nearest neighbors. This is called the XNET connection. Second, 16 PE's (a 4×4 sub-mesh) are grouped into a cluster. The 1024 clusters are organized as a 10-dimensional hypercube with each cluster representing a node in the hypercube. This is called the global router. Processors can communicate with each other by using the global router. Mesh communications are about 200 times faster than global routing requests for transmitting 32-bit data [20, 28]. The MasPar also has an Array Control Unit (ACU) for controlling the PE's and executing sequential instructions. The MasPar PE is a very slow processor. In comparison, SUN SPARC II is more than 200 times faster than a MasPar PE, while SUN SPARC 10/41 is more than 230 times faster.

We used the MPL high-level programming language for coding our programs. The current version of the MPL compiler [19, 20] is an extension of the ANSI C language [16] with data parallel constructs and a library of parallel primitives. (For details of the MPL language, see [23, 24]. An introduction to MPL is also given in [26].) In MPL, a variable can be declared with or without the attribute *plural*. A plural variable has a local copy (possibly with different values) in each PE, while a variable without the plural attribute has only one copy in the ACU. During each computation step, each PE decides whether or not to participate in the current computation by the value of a local flag. A PE that participates in a computation step is called *active*. Any step that uses plural variables will be executed by each active PE. For details, see [13, 26]. It is important to note that the current version of the MPL language does not support the use of virtual processors. Thus we had to design and implement our own scheme for virtual processing.

3 High-Level Description of Our Implementation

In our earlier implementation of parallel graph algorithms without virtual processing [13], we first provided a general mapping between the architecture of the MasPar and the schematic

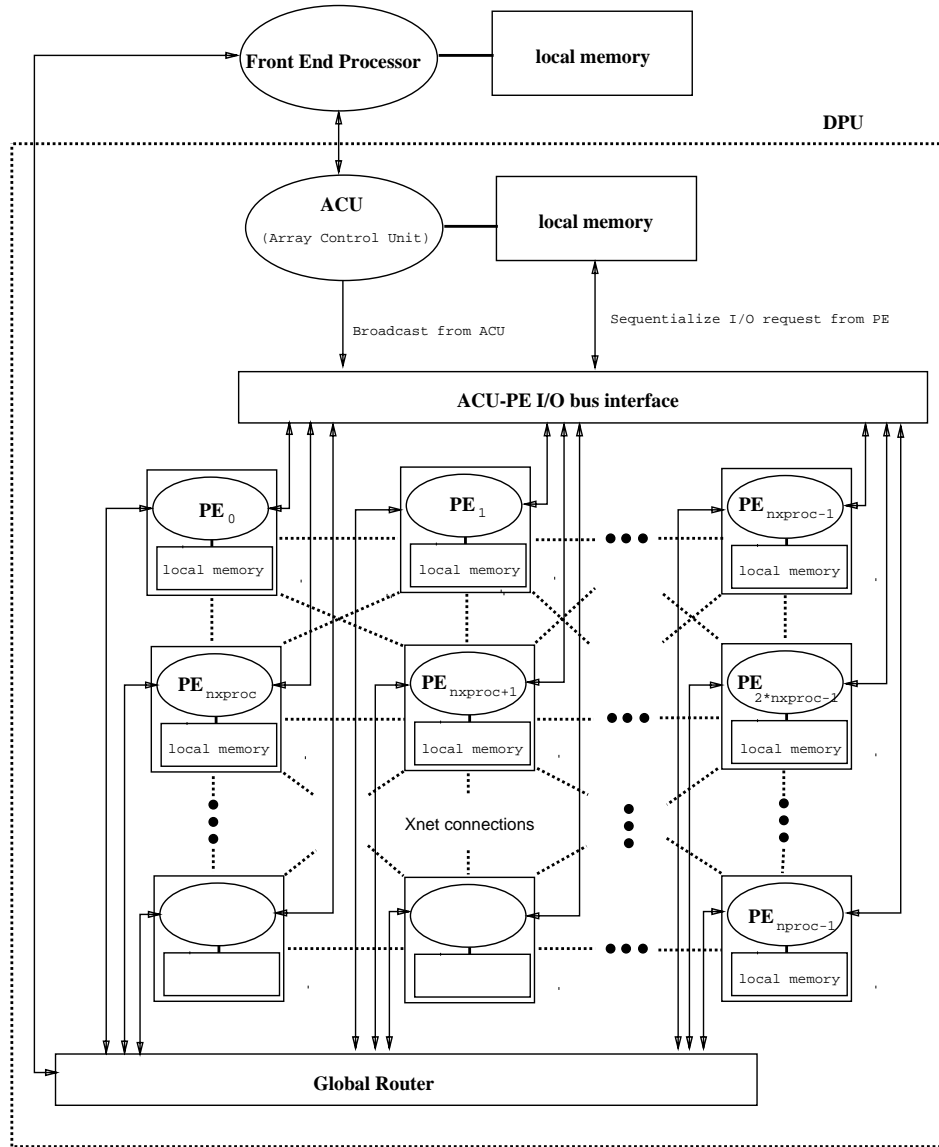


Figure 1: System architecture for the MasPar MP-1 computer.

structure of the PRAM model. This mapping scheme took advantage of some of the special properties of the MasPar, although it was not fine-tuned for each individual routine. This mapping scheme will be described in Section 4.1. (This approach has been used in simulating PRAM algorithms on various parallel architectures, e.g. see the section on simulating PRAM algorithms in [17]. However, most of the previous results do not have any implementation details and provide no performance data.) Using this mapping, we then coded each simple parallel primitive on the MasPar. While coding each primitive, we utilized the special properties of the MasPar to fine-tune our code. Since each parallel primitive is very easy to code, one would expect the fine-tuning step to be much simpler than the fine-tuning step of a complicated algorithm. We implemented a set of parallel graph algorithms without virtual processing by calling the parallel primitives we coded and routines provided in the system library as reported in [13].

Due to the constraints imposed by the programming environment on the MasPar, the above implementation requires the size of the input to be no more than the number of available physical processors. However, the parallel primitives coded can be used with any number of processors by invoking Brent’s scheduling principle [6, 15] to simulate several virtual processors on one physical processor. To do this, we extended our mapping scheme to handle the allocation and simulation of virtual processors. The extended mapping will be described in Section 4.2.

Using our original code when no virtual processors are used [13] as a blueprint and the extended mapping as a guideline, we transformed our code to handle the allocation of virtual processors. Since the MPL language does not support virtual processing, we had to implement our own scheme for virtual processing. To do this, we re-coded and fine-tuned the set of parallel primitives identified in [13] and several system library routines to handle the allocation of virtual processors efficiently. Then we implemented a set of parallel graph algorithms by calling these parallel primitives and system routines. The primitives and graph algorithms we implemented are described in Section 5.

4 Mapping Strategy

In this section, we briefly describe the mapping scheme we used in [13] to map the PRAM model onto the MasPar architecture. We then describe the mapping scheme we used in allocating virtual processors.

4.1 Mapping of the PRAM Model onto the MasPar Architecture

We briefly summarize the mapping scheme used in [13] to map a PRAM onto the MasPar architecture when the two machines have the same number of processors. We mapped part of the local memory in each PE and the local memory of the ACU onto the PRAM global

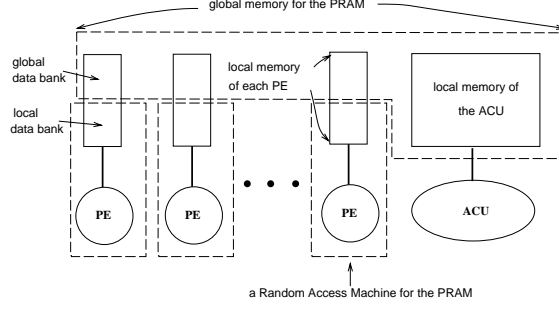


Figure 2: Mapping of the MasPar architecture onto the PRAM model.

memory. The major difference between the PRAM model and the MasPar architecture is the issue of global memory access. We partitioned the local memory bank of each PE into two halves. One half, which we call *the global data bank of each PE*, was mapped onto part of the global memory bank and the other half, which we call *the local data bank of each PE*, was used for storing local data for local computations. The entire local memory of the ACU was made part of the global memory of the PRAM model. When implementing a 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 read-only data into the local memory bank of the ACU and arranged for the ACU to broadcast the needed data to all PE's. We illustrate the mapping in Figure 2. More details of this mapping can be found in [13].

4.2 Mapping of the Virtual Processors onto the MasPar Architecture

In our programs, each virtual processor (or VPE) is given a unique ID ranging from 0 to $vnproc - 1$, where $vnproc$ is the number of virtual processors. (Note that $nproc$ is the number of physical processors and they are organized as an $nproc \times nproc$ mesh. For the machine that we used, $nproc = 16,384$ and $nproc = nyproc = 128$.) The number of virtual processors per physical processor is $vpr = \left\lceil \frac{vnproc}{nproc} \right\rceil$. The virtual processors are arranged into a 2-dimensional $vnproc \times vnproc$ mesh.

For our implementation, we used the so-called *hierarchical partitioning scheme* [22]. Each physical processor simulated a $vpr \times 1$ sub-mesh of virtual processors. Thus given an $nproc \times nyproc$ 2-dimensional mesh, the virtual machine being simulated is an $(nproc \cdot vpr) \times nyproc$ 2-dimensional mesh. (The implementation of bitonic sort [29] with virtual processing used the same mapping scheme as ours.) We illustrate the mapping in Figure 3. The reason for our choice is that in our implementation of parallel algorithms, we frequently need to use operations that can utilize the locality of data (e.g. the prefix sum (scan) operator [4]). This type of data partitioning enables us to preserve the locality of data.

Once our code decided on the vpr value, each plural variable allocated in the code in

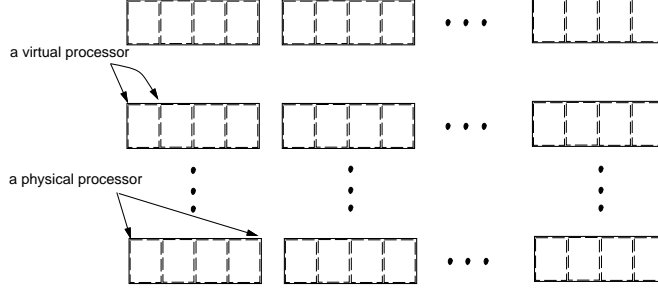


Figure 3: Mapping of 4 virtual processors onto each PE.

[13] was transformed into a plural array of vpr elements in our new code. (The selection of the vpr value is discussed at the end of Section 5.1.3.) The i th element in the j th physical processor corresponded to the local copy of virtual processor $(j-1) \cdot vpr + i$. Variables used in the code in [13] without the plural attribute were not changed in our new code. An extra flag (called *active*) in each virtual processor was allocated in our new code to indicate whether its corresponding virtual processor was active during each step of computation. Thus given a plural variable *data* and a VPE with the ID w , the local copy of *data* was stored in the $(w \bmod vpr)$ th element of the local array *data* in the PE with the ID $\lfloor \frac{w}{vpr} \rfloor$.

5 Implementation of Parallel Graph Algorithms

In this section, we first describe the implementation of several data structures. Then we describe the parallel graph algorithms library that we have built.

5.1 Data Structures

5.1.1 Array and Linked List

Given the value of vpr , we mapped a global memory array used in a PRAM algorithm onto the MasPar by putting the i th element of the array into the i th VPE. Thus this element will be allocated in the $(i \bmod vpr)$ th element of a local array on the $(\lfloor \frac{i}{vpr} \rfloor)$ th physical processor. We mapped a linear linked list used in a PRAM algorithm by putting each element in the list into a different VPE. Pointers in PRAM were replaced by the ID's of VPE's.

5.1.2 Tree

We represented an edge in an undirected tree by two directed edges of opposite directions. A tree was represented by a list of directed edges. In implementing the tree data structure on the MasPar, we put one directed edge in one VPE with the requirement that the set of edges that are incoming to the same vertex have to be allocated on a consecutive segment of VPE's. Each of the two copies of an undirected edge kept a reverse pointer which pointed to the location of the other copy of the same edge. Using this representation, we can use

the XNET connection to perform inter-processor communications needed for computing an Euler tour on a tree. Since computing an Euler tour is one of the most common subroutines on trees used by parallel graph algorithms, we saved time by using this mapping.

5.1.3 Undirected Graph

In our implementation without virtual processing [13], a general undirected graph was represented by a list of edges. Each edge had two copies with the two end points interchanged. We placed an edge on a MasPar PE with the requirement that the two copies of the edge have to be located on adjacent PE's. The reason for using this data structure was twofold. First, we wanted a tree to be represented by a list of edges such that edges incident on a node were allocated in a continuous segment of processors for the ease of finding an Euler tour in a tree. Representing an undirected edge by its two corresponding directed versions was consistent with the representation of a tree. Second, undirected graph algorithms often needed to perform operations on nodes based on information stored on the edges incident on them. Since an undirected edge has two end points, each edge had to perform operations on each of its two end points. Thus we needed two processors to handle one undirected edge. When virtual processing was involved, the natural candidate for our mapping was to allocate each copy of an edge on a different VPE.

Let m and n be the numbers of (undirected) edges and nodes in the input graph, respectively. Using the naive strategy for allocating undirected graphs described in the previous paragraph, we determined the value of vpr by computing the least power of 2 that is greater than or equal to $\left\lceil \frac{2m+2}{n_{proc}} \right\rceil$. Edges were allocated among virtual processors with the ID's from 2 to $2m+1$. (For the easy of programming, we did not use the first two virtual processors for storing edges.) The i th node was allocated to the virtual processor with the ID i . Initially, we coded the routine for finding a spanning forest[‡] with virtual processing using this simple strategy.

In the case when m was much greater than n , this type of data allocation scheme was not balanced since only a small portion of the machine was performing computations related to nodes. The other drawback in using this type of allocation came from the types of operations that were usually used in parallel graph algorithms. It is often the case that information related to edges incident on a node v had to be collected to produce data that will be stored in the processor that was allocated for v . In performing these operations, data will compete with each other to reach a small segment of processors that are physically connected to each other. The delay for this type of inter-processor communications was very large. In order to improve the performance of our code, we considered alternative strategies.

[‡]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.

Dynamic Load Balancing One possible solution for the above problem was to compute different vpr values for nodes and for edges. However, for this we would have to revise our code for parallel primitives such that each primitive knew whether it was performing operations on edges or on nodes. Also, the code for our graph algorithms would have to be changed. This would result in a more complicated implementation. Instead of going through such a serious revision, we came up with the following simple method that did not require us to change other programs. We first computed the number of virtual processors per node to be $nfactor = \left\lfloor \frac{vpr \cdot nproc}{n} \right\rfloor$. We then allocated the i th node to the $(i \cdot nfactor)$ th virtual processor. We had to make sure that the node numbers referred to in each edge are changed accordingly. This was done by multiplying $nfactor$ to every node number used in the edge list. We then performed all of our computations as if the number of nodes is $n \cdot nfactor$. (This is equivalent to adding $n \cdot (nfactor - 1)$ isolated vertices into the input graph.) After performing the computation, data related to nodes allocated in every other $nfactor$ virtual processors was collected. By performing simple preprocessing and post-processing, we evenly distributed all nodes and did not have to track the value of vpr during each operation. Our previous code for finding a spanning forest with virtual processing could be used with minor modification.

Note that we could apply the same technique to several data structures used in our programs. For example, our graph algorithms often found a spanning forest in the input graph and obtained an Euler tour of each of the tree in the spanning forest. The total number of edges in the Euler tours of the forest was $2n - 2$. We could apply the same technique to achieve a better load balancing by evenly distributing tour edges among physical processors. Our graph algorithms also performed range minimum queries on an array of elements whose size was $2n - 2$. We could also use this technique to achieve a better load balancing by evenly distributing elements in the array among physical processors.

We tested the implementation of our parallel program for finding a spanning forest on graphs of three different edge densities: (1) dense graphs where $m = \frac{n^2}{4}$; (2) intermediate-density graphs where $m = n^{1.5}$; (3) sparse graphs where $m = \frac{3n}{2}$. Performance data is shown in Figure 4 for this problem with and without the usage of dynamic load balancing. Figure 4 shows that by using dynamic load balancing, our parallel program ran about 12 times faster than our parallel program without dynamic load balancing on dense graphs. On intermediate-density graphs, it was about 8 times faster. On sparse graphs, it was about 1.5 times faster. We would expect this type of behavior as dynamic load balancing provides more help as the graph gets denser.

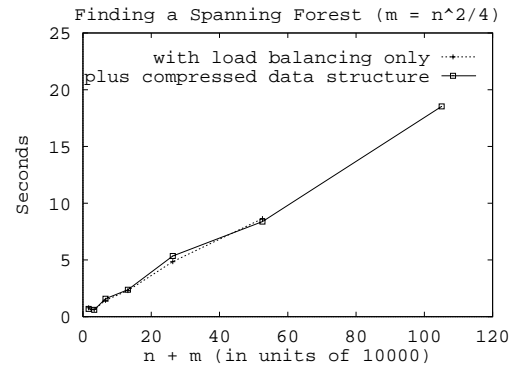
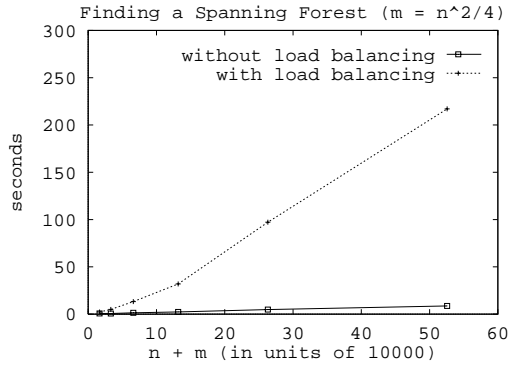
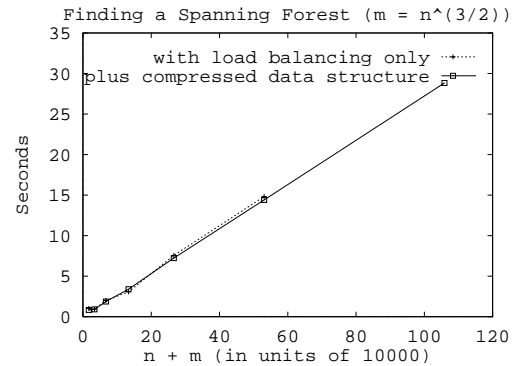
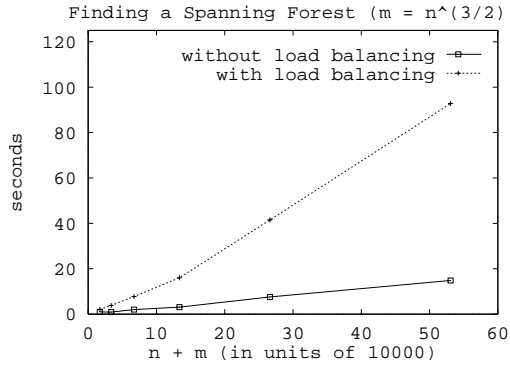
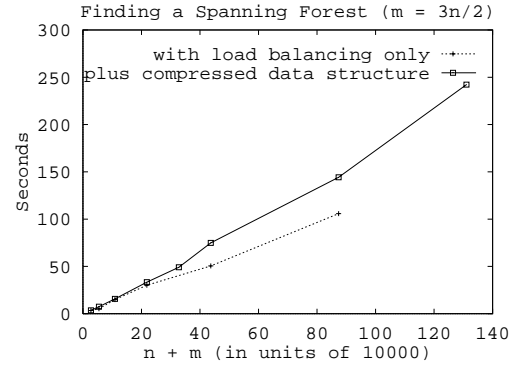
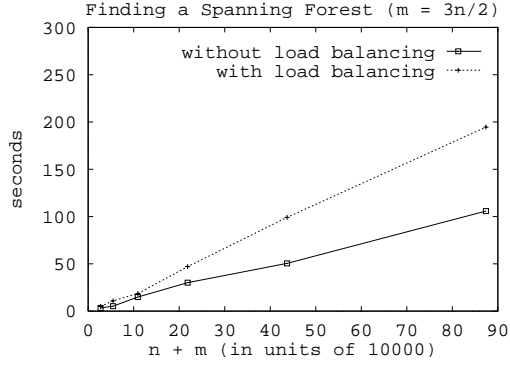


Figure 4: Illustrating the performance data for our parallel program for finding a spanning forest in graphs with and without dynamic load balancing.

Figure 5: Illustrating the performance data for our parallel program for finding connected components in graphs with and without the use of compressed data structure. Both programs were run using the same amount of memory per physical processor. Note that we can run inputs whose sizes are twice as large using the compressed data structure for graphs.

Compressed Data Structure A major goal of our implementation was to run inputs whose sizes are as large as possible. Since we have a limited amount of memory space per physical processor, we wanted to minimize the amount of space used by each edge without paying too much overhead in computation. It turns out that except for the case of representing a tree for finding an Euler tour, we can easily simulate the effect of having two processors handling one undirected edge by performing computations twice, one from each direction. Thus our program only allocated one processor to handle each edge. A side effect of this allocation scheme is that we had to write an expansion routine to convert this compressed representation into the tree format if we needed to build an Euler tour. In summary, our program first allocated vpr virtual processors per physical processor, where vpr is the least power of 2 that is greater than or equal to $\left\lceil \frac{m}{nproc} \right\rceil$. In the case when $2n > vpr \cdot nproc$ and a spanning tree format was needed, we doubled the value of vpr and called the expansion routine to transform the compressed data structure into the normal graph representation.

The performance data for running our program with and without the usage of compressed data structures for graphs to find a spanning forest are illustrated in Figure 5. Figure 5 shows that our program ran at about the same speed with or without the usage of compressed data structures on dense graphs and intermediate-density graphs. Note that by using compressed data structures, we could double the size of the largest graph we could handle. For sparse graphs, we had to pay a little overhead in using compressed data structures. Since the overhead was small, we decided to use compressed data structures for graphs though the code became a bit longer. Thus when allocating vpr virtual processors to each physical processor, we could run our programs on graphs with $vpr \cdot nproc$ nodes and $vpr \cdot nproc$ edges if we did not require the usage of a spanning forest in the program. We could run programs on graphs with $\frac{vpr \cdot nproc}{2}$ nodes and $vpr \cdot nproc$ edges if we had to use a spanning forest representation during the computation. Without the usage of compressed data structures, we could only run our programs on graphs with half the number of edges.

5.2 The Parallel Graph Algorithms Library

To build our parallel graph algorithms 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 and routines provided in the system library. The structure of the whole library is shown in Figure 6.

5.2.1 Routines in the System Library and the Kernel

We briefly describe the routines in the kernel. All of these routines are based on PRAM algorithms that run in $O(\log n)$ time for an input of size n . Although some of them are not theoretically optimal algorithms in that they perform $\Theta(n \log n)$ work, they are within

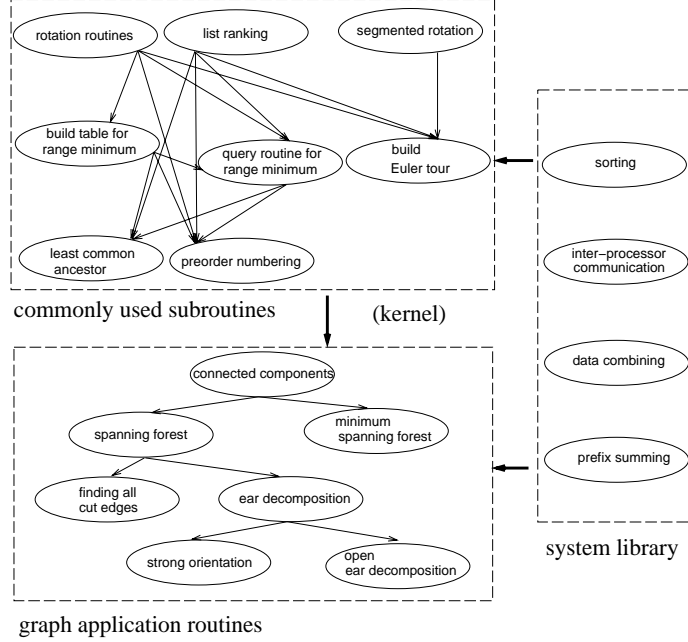


Figure 6: The structure of the routines we built for the parallel graph algorithms library. The kernel of the library will be used by the application routines. In our coding, we also use routines provided in the system library. An arrow from one node to another node means the routine at the tail of the arrow (upper) is used by the routine at the head of the arrow (lower).

an $O(\log n)$ factor of optimality, and they are very simple. These routines are as follows. (1) List ranking [15]. (2) Rotation. This routine rotates the data stored in a processor with ID i to the processor with ID $(i + d) \bmod P$, where d is an input to the routine and P is the number of PE's in the system. (3) Segmented rotation. We store data in each processor and partition the set of processors into sequences of consecutive segments. This routine rotates the data stored in each processor within each segment. Data within each segment are rotated in a way similar to the rotation routine described in (2). (4) Range minimum [36]. (5) Euler tour construction [36]. (6) Preorder numbering [36]. (7) Least common ancestor [36].

When implementing the above routines in the kernel without virtual processing [13], we also used the following routines that are provided in the system library. (1) Sorting. (2) Prefix sums. (3) Inter-processor communication. (4) Data combining. In our implementation of parallel algorithm with virtual processing, we also used the above routines. Since the MasPar does not provide virtual processing for these system routines, we coded and fine-tuned all of these routines with virtual processing except sorting. For sorting with virtual processing, we used the package developed in [29]. Since the sorting package in [29] can only handle the case when the number of virtual processors simulated by each physical processor is a power of two, our code inherited the same restriction.

5.2.2 Graph Application Routines

We implemented parallel algorithms for the following problems using the above kernel. (1) Spanning forest [2]. (2) Minimum cost spanning forest [2]. (3) Cut edges [31]. (4) Ear decomposition of a two-edge connected undirected graph [31]. (5) Open ear decomposition of a biconnected undirected graph [31] (this algorithm was not implemented in [13]). (6) Strong orientation of a two-edge connected undirected graph [31].

6 Performance Analysis

We tested our code by generating test graphs and measuring the performance of the code on these test graphs. In addition to testing our parallel code for the problems listed in Section 5.2.2, we also took the implementation of their corresponding sequential algorithms described in [13] and tested them on large inputs using SUN SPARC workstations. The corresponding sets of performance data were compared and studied. Note that a MasPar MP-1 PE is about 200 times slower than a SUN SPARC II and about 230 times slower than a SUN SPARC 10/41. Thus it is to be expected that the performance of our sequential programs will be faster than our parallel programs in some cases, though the speed-up of our parallel implementation was quite good, given the parameters of the MasPar MP-1. One noteworthy feature of our parallel implementation is that it could handle inputs whose sizes are much larger than the sizes of the input that can be handled by our sequential implementation.

The organization of this section is as follows. We first describe the method we used in generating test graphs. Then we describe the way we tested our programs and the curve-fitting scheme we performed on the sets of performance data. Finally, we analyze the performance data. Since no curve-fitting was performed in our earlier work without virtual processing [13], for completeness, we include the data from [13] in our performance analysis using curve fitting.

6.1 Generation of Test Graphs

We tested our programs using graphs of three different edge densities as described in Section 5.1.3 and [13]. For testing the code for finding a spanning forest and a minimum spanning forest, we generated test graphs from the class of random graphs $G_{n,m}$ as described in [13]. In addition, a random cost in the range from 0 to 99,999 (with repetition) on each edge, instead of from 0 to 999 as used in [13], was generated for testing the routine for finding a minimum spanning forest.

Test graphs with a given edge density, a given size, and a given property (e.g. biconnectivity) were generated using a similar method described in [13]. We generated a biconnected test 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 at 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 open ears of random lengths to connect all isolated vertices. To add a non-trivial open ear, we chose a random length l , $1 \leq l \leq x$, where x is the number of remaining isolated vertices. We randomly picked two non-isolated vertices u and v (without repetition). We then randomly permuted 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 were generated. A two-edge connected test graph with n vertices and m edges was generated in a similar fashion by “growing” ears that could possibly be cycles [13]. The test graphs for finding cut edges were generated as in [13].

6.2 Testing Scheme

For each size and sparsity, we generated four different test graphs. We ran each program on each test graph for 10 iterations and recorded the average of the 40 trials. The results are plotted in figures 7 – 18.

We had access to a MasPar MP-1 machine with 16,384 processors and 32 kilobytes of available memory per processor. (The other 32 kilobytes of memory in each processor was not available to us.) We were able to test all of our programs except the one for finding an open ear decomposition for the value of vpr up to 64. We were only able to run our parallel program for finding an open ear decomposition for the value of $vpr = 32$. Note that for testing dense graphs and intermediate-density graphs, we could run programs on graphs with $m = vpr \cdot 16,384$. For testing sparse graphs ($m = \frac{3}{2}n$), we could only run inputs with $m = \frac{3}{4} \cdot vpr \cdot 16,384$ since a tree data structure is required in the computation and we needed $2n = \frac{4}{3}m$ virtual processors to represent it. Our parallel programs for finding a spanning forest and for finding a minimum spanning forest used 24 kilobytes per physical processor when the value of vpr was 64. The rest of the programs used 32 kilobytes for the largest-sized inputs that we have tested. We spent about 2 months to obtain all of our performance data.

We ran the set of sequential algorithms implemented in [13] on a SUN SPARC 10/41 machine with 32 megabytes of memory and about 80 megabytes of swapping space on input sizes greater than 16,384. We tested the sequential programs on larger and larger inputs until either the programs complained that the usage of the memory is too much or we waited more than 1 day while there was only one active job running on the machine. For sparse graphs, our sequential programs ran out of available memory before we could obtain performance data that was worse than the corresponding parallel program. However, on dense graphs

and intermediate-density graphs, our parallel algorithms run much faster (in real CPU time) than their sequential counterparts. The likely reason is that we use a depth-first search in our sequential programs, which is a recursive program whose depth of recursion could be as large as the number of nodes in the graph.

Our sequential programs were implemented with the help of the graph package NETPAD [7] developed in Bellcore as described in [13]. NETPAD uses a lot of extra memory in creating a standard graph data structure. Thus we might save space by coding the sequential algorithms from scratch. We also note that the turn-around time (wall-clock time) for each of our sequential programs was very large when we used more than 80% of the main memory even if the system had only one job active, though our time measurement routine would report only a small fraction of the turn-around time. For example, for finding a minimum spanning forest sequentially on graphs with 300,000 edges, the time measurement routine reported a total usage of 110 seconds for 10 iteration of our program. However, the turn-around time was about 20 hours. We conjecture the reason might be that the architecture of SPARC 10/41 handles swapping poorly. We were unable to find better routines for measuring the performance of our sequential programs on the SPARC 10/41. For 5 of our 6 parallel programs, we were able to obtain sequential performance data that was worse than their parallel counterpart by testing large inputs. For the sequential algorithm for finding a minimum spanning forest, the turn-around time was too long when the input graph had more than 300,000 edges. As a result, we did not obtain further performance data for finding a minimum spanning forest such that we could observe the place where the sequential performance was worse than the parallel performance as shown in other programs. Overall, we spent more than 2 months in getting all of the performance data for our sequential programs. For all problems, we could handle input whose sizes are 4 to 5 times larger using our parallel code.

The performance data when the input size is within 16,384 is taken from [13]. In [13], sequential programs were run on a SPARC II workstation for input sizes up to 16,384; parallel programs without virtual processing were run on a MasPar MP-1 computer with 16,384 processors using 4 kilobytes of memory per PE.

6.3 Least-Squares Curve Fitting

We applied the least-squares fit package in Mathematica [38] to the data we obtained. We used the following method to find the fitted curves for our performance data. We first derived the theoretical asymptotical running time for our parallel program. For example, our code for finding a spanning forest in a graph with n nodes and m edges runs in $O(\frac{n}{p} \cdot \log^3 n)$ time using p processors since we used an $O(\log^2 n)$ time bitonic sorting routine in implementing global concurrent write operations. We first used Mathematica to find coefficients c_0 , c_1 , c_2 , c_3 and c_4 such that the function $c_0 + c_1 \cdot x + c_2 \cdot x \cdot \log x + c_3 \cdot x \cdot \log^2 x + c_4 \cdot x \cdot \log^3 x$ best fit

the set of experimental data that we obtained with virtual processing. For the data obtained without virtual processing, we used the function $c_0 + c_1 \cdot \log x + c_2 \cdot \log^2 x + c_3 \cdot \log^3 x$.

If any of the coefficients was negative, we forced the negative coefficient c_i with the largest integer i to be zero and perform the fitting once again. We iterated this process until all coefficients were not negative. We also performed the least-squares fit for performance data of the sequential programs when the amount of memory used in the program was within the capacity of the main memory.

To test the goodness of the curve we obtained, we computed the *average error* as the square root of $\frac{1}{k} \cdot \sum_{i=1}^k \left(\frac{y_i - f(x_i)}{f(x_i)} \right)^2$, where k is the number of experimental data points, f is the function that describes the fitted curve and y_i is the experimental value when the input size is x_i .

6.4 Analysis

In Section 6.4.1 through Section 6.4.6, we present the performance of our code for each of our six graph problems. The data for programs without virtual processing is taken from [13]. In the following, x denotes the size of the input and x' is the size of the input in units of 10,000. In interpreting the following data, note that we present the fitting curves in terms of x when no virtual processors are used and in terms of x' when virtual processors are used. There is a further compression by a factor of 2 due to the compressed data structure when virtual processors are used. The function value of each fitted curve is the running time in seconds.

6.4.1 Finding a Spanning Forest

For the parallel implementation, we modified the CRCW PRAM algorithm in [2] 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 detected to be 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. The sequential algorithm that we implemented is the simple linear time depth-first search algorithm.

The performance data without and with virtual processing are shown in Figures 7 and 8 respectively. The fitted curves for the parallel performance data without virtual processing are $0.0003 \log^3 x$ (with 8.9% average error), $0.000095 \log^3 x + 0.11$ (with 4.2% average error), and $0.00011 \log^3 x + 0.11$ (with 5.5% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data are $0.000023x$ (with 5.7% average error), $0.000011x$ (with

4.4% average error), and $0.00001x$ (with 4.1% average error). The corresponding fitted curves for the parallel performance data with virtual processing are $0.0014x' \log^3 x' + 1.32x' + 1.41$ (with 7.1% average error), $0.0000091x' \log^3 x' + 0.27x' + 0.028$ (with 3.2% average error), and $0.000074x' \log^3 x' + 0.15x' + 0.45$ (with 9.1% average error). The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.17x'$, $0.17x'$, and $0.15x'$.

6.4.2 Finding a Minimum Spanning Forest

For the parallel implementation, we modified the algorithm in [2] 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 edge with exactly one end point in the set using the concurrent write operation. This edge determines which other set of vertices is to be merged with its set. Once the merge is completed, the edge that caused the merging is marked as one of the edges in the minimum cost spanning forest. For sequential implementation, we implemented the $O(n + m \log n)$ -time Kruskal's algorithm [35] for finding a minimum cost spanning forest. Although faster algorithms are known for this problem, we implemented Kruskal's algorithm for its simplicity.

The performance data without and with virtual processing are shown in Figures 9 and 10 respectively. The fitted curves for the parallel performance data without virtual processing are $0.00033 \log^3 x + 0.12$ (with 4.5% average error), $0.00022 \log^3 x + 0.095$ (with 6.3% average error), and $0.00021 \log^3 x + 0.058$ (with 9.6% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data are $0.0000057x \log x$ (with 9.9% average error), $0.0000014x \log x + 0.0000042x + 0.011$ (with 10.4% average error), and $0.00000093x \log x + 0.0000044x + 0.0099$ (with 8.9% average error). The corresponding fitted curves for the parallel performance data with virtual processing are $0.0015x' \log^3 x' + 0.78x' + 2.46$ (with 14% average error), $0.00037x' \log^3 x' + 0.68x' + 0.11$ (with 0.00021x' log^3 x' + 0.67x' (with 7.2% average error)). The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.1x' \log x' + 5.44$, $0.056x' \log x' + 1.78$, and $0.049x' \log x' + 1.22$.

6.4.3 Finding All Cut Edges

Our parallel implementation is based on [31]. We first obtained 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 a proper ancestor of v . This can be determined by using the Euler

tour technique and the range minimum queries. For sequential implementation, we used a linear time algorithm for finding all cut edges in the graph based on depth-first search [31].

The performance data without and with virtual processing are shown in Figures 11 and 12 respectively. The fitted curves for the parallel performance data without virtual processing are $0.0004 \log^3 x + 0.013$ (with 8.1% average error), $0.00014 \log^3 x + 0.18$ (with 3.6% average error), and $0.00015 \log^3 x + 0.17$ (with 3.8% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data are $0.000023x + 0.00049$ (with 5.7% average error), $0.000011x + 0.00035$ (with 4.0% average error), and $0.00001x + 0.000051$ (with 5.4% average error). The corresponding fitted curves for the parallel performance data with virtual processing are $0.0019x' \log^3 x' + 1.44x' + 1.59$ (with 10.2% average error), $0.00018x' \log^3 x' + 0.61x' + 0.27$ (with 2.9% average error), and $0.00043x' \log^3 x' + 0.49x' + 0.73$ (with 3.7% average error). The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.22x'$, $0.18x'$, and $0.16x'$.

6.4.4 Finding an Ear Decomposition

For the parallel implementation, we used the PRAM parallel algorithm in [31] for finding an ear decomposition on a 2-edge connected graph by calling the sorting routine, routines in the kernel and the routine for finding a spanning forest. For sequential implementation, we used a linear time algorithm for finding an ear decomposition based on depth-first search [31].

The performance data without and with virtual processing are shown in Figures 13 and 14 respectively. The fitted curves for the parallel performance data without virtual processing are $0.0004 \log^3 x + 0.021$ (with 7.7% average error), $0.00014 \log^3 x + 0.19$ (with 3.5% average error), and $0.00015 \log^3 x + 0.18$ (with 3.8% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively. The corresponding fitted curves for the sequential performance data are $0.000093x$ (with 13.4% average error), $0.000084x$ (with 4.9% average error), and $0.000083x$ (with 4.9% average error). The corresponding fitted curves for the parallel performance data with virtual processing are $0.0014x' \log^3 x' + 1.43x' + 0.36$ (with 5.0% average error), $0.00036x' \log^3 x' + 0.33x' + 0.2$ (with 11.0% average error), and $0.00064x' \log^3 x' + 0.21x' + 0.91$ (with 5.9% average error). The corresponding fitted curves for the sequential performance data when the data is within the main memory are $0.57x'$, $0.72x'$, and $0.68x'$.

6.4.5 Finding an Open Ear Decomposition

For the parallel implementation, we used the PRAM algorithm in [31] for finding an open ear decomposition. This routine is obtained by modifying the ear decomposition algorithm

mentioned in the previous section. The sequential ear decomposition algorithm mentioned in the previous section [31] also finds an open ear decomposition on a biconnected graph.

The performance data without and with virtual processing are shown in Figures 15 and 16 respectively. The fitted curves for the parallel performance data without virtual processing are $0.00041 \log^3 x + 0.26$ (with 5.8% average error), $0.00033 \log^3 x + 0.27$ (with 7.8% average error), and $0.00017 \log^3 x + 0.44$ (with 4.8% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively, where x is the size of the input and the function value is the running time in seconds. The corresponding fitted curves for the parallel performance data with virtual processing are $0.0017x' \log^3 x' + 1.59x' + 0.24$ (with 13.9% average error), $0.0014x' \log^3 x' + 1.03x' + 0.36$ (with 9.0% average error), and $0.0024x' \log^3 x' + 0.057x' \log x' + 0.96x' + 0.5$ (with 7.1% average error). Note that the sequential performance data for finding an open ear decomposition is the same as the sequential performance data for finding an ear decomposition. We will not restate them here.

6.4.6 Finding a Strong Orientation

For the parallel implementation, we first obtained an ear decomposition for the input graph. Then we directed the edges of each ear so that each ear forms a directed path or a directed cycle. Observe that the ear decomposition algorithm first obtained rooted spanning tree T . The edges in an ear are of the form $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k), (v_k, u_r), (u_r, u_{r-1}), (u_{r-1}, u_{r-2}), \dots, (u_2, u_1)$, 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 directed every non-tree edge (u, v) from u to v where u has a smaller ID than that of v . Then we assigned directions to tree edges in such a way that the edges in an ear formed a directed path or directed cycle and the first two ears together formed a directed cycle. For sequential implementation, we used a linear time algorithm for finding a strong orientation based on a recursive version of depth-first search [34].

The performance data without and with virtual processing are shown in Figures 17 and 18 respectively. The fitted curves for the parallel performance data without virtual processing are $0.00039 \log^3 x + 0.033$ (with 7.7% average error), $0.00015 \log^3 x + 0.18$ (with 4.6% average error), and $0.00016 \log^3 x + 0.17$ (with 4.1% average error). for sparse graphs, intermediate-density graphs and dense graphs respectively, where x is the size of the input and the function value is the running time in seconds. The corresponding fitted curves for the sequential performance data are $0.000025x$ (with 5.6% average error), $0.000016x + 0.00028$ (with 4.7% average error), and $0.000015x + 0.0005$ (with 7.3% average error). The corresponding fitted curves for the parallel performance data with virtual processing are $0.0021x' \log^3 x' + 1.39x' + 1.78$ (with 5.8% average error), $0.00012x' \log^3 x' + 0.42x' + 0.032$ (with 2.6% average error), and $0.00058x' \log^3 x' + 0.23x' + 0.89$ (with 5.1% average error). The corresponding fitted curves for the sequential performance data when the data is within the main memory are

	$m = 3n/2$		$m = n^{3/2}$		$m = n^2/4$	
	no vpr $m = 8,191$ (seconds)	$vpr = 16$ $m = 262,142$ (seconds)	no vpr $m = 8,191$ (seconds)	$vpr = 16$ $m = 262,142$ (seconds)	no vpr $m = 8,191$ (seconds)	$vpr = 16$ $m = 262,142$ (seconds)
Spanning Forest	1.01	74.86	0.41	7.23	0.39	5.35
Minimum Spanning Forest	1.05	51.97	0.73	18.58	0.70	19.69
All Cut Edges	1.17	83.36	0.61	17.92	0.57	15.85
Ear Decomposition	1.19	72.54	0.60	11.32	0.58	8.71
Open Ear Decomposition	1.47	90.35	1.11	33.69	0.94	33.50
Strong Orientation	1.20	75.35	0.63	11.61	0.60	9.06

Table 1: Performance data for our parallel programs with and without virtual processing. The data for parallel programs without virtual processing is from [13].

$0.31x'$, $0.25x'$, and $0.22x'$.

6.5 Overhead for Implementing Virtual Processors

We compared the amount of time used by our parallel programs with and without virtual processing. The performance data is shown in Table 1. Note that we ran 5 of our 6 programs for the value of vpr up to 64 using no more than half of the available memory in the system. The one program that we could run only up to the value of $vpr = 32$, was the open ear decomposition routine. Also, when $vpr = 32$, our code for open ear decomposition could not handle inputs of size $32 \cdot 16,384$ on sparse graphs. (See Section 6.2 for details.) Hence in Table 1, we use $vpr = 16$ to show the performance of our parallel code when the virtual processors simulated in each physical processor were all active. The performance data with no virtual processing is from [13]. Our implementation of parallel algorithms with virtual processing had excellent speed-ups on dense graphs and intermediate-density graphs in relation to the implementation without virtual processing. For example, for finding an ear decomposition on dense graphs, we used 15 times more CPU time with virtual processing while handling graphs that were 32 times larger. For sparse graphs, the overhead was fairly large. The reason might be that for sparse graphs, using virtual processors increased the degree of concurrency when concurrent read or write is used. Since we could not offset it by the use of dynamic load balancing, our implementation had a big overhead on sparse graphs. We also note that the overhead for implementing the open ear decomposition algorithms is about twice as large as the overhead for implementing other algorithms.

7 Conclusion and Future Work

We have implemented a set of parallel algorithms for undirected graphs on the MasPar MP-1 to handle sizes of the input that are larger than the number of available physical processors. We tested our parallel programs on inputs whose sizes were up to 64 times larger than the number of physical processors and compared their performance with corresponding

sequential programs. Note that by using the full configuration of the current machine, we can simulate up to 128 virtual processors per physical processor. However, sharing the machine with other users limited us to use only half of the available memory in each processor. Thus if the full machine had been available, we could have run our programs on graphs with one million nodes and two million edges.

We note the following observations.

- By using the high-level structure of the PRAM algorithms as building blocks, our coding and debugging effort was relatively small. We wrote more than 12,000 lines of parallel code for the set of parallel graph algorithms that we implemented with virtual processing. All of the work reported here (include testing) was done within one year. Note that 4,000 lines of parallel code were written in 12 weeks in [13] for the same set of parallel programs with no virtual processing. We consider our strategy for implementing parallel graph algorithms to be promising.
- We examined the variation in data we obtained on the four different test graphs of a given size and a given edge density. Most of the data points ($> 90\%$) were within 7% of their average values. Less than 5% of the data points were more than 15% away from their average values.
- We compared the experimental data points with the computed points on the fitted curves. For programs without virtual processing, the average error for fitted curves on sparse graphs was usually larger than the average error for fitted curves on dense graphs and intermediate-density graphs. The average error was about 10% for all data sets with virtual processing. We also note that the average error for fitted curves on dense graphs and intermediate-density graphs without virtual processing is almost the same whether we fit the data with functions dominated by $\log^3 x$ or dominated by $\log^2 x$, though we used functions dominated by $\log^3 x$ in the paper. However, the average error for sparse graphs is about twice as large if we use functions dominated with $\log^2 x$ instead of $\log^3 x$. With virtual processing, the best fit in all cases was obtained when the function was dominated by $x' \log^3 x'$.

Our fitted curves fit quite well on the experimental performance data. The fitted curves showed that the dominant term in our parallel code was $\log^3 n$ both with and without virtual processing. We conjecture the reason might be that our graph algorithms usually compute by performing $O(\log n)$ iterations and if each iteration takes $\Theta(\log^2 n)$ time, the overall time complexity is $O(\log^3 n)$.

- Sequential implementations usually performed badly when a fraction of their data were placed out of the main memory. Note that our sequential programs used extra memory because we used NETPAD. Thus the biggest size inputs that one can run on a SPARC 10/41 would be somewhat larger than what we have shown here if a more efficient

coding of the graph data structure is used.

- Although each MasPar MP-1 PE is much slower than the SPARC workstation, we found that in most of the cases, parallel programs in fact runs faster in real time compared to sequential programs. In particular, our parallel programs were much faster on dense graphs and intermediate-density graphs than on sparse graphs. We traced our parallel program for finding a spanning forest and noticed that by using our dynamic load balancing technique, the performance of a concurrent read or write was not too bad on a dense graph compared to the performance of the same operations on a sparse graph. Recall that our algorithm for finding a spanning forest obtained a spanning forest by repeatedly growing forests in parallel in a loop until the size of any tree in the current forest could not be expended. For dense graphs, the parallel algorithms terminated in fewer iterations than on sparse graphs. Thus the running time was much smaller on dense graphs than on sparse graphs. Our parallel code can also handle much larger inputs than our sequential code.
- We found that our sequential program for finding a spanning forest used about 45 megabytes of memory on the largest-sized inputs. Our parallel program used no more than 24 kilobytes of memory per physical processor on inputs whose sizes were more than 4 times larger than the size of the largest inputs for the sequential program. Since there are 16,384 physical processors in the MasPar MP-1, the total memory used in our parallel program was no more than 384 megabytes. Hence we used about 8 times more memory in our parallel programs while we could run inputs whose sizes were 4 times larger than the largest input size on the SPARC 10/41. In most cases, when testing the largest size inputs, our parallel code ran faster than their sequential counterparts on dense graphs and intermediate-density graphs even when the input size was 4 times larger.
- The current architecture of the MasPar MP-1 is not adequate to run programs that require a lot of memory per physical processor. The limitation of only having 64 kilobytes of memory per physical processor prevents us from running inputs of larger sizes. No support from the operating system for using virtual memory also inhibits us from running larger examples. It is reported in [27] that the new MasPar MP-2 upgrades the raw computation power of each individual processor while keeping its communication hardware and limitation of memory space unchanged. For our application, we feel that the amount of memory in each processor should be increased and the bandwidth of the communication channel should be enlarged before the upgrading of the processor computation power.

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

- The lack of a good graph manipulation package like NETPAD for handling large graphs

makes it difficult to debug our programs. In [13], NETPAD was able to help the debugging and testing of our parallel implementation after we built an interface to use it on the MasPar. In our current implementation, the sizes of the graphs became too large for NETPAD to handle. Work should be done for graph manipulation (especially visualization) packages on large graphs.

- Our current implementation requires that vpr , the number of virtual processors simulated by each physical processor, be a power of 2 because of a bitonic sorting package [29] that we are using. We would like to replace this sorting package by a sorting routine that can simulate any number of virtual processors per physical processor.
- We note that our current implementation has a very large overhead on sparse graphs. More work has to be done to improve the running time on graphs that are very sparse.

References

- [1] R. Anderson and J. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 168–177, 1992.
- [2] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Tran. on Computers*, pages 1258–1263, October 1987.
- [3] T. Blank. The MasPar MP-1 architecture. In *Proc. of COMPCON Spring 90 – 35th IEEE Computer Society International Conference*, pages 20–40, 1990.
- [4] G. E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, M.I.T., October 1989.
- [5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3th ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, 1991.
- [6] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.
- [7] N. Dean, M. Mevenkamp, and C. L. Monma. NETPAD: An interface graphics system for network modeling and optimization. In *Proc. Computer Science & Operations Research: New Developments in their Interfaces*, pages 231–243. Pergamon Press, 1992.
- [8] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10:657–675, 1981.
- [9] B. Dixon and A. K. Lenstra. Factoring integers using SIMD sieves. Manuscript, 1992.
- [10] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. Manuscript, 1992.
- [11] W. Hightower, J. Prins, and J. Reif. Implementations of randomized sorting on large parallel machines. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 158–167, 1992.
- [12] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, 1986.
- [13] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In *AMS Proc. of DIMACS Workshop on Computational Support for*

Discrete Math., to appear. Also available as TR-92-38, Dept. of Comp. Sci., Univ. of Texas at Austin.

- [14] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [15] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. North Holland, 1990.
- [16] B. W. Kernighan and D. M. Ritchie. *The C Programming language*. Prentice Hall, Englewood Cliffs, NJ, 1988. Second Edition.
- [17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
- [18] C. Leiserson, Z. S. Abuhamdeh, D. Douglas, C. R. Feynmann, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S-W Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 272–287, 1992.
- [19] MasPar Computer Co. *MasPar Parallel Application Language (MPL) Reference manual*, version 2.0 edition, March 1991.
- [20] MasPar Computer Co. *MasPar Parallel Application Language (MPL) User Guide*, version 2.0 edition, March 1991.
- [21] MasPar Computer Co. *MasPar System Overview*, version 2.0 edition, March 1991.
- [22] MasPar Computer Co. *MasPar Data Display Library (MPDDL) Reference manual*, version 3.0, rev. a6 edition, July 1992.
- [23] MasPar Computer Co. *MasPar Parallel Application Language (MPL) Reference manual*, version 3.0, rev. a3 edition, July 1992.
- [24] MasPar Computer Co. *MasPar Parallel Application Language (MPL) User Guide*, version 3.1, rev. a3 edition, November 1992.
- [25] B. Narendran and P. Tiwari. Polynomial root-finding: Analysis and computational investigation of a parallel algorithm. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 178–187, 1992.
- [26] R. Pickering and J. Cook. A first course in programming the DECmpp/Sx. Technical report, para//lab, Dept. of Informatics, Univ. of Bergen, N-5020 Bergen, Norway, 1993. Series of Parallel Processing: A Self-Study Introduction.

- [27] L. Prechelt. Comparison of MasPar MP-1 and MP-2 communication operations. Technical Report 16/93, Institute für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Germany, April 1993.
- [28] L. Prechelt. Measurements of MasPar MP-1216A communication operations. Technical Report 01/93, Institute für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1993.
- [29] J. F. Prins and J. A. Smith. Parallel sorting of large arrays on the MasPar MP-1. In *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 59–64, 1990.
- [30] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.
- [31] 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-Kaufmann, 1993.
- [32] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan-Kaufmann, 1993.
- [33] J. T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2:484–521, October 1980.
- [34] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.
- [35] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM Press, Philadelphia, PA, 1983.
- [36] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.
- [37] U. Vishkin. Structural parallel algorithmics. In *Proc. 18th ICALP*, volume LNCS #510, pages 363–380. Springer-Verlag, 1991.
- [38] S. Wolfram. *MathematicaTM A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

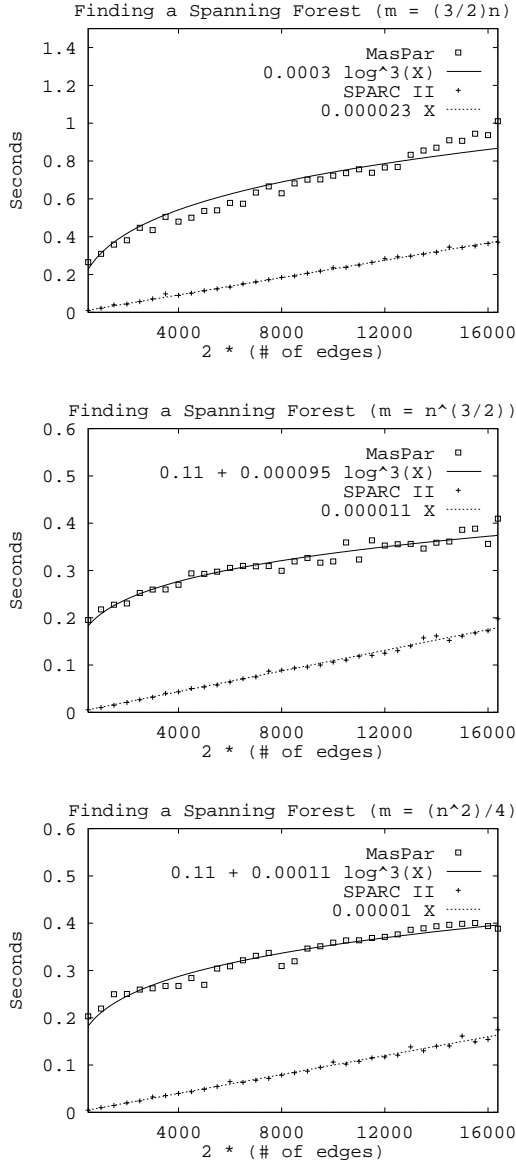


Figure 7: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding connected components without virtual processing.

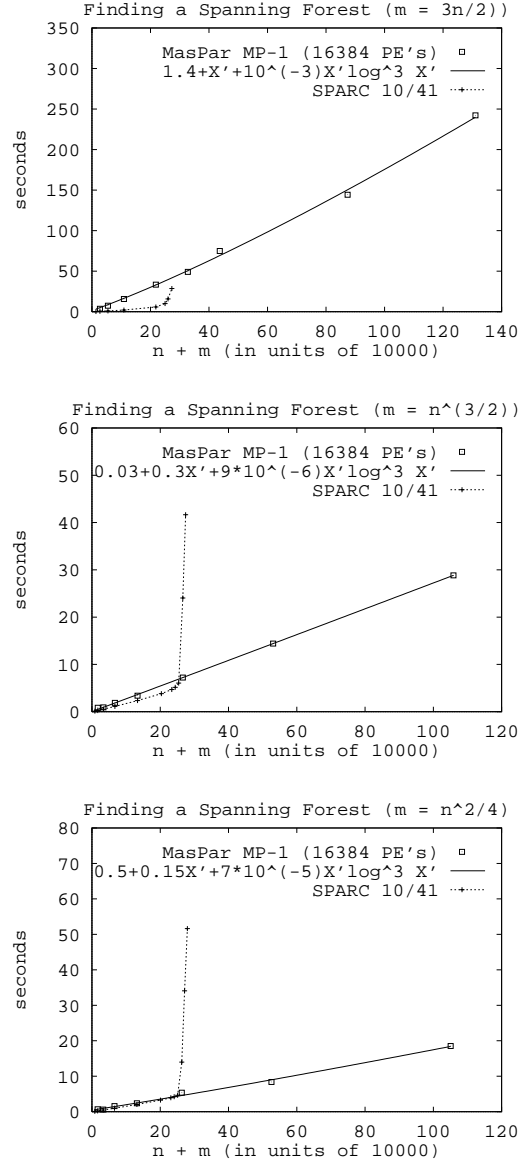


Figure 8: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding a spanning forest with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $0.17x$, $0.17x$, and $0.15x$, respectively, from the top to the bottom.

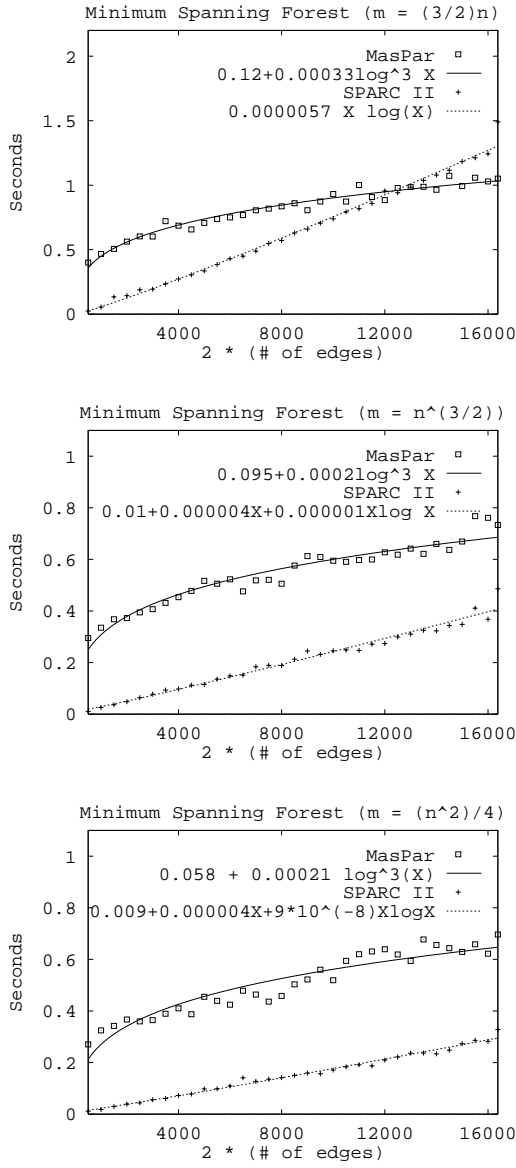


Figure 9: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding a minimum spanning forest without virtual processing.

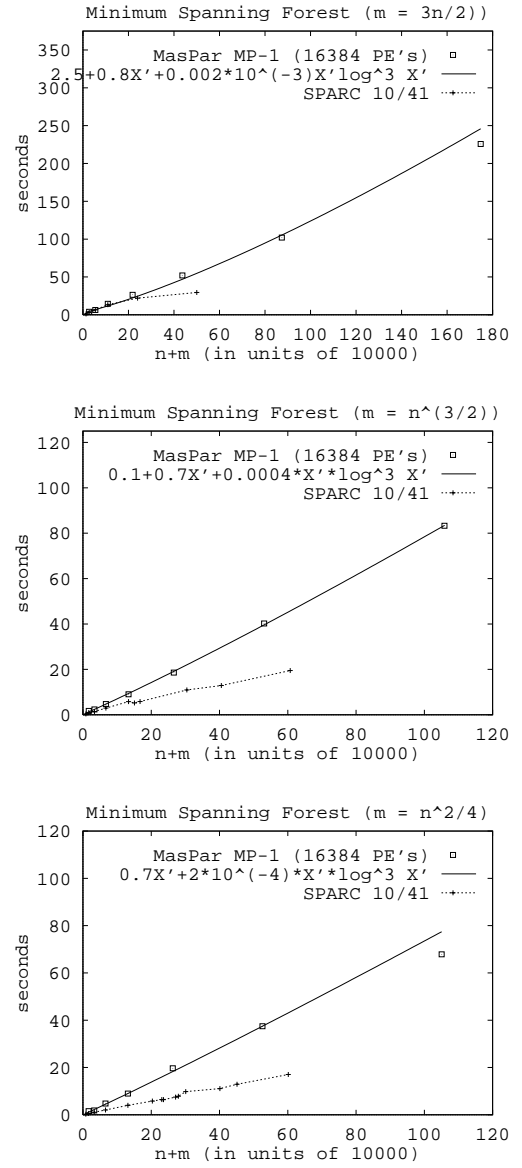


Figure 10: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding a minimum spanning forest with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $5.44 + 0.20x \log x$, $1.78 + 0.11x \log x$, and $1.22 + 0.10x \log x$, respectively, from the top to the bottom.

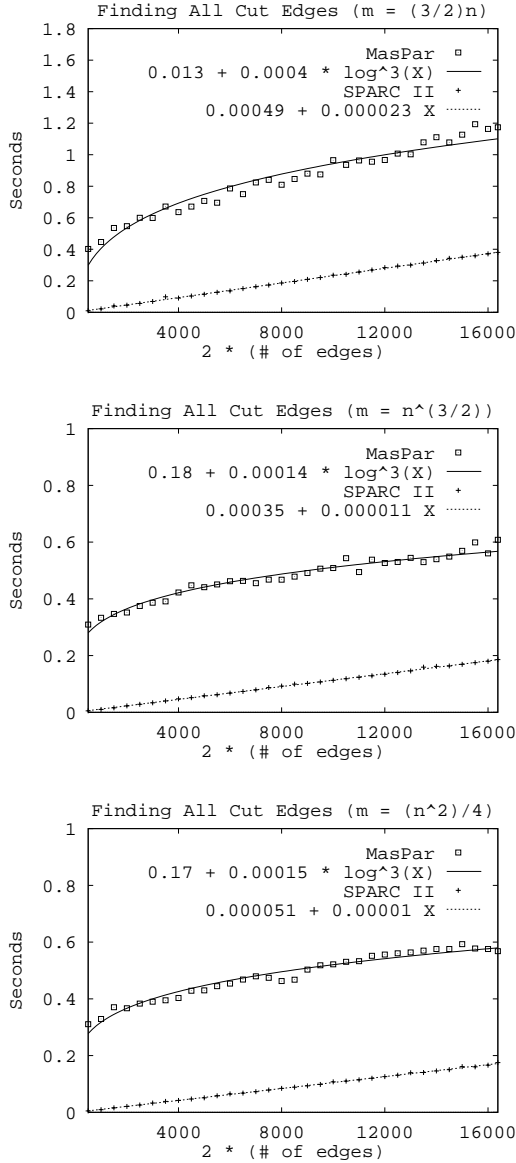


Figure 11: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding all cut edges without virtual processing.

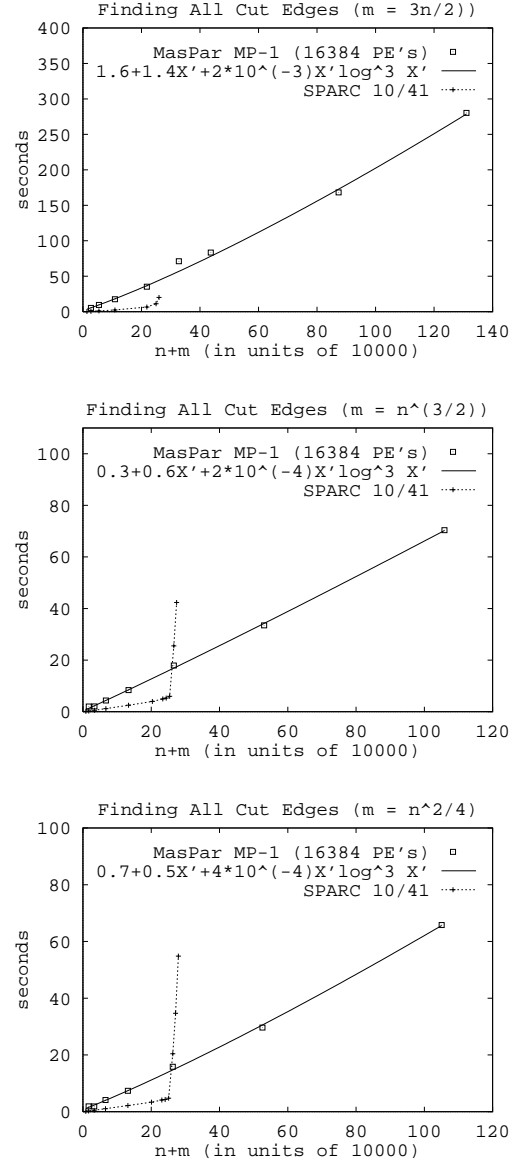


Figure 12: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding all cut edges with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $0.22x$, $0.18x$, and $0.16x$, respectively, from the top to the bottom.

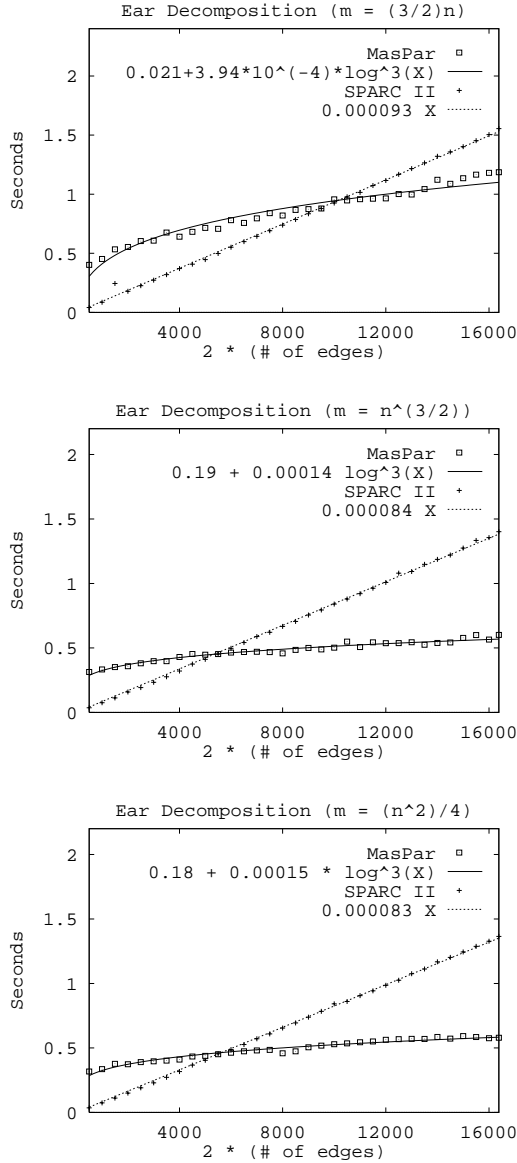


Figure 13: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding an ear decomposition on a two-edge connected graph without virtual processing.

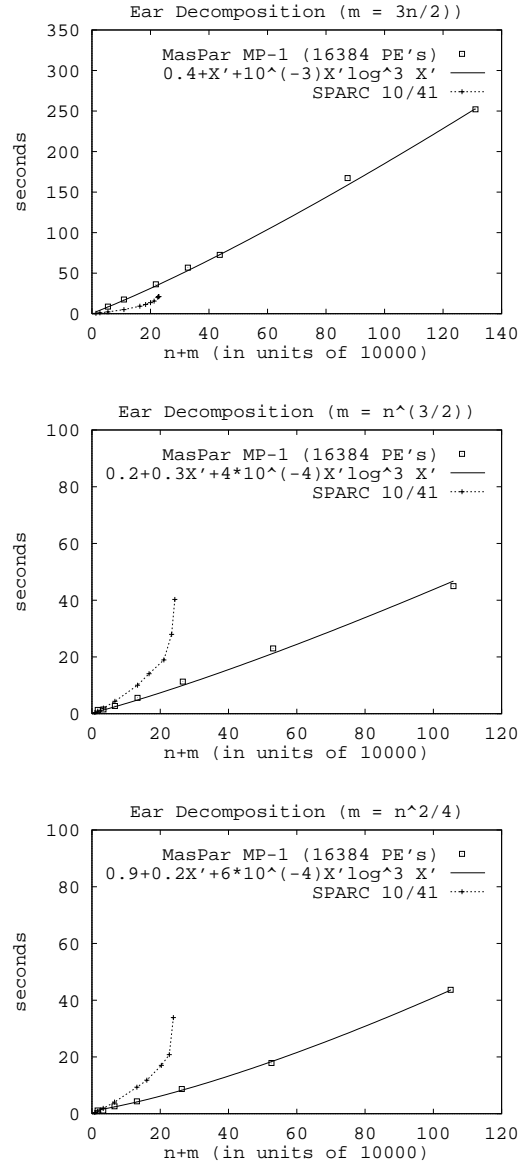


Figure 14: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding an ear decomposition with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $0.57x$, $0.72x$, and $0.68x$, respectively, from the top to the bottom.

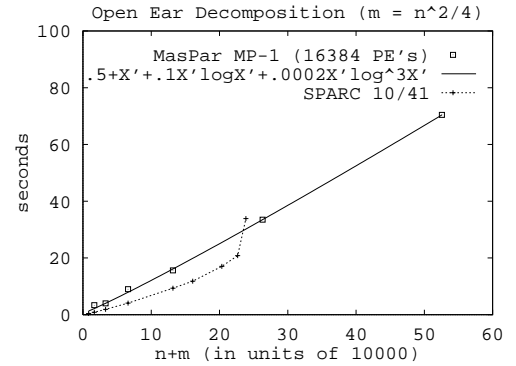
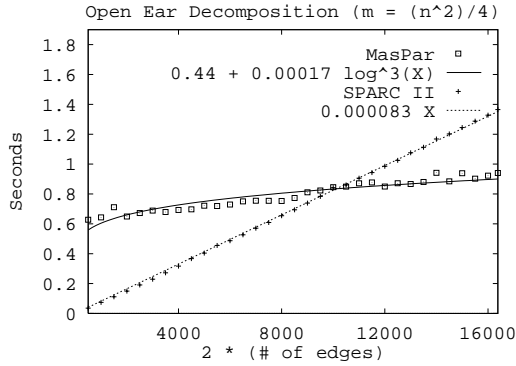
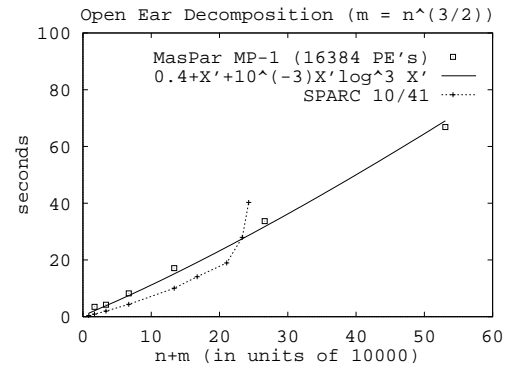
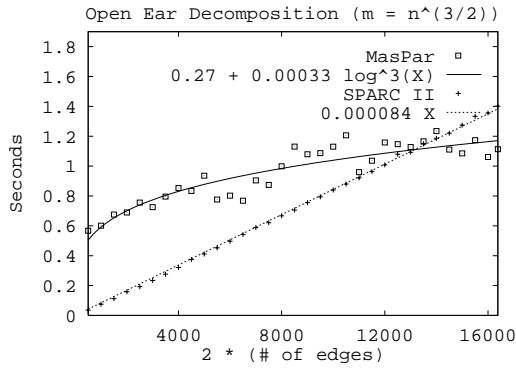
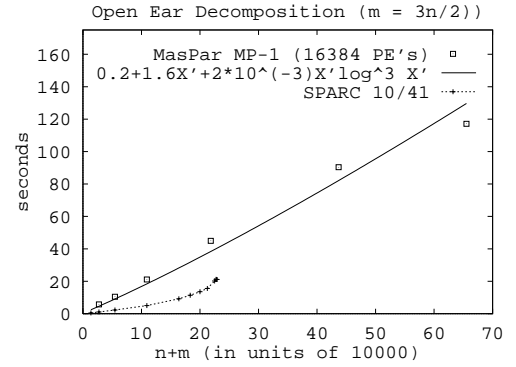
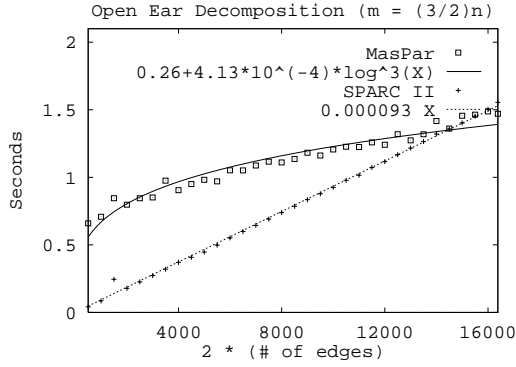


Figure 15: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding an open ear decomposition on a biconnected graph without virtual processing.

Figure 16: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding an open ear decomposition with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $0.57x$, $0.72x$, and $0.68x$, respectively, from the top to the bottom.

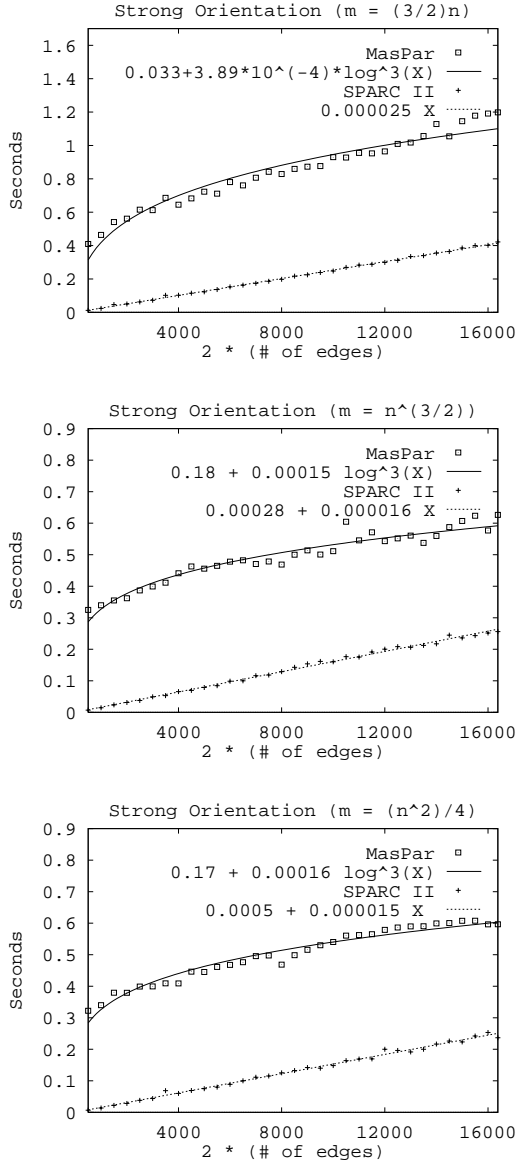


Figure 17: Relative performance of the sequential program on a SPARC II workstation and the parallel program on the MasPar MP-1 for finding a strong orientation on a two-edge connected graph without virtual processing.

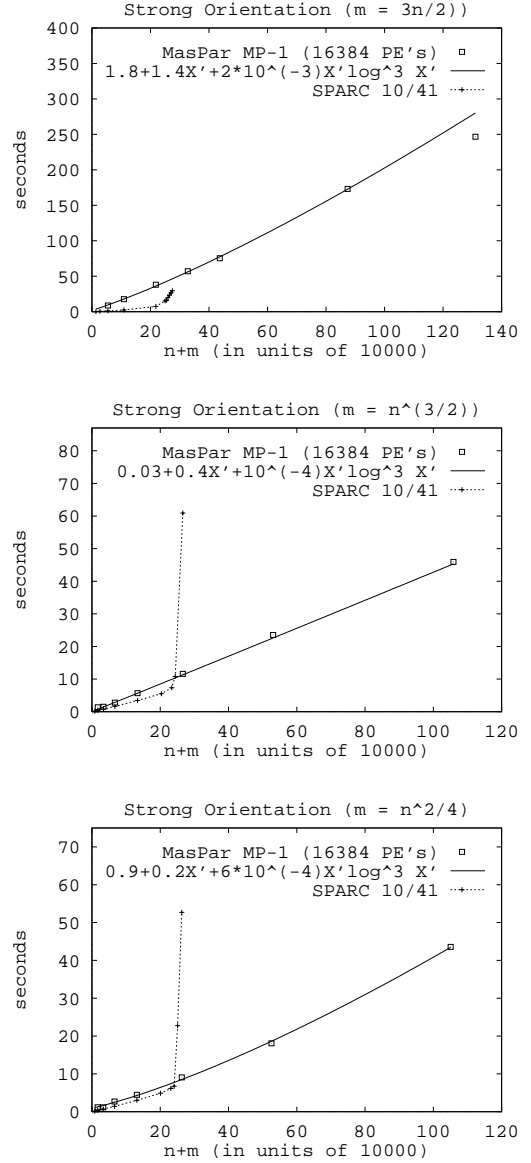


Figure 18: Relative performance of the sequential program on a SUN SPARC 10/41 workstation and the parallel program on the MasPar MP-1 for finding a strong orientation with virtual processing. The least-squares-fit curves for the performance data of the sequential program when $< 80\%$ of the main memory are used are $0.31x$, $0.25x$, and $0.22x$, respectively, from the top to the bottom.