CGMGRAPH/CGMLIB: IMPLEMENTING AND TESTING CGM GRAPH ALGORITHMS ON PC CLUSTERS AND SHARED MEMORY MACHINES

Albert Chan[1]
Frank Dehne[2]
Ryan Taylor[3]

Abstract

In this paper, we present CGM*graph*, the first integrated library of parallel graph methods for PC clusters based on Coarse Grained Multicomputer (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://www.scs.carleton.ca/~cgm`.

In the experimental part of this paper, we demonstrate the performance of our methods on four different architectures: a gigabit connected high performance PC cluster, a smaller PC cluster connected via fast ethernet, a network of workstations, and a shared memory machine. Our experiments show that our library provides good parallel speedup and scalability on all four 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.

Key words: Coarse-grained multiprocessor, graph algorithms, library, parallel algorithms implementation, processor cluster

The International Journal of High Performance Computing Applications, Volume 19, No. 1, Spring 2005, pp. 81–97
DOI: 10.1177/1094342005051196
© 2005 Sage Publications

1 Introduction

In this paper, we present CGM*graph*, the first integrated library of Coarse Grained Multicomputer (CGM; Dehne et al. 1993) methods for graph problems including list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition. 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 Guérin Lassous et al. (2000), CGM*graph* implements both a randomized as well as a deterministic list ranking method. Our experimental results for randomized list ranking are similar to those reported in Guérin Lassous et al. (2000). Our implementations of deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition are, to our knowledge, the first efficient implementations for PC clusters.

CGM*graph* and CGM*lib* are based on the CGM/BSP model (Valiant 1990; Dehne et al. 1993) and are optimized for PC clusters. In the experimental part of this paper, we show the performance of our methods on four different architectures: *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. The *THOG* cluster is a gigabit connected high performance cluster, *CGM1* is a smaller cluster connected via fast ethernet, *ULTRA* is a network of workstations, and *SUNFIRE* is a shared memory cluster. Our experiments show that our library provides good relative parallel speedup and scalability on all four 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. The communication overhead is, in most cases, dominated by the local computation time, which implies good relative speedup in practice.

Both the CGM*lib* and CGM*graph* libraries are freely available for download at http://www.scs.carleton.ca/~cgm, together with a library installation script. The aim of these libraries is to make efficient parallel graph methods available to a wider community of researchers who can utilize them as building blocks for other parallel programming projects (Chan and Dehne 2003).

[1]DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE FAYETTEVILLE STATE UNIVERSITY FAYETTEVILLE, NC 28301, USA

[2]SCHOOL OF ICT GRIFFITH UNIVERSITY NATHAN, QLD 4111, AUSTRALIA (F.DEHNE@GRIFFITH.EDU.AU; HTTP://WWW.DEHNE.NET)

[3]SCHOOL OF COMPUTER SCIENCE CARLETON UNIVERSITY OTTAWA, CANADA K1S 5B6

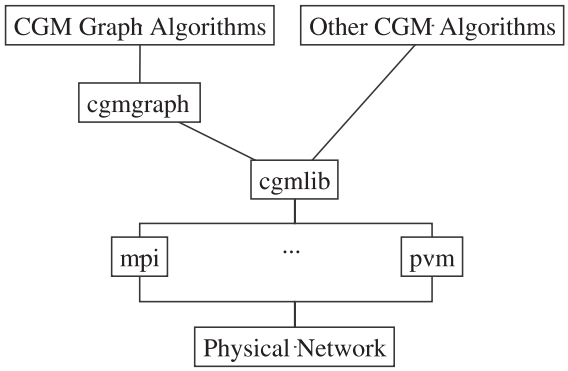

Fig. 1 General use of CGM*lib* and CGM*graph* (PVM implementation not yet available).

2 Library Overview and Experimental Setup

Figure 1 illustrates the general use of CGM*lib* and CGM*graph*, and Figure 2 shows the class hierarchy of the main classes in CGM*lib* and CGM*graph*. Note that all classes in CGM*graph*, except `EulerNode`, are independent. Both libraries require an underlying communication library such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine). 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. At this moment, an MPI implementation of `Comm` is available as part of CGM*lib*. The aim of our design is to provide portability of CGM*lib* and CGM*graph* across different architectures and communication libraries. For any other communication library, an implementation of `Comm` is sufficient to port the entire CGM*lib* and CGM*graph*. As shown in Figure 1, CGM*lib* provides basic CGM functionality (sorting, prefix sum, etc.) and CGM*graph* uses CGM*lib* in order to implement graph algorithms. The design separates the two libraries so that other CGM algorithms (in addition to the CGM graph library provided) can also make use of the CGM*lib* methods.

The performance of our library was evaluated on four parallel platforms: *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*.

The *THOG* cluster is located in the High Performance Computing Virtual Laboratory (HPCVL) at Carleton University. This cluster consists of 64 nodes, each with two Xeon processors. The nodes are of two different generations, with processors at 1.7 or 2.0 GHz, 1.0 or 1.5 GB RAM, and 60 GB 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 *CGM1* cluster is located in the Carleton/Dalhouise CGMlab. This cluster consists 32 processor nodes. Each node has two Xeon processors at 1.8 GHz with 1.0 GB RAM and 80 GB disk storage. The nodes are interconnected via 100 Mb Switched Fast Ethernet. The operating system is Linux Red Hat 7.1 together with LAM-MPI version 6.5.6.

The *ULTRA* platform is an older network of workstations located in the Carleton University School of Computer Science graduate laboratory. The network consists of 10 Sun Sparc Ultra 10 workstations. The processor speed is 440 MHz. Each processor has 256 MB RAM. The nodes are interconnected via 100 Mb Switched Fast Ethernet. The operating system is Sun OS 5.7 and LAM-MPI version 6.3.2.

The *SUNFIRE* machines are located in the HPCVL lab at Queen's University. The *SUNFIRE* machines consist of eight Sun Fire 6800 and two Sun Fire 15000 servers. A total of 336 processors with 786 GB RAM is distributed throughout these machines. The processors inside each of these machines share their machine's memory, so shared memory communication is used. For communication between processors in different machines, Gigabit Ethernet links are used. Figure 3 shows the system view of the *SUNFIRE* machines. The operating system is Sun OS 5.9 and Sun's implementation of the MPI library. The software development environment is Sun's Forte 6 Development Suite. Our tests are done using the processors within a single box. Therefore, our results on the *SUNFIRE* reflects the performance on shared memory machines.

For the remainder, p denotes the number of processors, m denotes the total memory size of the parallel machine, m/p is the memory per processor, and n denotes the total input data size. All times reported in the remainder of this paper are wall clock times in seconds. Communication times reported are the accumulated wall clock times spent by the MPI method calls. The reported computation times are the difference between the total wall clock times and the communication times. Note that the computation times indicated include system time (e.g. garbage collection overhead). The input data sets for our tests consist of randomly created test data.

Parallel graph methods are obviously targeted towards very large graphs and the experiments should reflect that. Unfortunately, test data sets of different sizes had to be chosen for the different platforms because of the smaller memory capacity of *CGM1* and in particular *ULTRA*. While the *SUNFIRE* and *THOG* could run our code on data sets of size n = 10,000,000, *CGM1* and *ULTRA* have far less memory and could only handle data sets of size n = 5,000,000 and n = 100,000, respectively.

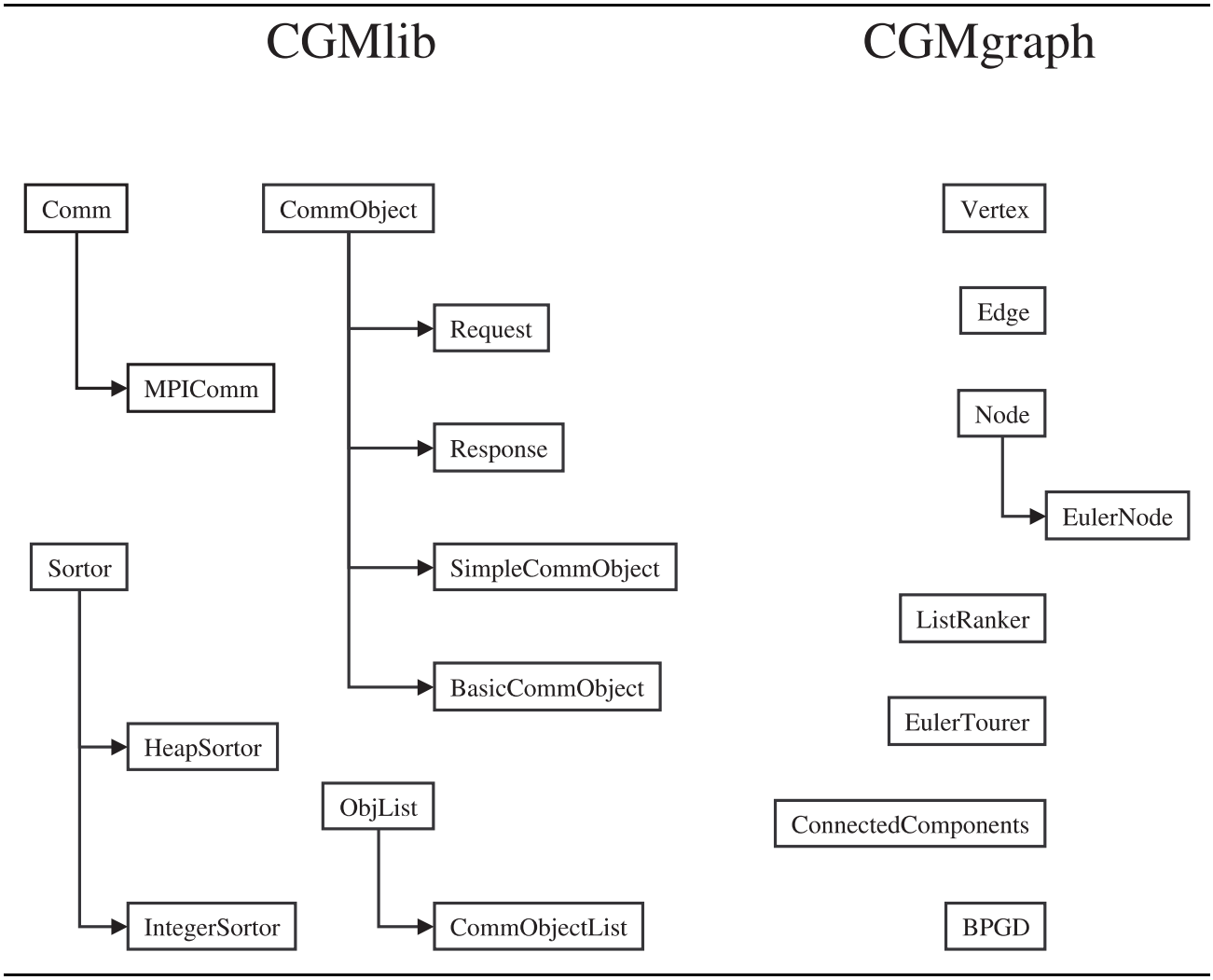

Fig. 2 Hierarchy of main classes in CGM*lib* and CGM*graph*.

Our experiments consisted of running CGM*graph* and CGM*lib* methods for variable numbers of processors on each machine. For each experiment, we measured the total wall clock time as well as the wall clock time for the communication and the computation portion of the experiment. Each data point in the diagrams presented in the remainder of this paper represents the average of three experiments for CGM*graph* and 10 experiments for CGM*lib*.

During the development of CGM*lib* and CGM*graph*, we observed irregular performance for large input data sizes. Investigation showed that this was caused by the C++/Linux memory management. When our code releases unused memory, the memory management system attempts to defragment the memory. This defragmentation occurs at unpredictable times and can lead to performance degradation. To remedy the problem, we provide our own memory management for the class `ObjList` where arrays are allocated manually and memory is managed by our code. Our experience has shown that this can bring very substantial performance improvements.

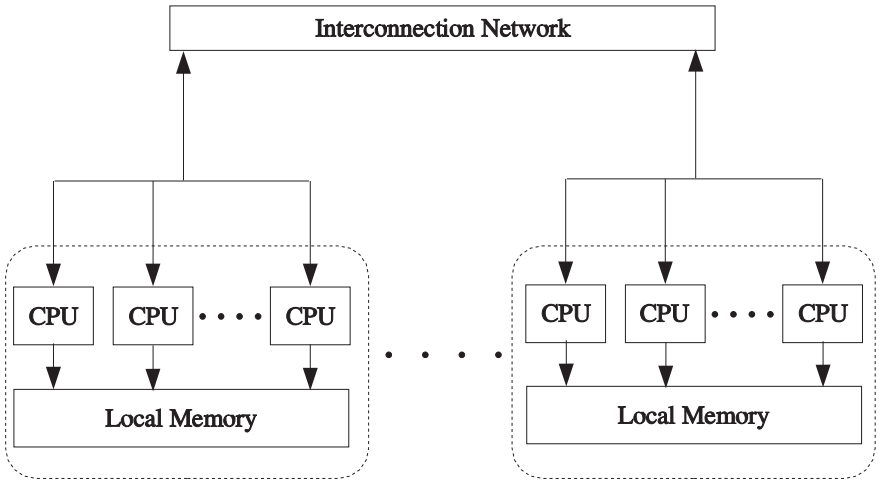

Fig. 3 System view of the *SUNFIRE* 6800 machines.

3 CGMlib: 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 the following.

- `oneToAllBCast (int source, CommObjectList &data)`: broadcast the list `data` from processor number `source` to all processors.
- `allToOneGather (int target, CommObjectList &data)`: execute a gather operation on the lists `data` from every processor 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 will contain `expectedN` elements. `expectedN = -1` indicates that every processor receives 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 data in the form of lists of type `CommObjectList`. A `CommObjectList` is a list of `CommObject` elements. The `CommObject` interface defines the operations which every object that is to be sent/received has to support. A class `SimpleCommObject` is provided which puts an appropriate wrapper around any C++ object (as long as it does not contain pointers). A class `BasicCommObject` is provided which puts an appropriate wrapper around the basic C++ data types (int, float, etc.).

The above operations are defined for a CGM with local memory $m/p \geq O(p)$. For larger local memory with $m/p \geq O(p^2)$, simpler versions `allToAllBCast2 (...)` and `arrayBalancing2 (...)` for all-to-all broadcast and array balancing, respectively, are available.

The library is designed such that everything is built on top of the `Comm` interface which encapsulates basic send/receive operations. This allows different implementations of `Comm` to support different platforms. For the current CGM*lib*, we provide an MPI implementation `MPIComm`. In the future it is possible to add, for example, a PVM

implementation of `Comm` which will enable the entire CGM*lib* and CGM*graph* to run on a PVM platform as well.

3.2 CGM UTILITIES

3.2.1 CGM Parallel Prefix Sums **Definition 1** Given a list of n items $A = \{a_1, a_2, \cdots, a_n\}$ and an associative operator $\otimes$, the prefix sum operation over A and $\otimes$ is to calculate another list of n items $B = \{b_1, b_2, \cdots, b_n\}$ such that $b_i = a_1 \otimes a_2 \otimes \cdots b_i$.

A CGM implementation of the parallel prefix sum algorithm is available. The algorithm is simple. The method `calculatePrefixSum (CommObjectList &result, CommObjectList &data)` performs the following steps. Each processor calculates a local prefix sum. The local total sum values are sent to one processor via a `allToOneGather`. That processor calculates a prefix sum of the total sums and sends the results back to the other processors via an `hRelation` operation. Finally, each processor adjusts its local prefix sum values. The operator used for the prefix sum is flexible and is set via the class constructor.

3.2.2 CGM Parallel Sorting A method `sort (CommObjectList &data)` is provided which sorts the union of the lists `data` from all processors. The comparison operator used for the sort is flexible and is set via the class constructor. We use the deterministic parallel sample sort methods in Chan and Dehne (1998, 1999) and Shi and Schaeffer (1992). Both methods require a local sequential sort method for which the default in CGM*lib* is Heapsort. However, any other local sequential sort method can be assigned via the class constructor. While the method of Shi and Schaeffer (1992) requires $m/p \geq O(p^2)$, the algorithm in Chan and Dehne (1998, 1999) requires only $m/p \geq O(p)$ but is slightly more complex. Depending on the local memory available, the user can select between the two sorting methods via a method `setScalability`. In theory, any fixed scalability can be achieved via the algorithm in Goodrich (1996). However, this method is very complex and incurs large overheads. Therefore, the method in Goodrich (1996) is not implemented in this library.

3.2.3 CGM Request System In some CGM algorithms, it is necessary for processors to obtain information from other processors. This can be done by sending requests. That is, every processor sends to other processors requests for locally stored data items, and these items are then delivered to the requesting processors. In the CGM model, the requests are all handled jointly via h-Relation and sort operations. 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.

3.2.4 Other CGM Utilities Various utilities are provided. The most important is the class `CGMTimers`. We provide six timers within `CGMTimers` which measure computation time, communication time, and total time, both in wall clock time and CPU ticks. Another utility provided is a parallel random number generator. Its main purpose is to make sure that all processors use different seeds for the pseudo random number computation.

3.3 PERFORMANCE EVALUATION

We have measured the performance of our communication operations and utilities on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. In the following, we present the results for h-Relation, prefix sum and parallel sort operations. For each operation, we measured the performance on *THOG* with $n = 5{,}000{,}000$ and $n = 10{,}000{,}000$, on *CGM1* with $n = 5{,}000{,}000$, on *ULTRA* with $n = 100{,}000$, and on *SUNFIRE* with $n = 10{,}000{,}000$. The data consist of randomly created integers. For the h-Relation, the array `ns` is a different array of random integers for each processor.

Figure 4 shows the performance of our h-Relation implementation on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. (See Thakur et al. 2005 for related studies.) Figure 5 shows the performance of our prefix sum implementation, and Figure 6 shows the performance of our parallel sort implementation. For $n = 5{,}000{,}000$, *THOG* is about twice as fast as *CGM1*. Both machines have comparable processors, and the computation times are indeed similar. However, *THOG* has a much more powerful switch, which results in much faster communication times. For our h-Relation implementation, we observe that the curves for the total wall clock times are in all four diagrams similar to $1/p$ for $p \geq 10$ but then become more flat. This is particularly pronounced for *THOG* where we have a larger number of processors available. It appears to be caused by the communication times. The computation portion always appear to be similar to $1/p$ but the communication times appear to go flat for $p > 10$. This is caused by the fact that the total volume of data communicated through the switch is the same, regardless of p. For small p, sending the data to the switch through more ports appears to bring some improvement in speed but this effect disappears for larger p. This is an important observation. The communication time for an h-Relation operation can essentially be seen as a function of the total data volume only, independent of the number of processors.

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 $1/p$.

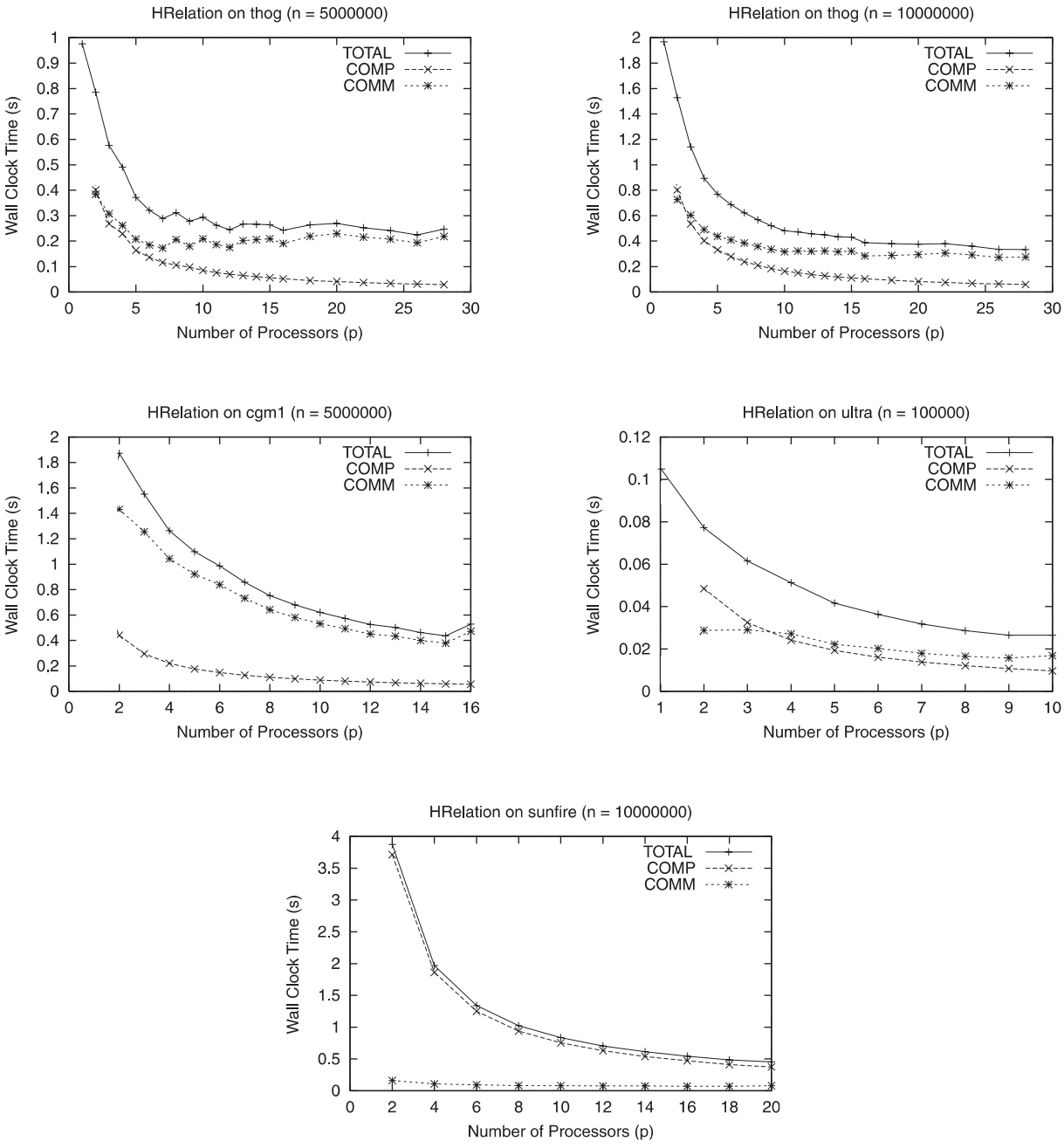

Fig. 4 Performance of the h-Relation implementation ($n = hp$).

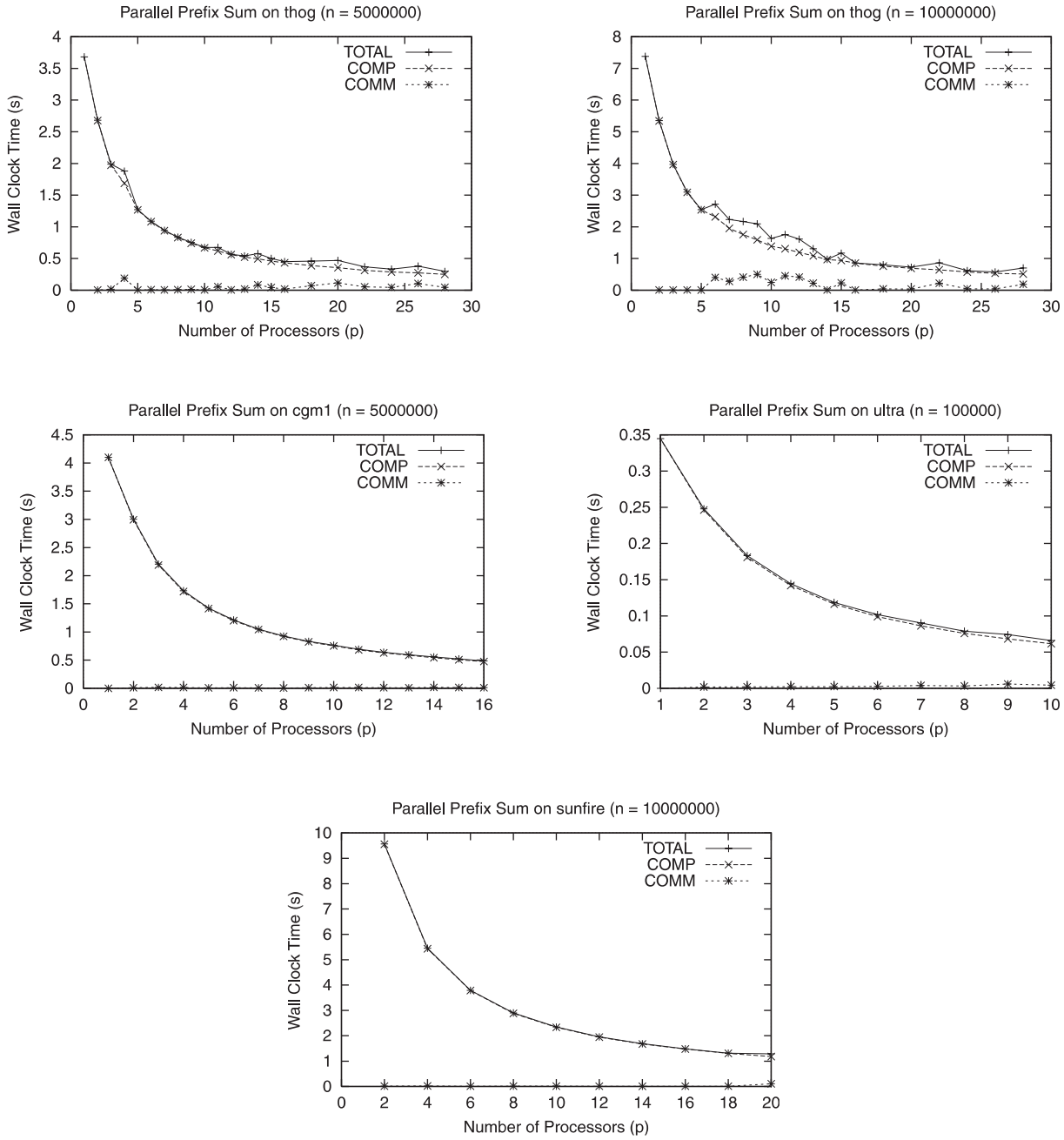

Fig. 5 Performance of the deterministic list ranking algorithm.

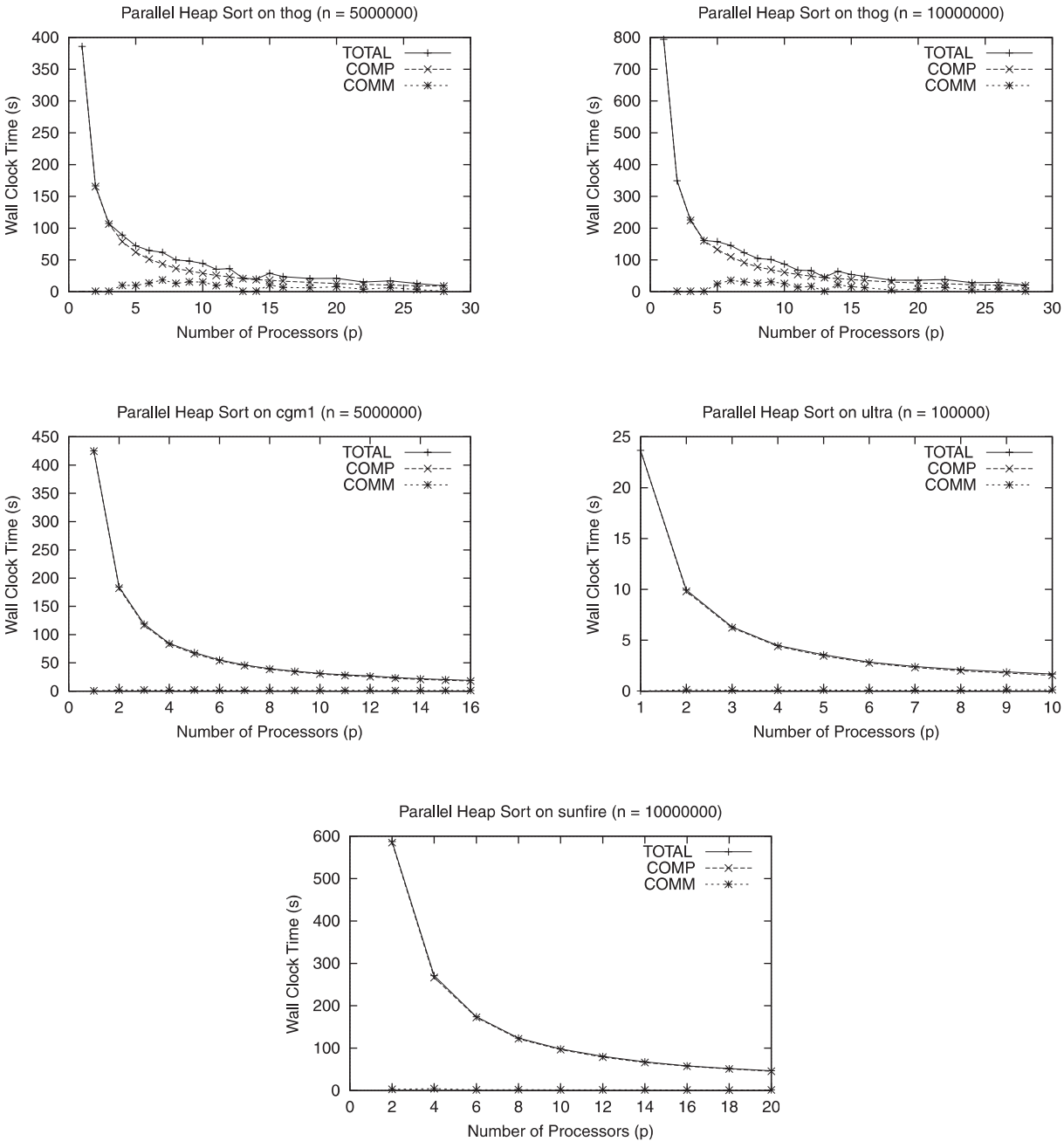

Fig. 6 Performance of the parallel sort implementation.

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 $1/p$.

Also noticeable is that for the result on *SUNFIRE*, the communication time is close to zero. This is because the *SUNFIRE* is a shared memory machine.

4 CGMgraph: Parallel List Ranking and Euler Tour

CGM*graph* provides a list ranking method `rankTheList (ObjList <Node> &nodes, bool randomized, Comm *comm)` which implements a randomized method as well as a deterministic method depending on the boolean `randomized = TRUE/FALSE`, respectively. The input to the list ranking method is a linear linked list where each node has four integer fields: `index` is a non-negative integer index. The indices need not be consecutive. Field `next` is a pointer that points to the next node in the linked list. The pointer is stored as the index of the next node. Field `rank` is an integer to be filled by the `rankTheList (...)` method to represent the rank of the node. Field `tail` is a pointer to be filled by the `rankTheList (...)` method to represent the last node in the linked list.

CGM*graph* also provides a method to compute the Euler tour traversal of a forest. The method is `getEulerTour (ObjList<Vertex> &r, ObjList<Vertex> &v, ObjList<Edge> &e, ObjList<EulerNode> &eulerNodes,Comm *comm)`.

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.

4.1 RANDOMIZED LIST RANKING

We implemented a simplified version of the randomized list ranking method described in Dehne and Song (1996), Reid-Miller (1994), and Sibeyn (1998, 1999). The following is an outline of our method which proceeds in three phases.

Phase 1. A randomized method is used to approximate a p-sample, which is defined as a subset of n/p selected nodes with the property that two consecutive selected nodes in the linked list have a distance of $O(p^k)$, $k = O(1)$, with respect to the linked list. The method simply selects n/p random nodes. Clearly, the average distance between two selected nodes is p. However, what is more important is the maximum distance between two selected nodes which is, with high probability, bounded by $O(p \log n)$.

Phase 2. Using simulated pointer jumping (Reif 1993), implemented via the CGM Request System (Section 3.2), every list node obtains a pointer to the next selected node in the list, together with the distance to that node. All selected nodes are then compressed into one processor and sequential weighted list ranking is applied where the weights are the distances obtained in the previous step.

Phase 3. All other (non-selected) nodes determine their rank by obtaining the rank of the next selected node (as determined in the previous step) and adding to it the distance obtained through the simulated pointer jumping. The total number of h-Relation operations for all three phases is, with high probability between $O(p \log n)$ and $O(\log p + \log \log n)$. For practical purposes, $\log \log n$ can be assumed to be bounded by a constant (approximately 4 for our data sets).

4.2 DETERMINISTIC LIST RANKING

We also implemented the deterministic list ranking method described in Dehne et al. (2002). The deterministic algorithm is much more involved than the randomized method. The following is a brief outline of the algorithm. For more details, see Dehne et al. (2002). The basic structure of the deterministic method is the same as in the randomized case. However, a different approach is used for determining the selected nodes in Phase 1. The approach consists of a combination of a CGM adaptation of deterministic coin tossing (Reif 1993) and various merge steps. At the end of this phase, the set of selected nodes has the property that every node has a distance of at most $O(p^2)$ from the next selected node in the list. This guarantees that the entire algorithm requires between $c \log p$ and $2c \log p$ h-Relation operations in total. The drawback of this method is that it is more involved and incurs considerably larger constant factors in the running times.

4.3 EULER TOUR

The Euler tour traversal of a tree is simple, once a list ranking method is available. A Euler tour traversal in linked list format is generated by converting each tree edge into two directed edges. The edges have to be linked together in the appropriate way to avoid self loops. See Dehne et al. (2002) for more details. Then, list ranking is applied to this linked list.

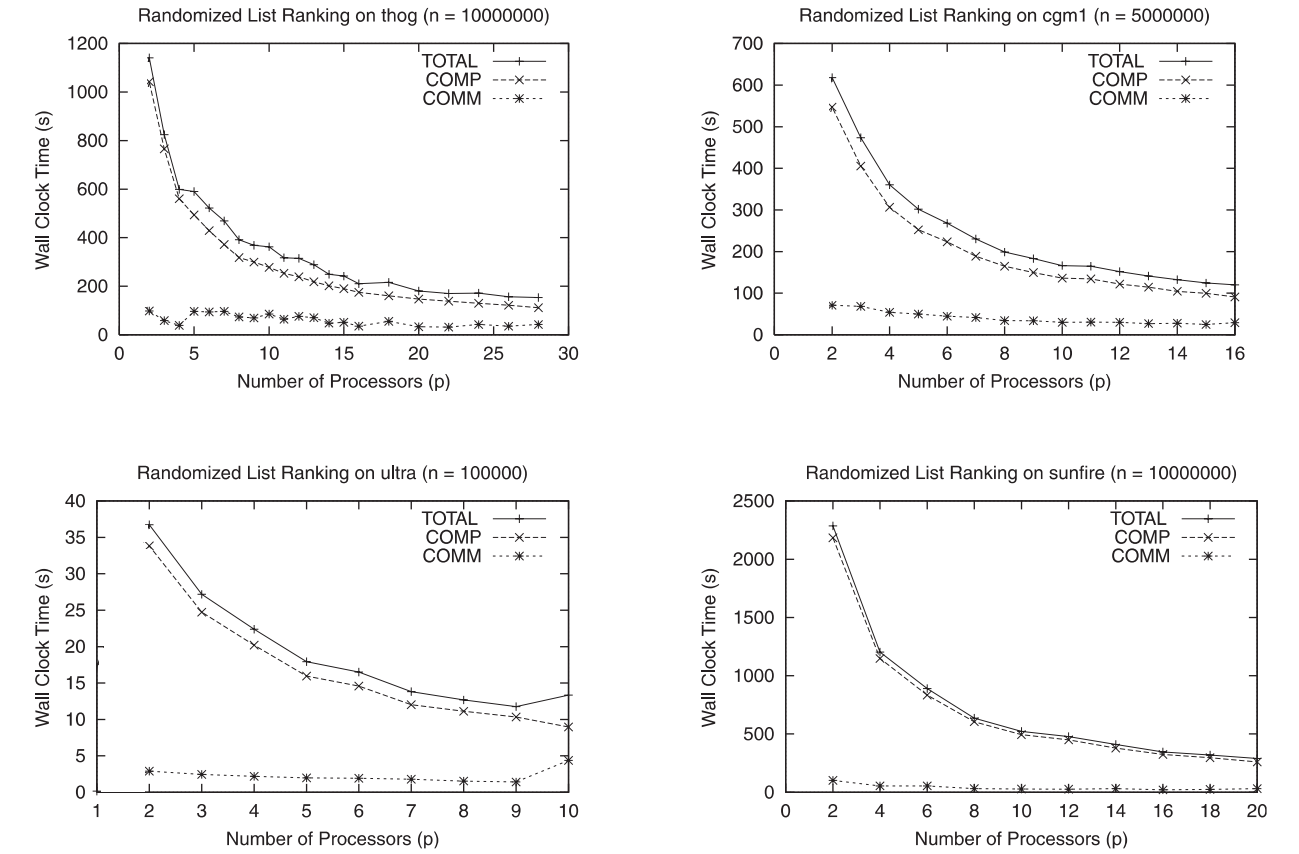

Fig. 7 Performance of the randomized list ranking algorithm.

4.4 PERFORMANCE EVALUATION

We have measured the performance of our randomized and deterministic list ranking methods on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. In the following, we present the results of our experiment. For each operation, we measured the performance on *THOG* with $n = 10{,}000{,}000$, on *CGM1* with $n = 5{,}000{,}000$, on *ULTRA* with $n = 100{,}000$, and on *SUNFIRE* with $n = 10{,}000{,}000$.

Figure 7 shows the performance of the randomized list ranking algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. We observe that for all four machines, the communication time is only a small portion of the total time and appears to be fixed even with growing p. The number of rounds measured in our experiments was usually between 6 and 8, with small fluctuations. Recall that each round has a fixed communication time (except for small p), as discussed in Section 3.3 and that the randomized list ranking requires between $O(p \log n)$ and $O(\log p + \log \log n)$ h-Relation operations. With $\log n$ in the range [1, 5] and $\log \log n$ approximately 4, we expect between five and nine h-Relation operations. What is somewhat surprising at first is that, for our experiments, the number of h-Relation operations remained always in the [6, 8] middle range. However, even if it would not, the fluctuation in number of h-Relation operations would not be large. The experiments show that the $\log p$ factor is growing so slowly in practice that it has little influence on the measured communication time which is essentially independent of p. 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 8 shows the performance of the deterministic list ranking algorithm. Again, we observe that for all four machines, the communication time is a small, essentially

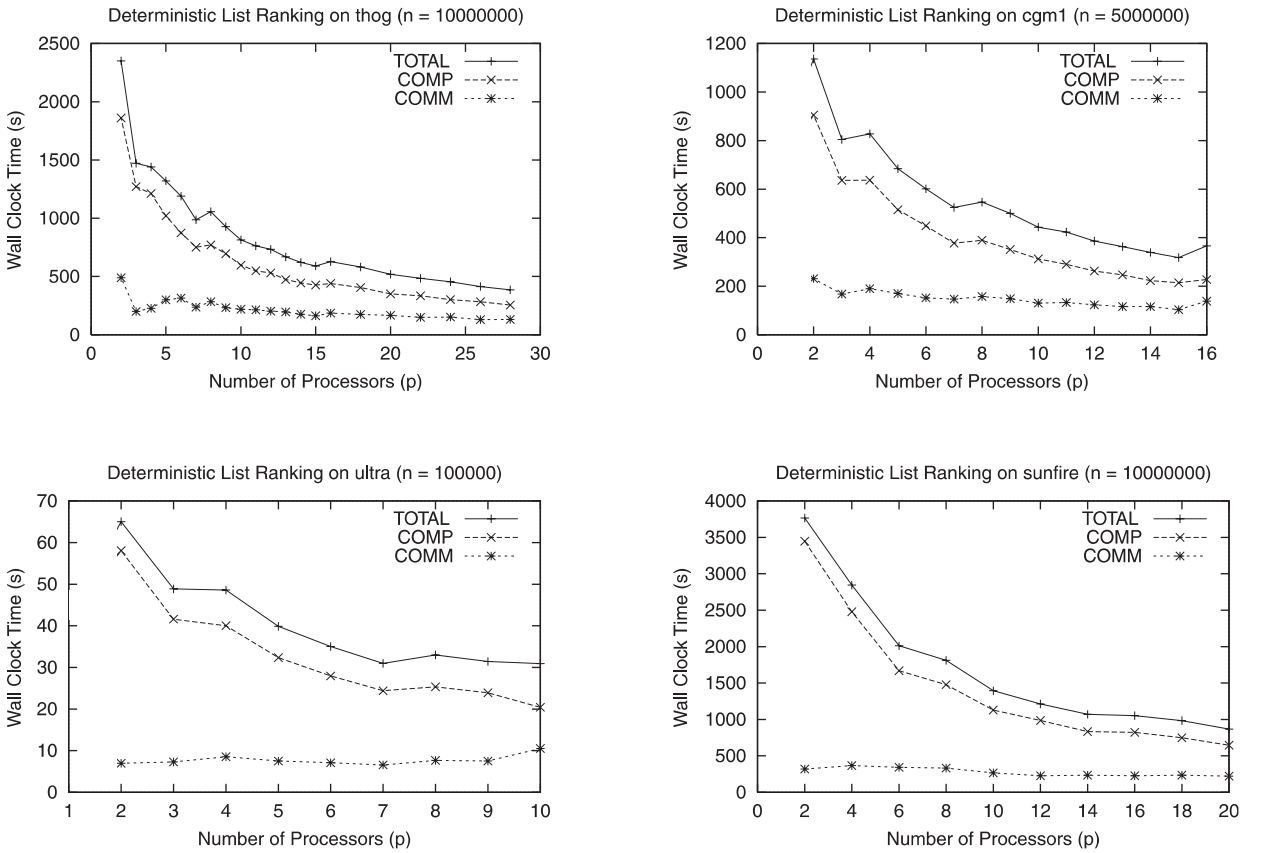

Fig. 8 Performance of the deterministic list ranking algorithm.

fixed, portion of the total time (though somewhat larger than in Figure 7). 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 the number of h-Relation operations grows with $\lfloor \log p \rfloor$, which is 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 9 shows the performance of the Euler tour algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. 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 $p \geq 10$, and on *SUNFIRE* for n = 10,000,000 only with $p \geq 6$.

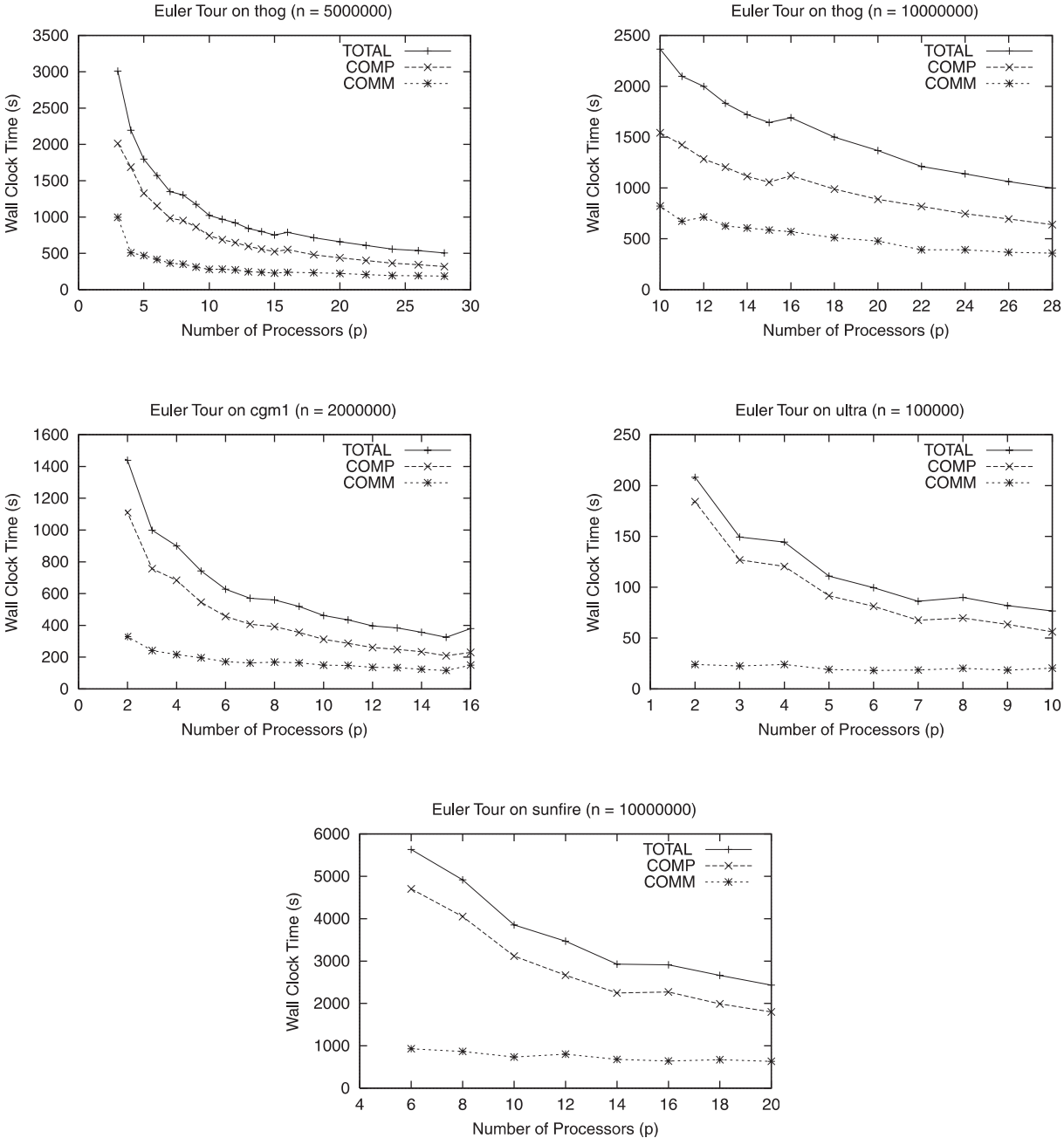

Fig. 9 Performance of the Euler tour algorithm.

5 CGMgraph: Parallel Connected Components and Spanning Forest

CGM*graph* provides a method called `findConnectedComponents (Graph &g, Comm *comm)` for the computation of the connected components of a graph. It also has a method `findSpanningForest (Graph &g, ObjList<Vertex> &spanningRoots, ObjList <Edge> &spanningTreeEdges, Comm *comm)` for the calculation of a 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.

5.1 OUTLINE OF METHOD

We implemented the connected component method described in Caceres et al. (2000). The method also provides immediately a spanning forest of the given graph. The following is a brief outline of the algorithm. For more details, see Caceres et al. (2000). The method consists of two phases.

Phase 1. Using the CGM Request System (Section 3.2), $\log p$ iterations of the standard PRAM algorithm for connected component computation (Reif 1993) are simulated on the CGM. Each round of the PRAM algorithm creates super vertices by combining adjacent vertices (or super vertices). Each round reduces the number of vertices or super vertices by at least a constant factor. Therefore, after $\log p$ iterations, the number of (super) vertices is reduced to $O(n/p)$.

Phase 2. Each processor obtains a copy of the $O(n/p)$ (super) vertices and stores a subset of the edges (which could be still more than n/p in total). Each processor first builds a spanning forest for its subset of edges and then $\log p$ binary merge steps are performed in which pairs of processors merge their two spanning forests into one spanning forest. After $\log p$ merging steps, we obtain one single spanning forest with at most $O(n/p)$ edges.

5.2 PERFORMANCE EVALUATION

We have measured the performance of the connected components and spanning forest algorithms on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. In the following, we present the results of our experiment. For each operation, we measured the performance on *THOG* with n = 10,000,000, on *CGM1* with n = 5,000,000, on *ULTRA* with n = 100,000, and on *SUNFIRE* with n = 10,000,000.

Figure 10 shows the performance of the connected components algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. Figure 11 shows the performance of the spanning forest algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. 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 10 and 11 are very similar. (Note that the second graph in Figure 11 starts only at $p = 2$.) Again, we observe that for all four 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 $1/p$.

6 CGMgraph: Parallel Bipartite Graph Recognition

CGM*graph* provides a method `isBipartiteGraph (Graph &g, Comm *comm)` for recognizing whether a graph is a bipartite graph, i.e. a graph that can be divided into two vertex sets such that no edge connects vertices in the same set. The input is a graph represented as a list of vertices and a list of edges.

6.1 OUTLINE OF METHOD

We implemented the bipartite graph recognition algorithm described in Dehne et al. (2002). The following is a brief outline of the algorithm. For more details, see Dehne et al. (2002).

The method proceeds in four steps. Step 1: a spanning forest of the given graph is computed using the method in Section 5. Step 2: using the method in Section 4, for each spanning tree in the spanning forest, its Euler tour is computed and for each node, the distance to the root of its spanning tree is determined. Step 3: the nodes are categorized into two groups, the nodes with an odd distance to their root and the nodes with an even distance to their root. Step 4: if there is any edge whose two vertices do not belong to different groups, then the graph is not bipartite; otherwise the graph is bipartite, and the two groups partition the vertices into the two disjoint vertex sets given in the definition of bipartite graphs.

6.2 PERFORMANCE EVALUATION

We have measured the performance of the bipartite graph recognition algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. We measured the performance on *THOG* with n = 5,000,000 and n = 10,000,000, on *CGM1* with n = 5,000,000, on *ULTRA* with n = 100,000, and on *SUNFIRE* with n = 10,000,000.

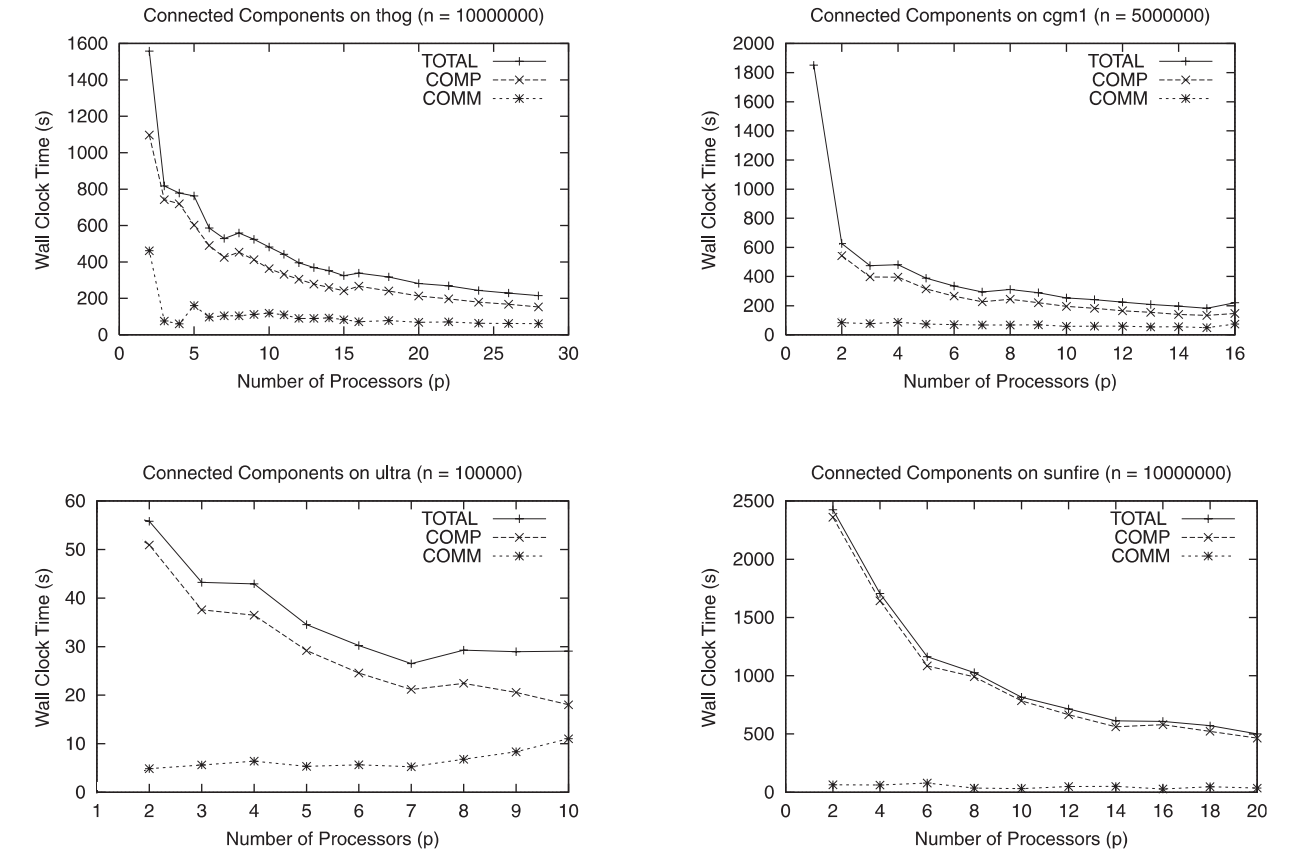

Fig. 10 Performance of the connected components algorithm.

Figure 12 shows the performance of the bipartite graph recognition algorithm on *THOG*, *CGM1*, *ULTRA* and *SUNFIRE*. 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*, *CGM1* and *SUNFIRE*, 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. For *THOG* and *CGM1*, the computation and communication times become equal for larger p but only because of the decrease in computation time. For *SUNFIRE*, the communication time is still less than one half of the computation even for large p ($p = 20$), meaning that the *SUNFIRE* has a behavior similar to the *THOG* and *CGM1* networks.

7 Summary and Future Work

In this paper, we have presented CGM*graph*, the first integrated library of parallel graph methods for PC clusters solving deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition. We have also presented 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. In our

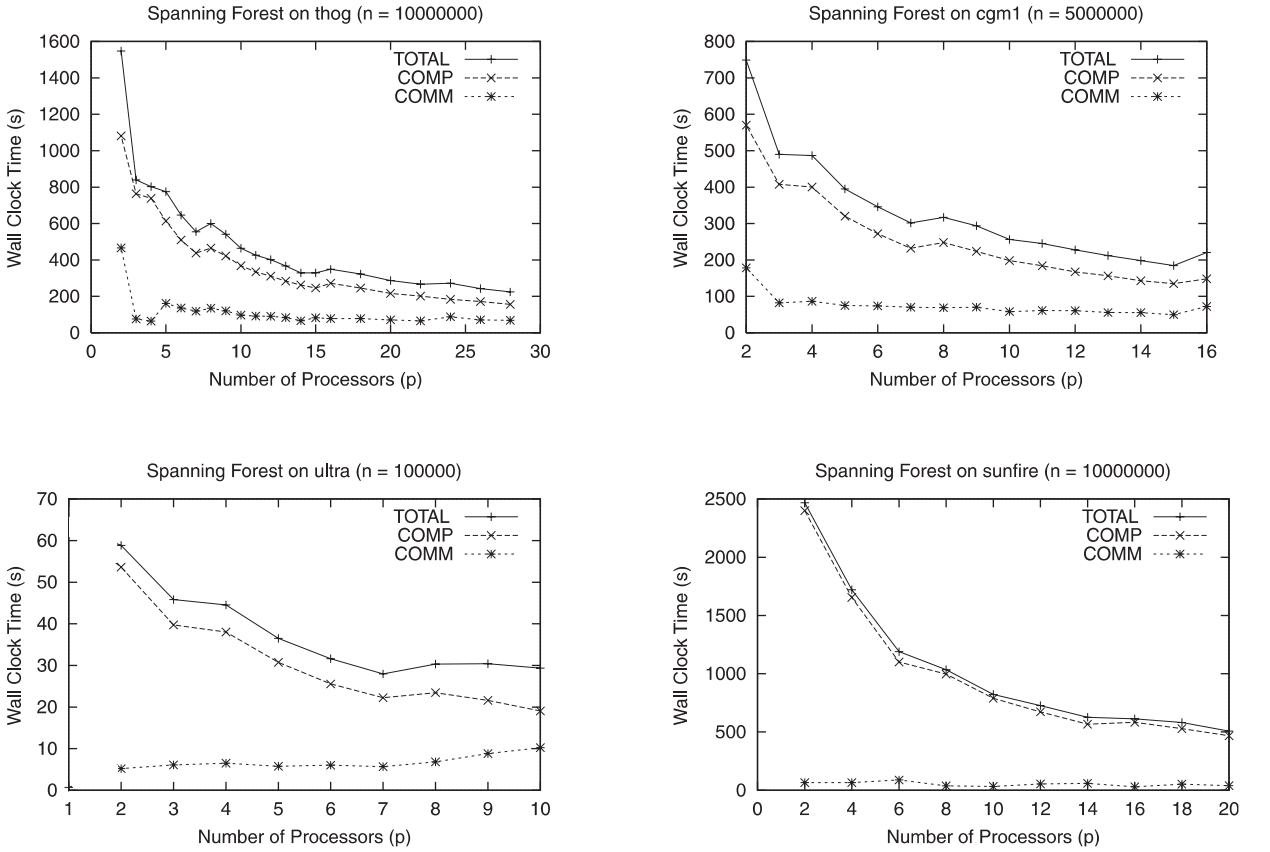

Fig. 11 Performance of the spanning forest algorithm.

experiments, we demonstrated the performance of our methods on four different architectures and showed that our library provides good relative parallel speedup and scalability on all four platforms.

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 (http://www.algorithmic-solutions.com/).

ACKNOWLEDGMENTS

This research is partially supported by the Natural Sciences and Engineering Research Council of Canada. A preliminary version of this paper has appeared in the Proceedings of 10th European PVM/MPI User's Group Meeting (Euro PVM/MPI'03).

AUTHOR BIOGRAPHIES

Albert Chan received a Ph.D. in Computer Science from Carleton University (Ottawa, Canada) in 2003. He is currently Assistant Professor at Fayetteville State University in Fayetteville, North Carolina, USA.

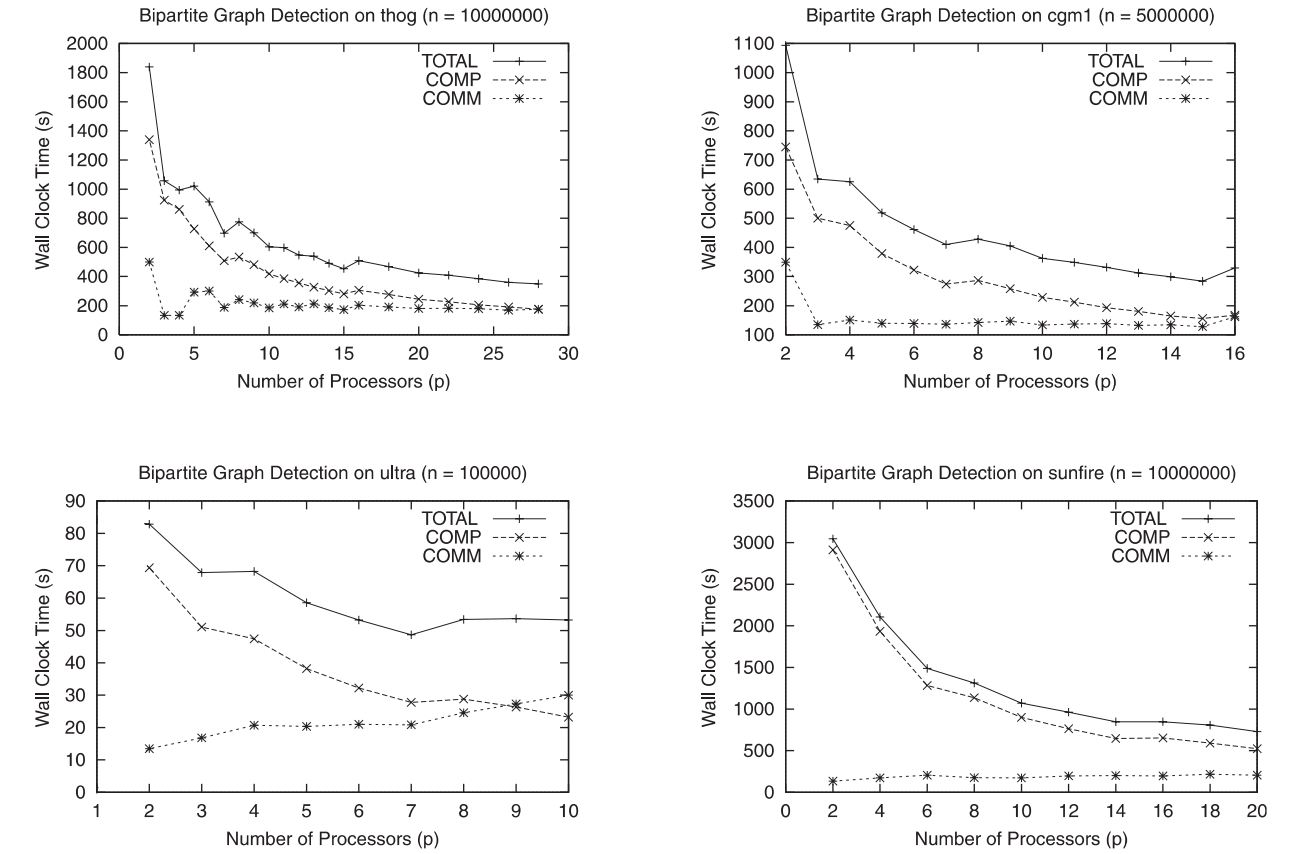

Fig. 12 Performance of the bipartite graph recognition algorithm.

Frank Dehne received a Ph.D. in Computer Science from the University of Wuerzburg (Germany) in 1986. From 1986 to 2004, he was a Professor at Carleton University (Ottawa, Canada). Since April 2004, he is a Professor at Griffith University in Brisbane, Australia. Professor Dehne is a Senior Member of the IEEE, and Editor of *IEEE Transactions on Computers* and *Information Processing Letters*.

Ryan Taylor is a Master student at Carleton University (Ottawa, Canada).

References

Caceres, E., Chan, A., Dehne, F., and Prencipe, G. 2000. Coarse grained parallel algorithms for detecting convex bipartite graphs. *Proceedings of the 26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000), Lecture Notes in Computer Science* Vol. 1928, Springer-Verlag, Berlin, pp. 83–94.

Chan, A. and Dehne, F. 1998. A note on coarse grained parallel integer sorting. Technical Report TR-98-06, School of Computer Science, Carleton University. Available at http://www.scs.carleton.ca/.

Chan, A. and Dehne, F. 1999. A note on coarse grained parallel integer sorting. *Parallel Processing Letters* 9(4):533–538.

Chan, A. and Dehne, F. 2003. CGM*graph*/CGM*lib*: implementing and Testing CGM Graph Algorithms on P Clusters. *Proceedings of the 10th European PVM/MPI Users Group Meeting (Euro PVM/MPI03), Lecture Notes in Computer Science* Vol. 2840, Springer-Verlag, Berlin, pp. 117–125.

Dehne, F., Ferreira, A., Caceres, E., Song, S.W., and Roncato, A. 2002. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica* 33(2):183–200.

Dehne, F. and Song, S.W. 1996. Randomized parallel list ranking for distributed memory multiprocessors. *Asian Computer*

Science Conference (ASIAN '96), Lecture Notes in Computer Science Vol. 1179, Springer-Verlag, Berlin, pp. 1–10.

Dehne, F., Fabri, A., and Rau-Chaplin, A. 1993. Scalable parallel geometric algorithms for coarse grained multicomputers. *ACM Symposium on Computational Geometry*, San Diego, CA, May, pp. 298–307.

Goodrich, M.T. 1996. Communication efficient parallel sorting. *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, Philadelphia, PA, May.

Guerin Lassous, I., Gustedt, J., and Morvan, M. 2000. 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).

Reid-Miller, M. 1994. List Ranking and List Scan on the Cray C-90. *ACM Symposium on Parallel Algorithms and Architectures*, Cape May, NJ, June, pp. 104–113.

Reif, J. 1993. *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA.

Shi, H. and Schaeffer, J. 1992. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing* 14:361–372.

Sibeyn, J.F. 1998. List ranking on meshes. *Acta Informatica* 35(7):543–566.

Sibeyn, J.F., Guillaume, F., and Seidel, T. 1999. Practical parallel list ranking. *Journal of Parallel and Distributed Computing* 56(2):156–180.

Thakur, R., Rabenseifner, R., and Gropp, W. 2005. Optimization of collective communication operations in MPICH. *High Performance Computing Applications* 19(1):49–66.

Valiant, L. 1990. A Bridging Model for Parallel computation. *Communications of the ACM* 33(8).