Parallel Computation on Interval Graphs Using PC Clusters: Algorithms and Experiments

A. Ferreira[1], I. Guérin Lassous[2], K. Marcus[3] and A. Rau-Chaplin[4]

[1] CNRS, INRIA, Projet SLOOP, BP 93, 06902 Sophia Antipolis, France.
ferreira@sophia.inria.fr.
[2] LIAFA – Université Paris 7, Case 7014, 2, place Jussieu, F-75251 Paris Cedex 05.
guerin@liafa.jussieu.fr.
[3] Eurecom, 2229, route des Cretes - BP 193, 06901 Sophia Antipolis cedex - France.
marcus@eurecom.fr.
[4] Faculty of Computer Science, Dalhousie University, P.O. Box 1000, Halifax, NS, Canada B3J 2X4. arc@cs.dal.ca.

Abstract. The use of PC clusters interconnected by high performance local networks is one of the major current trends in parallel/distributed computing. We give coarse-grained, BSP-like, parallel algorithms to solve many problems arising in the context of interval graphs, namely connected components, maximum weighted clique, BFS and DFS trees, minimum interval covering, maximum independent set and minimum dominating set. All of the described p-processor parallel algorithms require only constant or $O(\log p)$ number of communication rounds and are efficient in practice, as demonstrated by our experimental results obtained on a Fast Ethernet based PC cluster.

1 Introduction

The use of PC clusters interconnected by high performance local networks with raw throughput close to 1Gb/s and latency smaller than $10\mu s$ is one of the major current trends in parallel/distributed computing. The local networks are either realized with off-the-shelf hardware (e.g. Myrinet and Fast Ethernet), or application-driven devices, in which case additional functionalities are built-in, mainly at the memory access level. Such cluster-based machines (called henceforth PCC's) typically utilize some flavour of Unix and any number of widely available software packages that support multi-threading, collective communication, automatic load-balance, and others. Note that such packages typically simplify the programmers task by both providing new functionality and by promoting a view of the cluster as a single virtual machine. Clusters based on off-the-shelf hardware can yield effective parallel systems for a fraction of the price of machines using special purpose hardware. This kind of progress may thus be the key to a much wider acceptance of parallel computing, that has been postponed so far, perhaps primarily due to issues of cost and complexity.

Although a great deal of effort has been undertaken on system-level and programming environment issues as described above, little attention has been paid to methodologies for the design of algorithms for this kind of parallel systems.

Despite the availability of a large number of built-in and/or highly optimized procedures, algorithms are still designed at the machine level and claims to portability lay only on the fact that they are implemented using communication libraries such as PVM or MPI.

In this paper we show that theoretical (BSP-like) coarse-grained models are well adapted to PCC's. In particular, algorithms designed for such models are portable and their theoretical and practical performance are closely related. Furthermore, they allow a reduction on the costs associated with software development since the main design paradigm is the use of existing sequential algorithms and communication sub-routines, usually provided with the systems.

Our approach will be to study a class of problems from the start of the algorithm design task until the implementation of the algorithms on a PCC. The class of problems will be those arising on a family of intervals on the real line which can model a number of applications in scheduling, circuit design, traffic control, genetics, and others [16].

Previous Work

This class of problems has been studied extensively in the parallel setting and many work-optimal fine-grained PRAM algorithms have been described in the literature [3, 14, 15, 16]. Their sequential complexity is $\Theta(n \log n)$ in all cases.

Whereas fine-grained PRAM algorithms are likely to be efficient on fine-grained shared memory architectures, it is common knowledge that they tend to be impractical on PCC's due to their failure to exploit locality. Therefore, there has been a recent growth of interest in coarse-grained computational models [4, 5, 18] and the design of coarse-grained algorithms [5, 6, 8, 10, 13].

The BSP model, described by Valiant [18], uses slackness in the number of processors and memory mapping via hash functions to hide communication latency and provide for the efficient execution of fine grained PRAM algorithms on coarse-grained hardware. Culler et. al. introduced the LogP model which, using Valiant's BSP model as a starting point, focuses on the technological trend from fine grained parallel machines towards coarse-grained systems and advocates portable parallel algorithm design [4]. Other coarse grained models focus more on utilizing local computation and minimizing global operations. These include the Coarse-Grained Multicomputer (CGM(n,p)) model used in this paper [5], where n is the input size and p the number of processing elements. In this mixed sequential/parallel setting, there are three important measures of any coarse-grained algorithm, namely, the amount of *local computation* required, the number and type of *global communication phases* required and the *scalability* of the algorithm, that is, the range of values for the ratio $\frac{n}{p}$ for which the algorithm is efficient and applicable. We refer to [5, 9, 10] for more details on this model.

Recently, Cáceres et al. [2] showed that many problems in general graphs, such as list ranking, connected components and others, can be solved in $O(\log p)$ communication rounds in BSP and CGM. However, unlike general graphs, interval graphs can be more easily partitioned and treated in the distributed memory setting. Since each interval is given by its two extreme points, they can be sorted

by left and/or right endpoints and distributed according to this ordering. This partitioning allows us to design less complex parallel algorithms; moreover, the derived algorithms are easier to implement and faster both in theory and in practice.

The following result will be used in the remaining to achieve a constant number of communication rounds in the solution of many problems.

Theorem 1. [13] *Given a set S of n items stored $O(n/p)$ per processor on a CGM(n,p), $n/p \geq p$, sorting S takes a constant number of communication rounds.* □

The algorithms proposed for the CGM are independent of the communication network. Moreover, it was proved that the main collective communication operations can be implemented by a constant number of calls to global sort ([5]). Hence, by Theorem 1, these operations take a constant number of communication rounds. However, in practice these operations will be implemented through built-in, optimized system-level routines. In the remainder, let $T_S(n,p)$ denote the time complexity of a global sort in the CGM.

Our Work

We describe constant communication round coarse-grained parallel algorithms to solve a set of the standard problems arising in the context of interval graphs [16], namely connected components [3], maximum weighted clique [15] and breadth-first-search (BFS) and depth-first-search (DFS) trees [14]. We also propose $O(\log p)$ communication round algorithms for optimization problems as minimum interval covering, maximum independent set [15] and minimum dominating set [17].

In order to demonstrate the practicability of our approach, we implemented three of the above algorithms on a PCC interconnected by a Fast Ethernet backbone. Because of the paradigms used, the programs were easy to develop and are quite portable. The results presented in this paper show that high performance can be achieved with off-the-shelf PCC's along with the right model for algorithm design. Interestingly, super-linear speedups were observed in some cases due to memory swapping effects. Using multiple processors allows us to effectively utilize more RAM and therefore allows computation on data sets that are simply too large to be effectively processed on single processor machines.

In Section 2 the required basic operations are described. Then, in Section 3, chosen problems in interval family model are presented, and solutions are proposed using the basic operations from Section 2. In Section 4, we describe experiments on a Fast Ethernet based PCC. We close the paper with some conclusions and directions for further research.

2 Basic Operations

In the CGM model, any parallel prefix (suffix) associative function to be performed in an array of elements can be done in $O(1)$ communication steps, since

each processor can compute locally the function, and then with a total exchange all the processors get to know the partial result of all the other processors and can compute the final result for each element in the array.

We will also use the pointer-jump operation, to identify the elements in a linked list. This operation can be easily done in $O(\log p)$ communication steps, at each step each processor keeps track of the pointers of its elements.

2.1 Interval Operations

In the following algorithms two functions will be widely used, the *Ominright* and *Omaxright* [1]. Given an interval I, *Omaxright*(I) (*Ominright*(I)) denotes, among all intervals that intersect I, the one whose right endpoint is the furthest right (left). The formal definition is the following.

$$Omaxright(I_i) = \begin{cases} I_j, & \text{if } b_j = \max\{b_k | a_k \leq b_i < b_k\} \\ nil, & \text{otherwise.} \end{cases}$$

The function *Omaxright* can be computed with time complexity $O(T_S(n,p))$, as follows.

1. Sort the left endpoints of the interval in ascending order as $a'_1, a'_2, \ldots, a'_n$.
2. Compute the prefix maxima of the corresponding sequence $b'_1, b'_2, \ldots, b'_n$ of right endpoints and let the result be $b''_1, b''_2, \ldots, b''_n$. ($b''_k = \max_{1 \leq i \leq k}\{b'_i\}$.)
3. For every i $(1 \leq i \leq n)$ compute the rank $r(i)$ of b_i with respect to $a'_1, a'_2, \ldots, a'_n$
4. For every i $(1 \leq i \leq n)$, set *Omaxright*(I_i) = I_j, such that $b_j = b''_{r(i)}$ and $b_i \neq b''_{r(i)}$; otherwise set *Omaxright*(I_i) = *nil*.

We define also the parameter First($\mathcal{I}$) as the segment I which "ends first", that is, whose right endpoint is the furthest left:

$$\text{First}(\mathcal{I}) = I_j, \text{ with } b_j = \min\{b_i | 1 \leq i \leq n\}.$$

To compute it, we need only to compute the minimum of the sequence of right endpoints of intervals in the family $\mathcal{I}$.

Finally, we will use the function next(I) : $\mathcal{I} \to \mathcal{I}$ defined as

$$\text{next}(I_i) = \begin{cases} I_j, & \text{if } b_j = \min\{b_k | b_i < a_k\}, \\ nil, & \text{otherwise.} \end{cases}$$

That is, *next*(I_i) is the interval that ends farthest to the left among all the intervals beginning after the end of I_i. To compute next(I_i), $1 \leq i \leq n$, we use the same algorithm used for *Omaxright*(I_i), with a new step 2.

1. Sort the left endpoints of the interval in ascending order as $a'_1, a'_2, \ldots, a'_n$.

2. Compute the suffix minima of the corresponding sequence $b'_1, b'_2, \ldots, b'_n$ of right endpoints and let the result be $b''_1, b''_2, \ldots, b''_n$. ($b''_k = \min_{k \leq i \leq n}\{b'_i\}$.)
3. For every i $(1 \leq i \leq n)$ compute the rank $r(i)$ of b_i with respect to $a'_1, a'_2, \ldots, a'_n$
4. For every i $(1 \leq i \leq n)$, set $Next(I_i) = I_j$, such that $b_j = b''_{r(i)}$ and $b_i \neq b''_{r(i)}$; otherwise set $Next(I_i) = nil$.

It is easy to see that the above procedure implements the definition of next(I_i), with the same complexity as for computing $Omaxright(I_i)$.

3 Interval Graph Problems and Algorithms

Formally, given a set n of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ on a line, the corresponding *interval graph* $G = (V, E)$ has the set of nodes $V = \{v_1, \ldots, v_n\}$, and there is an edge in E between nodes v_i, v_j if and only if $I_i \cap I_j \neq \emptyset$.

In this section, solutions for some important problems in interval graphs are proposed for the CGM model. Some of these algorithms use techniques derived from their corresponding PRAM algorithms while others require different methods, e.g. to compute the connected components, as shown below.

3.1 Maximum Weighted Clique

A *clique* is a set of nodes that are mutually adjacent. In the *maximum weighted clique problem* for an interval graph, we want to know the maximum weight of such a set, given weights $p(I_i) \geq 0$ on the intervals, and identify a maximum weighted clique by marking its nodes. The CGM algorithm is as follows:

1. Sort the endpoints of the segments such that each processor receives $2n/p$ endpoints.
2. Assign to each endpoint c_i a weight w_i defined by

$$w_i = \begin{cases} p(I_j), & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n, \\ -p(I_j), & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n, \end{cases}$$

3. Compute the prefix sum of the resulting weighted sequence - the maximum obtained is the cardinality of a maximum clique; let $d_1, \ldots, d_{2n}$ denote the resulting sequence.
4. Consider the sequence $e_1, \ldots, e_{2n}$ obtained by replacing every d_j corresponding to a right endpoint of an interval with -1 and compute the rightmost maximum of the resulting sequence; this occurs at a_k.
5. Broadcast a_k. Every interval I_u such that $a_u \leq a_k < b_u$ is marked to be in the final maximum weighted clique.

Due to space limitations, the correctness and the complexity of the algorithm can be found in [7].

Theorem 2. *The maximum weighted clique problem in an interval graph of size n can be solved on a CGM(n, p) in $O(T_S(n, p) + n/p)$ time, with a constant number of communication rounds.* □

3.2 Connected Components

The *connected components* of a graph G are the maximal connected subgraphs of G. The *connected components problem* consists of assigning to each node the label of the connected component that contains it. For the CGM(n,p) we have the following algorithm:

1. Sort the intervals by left endpoints distributing n/p elements to each processor.
2. Each processor P_i computes the connected components for the subgraph corresponding to its n/p intervals, giving labels to the components and associating the labels to the nodes.
3. Each processor detects the farthest right segment amongst its n/p intervals - tail t_i - and broadcasts it (with its label) to all other processors.
4. Each processor checks if any of the tails intersects its components, and updates its local labels using in each case the smallest such new label.
5. Each processor P_i records the pair (t_i, new label) and sends it to processor P_0.
6. Processor P_0 performs a connected components algorithm on the tails and updates the tail labels using the smallest such new labels and sends
   the tails and their new labels to all processors.
7. Each processor updates the labeling accordingly.

Due to space limitations, the correctness and the complexity of the algorithm can be found in [7].

Theorem 3. *The connected components problem in interval graphs can be solved on a CGM(n,p) in $O(T_S(n,p)+n/p)$ time, with a constant number of communication rounds.* □

3.3 BFS and DFS Tree

The problem of finding a Breadth First Search Tree in an interval graph reduces to the problem of computing the function *Omaxright* described earlier. The tree given by the edges $(I_i, Omaxright(I_i))$ is a BFS tree [14]. And the tree formed by the edges $(I_i, Ominright(I_i))$ is a DFS tree [14]. The algorithm is the following:

1. Compute $Omaxright(I_i)$, for $1 \leq i \leq n$.
2. Let $father(I_i) = Omaxright(I_i)$.
3. The edges $(I_i, father(I_i))$ form a BFS tree.

With the appropriate modifications, this algorithm may be used to find a DFS tree. The obtained BFS and DFS trees have their roots in the segments ending farthest to the right in each connected component. With respect to its complexity, the algorithm takes a constant number of communication steps and requires a total running time of $O(T_S(n,p)+n/p)$.

Theorem 4. *Given an interval graph G, BFS and DFS trees can be found using a CGM(n,p) in $O(T_S(n,p)+n/p)$ time, with a constant number of communication rounds.* □

3.4 Minimum Interval Covering

Given a family $\mathcal{I}$ of intervals and a special interval $J=(J_a,J_b)$, the problem of the minimum interval covering is to find a subset $\mathcal{J}\subseteq\mathcal{I}$ such that $J\subseteq\cup(\mathcal{J})$, and $|\mathcal{J}|$ is minimum; i.e., to find the minimum number of intervals in $\mathcal{I}$ needed to cover J. To solve this problem we may only consider the intervals $I_i=(a_i,b_i)\in(\mathcal{I})$ such that $b_i\geq J_a$ and $a_i\leq J_b$. Let $\mathcal{I}_J$ be the family of the intervals in $\mathcal{I}$ satisfying this condition.

An algorithm to solve this problem is as follows:

1. Compute *Omaxright*(I), $I\in\mathcal{I}_J$.
2. Find the interval $I_{\texttt{init}}$ such that $b_{\texttt{init}}=\max\{b_k|a_k\leq J_a\}$.
3. Mark $I_{\texttt{init}}$ and all the intervals in the path given by *Omaxright* pointers beginning at $I_{\texttt{init}}$.

Due to space limitations, the correctness and the complexity of the algorithm can be found in [7].

Theorem 5. *The minimum interval covering problem in interval graphs can be solved using a CGM(n,p) in $O(T_S(n,p)+\log p)$ time, with $O(\log p)$ communication rounds.* □

3.5 Maximum Independent Set and Minimum Dominating Set

The first problem consists of finding a largest set of mutually non-overlapping intervals in the family $\mathcal{I}$, called the *maximum independent set*. The second problem consists of finding a *minimum dominating set*, i.e., a minimum set of intervals which are adjacent to all remaining intervals in the family $\mathcal{I}$. To solve these problems, we simply show a coarse-grained implementation of the algorithms proposed in [17]. In fact, it can be shown that both problems can be solved by building a linked list from First$(\mathcal{I})$:

MAXIMUM INDEPENDENT SET:

1. Compute First$(\mathcal{I})$
2. Compute next(I_i), for i, $1\leq i\leq n$
3. Let father(I_i) = next(I_i), for i, $1\leq i\leq n$
4. Using the pointer-jump operation, mark all the intervals in the linked list given by *father* and beginning at First$(\mathcal{I})$

MINIMUM DOMINATING SET:

1. Compute First($\mathcal{I}$)
2. Compute Omaxright(I_i), for $1 \leq i \leq n$
3. Compute next(I_i), for $1 \leq i \leq n$
4. Let father(I_i) = Omaxright(next(I_i)), for $1 \leq i \leq n$
5. Using the pointer-jump operation, mark all the intervals in the linked list given by *father* and beginning at Omaxright(First($\mathcal{I}$))

The number of communication rounds in each of the algorithms is $O(\log p)$, giving us a total time complexity of $O(T_S(n,p) + \log p)$ and $O(\log p)$ communication rounds. Their correctness stems from the arguments in [17].

Theorem 6. *The maximum independent set and the minimum dominating set problems in interval graphs can be solved using a CGM(n,p) in $O(T_S(n,p)+\log p)$ time, with $O(\log p)$ communication rounds.* □

4 Experimental Results

This section describes the implementations of three of the algorithms presented previously. Our aim here is to demonstrate that these algorithms are not only theoretically efficient but that they lead to simple fast codes in practice. They were implemented on a Fast Ethernet-PCC platform which consists of a set of 12 Pentium Pro 200 Mhz processors each with 64M of RAM that are linked by a 100Mb/s Fast Ethernet network. The processors run the Linux Operating System and the programs are written in *C* utilizing the *PVM* communication library [11] for all interprocessor communications.

Since many of the algorithms rely on sorting, the choice of the sorting method was critical. In the following we first present the implemented sort and its performance before describing the implementation and performance of our algorithms.

4.1 Global Sort

The sorting algorithm implemented is described in [12]. The algorithm requires a constant number of communication steps and its single drawback is that data may not be equally distributed at the end of the sort. Nevertheless, a partial sum procedure and a routing can be used to redistribute the data with a constant number of communication rounds so that each processor stores $\frac{n}{p}$ data in its memory.

Figure 1 shows the execution time for the global sort on an array of integers with the data redistributed in comparison to the sequential performance of quicksort (from the standard *C* library). The results shown are the average of ten execution times over ten different inputs generated randomly. The abscissa represents n the size of the input array and the ordinate the execution time in seconds.

For less than 7,000,000 integers, the achieved speedup is about 2.5 for four processors and 6 for twelve processors. Beyond the size of 7,000,000 integers, the

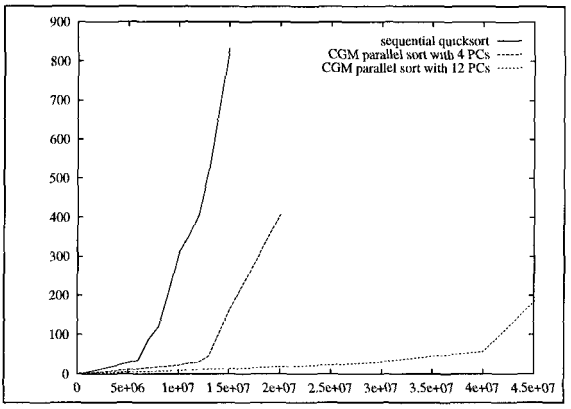

Fig. 1. Sorting on the Fast Ethernet-PCC.

memory swapping effects increase significantly the execution time on a single processor and super-linear speedup is obtained, with 10 for four processors and 35 for twelve processors. Sorting 40,000,000 integers takes less than one minute with twelve processors.

4.2 Maximum Weighted Clique

The algorithm requires a constant number of communication rounds and $O(\frac{n}{p}log\frac{n}{p})$ local operations. Note that only the local computations involved in the sort require $O(\frac{n}{p}log\frac{n}{p})$ operations, whereas all the other steps require only $O(\frac{n}{p})$ operations.

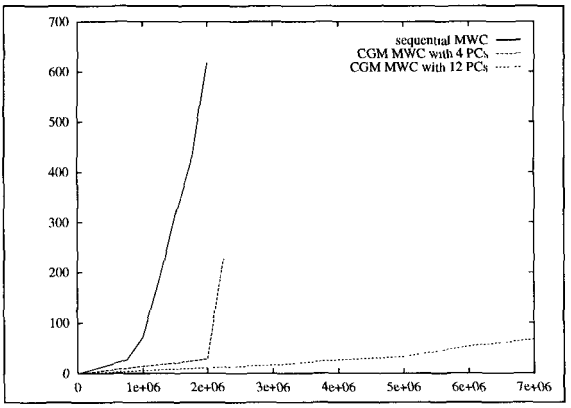

Fig. 2. Maximum Weighted Clique on the Fast Ethernet-PCC.

Figure 2 presents the execution time when the number of intervals increases.

With a graph having less than 1,000,000 intervals, the speedup is 2.5 with four processors, whereas it is equal to 7 for twelve processors. Beyond 1,000,000 intervals, the speedup is 10 for four processors and 35 for twelve processors, these superlinear timings being due to memory swapping effects. Again note that with this algorithm larger data sets than in the sequential case can be handled in a reasonable time.

4.3 Connected Components

As in the maximum weighted clique algorithm above, here also only the sort requires $O(\frac{n}{p}log\frac{n}{p})$ local operations, all the other steps being linear in $\frac{n}{p}$.

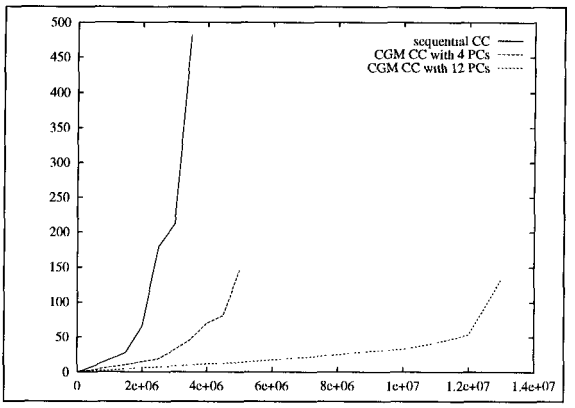

Fig. 3. Connected Components on the Fast Ethernet-PCC.

Figure 3 shows the execution time in seconds as the size of the input increases. The achieved speedup is approximately 2.5 for four processors and 7 for twelve processors for a graph having at most 2,000,000 intervals. With more intervals, the speedup is 8 for four processors and 30 for twelve processors. Also observe that with one processor, at most 2 million of data can be processed whereas twelve processors can process 12 million of data reasonably. Beyond 12 million data items the execution time increases more steeply due to memory swapping effects, even with twelve processors.

4.4 BFS Tree

The achieved speedup is 2 for four processors and 6 for twelve processors with at most 2 million of intervals. Beyond this size, the speedup becomes 7 for four processors and 20 for twelve processors. The measured times are slower than those obtained for the previous algorithms, because two steps of the function Omaxright require $O(\frac{n}{p}log\frac{n}{p})$ operations, whereas for the previous problems only one step required this number of local operations.

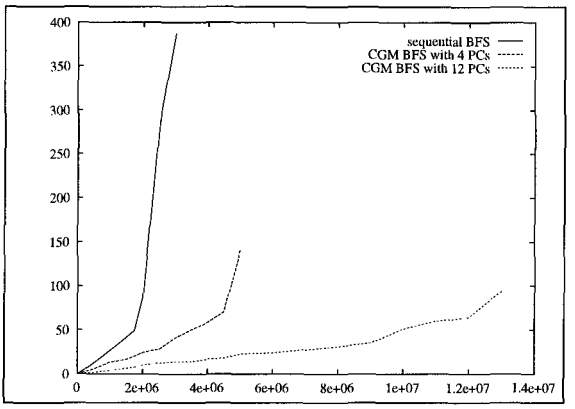

Fig. 4. BFS Tree on the Fast Ethernet-PCC.

5 Conclusion

In this paper we have shown how to solve many important problems on interval graphs using a coarse-grained parallel computer such as a cluster of PC's. The proposed algorithms were shown to be theoretically efficient, easy to implement and fast in practice. We believe this can largely be attributed to the use of the CGM model which accounts for distributed memory effects, mixes sequential and parallel coding, and encourages the use of a constant or very small number of communication rounds.

Note that the use of the CGM model, which was primarily developed for algorithm design in the context of interconnection networks, has led to efficient implementations even in the context of a bus-based network like Ethernet. We speculate that this is due to several factors including: 1) the model focuses on sending a small number of large messages rather than a large number of small ones 2) it relies on standard, and typically well optimized, communications operations and 3) it focuses on reducing the number of communication rounds and therefore the number on interdependencies between rounds. Of course at some point such bus-based networks always become saturated and more attention must be paid to bandwidth and broadcast conflict concerns, particularly as one scales up. We are currently exploring how such concerns can best be dealt with within the context of a CGM-like model.

References

1. A.A. Bertossi and M.A. Bonuccelli. Some Parallel Algorithms on Interval Graphs. *Discrete Applied Mathematics*, 16:101–111, 1987.
2. E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proc. of ICALP'97*, pages 131–143. Lecture Notes in Computer Science. Springer-Verlag, 1997.

3. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
4. D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Shacuser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Princ. and Practice of Parallel Programming*, pages 1–12, 1993.
5. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. 9th ACM Symp. on Computational Geometry*, pages 298–307, 1993.
6. M. Diallo, A. Ferreira, and A. Rau-Chaplin. Communication-efficient deterministic parallel algorithms for planar point location and 2d Voronoi diagram. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science - STACS'98*, Lecture Notes in Computer Science, Paris, France, February 1998. Springer Verlag.
7. A. Ferreira, I. Guerin-Lassous, K. Marcus, and A. Rau-Chaplin. Parallel computation of interval graphs on PC clusters: Algorithms and experiments. RR LIAFA 97-30, University of Paris 7, http://www.liafa.jussieu.fr/~guerin/biblio.html, 1997.
8. A. Ferreira, C. Kenyon, A. Rau-Chaplin, and S. Ubéda. d-dimensional range search on multicomputers. *Algorithmica*, in press. Special Issue on Coarse Grained Algorithms.
9. A. Ferreira and M. Morvan. Models for parallel algorithm design: An introduction. In A. Migdalas, P. Pardalos, and S. Storoy, editors, *Parallel Computing in Optimization*, pages 1–26. Kluwer Academic Publisher, Boston (USA), 1997.
10. A. Ferreira, A. Rau-Chaplin, and S. Ubéda. Scalable 2d convex hull and triangulation for coarse grained multicomputers. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, San Antonio, USA*, pages 561–569. IEEE Press, October 1995.
11. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R Manchek, and V. Sunderman. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, 1994.
12. A.V Gerbessiotis and L.G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, pages 251–267, 1994.
13. M.T. Goodrich. Communication-efficient parallel sorting. In *Proc. of 28th Symp. on Theory of Computing*, 1996.
14. S.K. Kim. Optimal Parallel Algorithms on Sorted Intervals. In *Proc. 27th Annual Allerton Conference Communication, Control and Computing*, volume 1, pages 766–775, 1990.
15. A. Moitra and R. Johnson. PT-Optimal Algorithms for Interval Graphs. In *Proc. 26th Annual Allerton Conference Communication, Control and Computing*, volume 1, pages 274–282, 1988.
16. S. Olariu. Parallel graph algorithms. In A. Zomaya, editor, *Handbook of Parallel and Distributed Computing*, pages 355–403. McGraw-Hill, 1996.
17. S. Olariu, J.L. Schwing, and J. Zhang. Optimal Parallel Algorithms for Problems Modelled by a Family of Intervals. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):364–374, 1992.
18. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.