CGM*graph*/CGM*lib*: Implementing and Testing CGM Graph Algorithms on PC Clusters

Albert Chan and Frank Dehne

School of Computer Science, Carleton University, Ottawa, Canada
`http://www.scs.carleton.ca/~achan` and `http://www.dehne.net`

Abstract. In this paper, we present CGM*graph*, the first integrated library of parallel graph methods for PC clusters based on CGM algorithms. CGM*graph* implements parallel methods for various graph problems. Our implementations of deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph detection are, to our knowledge, the first efficient implementations for PC clusters. Our library also includes CGM*lib*, a library of basic CGM tools such as sorting, prefix sum, one to all broadcast, all to one gather, h-Relation, all to all broadcast, array balancing, and CGM partitioning. Both libraries are available for download at `http://cgm.dehne.net`.

1 Introduction

Parallel graph algorithms have been extensively studied in the literature (see e.g. [15] for a survey). However, most of the published parallel graph algorithms have traditionally been designed for the theoretical PRAM model. Following some previous experimental results for the MASPAR and Cray [14, 6, 18, 17, 11], parallel graph algorithms for more "PC cluster like" parallel models like the BSP [19] and CGM [7, 8] have been presented in [12, 1–4, 9, 10]. In this paper, we present CGM*graph*, the first integrated library of CGM methods for various graph problems including list ranking, Euler tour, connected components, spanning forest, and bipartite graph detection. Our library also includes a library CGM*lib* of basic CGM tools that are necessary for parallel graph methods as well as many other CGM algorithms: sorting, prefix sum, one to all broadcast, all to one gather, h-Relation, all to all broadcast, array balancing, and CGM partitioning. In comparison with [12], CGM*graph* implements both a randomized as well as a deterministic list ranking method. Our experimental results for randomized list ranking are similar to the ones reported in [12]. Our implementations of deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph detection are, to our knowledge, the first efficient implementations for PC clusters. Both libraries are available for download at `http://cgm.dehne.net`. We demonstrate the performance of our methods on two different cluster architectures: a gigabit connected high performance PC cluster and a network of workstations. Our experiments show that our library provides good parallel speedup and scalability on both platforms. The communication overhead is, in most cases, small and does not grow significantly with

an increasing number of processors. This is a very important feature of CGM algorithms which makes them very efficient in practice.

2 Library Overview and Experimental Setup

Figure 1 illustrates the general use of CGM*lib* and CGM*graph* as well as the class hierarchy of the main classes. Note that all classes in CGM*graph*, except `EulerNode`, are independent. Both libraries require an underlying communication library such as MPI or PVM. CGM*lib* provides a class `Comm` which interfaces with the underlying communication library. It provides an interface for all communication operations used by CGM*lib* and CGM*graph* and thereby hides the details of the communication library from the user.

The performance of our library was evaluated on two parallel platforms: *THOG*, and *ULTRA*. The *THOG* cluster consists of $p = 64$ nodes, each with two Xeon processors. The nodes are of two different generations, with processors at 1.7 or 2.0GHz, 1.0 or 1.5GB RAM, and 60GB disk storage. The nodes are interconnected via a Cisco 6509 switch using Gigabit ethernet. The operating system is Linux Red Hat 7.1 together with LAM-MPI version 6.5.6. The *ULTRA* platform is a network of workstations consisting of $p = 10$ Sun Sparc Ultra 10. The processor speed is 440MHz. Each processor has 256MB RAM. The nodes are interconnected via 100Mb Switched Fast Ethernet. The operating system is Sun OS 5.7 and LAM-MPI version 6.3.2.

All times reported in the remainder of this paper are wall clock times in seconds. Each data point in the diagrams represents the average of three experiments (on different random test data of the same size) for CGM*graph* and ten experiments for CGM*lib*. The input data sets for our tests consisted of randomly created test data. For inputs consisting of lists or graphs, we generated random lists or graphs as follows. For random linked lists, we first created an arbitrary linked list and then permuted it over the processors via random permutations. For random graphs, we created a set of nodes and then added random edges. Unfortunately, different test data sizes had to be chosen for the different platforms because of the smaller memory capacity of *ULTRA*.

3 CGM*lib*: Basic Infrastructure and Utilities

3.1 CGM Communication Operations

The basic library, called CGM*lib*, provides basic functionality for CGM communication. An interface, `Comm`, defines the basic communication operations such as

- `oneToAllBCast(int source, CommObjectList &data)`: Broadcast the list `data` from processor number `source` to all processors.
- `allToOneGather(int target, CommObjectList &data)`: Gather the lists `data` from all processors to processor number `target`.

- `hRelation(CommObjectList &data, int *ns)`: Perform an h-Relation on the lists `data` using the integer array `ns` to indicate for each processor which list objects are to be sent to which processor.
- `allToAllBCast(CommObjectList &data)`: Every processor broadcasts its list `data` to all other processors.
- `arrayBalancing(CommObjectList &data, int expectedN=-1)`: Shift the list elements between the lists `data` such that every processor contains the same number of elements.
- `partitionCGM(int groupId)`: Partition the CGM into groups indicated by `groupId`. All subsequent communication operations, such as the ones listed above, operate within the respective processor's group only.
- `unPartitionCGM()`: Undo the previous partition operation.

All communication operations in CGM*lib* send and receive lists of type `Comm ObjectList`. A `Comm ObjectList` is a list of `CommObject` elements. The `Comm Object` interface defines the operations which every object that is to be sent or received has to support.

3.2 CGM Utilities

- Parallel Prefix Sum:
`calculatePrefixSum (CommObjectList &result, CommObjectList &data)`.
- Parallel Sorting: `sort(CommObjectList &data)` using the deterministic parallel sample sort methods in [5] and [16].
- Request System for exchanging data requests between processors: The CGM*lib* provides methods `sendRequests(...)` and `sendResponses(...)` for routing the requests from their senders to their destinations and returning the responses to the senders, respectively.
- Other CGM Utilities: A class `CGMTimers` (with six timers measureing computation time, communication time, and total time, both in wall clock time and CPU ticks) and other utilities including a parallel random number generator.

3.3 Performance Evaluation

Figure 2 shows the performance of our prefix sum implementation, and Figure 3 shows the performance of our parallel sort implementation. For our prefix sum implementation, we observe that all experiments show a close to zero communication time, except for some noise on *THOG*. The prefix sum method communicates only very few data items. The total wall clock time and computation time curves in all four diagrams are similar to $1/p$. For our parallel sort implementation, we observe a small fixed communication time, essentially independent of p. This is easily explained by the fact that the parallel sort uses of a fixed number of h-Relation operations, independent of p. Most of the total wall clock time is spent on local computation which consists mainly of local sorts of n/p data. Therefore, the curves for the local computation and the total parallel wall clock time are similar to $1/p$.

4 CGM*graph*: Parallel Graph Algorithms Utilizing the CGM Model

CGM*graph* provides a list ranking method `rankTheList(ObjList<Node> &nodes, ...)` which implements a randomized method as well as a deterministic method [15, 9]. The input to the list ranking method is a linear linked list where Tthe pointer is stored as the index of the next node. CGM*graph* also provides a method `getEulerTour(ObjList <Vertex> &r, ...)` for Euler tour traversal of a forest [9]. The forest is represented by a list of vertices, a list of edges and a list of roots. The input to the Euler tour method is a forest which is stored as follows: `r` is a set of vertices that represents the roots of the trees, `v` is the input set of vertices, `e` is the input set of edges, and `eulerNodes` is the output data of the method. We implemented the connected component method described in [9]. The method also provides immediately a spanning forest of the given graph. CGM*graph* provides a method `findConnectedComponents(Graph &g, Comm *comm)` for connected component computation and a method `findSpanningForest(Graph &g, ...)` for the calculation of the spanning forest of a graph. The input to the above two methods is a graph represented as a list of vertices and a list of edges. We also implemented the bipartite graph detection algorithm described in [3]. CGM*graph* provides a method `isBipartiteGraph(Graph &g, Comm *comm)` for detecting whether a graph is a bipartite graph. As in the case of connected component and spanning forest, the input is a graph represented as a list of vertices and a list of edges.

4.1 Performance Evaluation

In the following, we present the results of our experiment. For each operation, we measured the performance on *THOG* with $n = 10,000,000$ and on *ULTRA* with $n = 100,000$.

Figure 4 shows the performance of the deterministic list ranking algorithm. Again, we observe that for both machines, the communication time is a small, essentially fixed, portion of the total time. The deterministic list ranking requires between $c \log p$ and $2c \log p$ h-Relation operations. With $\log p$ in the range $[1, 5]$, we expect between c and $10c$ h-Relation operations. Since the deterministic algorithm is more involved and incurs larger constants, c may be around 10 which would imply a range of $[10, 100]$ for the number of h-Relation operations. We measured usually around 20 h-Relation operations. The number is fairly stable, independent of p, which shows again that $\log p$ has little influence on the measured communication time. The small increases for $p = 4, 8, 16$ are due to the fact that that the number of h-Relation operations grows with $\lfloor \log p \rfloor$, which gets incremented by 1 when p reaches a power of 2. In summary, since the communication time is only a small fixed value and the computation time is dominating and similar to $1/p$, the entire measured wall clock time is similar to $1/p$. Figure 5 shows the performance of the Euler tour algorithm on *THOG* and *ULTRA*. Our implementation uses the deterministic list ranking method for the Euler tour computation. Not surprisingly, the performance is essentially

the same as for deterministic list ranking. Due to the fact that all tree edges need to be duplicated, the data size increases by a factor of three (original plus two copies). This is the reason why we could execute the Euler tour method on *THOG* for $n = 10,000,000$ only with $p \geq 10$.

Figure 6 shows the performance of the connected components algorithm on *THOG* and *ULTRA*. Figure 7 shows the performance of the spanning forest algorithm on *THOG*, and *ULTRA*. The only difference between the two methods is that the spanning forest algorithm has to create the spanning forests after the connected components have been identified. Therefore, the times shown in Figures 6 and 7 are very similar. Again, we observe that for both machines, the communication time is a small, essentially fixed, portion of the total time. The connected component method uses deterministic list ranking. It requires $c \log p$ h-Relation operations with $\log p$ in the range $[1, 5]$. The communication time observed is fairly stable, independent of p, which shows that the $\log p$ factor has little influence on the measured communication time. The entire measured wall clock time is dominated by the computation time and similar to $1/p$.

Figure 8 shows the performance of the bipartite graph detection algorithm on *THOG*, and *ULTRA*. The results mirror the fact that the algorithm is essentially a combination of Euler tour traversal and spanning forest computation. The curves are similar to the former but the amount of communication time is now larger, representing the sum of the two. This effect is particularly strong on *ULTRA* which has the weakest network. Here, the $\log p$ in the number of communication rounds actually leads to a steadily increasing communication time which, for $p = 9$ starts to dominate the computation time. However, for *THOG* and *CGM1*, the effect is much smaller. For these machines, the communication time is still essentially fixed over the entire range of values of p. The computation time is similar to $1/p$ and determines the shape of the curves for the entire wall clock time. The computation and communication times become equal for larger p but only because of the decrease in computation time.

5 Future Work

Both, CGM*lib* and CGM*graph* are currently in beta state. Despite extensive work on performance tuning, there are still many possibilities for fine-tuning the code in order to obtain further improved performance. Of course, adding more parallel graph algorithm implementations to CGM*graph* is an important task for the near future. Other possible extensions include porting CGM*lib* and CGM*graph* to other communication libraries, e.g. PVM and OpenMP. We also plan to integrate CGM*lib* and CGM*graph* with other libraries, in particular the LEDA library [13].

References

1. P. Bose, A. Chan, F. Dehne, and M. Latzel. Coarse Grained Parallel Maximum Matching in Convex Bipartite Graphs. In *13th International Parallel Processing Symposium (IPPS'99)*, pages 125–129, 1999.

2. E. Cacere, A. Chan, F. Dehne, and G. Prencipe. Coarse Grained Parallel Algorithms for Detecting Convex Bipartite Graphs. In *26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000)*, volume 1928 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2000.
3. E. Caceres, A. Chan, F. Dehne, and G. Prencipe. Coarse Grained Parallel Algorithms for Detecting Convex Bipartite Graphs. In *26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000)*, volume 1928 of *Springer Lecture Notes in Computer Science*, pages 83–94, 2000.
4. E. Caceres, A. Chan, F. Dehne, and S. W. Song. Coarse Grained Parallel Graph Planarity Testing. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*. CSREA Press, 2000.
5. A. Chan and F. Dehne. A Note on Coarse Grained Parallel Integer Sorting. *Parallel Processing Letters*, 9(4):533–538., 1999.
6. S. Dascal and U. Vishkin. Experiments with List Ranking on Explicit Multi-Threaded (XMT) Instruction Parallelism. In *3rd Workshop on Algorithms Engineering (WAE-99)*, volume 1668 of *Lecture Notes in Computer Science*, page 43 ff, 1999.
7. F. Dehne. Guest Editor's Introduction, Special Issue on Coarse Grained Parallel Algorithms. *Algorithmica*, 24(3/4):173–176, 1999.
8. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *ACM Symposium on Computational Geometry*, pages 298–307, 1993.
9. F. Dehne, A. Ferreira, E. Caceres, S. W. Song, and A. Roncato. Efficient Parallel Graph Algorithms for Coarse Grained Multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.
10. F. Dehne and S. W. Song. Randomized Parallel List Ranking for Distributed Memory Multiprocessors. In *Asian Computer Science Conference (ASIAN '96)*, volume 1179 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1996.
11. T. Hsu, V. Ramachandran, and N. Dean. Parallel Implementation of Algorithms for Finding Connected Components in Graphs. In *AMS/DIMACS Parallel Implementation Challenge Workshop III*, 1997.
12. Isabelle Gurin Lassous, Jens Gustedt, and Michel Morvan. Feasability, Portability, Predictability and Efficiency : Four Ambitious Goals for the Design and Implementation of Parallel Coarse Grained Graph Algorithms. Technical Report RR-3885, INRIA, http://www.inria.fr/rrrt/rr-3885.html.
13. LEDA library. http://www.algorithmic-solutions.com/.
14. Margaret Reid-Miller. List Ranking and List Scan on the Cray C-90. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 104–113, 1994.
15. J. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan and Kaufmatin Publishers, 1993.
16. H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
17. Jop F. Sibeyn. List Ranking on Meshes. *Acta Informatica*, 35(7):543–566, 1998.
18. Jop F. Sibeyn, Frank Guillaume, and Tillmann Seidel. Practical Parallel List Ranking. *Journal of Parallel and Distributed Computing*, 56(2):156–180, 1999.
19. L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), 1990.

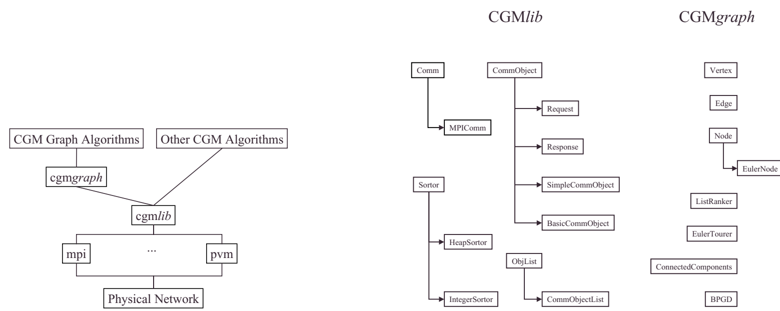

Fig. 1. Overview of CGM*lib* and CGM*graph*

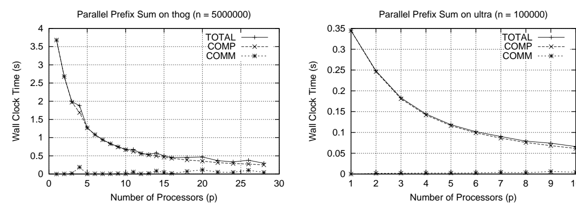

Fig. 2. Performance of our prefix sum implementation on *THOG* and *ULTRA*

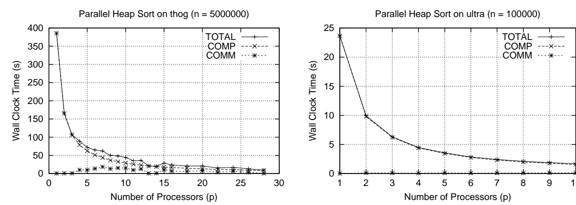

Fig. 3. Performance of our parallel sort implementation on *THOG* and *ULTRA*

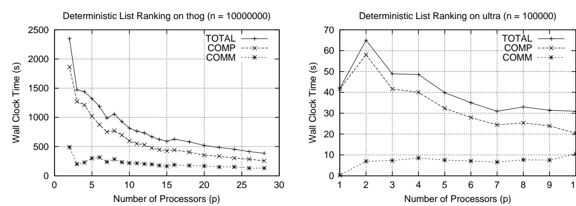

Fig. 4. Performance of the deterministic list ranking algorithm on *THOG* and *ULTRA*

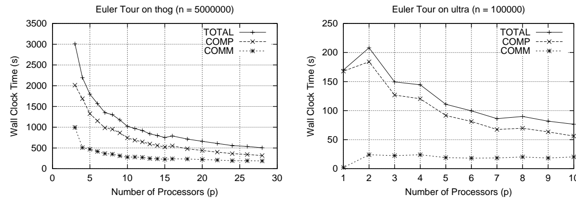

Fig. 5. Performance of the Euler tour algorithm on *THOG* and *ULTRA*

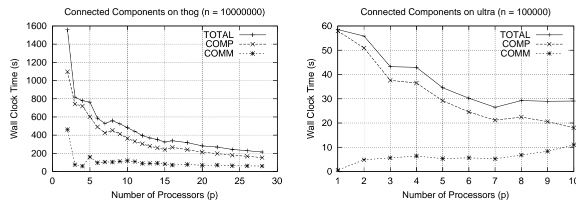

Fig. 6. Performance of the connected components algorithm on *THOG* and *ULTRA*

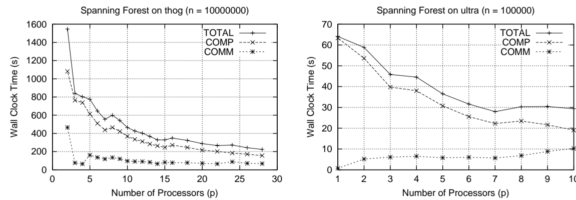

Fig. 7. Performance of the spanning forest algorithm on *THOG* and *ULTRA*

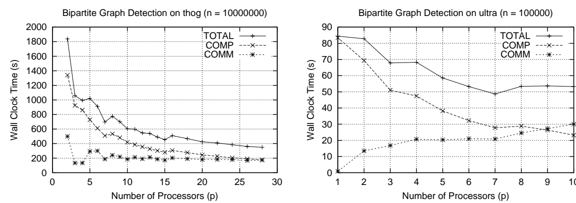

Fig. 8. Performance of the bipartite graph detection algorithm on *THOG* and *ULTRA*