Implementation of Parallel Graph Algorithms: A Proposal for the Third DIMACS International Algorithm Implementation Challenge

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

Bellcore
Morristown, NJ 07960
USA
nate@bellcore.com

January 7, 1994

Abstract

In this abstract, we describe the parallel algorithms that we have implemented [5, 6] and will implement for the Third DIMACS International Algorithm Implementation Challenge. We also describe our implementation strategy, the parallel computer MasPar MP-1 that we are going to use, formats and generation of input data, and our benchmark.

1 Introduction

In [5, 6], we have reported the implementation of a set of parallel graph algorithms on the MasPar MP-1. All of the non-trivial parallel graph algorithms (e.g, the algorithm for finding an ear decomposition) that we have implemented called the routine for finding connected components before performing any further computation. We have noticed that on very sparse graphs, more than 60% of the computation time for non-trivial parallel graph algorithms is spent on finding connected components. On very dense graphs, this ratio is more than one third. Thus it is critical to speed up the routines for finding connected components.

For the DIMACS Implementation Challenge, we propose to further improve the performance of the set of parallel graph algorithms implemented in [5, 6] by fine tuning the code for finding connected components as given in [6]. We will also try different underlying algorithms for this problem to achieve better speed-ups than the one we achieved in [6]. Our proposed work will build on the work described in [6]. If time allows, we will also test the performance of the code for some non-trivial graph algorithms using the new code for finding connected components.

The organization of this proposal is as follows. We first describe the general approach we will use in implementing graph algorithms in Section 2. We then describe the graph algorithms that we will implement in Section 3. For completeness, we also include the set of parallel graph algorithms that were implemented in [5, 6]. We describe the computing platform in Section 4, the proposed input specifications in Section 6, and benchmarks in Section 5. Finally, we describe the proposed input generators in Section 7 and concluding remarks in Section 8.

2 Experiment Approach

Our implementation 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 [2, 3, 4, 17]. Instead of using a new model, we have performed a direct implementation of parallel algorithms based on the popular PRAM model [7, 8, 16]. 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 [15, 18]. 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.

3 Graph Algorithms

We plan on implementing the following graph algorithms.

1. **Connected components.** Since finding connected components of an undirected graph is one of the most important primitives for solving graph problems in parallel, we will implement several different programs and compare their performances.

   We will first implement the arbitrary CRCW algorithm as described in [1]. This algorithm runs in $O(\log n)$ time using $O(n + m)$ processors on a graph with n vertices and m edges. Using the standard $O(\log p)$-time-$O(p)$-processor simulation technique to translate concurrent writes issued by p processors into exclusive writes, [5, 6] implemented the CREW version of this algorithm. This algorithm runs in $O(\log^2 n)$ time using $O(n + m)$ processors on a graph with n vertices and m edges. We will further fine tune this code to achieve better performance. In addition to the previous two deterministic algorithms, we will also implement a very simple randomized algorithm (see e.g., Chapter 4 in [16]). This algorithms runs in $O(\log n)$ time using $O(n + m)$ processors on an arbitrary CRCW model.

2. **Spanning forest.** We modified the algorithm in [1] for finding connected components to find a spanning forest of the input graph.

3. **Minimum cost spanning forest.** Given the input graph with an integer weight assigned on each edge, we modified the algorithm in [1] for finding connected components to find a minimum cost spanning forest of the input graph.

4. **Ear decomposition of a two-edge connected undirected graph.** We implemented the PRAM parallel algorithm in [15] for finding an ear decomposition.

5. **Open ear decomposition of a biconnected undirected graph.** We implemented the PRAM parallel algorithm in [15] for finding an open ear decomposition.

6. **Strong orientation of a two-edge connected undirected graph.** The algorithm for this problem is as follows. We first find an ear decomposition for the input graph. Then we direct the edges of each ear so that each ear forms a directed path or a directed cycle.

7. **Cut edges.** The algorithm for finding all cut edges in a connected graph is as follows. We first find a rooted spanning tree of the input graph G. A cut edge is a tree edge (u, v), where u is the parent of v and there is no non-tree edge (x, y) in G such that either x or y is a descendant of v or equal to v and the least common ancestor of x and y is an ancestor of u or equal to u.

4 Computing Platform

All of our parallel implementations will be done on an MasPar MP-1 with 16,384 processors. The MasPar computer [10] 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 description of the hardware architecture and the software environment of MasPar can be found in [5].

5 Benchmarks

The NETPAD software [11, 12, 13] is a general tool for graph manipulations and graph algorithm design. It uses the X-window system [14], and one can edit graphs and display them easily. The NETPAD system also provides a rich set of basic graph manipulation operations. By calling these operations in one's own program (preferably written in C [9]), users can implement their own graph algorithms easily.

We will use NETPAD to generate our test data. More details are given in Section 7. We will also use programs in NETPAD as a benchmark. For each parallel graph algorithms we have implemented, we will compare the performance of our parallel implementation and the performance of the sequential implementation in NETPAD. By scaling the performance data according to the relative power of the sequential CPU and each processor in the parallel machine, we can obtain meaningful speedup data.

6 Input Specifications

A tree is represented by its adjacency edge list. Thus an undirected edge will have two different copies, one for each direction. An undirected graph is represented by a list of edges in it. Unlike trees, each undirected edge will appear only once.

7 Input Generators

We will test our programs using 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 $m = \frac{3n}{2}$. We generate graphs with a given property (e.g., two-edge-connected and biconnected) using methods described in [5, 6]. For each size and sparsity, we will generate four different test graphs. We will run each program on each test graph for 10 iterations and record the average of the 40 trials.

8 Concluding Remarks

Our parallel implementation will build on the work reported in [5, 6] with the extension of trying different codes for the basic primitive of finding connected components. Since the routine for finding connected components is the fundamental building block for parallel graph algorithms, we hope to improve the overall performance of the graph algorithm library as described in [5, 6]. We will run our code on inputs as large as possible within the hardware limitation of the parallel machine we use. By doing this, we hope to see the advantage of using our parallel code on larger-sized inputs.

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

[3] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10:657–675, 1981.

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

[5] 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.

[6] 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, 1993.

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

[8] 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.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming language.* Prentice Hall, Englewood Cliffs, NJ, 1988. Second Edition.

[10] MasPar Computer Co. *MasPar System Overview*, version 2.0 edition, March 1991.

[11] M. Mevenkamp. *NETPAD Programmer's Guide.* Bellcore, August 1991.

[12] M. Mevenkamp. *NETPAD Reference Guide.* Bellcore, August 1991.

[13] M. Mevenkamp, N. Dean, and C. L. Monma. *NETPAD User's Guide.* Bellcore, August 1991.

[14] A. Nye. *X Window System User's Guide.* O'Reilly & Associates, Inc., 1988.

[15] 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.

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

[17] J. T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2:484–521, October 1980.

[18] U. Vishkin. Structural parallel algorithmics. In *Proc. 18th ICALP*, volume LNCS #510, pages 363–380. Springer-Verlag, 1991.