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

Tsan-sheng Hsu[*†]
Inst. of Information Science
Academia Sinica, Nankang
Taipei 115, Taiwan, ROC
tshsu@iis.sinica.edu.tw

Vijaya Ramachandran[*‡]
Dept. of Computer Sciences
Univ. of Texas at Austin
Austin, TX 78712, USA
vlr@cs.utexas.edu

Nathaniel Dean[§]
S/W Production Research
AT&T Bell Labs.
Murray Hill, NJ 07960, USA
nate@research.att.com

Abstract

We describe our implementation, with virtual processing, of several parallel graph algorithms on a 16,384-processor MasPar MP-1. We present extensive test data on our code.

1 Introduction

This paper describes an on-going project for implementing parallel graph algorithms on the massively parallel machine MasPar MP-1. The main focus of our project has been the implementation of parallel algorithms that require large amounts of non-oblivious memory accesses. The first phase of this project is reported in [14].

There has been a fair amount of recent work on implementing combinatorial algorithms on massively parallel machines [3, 10, 15, 12, 24]. There also has been work reported on implementing combinatorial algorithms on a vector super computer [10, 27, 29], of which [10] implements parallel graph algorithms on a massively parallel machine. In [10], the problem of finding connected components is extensively studied by applying various fine-tuning techniques on several algorithms to achieve best performance on several classes of graphs. Our work has a different focus: we address the issue of implementing an extensible parallel graph algorithms library, which includes the problem of finding connected components. We developed general implementation and fine-tuning techniques without expending too much effort on optimizing each individual routine.

In [14], we reported the implementation of several parallel graph algorithms on the MasPar MP-1 [2] using the parallel language MPL [22] which is an extension of the ANSI C language. The MPL language requires the user to specify the physical organization of the processors used in the program. Our implementation in [14] 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. (The same data allocation scheme is used in the more recent work reported in [10].) 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. For the MasPar MP-1 that we used, *nproc* = 16,384 processors. In the current paper, we report the second phase of this work. By using better data allocation schemes and virtual processing, we implemented these algorithms to handle inputs of size greater than 16,384.

Over the past decade there has been a large amount of work in the theory of efficient, highly parallel graph algorithm design [18, 19, 20, 28]. Parallel algorithms that run in polylog time with a linear or sub-linear number of processors have been developed for several fundamental problems on undirected graphs including connected components and spanning forest[%] [1, 5, 7, 9, 10, 11, 17], minimum spanning forest (MSF) [1, 5, 6], ear decomposition and 2-edge connectivity [21, 23, 25], open ear decomposition and biconnectivity [21, 23, 25, 31], triconnectivity [8] and planarity

*Supported by NSF Grant CCR-90-23059 and Texas Advanced Research Projects (TARP) Grant 003658480.

†Also supported by an IBM graduate fellowship while this author was with Univ. of Texas at Austin.

‡Also supported by TARP Grant 003658386.

§ Work performed while this author was at Bellcore, Morristown, NJ.

%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.

1063-7133/95 $4.00 © 1995 IEEE

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

[26]. All of these algorithms (with the exception of some algorithms for MSF) have the additional feature that they serialize into linear-time sequential algorithms. However, these algorithms are quite different from earlier depth-first search based linear time algorithms [30] in that they are very modular in structure. The algorithm for ear decomposition calls subroutines for several basic problems such as connected components, spanning forest, the Euler tour technique on trees [31], least common ancestors in trees [31] and range minima [31]. More complex algorithms, such as those for triconnectivity and planarity, call subroutines for open ear decomposition, in addition to calling subroutines for more basic problems. Thus an implementation of parallel algorithms for undirected graphs would have to proceed in a bottom-up fashion, starting with an implementation of basic primitives, and successively building up to more complex algorithms.

Our implementation followed the above strategy. We have implemented a large number of parallel algorithms as listed below, grouped into three main categories. We have incorporated virtual processing in all of the algorithms listed below:

1. *Basic Primitives:* Prefix sums (scans) and various routing routines. These routines are provided in MPL as system primitives without virtual processing, but we needed to code in virtual processing.
2. *Kernel:* List ranking, rotations, Euler tour on trees, tree functions such as preorder, least common ancestors in trees and range minima. Although these are not graph algorithms, they are commonly used subroutines in the graph algorithms that we implemented.
3. *Graph Algorithms:* Connected components and spanning forest, minimum spanning forest, ear decomposition, cut edges, strong orientation and open ear decomposition.

In this paper, we describe our implementation, with virtual processing, of the graph algorithms listed above. In a companion paper [13] we describe our implementation and algorithm choices for the basic primitives and kernel problems.

We tested all of our code on random graphs of various edge densities. We performed a least squares fit on our data, both with and without virtual processing (the results in [14] do not include any least squares fit). We also compared the performance of our code for virtual processing with the data reported in our earlier work [14] without virtual processing.

The rest of the paper is organized as follows. Section 2 describes the mapping of virtual processors on to the MasPar MP-1. Section 3 describes our implementation strategy and two optimizing features that we incorporated. Section 4 describes our testing scheme, including the least squares fit, and our code for the six graph algorithms, and provides extensive data. Section 5 provides some concluding remarks.

Due to space limitation, some details of our implementation and performance data are omitted in this abstract. They can be found in the full paper [16].

2 Mapping Strategy

Let *nproc* and *vnproc* be the number of physical processors and virtual processors, respectively. For the MasPar MP-1 we used, $nproc = 16{,}384$, and these processors are organized as an $nxproc \times nyproc$ mesh, where $nxproc = nyproc = 128$. In our programs, each virtual processor (or VPE) is given a unique ID ranging from 0 to $vnproc - 1$. 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 $vnxproc \times vnyproc$ mesh. The choice of the *vpr* value (and hence the *vnproc* value) is discussed in Section 3.

For our implementation, we used the so-called *hierarchical partitioning scheme*. Each physical processor simulated a $vpr \times 1$ sub-mesh of virtual processors. Thus given an $nxproc \times nyproc$ 2-dimensional mesh, the virtual machine being simulated is an $(nxproc \cdot vpr) \times nyproc$ 2-dimensional mesh. (This is the same mapping scheme used in the implementation of bitonic sort with virtual processing [24].) The reason for our choice is that in our implementation, we frequently need to use operations that utilize locality of data (e.g., the prefix sum (scan) operator). Our data partitioning scheme preserves locality of data.

3 Implementation of Parallel Graph Algorithms

To build our parallel graph algorithms library, we first wrote a kernel that includes all of the subroutines commonly used in 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 1.

Initially, we coded the routine for finding a spanning forest with virtual processing using the simple graph representation of a list of edges where each undirected edge has two copies with the two end points interchanged. Let m and n be the numbers of (undirected) edges and nodes in the input graph, respectively. Using the naive strategy for allocating the

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

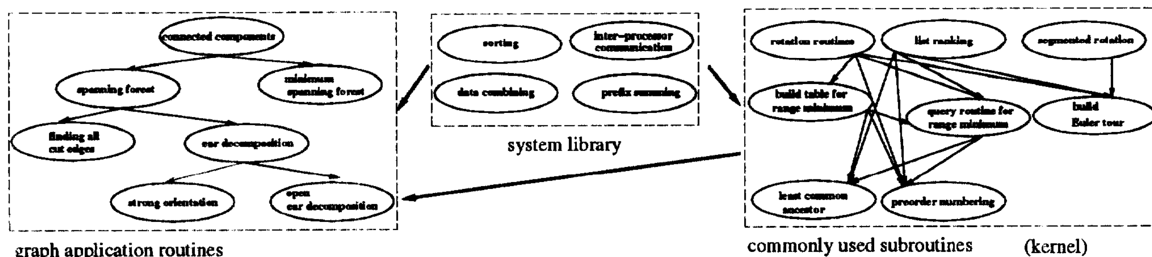

Figure 1: *The structure of the routines we built for the parallel graph algorithms library. An arrow from one node to another node means the routine at the tail of the arrow (upper) is used by the routine at the head of the arrow (lower).*

edges in an undirected graph 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 $\left\lceil \frac{2m+2}{nproc} \right\rceil$. Edges were allocated among virtual processors with ID's from 2 to $2m+1$. (For ease of programming, we did not use the first two virtual processors for storing edges.) The ith node was allocated to the virtual processor with the ID i.

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 to be stored in the processor that was allocated for v. In performing these operations, data will compete with each other to reach a small number of processors. The delay for this type of inter-processor communications is very large. In order to improve the performance of our code, we implemented the following two strategies.

3.1 Dynamic Load Balancing

One possible solution to the above problem is 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 ith 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 was collected for nodes allocated to virtual processors with ID $(i \cdot nfactor)$, for each i, By performing these simple preprocessing and post-processing steps, we evenly distributed all nodes and did not have to track the value of vpr during each operation. Thus, with minor modifications, we could use our previous code with the data allocation scheme given in Section 3 to find a spanning forest with virtual processing.

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 tree in the spanning forest. The total number of edges in the Euler tours of the forest was at most $2n-2$. We applied 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 also used 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 random graphs of three different edge densities: *sparse* (linear number of edges), *medium* ($n^{1.5}$ edges, where n is the number of vertices), and *dense* (quadratic number of edges). Performance data is shown in Figure 2 for this problem with and without the usage of dynamic load balancing on dense graphs. We find that by using dynamic load balancing, our parallel program ran about 12 times faster than our parallel program without dy-

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

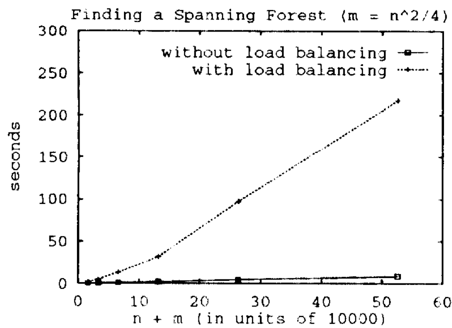

Figure 2: *Performance data for our parallel programs, with and without dynamic load balancing, for finding a spanning forest in a dense graph.*

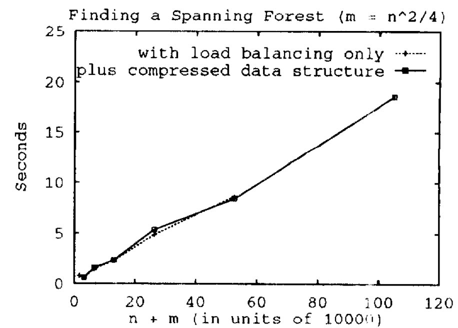

Figure 3: *Performance data for our parallel programs, with and without the use of compressed data structure, for finding a spanning forest in a dense graph. Both programs were run using the same amount of memory per physical processor. Note that we can run inputs whose sizes are twice as large using the compressed data structure for graphs.*

namic load balancing on dense graphs. On medium 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 speed up as the input graph gets denser.

3.2 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 our program with and without the use of compressed data structures for dense graphs to find a spanning forest is given in Figure 3. We find that our program ran at about the same speed with or without the usage of compressed data structures on dense graphs and medium 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 small overhead for 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 use 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 use of compressed data structures, we could only run our programs on graphs with half the number of edges.

4 Performance Analysis

We tested our code by generating test graphs and measuring the performance of the code on these test graphs. The test graphs were of three types: dense, medium, and sparse and are defined in Section 3.1. For each size and edge density, we generated 4 test graphs. We ran each program on each test graph for 10 iterations and recorded the average of the 40 trials.

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 values of vpr up to 64. We were able to run our parallel program for finding an open ear decomposition only up to the value of

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

$vpr = 32$.

4.1 Least-Squares Curve Fitting

We applied the least-squares fit package in Mathematica [32] to the data we obtained. To test the goodness of the curve we obtained, we computed the *goodness of fit* [4], $\gamma(f)$, as $\frac{\sum_{i=1}^{k}(y_i-\overline{y})^2-\sum_{i=1}^{k}(y_i-f(x_i))^2}{\sum_{i=1}^{k}(y_i-\overline{y})^2}$ where k is the number of experimental data points, $\overline{y} = \frac{1}{k} \cdot \sum_{i=1}^{k} y_i$, f is the function that describes the fitted curve, and y_i is the experimental value when the input size is x_i. The value $\gamma(f)$ is 1 if f is a perfect fit of the input data. If $1-\gamma(f_1) < 1-\gamma(f_2)$ for two fitted curves f_1 and f_2 of a set of data, then f_1 is a better fit than f_2. We applied curve fitting to data obtained with and without virtual processing, since [14] did not perform any curve fitting.

We used the following method to find the fitted curves for our data. We first derived the theoretical asymptotic running time for our parallel programs. Let x be the size of the input graph. All of the graph algorithms we implemented run in $O(\frac{x}{p} \cdot \log^3 x)$ time* using p processors since we used the $O(\frac{x}{p} \log^2 x)$ time bitonic sort in implementing global concurrent write operations. Let $S = \{1, x, x \cdot \log x, x \cdot \log^2 x, x \cdot \log^3 x\}$. Given a set of functions, let the element in the set with the largest asymptotic value be the *dominating term*. For instance, the dominating term in S is $x \cdot \log^3 x$. We used Mathematica to find a least-squares fit function f where f is a linear combination of a subset of elements in S. Among all fitted curves for f whose coefficients are all non-negative, we picked the one with the largest goodness of fit that contained the dominating term. For data without virtual processing, the set S we used was $\{1, \log x, \log^2 x, \log^3 x\}$. We noticed that, when no virtual processing was used, the best fit in every case contained the dominating term $\log^3 x$. For data with virtual processing, in the few cases where the best fit did not contain the dominating term $x \cdot \log^3 x$, a function with the dominating term was close to the best. In view of this and the theoretical justification, we chose, for each piece of code, the curve with the best fit that contained the dominating term.

For each problem and each class of graphs tested, we present the fitted curve and its corresponding corresponding goodness of fit in Table 2. The data for programs without virtual processing is taken from [14].

*We use log to denote logarithm to the base 2.

4.2 Overhead for Implementing Virtual Processing

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 values 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 $vpr = 32$ was the open ear decomposition routine. Also, when $vpr = 32$, our code for open ear decomposition could not handle inputs of size $32 \cdot 16{,}384$ (i.e., 524,288) on sparse graphs. Hence in Table 1, we use $vpr = 16$ to show the performance of our parallel code when all virtual processors simulated in each physical processor were active. Our implementation of parallel algorithms with virtual processing had excellent speed-ups on dense graphs and medium 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.

5 Concluding Remarks

We have implemented a set of parallel algorithms for undirected graphs on the MasPar MP-1 to handle input sizes that are larger than the number of available physical processors. We wrote more than 12,000 lines of parallel code for the set of parallel graph algorithms that we implemented with virtual processing. For comparison, 4,000 lines of parallel code were written in [14] for the same set of parallel programs with no virtual processing. We tested our parallel programs on inputs whose sizes were up to 64 times larger than the number of physical processors. If we had been able to use the full configuration of our machine, we could have simulated up to 128 virtual processors per physical processor. However, since we were sharing the machine with other users, we could 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.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

	$m = 3n/2$		$m = n^{3/2}$		$m = n^2/4$	
	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 [14].*

We compared the experimental data points with the interpolated points on the fitted curves. For programs with virtual processing, the goodness of fit for 13 of the curves was very close to 1 (i.e., greater than 0.99). The goodness of fit for the remaining 5 curves was greater than 0.95. If we did not insist on the coefficients of the function corresponding to each fitted curve to be non-negative, the goodness of fit for these 5 curves is much closer to 1. For programs without virtual processing, the goodness of fit was slightly smaller. We also note that for all of our programs without virtual processing, a fitted curve with the dominating term $\log^3 x$ had better goodness of fit than a fitted curve with the dominating term $\log^2 x$. Let the *average error* of a fit f be $1 - \gamma(f)$. In some cases, the average error was twice as large if using fitted curves with the dominating term $\log^2 x$. For our programs with virtual processing, a fitted curve with the dominating term $x \cdot \log^3 x$ did not always have smaller average error than a fitted curve with the dominating term $x \cdot \log^2 x$; however, the difference between the average errors of two fitted curves was not too large.

The fitted curves showed that the dominating term for the number of operations in our parallel code was $x \cdot \log^3 x$ both with and without virtual processing, where x is the input size. This tracks theoretical predictions since our graph algorithms usually compute by performing $O(\log x)$ iterations and if each iteration takes $\Theta(\log^2 x)$ time using x processors, the total work is $O(x \cdot \log^3 x)$.

Our parallel programs were much faster on dense graphs and medium 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 this algorithm obtained a spanning forest by repeatedly growing forests in parallel until the size of any tree in the current forest could not be extended. 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.

References

[1] 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.

[2] T. Blank. The MasPar MP-1 architecture. In *Proc. of COMPCON Spring 90 – 35th IEEE Computer Society International Conference*, pages 20–40, 1990.

[3] G. E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, M.I.T., October 1989.

[4] S. C. Chapra and R. P. Canale. *Numerical Methods for Engineers*. McGraw-Hill, second edition, 1988.

[5] K. W. Chong and T. W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. In *Proc. 4th SODA*, pages 11–20, 1993.

[6] R. Cole, P. N. Klein, and R. E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proc. 6th SPAA*, pages 11–15, 1994.

[7] R. Cole and U. Vishkin. Approximate parallel scheduling. Part II: Applications to logarithmiic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[8] D. Fussel, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. *SIAM J. Comput.*, 22(3):587–616, 1993.

[9] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[10] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th SPAA*, pages 16–25, 1994.

[11] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *Proc. 6th SPAA*, pages 1–10, 1994.

[12] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, 1986.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

		$m = 3n/2$ (sparse graphs)		$m = n^{3/2}$ (medium graphs)		$m = n^2/4$ (dense graphs)	
		fitted curve f	$\gamma(f)$	fitted curve f	$\gamma(f)$	fitted curve f	$\gamma(f)$
Spanning Forest	w/o v.p.	$0.3\log^3 x$	0.9313	$0.095\log^3 x + 110$	0.9407	$0.11\log^3 x + 110$	0.9554
	v.p.	$1.5x'\log^3 x' + 1300x' + 1470$	0.9982	$0.016x'\log^3 x' + 270x'$	0.9999	$0.074x'\log^3 x' + 150x' + 450$	0.9949
Min. Span. Forest	w/o v.p.	$0.33\log^3 x + 120$	0.9625	$0.22\log^3 x + 95$	0.9253	$0.21\log^3 x + 58$	0.8959
	v.p.	$0.79x'\log^3 x' + 950x' + 3140$	0.9996	$0.068x'\log^3 x' + 28x'\log x' + 580x'$	0.9999	$0.2x'\log^3 x' + 8580$	0.9383
All Cut Edges	w/o v.p.	$0.4\log^3 x + 13$	0.9465	$0.14\log^3 x + 180$	0.9559	$0.15\log^3 x + 170$	0.9618
	v.p.	$1.4x'\log^3 x' + 1630x' + 1250$	0.9964	$0.18x'\log^3 x' + 610x' + 270$	0.9995	$0.43x'\log^3 x' + 490x' + 730$	0.9992
Ear Decomp.	w/o v.p.	$0.4\log^3 x + 21$	0.9491	$0.14\log^3 x + 190$	0.9535	$0.15\log^3 x + 180$	0.9605
	v.p.	$x'\log^3 x' + 1590x'$	0.9991	$1.3x'\log^3 x' + 4900$	0.9610	$0.64x'\log^3 x' + 210x' + 910$	0.9994
Open Ear Decomp.	w/o v.p.	$0.41\log^3 x + 260$	0.9408	$0.33\log^3 x + 270$	0.8401	$0.17\log^3 x + 440$	0.8688
	v.p.	$7.2x'\log^3 x' + 22100$	0.9252	$5.9x'\log^3 x' + 9800$	0.9600	$1.1x'\log^3 x' + 1130x' + 600$	0.9996
Strong Orient.	w/o v.p.	$0.39\log^3 x + 33$	0.9416	$0.15\log^3 x + 180$	0.9286	$0.16\log^3 x + 170$	0.9588
	v.p.	$0.46x'\log^3 x' + 1750x'$	0.9972	$0.13x'\log^3 x' + 5020$	0.9607	$0.58x'\log^3 x' + 230x' + 890$	0.9992

Table 2: *Fitted curves for the performance of our parallel code (in milliseconds) without virtual processing and with virtual processing. The goodness of fit $\gamma(f)$ for a curve f is defined in Section 4.1. In each curve, x is the input size and $x' = 10,000x$. The input size is $2 \cdot m$ for programs without virtual processing, and is $m + n$ for programs with virtual processing.*

[13] T.-s. Hsu and V. Ramachandran. Efficient implementation of virtual processing for some combinatorial algorithms on the MasPar MP-1. Manuscript, 1993.

[14] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 15, pages 165–198. American Mathematical Society, 1994.

[15] T.-s. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components. Presented at the 3rd DIMACS Implementation Challenge Workshop, 1994.

[16] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. Technical Report TR-93-14, Dept. of Computer Sciences, Univ. of Texas at Austin, July 22, 1993.

[17] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16:190–217, 1994.

[18] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[19] 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.

[20] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[21] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and *st*-numbering in graphs. *Theoret. Comput. Sci.*, pages 277–298, 1986.

[22] MasPar Computer Co. *MasPar Parallel Application Language (MPL) Reference manual*, version 3.0, rev. a3 edition, July 1992.

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

[24] 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.

[25] 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.

[26] V. Ramachandran and J. Reif. Planarity testing in parallel. *Jour. Comput. and Sys. Sci.*, 49(3):517-561, 1994. Special Issue for *FOCS '89.*

[27] M. Reid-Miller. List ranking and list scan on the CRAY C-90. In *Proc. 6th SPAA*, pages 104–113, 1994.

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

[29] T. J. Sheffler. Implementing the multiprefix operation on parallel and vector computers. In *Proc. 5th SPAA*, pages 377–386, 1993.

[30] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[31] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.

[32] S. Wolfram. *Mathematica™ A System for Doing Mathematics by Computer.* Addison-Wesley, 1988.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE