COARSE GRAINED PARALLEL COMPUTING ON HETEROGENEOUS SYSTEMS*

Pat Morin
Carleton University School of Computer Science
1125 Colonel By Dr.
Ottawa, Canada
K1S 5B6
email: *morin@scs.carleton.ca*

Keywords: parallel algorithms, heterogeneous computing, coarse grained multicomputer, bulk synchronous parallel

Abstract

Coarse grained parallel (CGP) computing models such as the coarse grained multicomputer (CGM), bulk synchronous parallel (BSP), and LogP models have received considerable attention recently from the parallel computing community. This paper examines a new application of CGP algorithms, namely in heterogeneous systems, and shows that this approach to heterogeneous computing has a number of advantages over traditional approaches. A hetegerogeneous CGP model of computation is defined, and a number of algorithms and basic communication operations are developed for this model. These algorithms have been implemented in the form of a reusable and extendable library which simplifies the task of programming heterogeneous systems. Empirical results are given which show that this approach performs very well in practice.

1 INTRODUCTION

Assessing the impact of heterogeneity in parallel computing systems is becoming increasingly important. Individuals with limited budgets can now build workstation clusters from off-the-shelf processing components and interconnection networks [4, 19]. High speed networks are being used to interconnect traditional supercomputers in order to direct large amounts of computing power at Grand Challenge problems [3]. Even traditional supercomputers usually consist of a very fast workstation host connected to a number of slower in-the-box processors.

*The author received funding from the Natural Science and Engineering Research Council of Canada. Part of the work was done while the author was a visitor at theHeinz-Nixdorf Institute at the University of Paderborn.

The three situations above, which cover nearly all modern parallel computing systems, are all potential examples of heterogeneous systems. (Here and throughout the remainder of the paper, the term *heterogeneous system* refers to a system in which processors have differing speeds.) In the case of workstations clusters, the processing components may be different because the system was grown incrementally and newly added processors are more modern than the originals. In the case of supercomputer clusters, the supercomputers may come from different manufacturers. Finally, in the case of traditional supercomputers, it may be useful to use the host processor, particularly for sequential portions of computations.

Traditionally, there have been two approaches to dealing with the varying processor speeds in such systems. The first and simplest approach, which we call the *ostrich approach* is to simply ignore the difference in processor speeds and use standard parallel algorithms. In many cases, this leads to the slowest processor becoming a bottleneck, and effectively reduces performance to that of a machine in which all processors are equally slow. This can result in decreased performance when slow processors are added to a system.

The second approach, which we call the *overpartitioning approach* is to break the problem into small subproblems, so that there are many more subproblems than processors, and assign subproblems to processors whenever they become idle. This approach also has its disadvantages. Decomposing the problem and merging the solutions to subproblems is not always easy, nor is coordinating the processors, and these tasks have an overhead associated with them. Even worse, because of the high latency of communications networks, many processor cycles are wasted waiting for the network to deliver subproblems. In most cases, a healthy dose of performance testing, algorithm analysis, and common sense is required to determine the optimum subproblem size, and this procedure must be repeated when the system configuration changes.

The approach taken in this paper is to modify fast parallel algorithms which have been shown to be efficient in homogeneous systems to run efficiently on heterogeneous systems. The class of algorithms we choose as our starting point is the class of *coarse grained parallel* (CGP) algorithms. Examples of such algorithms include algorithms for the *bulk synchronous parallel* (BSP) [22], *Coarse Grained Multicomputer* (CGM) [9], and *LogP* [7] models of parallel computation.

In these models a parallel computer is composed of p processors and is being used to solve a problem of size n, where typically $p \ll n$. The basic communication operation is the h-relation, an all-to-all communication operation in which no processor is the source or destination of more than h words. Algorithms based on these models work in supersteps, where a superstep consists of local computation, followed by global communication (routing an h-relation). The goal of algorithm design is to simultaneously minimize communication and computation.

The heterogeneous networks described above present a problem for CGP algorithms, since the slow processors in the network become a bottleneck for the computation. This is due to the fact that CGP algorithms are designed to distribute computation load evenly across processors. However, through careful modifications, these algorithms can be made to distribute computation load according to processor speeds without sacrificing efficiency.

This approach has the obvious advantage over the ostrich approach that it balances the computation according to processor speed and therefore improves performance (Section 7 bears this out with empirical evidence). This approach has two advantages over the overpartitioning approach. The first is that it minimizes the effects of latency (most of the algorithms described in Section 6 perform only a constant number of communication operations). The second is that it doesn't require extensive testing and measurements to determine optimum algorithm parameters. In fact, the only parameters used by the algorithms are the processor speeds.

The main contributions of this paper are the following:

1. The definition of a parallel computation model called HCGM which takes into account varying processor speeds—The model is simple enough to be easy to use, accurate enough to allow for the development of truly efficient algorithms, and portable enough to allow these algorithms to run efficiently on a wide variety of parallel architectures.
2. The identification of a number of communication patterns most commonly used in CGP algorithms and efficient HCGM algorithms for their implementation—These algorithms form the basis for translating existing CGP algorithms into HCGM algorithms.
3. A number of algorithms for the HCGM model—These algorithms are arrived at by describing existing CGM and BSP algorithms in terms of the previously mentioned communication patterns.
4. An implementation of these ideas—The implementation consists of a library of the previously mentioned communication patterns and some algorithms.

The remainder of the paper is organized as follows: Section 2 reviews related work. Section 3 describes the CGM model and defines a generalization of this model, the Heterogeneous CGM (HCGM). Section 4 examines common communication patterns used by CGP algorithms. Section 5 shows how these patterns can be implemented on the HCGM model. Section 6 presents a number of algorithms for the HCGM model based on the HCGM versions of the communication patterns. Section 7 describes an implementation of these ideas and presents some empirical results which validates both the model and the algorithms. Finally, Section 8 summarizes and suggests directions for future work in this area.

2 RELATED WORK

The topic of data partitioning in heterogeneous systems with simple fixed communication patterns is addressed in [6, 18], and semi-automatic methods of choosing the best partitioning scheme and parameters are described. Methods for the compile time scheduling of various types of parallel loops are described in [5]. The results in this paper go beyond these in that the problems addressed have much less structure than simple stencilling operations on 2D grids or uniform parallel loops whose communication patterns can be analyzed at compile time. In Section 6 algorithms are presented for sorting, median finding, and a number of computational geometry problems whose communication patterns are input dependent.

Methods for dynamic load balancing such as those described in [20, 14, 23] can also be applied to heterogeneous systems. All these methods fall into the overpatitioning strategy category. The advantages of our strategy over such overpartitioning strategies have been described in Section 1. These are the minimization of the effects latency and simplicity of the algorithm parameters.

In [25] a mathematical model of a network of workstations is described. In [24], the authors describe a stochastic performance prediction methodology for this model based on the task graph of the parallel application. Although this model is an accurate predictor of performance, it is not clear that the model leads to the development of efficient algorithms. In fact, in the matrix multiplication tests described in [24], a 2 processor configuration actually outperforms a 12 processor configuration.

An important difference between the model in [25, 24] and the HCGM model is that the HCGM model is not intended to predict exact running times of parallel algorithms on parallel machines. Rather, it is designed to distinguish between "good" and "bad" algorithms, i.e., if the model says that algorithm $\mathcal{A}$ is better than algorithm $\mathcal{B}$, then $\mathcal{A}$ should perform better than $\mathcal{B}$ when implemented. This makes the HCGM model simpler, which in turn leads to a much simpler algorithms analysis procedure.

3 CGM AND HCGM

In this section, we review the CGM model introduced by Dehne *et. al.* in [9] and define a generalization of this model which we call the HCGM. Although we use the CGM model as our starting point, similar modifications could be incorporated into the BSP model or LogP models.

A *coarse grained multicomputer*, CGM(m,p), consists of p identical processors, labelled $P_0,\ldots,P_{p-1}$. These processors are interconnected by a communication network capable of routing an h-relation with $h = O(\frac{m}{p})$. When discussing the performance of a CGM algorithm there are 3 items of interest: (1) computation time, (2) number of supersteps, and (3) restrictions on n, m, and p. As an example, the Sample Sort algorithm in [13] uses $\tilde{O}(\frac{n}{p}\log n)$ computation time and $\tilde{O}(1)$ supersteps on a CGM(n,p), where $\frac{n}{p} \geq p\log n$.[1]

A *heterogeneous coarse grained multicomputer* HCGM(m,p,s) consists of p possibly heterogeneous processors labelled $P_0,\ldots,P_{p-1}$. The value $s = \sum_{i=0}^{p-1} s_i$ represents the total speed of the parallel machine, where s_i represents the speed of P_i and is an integer. Each processor, P_i, can perform w units of work in $\frac{w}{s_i}$ time units. For conciseness, we define $s^{max} = \max\{s_i : 0 \leq i \leq p-1\}$ and $s^{min} = \min\{s_i : 0 \leq i \leq p-1\}$, i.e., s^{max} and s^{min} are the speeds of the fastest and slowest processors, respectively. Similarly, $P^{max} = P_{\min\{i:s_i=s^{max}\}}$ and $P^{min} = P_{\min\{i:s_i=s^{min}\}}$, i.e., P^{max} is a representative fastest processor, and P^{min} is a representative slowest processor.

The p processors of an HCGM(m,p,s) are interconnected by a network capable of routing any all-to-all communication in which the total amount of data exchanged is $O(m)$. However, these communication operations incur a penalty in computation time. If P_i is the source (resp. destination) of $O(b)$ bytes of information, then P_i incurs a penalty in computation time of $O(\frac{b}{s_i})$. This represents the local computation needed to pack (resp. unpack) messages into (resp. from) buffers. For example, the computation time associated with routing an h-relation is $\max\{\frac{h}{s_i} : 0 \leq i \leq p-1\} = \frac{h}{s^{min}}$.

Like a CGM algorithm, the performance of an HCGM algorithm is measured in terms of local computation time and the number of supersteps. Both of these quantities can be functions of n, p, s, and $s_0,\ldots,s_{p-1}$. Ideally, an HCGM(m,p,s) algorithm gives a speedup of s when compared to a uniprocessor machine with unit speed running the fastest sequential algorithm for the same problem. This speedup should be independent of the values of $s_0,\ldots,s_{p-1}$.

[1]We say that a randomized algorithm has running time $\tilde{O}(f(n))$ if the probability that the algorithm's running time exceeds $cf(n)$ is less than or equal to $1/poly(n)$ where $poly(n)$ is a polynomial whose degree depends on c. We say that such an algorithm has running time $O(f(n))$ *with high probability.*

We assume that the input to a HCGM(m,p,s) algorithm is initially distributed in a load balanced manner, that is, each P_i initially holds $\frac{s_i}{s}n$ input elements. At this point we note that the HCGM(m,p,s) model is equivalent to the CGM(m,p) model when $s_0 = s_1 = \cdots = s_{p-1} = 1$.

One possible approach to developing HCGM algorithms directly from BSP and CGM algorithms is to have each processor, P_i, simulate $s_i/\gcd(s_0,\ldots,s_{p-1})$ virtual CGM processors, where $\gcd(s_0,\ldots,s_{p-1})$ denotes the greatest common divisor of $s_0,\ldots,s_{p-1}$. Although this approach leads to perfect load balancing, it has at least three problems.

1. The overheads associated with automatically simulating virtual processors can have a significant negative impact on real running times. These overheads can be avoided by having implementors code the simulation by hand, but this adds complexity to the already difficult task of implementing parallel algorithms.
2. In some cases the number of supersteps in a CGM algorithm is a function of the number of processors, so increasing the number of processors by introducing virtual processors increases the number of supersteps.
3. Most CGP algorithms require restrictions on n and p in order to work efficiently, and increasing p by introducing virtual processors may violate these restrictions.

As an example of the difference between an optimal algorithm on the CGM model and the HCGM model, we consider the Sample Sort algorithm described in [13]. The original Sample Sort algorithm has a running time of

$$\begin{aligned}\tilde{O}\left(\max\left\{\frac{\frac{n}{p}\log n}{s_i}\right\}\right) &= \tilde{O}\left(\frac{\frac{n}{p}\log n}{s^{min}}\right)\\ &= \tilde{O}\left(\frac{n}{s^{min}p}\log n\right)\end{aligned}$$

on an HCGM(n,p,s), that is, the running time is dominated by the speed of the slowest processor so that the running time is the same as if we had p processor each with speed s^{min}. However, if the algorithm could somehow be modified so that during each round, each P_i received $\frac{s_i}{s}n$ keys then the running time would be given by

$$\tilde{O}\left(\frac{\frac{s_i n}{s}\log n}{s_i}\right) = \tilde{O}\left(\frac{n}{s}\log n\right),$$

which is optimal up to constant factors, since a single processor with unit speed requires $O(n\log n)$ time using the best sequential algorithm.

4 CGP COMMUNICATION PATTERNS

In this section common communication patterns used in CGP algorithms are discussed. A survey of the literature on CGP algorithms reveals that although there are many possible communication patterns available with the h-relation, most algorithms can be implemented using a small number of well defined communication patterns. These patterns are listed below. Beside each pattern is a reference to some algorithms which use it.

Pattern 1 (CGM-Prefix-Sum). [10, 9]. Compute the prefix sum of a sequence of n elements. Each processor locally computes the prefix sum of it's subsequence and sends the total sum to P_0. P_0 computes the prefix sum of this sequence and sends the ith element of this prefix sum to P_i. P_i then adds this value to each element of the prefix sum computed in the first step to obtain the prefix sum of the overall result.

Pattern 2 (CGM-Random-Sample). [13, 8]. Take a small random sample of the input. Each element is chosen as a sample element with probability $\frac{r}{n}$ where $r \leq \frac{n}{p}$ is the desired sample size. All the samples are then routed to P_0.

Pattern 3 (CGM-Random-Assign). [1, 2, 11]. Randomly assign each input element to a processor. Each processor places each of its elements into one of p buckets with equal probability. The contents of bucket i are then routed to P_i.

Pattern 4 (CGM-Linear-Partition). [13, 9]. Partition the input in such a way that each element at P_i is less than each element at P_j, for all $i < j$. The input is sampled using the Sample pattern. The sample is then sorted, and p *splitters* are chosen at uniform intervals from the sorted sample. Each of the input elements is then assigned to one of p buckets depending on which pair of splitters it falls between in the sorted order. Finally, the contents of bucket i are routed to P_i.

Pattern 5 (CGM-Circulate). This pattern takes two ordered lists A and B of size $O(n)$ as input. The computation proceeds in p rounds. During each round each processor sends and receives some portion of B of size $\frac{n}{p}$, and performs some computation on its locally stored portions of A and B. After the p rounds, each element of B has been stored in the same processor as each element in A during exactly one round. The nature of the computation performed in each round may vary, but the running time must be of the form $O(\frac{n^c}{p}|A_i|)$, where A_i is the sublist of A stored at P_i. This pattern, which is part of the folklore, is a simple technique that can be used to parallelize many sequential algorithms with running times of $O(n^2)$ or higher. Examples include a p round matrix multiplication algorithm and the Floyd-Warshall all pairs shortest path algorithm.

Fact 1. *The communication patterns described above can be implemented on a* CGM(n, p) *with the following running times and restrictions on n and p. For more details the reader is referred to the cited references.*

Pattern	*Supersteps*	*Computation*	*Restrictions*
1	$\tilde{O}(1)$	$\tilde{O}(\frac{n}{p})$	$\frac{n}{p} \geq p$
2	$\tilde{O}(1)$	$\tilde{O}(\frac{n}{p})$	$\frac{n}{p} \geq \log n$
3	$\tilde{O}(1)$	$O(\frac{n}{p})$	$\frac{n}{p} \geq \log n$
4	$\tilde{O}(1)$	$\tilde{O}(\frac{n}{p}\log p)$	$\frac{n}{p} \geq p \log n$
5	$O(p)$	$O(\frac{n^{c+1}}{p})$	$n \geq p$

The careful reader may have noticed that sorting, which is viewed by many as a basic communication operation, is not included in the list of communication patterns. We also view sorting as a basic communication operation, and note that sorting is nothing more than a Linear-Partition followed by a local sort.

5 COMMUNICATION PATTERNS ON AN HCGM

In this section, modifications to the patterns of Section 4 are given which allow them to run efficiently on an HCGM(n, p, s). Recall that the input to an HCGM algorithm is initially load balanced, that is, each processor, P_i, holds $\frac{s_i}{s}n$ elements. The modifications to the patterns are aimed at maintaining this load balanced state as much as possible.

Pattern 6 (HCGM-Prefix-Sum). To obtain a load balanced prefix sum computation we simply have processor P^{max} (rather than P_0) do the work of computing the intermediate prefix sum of size p in the second step.

Pattern 7 (HCGM-Random-Sample). The only possible form of load balancing for this pattern is to have the sample elements routed to P^{max} (rather than P_0) so that computations on the sample can be done as quickly as possible.

Pattern 8 (HCGM-Random-Assign). In order to include load balancing in this pattern, we need only change the probability with which an element is assigned to a bucket. In particular, the probability that an element is assigned to bucket i is given by $\frac{s_i}{s}$. In this way, the expected number of elements that arrive at P_i is $\frac{s_i}{s}n$.

Pattern 9 (HCGM-Linear-Partition). Adding load balancing to this pattern involves changing the manner in which the splitters are chosen from the r sorted sample keys. Rather than choosing the splitters at uniform intervals, the splitters are chosen so that the number of sample keys which fall between splitter i and splitter $i+1$ is $\lfloor \frac{s_i}{s} r \rfloor$. In this way, the expected number of input keys which fall between splitter i and splitter $i+1$ is approximately $\frac{s_i}{s}n$.

Pattern 10 (HCGM-Circulate). This pattern can be load balanced by distributing A such that P_i stores $\frac{s_i}{s}n$ elements of A.

Theorem 1. *The communication patterns described above can be implemented on an* HCGM(n, p, s) *with the following running times and restrictions on n, p and s.*

Patt.	*Supersteps*	*Computation*	*Restrictions*
6	$O(1)$	$O(\frac{n}{s})$	$\frac{s^{max}}{s}n \geq p$
7	$O(1)$	$\tilde{O}(\frac{n}{s})$	$\frac{s^{max}}{s}n \geq \log n$
8	$O(1)$	$\tilde{O}(\frac{n}{s}\log p)$	$\frac{s^{min}}{s}n \geq \log n$
9	$O(1)$	$\tilde{O}(\frac{n}{s}\log p)$	$\frac{s^{max}}{s}n \geq \frac{s}{s^{min}}\log n$
10	$O(p)$	$O(\frac{n^{c+1}}{s})$	$n \geq s$

Proof Sketch. Due to space limitations we can only outline the proofs and defer complete proofs to the full version of the paper.

Parts 6 and 7 can be seen by observing that each processor, P_i does $O(\frac{s_i}{s}n)$ work, except for P^{max}. In Part 6, P^{max} does an additional $O(p) \subseteq O(\frac{s^{max}}{s}n)$ work. In Part 7, P^{max} does an additional $\tilde{O}(r) \subseteq \tilde{O}(\frac{s^{max}}{s}n)$ work.

Part 8 can be proven using Chernoff bounds to show that the number of elements which arrive at P_i is $\tilde{O}(\frac{s_i}{s}n)$. The $\log p$ factor in the running time comes from the fact that a binary search must be used to find which processor each element is assigned to.

Part 9 can be proven by using Chernoff bounds to show that, for properly chosen r, the number of *samples* in the $c\frac{s_i}{s}n$ *keys* which follow splitter i is greater than $\lfloor \frac{s_i}{s}r \rfloor$, with high probability. Therefore, at most $c\frac{s_i}{s}n$ keys are assigned to P_i, with high probability.

Part 10 follows from the definition of the circulate pattern since the work done by P_i during a single round is $O(\frac{n^c}{p} \cdot \frac{s_i}{s}n)$. Over p rounds this becomes $O(\frac{s_i}{s}n^{c+1})$. □

6 HCGM ALGORITHMS

In this section, the work on communication patterns finally pays off in that it provides a large number of algorithms for the HCGM model. This is due to the fact that many existing CGM and BSP algorithms can be expressed solely in terms of commonly occuring communication patterns. Since HCGM versions of these patterns exist, so do HCGM versions of these algorithms. Rather than describe all HCGM algorithms in detail, we reference the original CGM or BSP algorithm on which it is based.

Theorem 2. *The following problems can be solved on an* HCGM(n,p,s), *provided that* $\frac{s^{max}}{s}n \geq \frac{s}{s^{min}} \log n$, *using the stated amount of computation time and* $O(1)$ *supersteps:*

1. *sorting* n *keys:* $\tilde{O}(\frac{n}{s}\log n)$,
2. m *priority queue operations on a priority queue of size* n*:* $\tilde{O}(\frac{m}{s}\log n)$,
3. *finding the median of* n *elements:* $\tilde{O}(\frac{n}{s})$.
4. *a number of 2 and 3-dimensional computational geometry problems on inputs of size* n*:* $\tilde{O}(\frac{n}{s}\log n)$, *and*
5. *computing the medial axis transform of a* $\sqrt{n} \times \sqrt{n}$ *image:* $O(\frac{n}{s})$.

Proof Sketch. Due to space limitations we show only how existing algorithms can be expressed in terms of common communication patterns and defer complete proofs to the full version of the paper.

1. Sorting can be implemented as a Linear-Partition followed by a local sort [13].
2. The priority queue insertion algorithm in [2] consists of a Random-Assign followed by repeated insertions into a local priority queue. The deletion algorithm can be implemented by having each processor delete $c\frac{s_i}{s}m$ keys from a local priority queue, sorting the deleted keys, and reinserting the keys with rank $> m$.
3. The selection algorithm in [12] can be implemented as a random sample, a global sort, and a prefix sum.
4. The computational geometry algorithms described in [9] use only one communication operation, global sorting.
5. The medial axis transform algorithm in [10] uses only scan (prefix-sum) operations and local computation.

□

Next, we point out some simple algorithms on matrices that are based on the Circulate pattern.

Theorem 3. *The following problems can be solved on a* HCGM(n^2,p,s) *using* $O(\frac{n^3}{s})$ *computation time and* $O(p)$ *supersteps: multiplication of two* $n \times n$ *matrices, Gaussian elimination on an* $n \times n$ *matrix, and the all pairs shortest path problem on* n *vertices.*

Proof Sketch. These three problems can be solved using the Circulate pattern. The set A consists of the columns of a matrix distributed in a load balanced manner. The set B consists of the rows or columns of a matrix distributed in p groups of size $\frac{n}{p}$. By circulating the B set among the processors, each processor, P_i, sees every element of the matrix, and can solve its subproblem of size $\frac{s_i}{s}n^2$ in the stated time bound. □

As these two theorems show, a large number of existing CGP algorithms can be made into HCGM algorithms with minimal effort. Given a library which includes the above communication patterns, then little extra effort is needed on the part of the programmer to include support for heterogeneous systems in her implementation.

7 IMPLEMENTATION RESULTS

The algorithms for the communication patterns described in Section 5 and some of the algorithms in Section 6 have been implemented as part of the PLEDA library, an ongoing project whose goal is to supply a portable library of efficient parallel data structures and algorithms [16]. This work builds on the LEDA library of sequential data structures and algorithms [17]. The library is written in C++ [21] and uses MPI [15] for message passing.

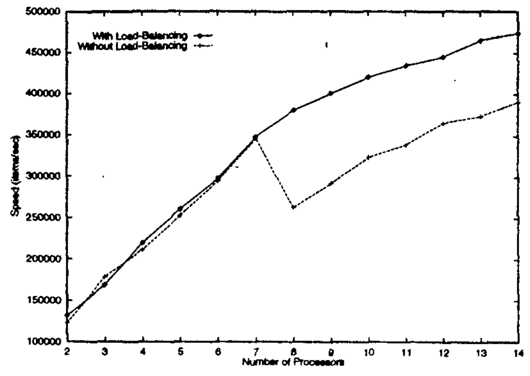

Figure 1: Performance of CGM and HCGM versions of Sample Sort.

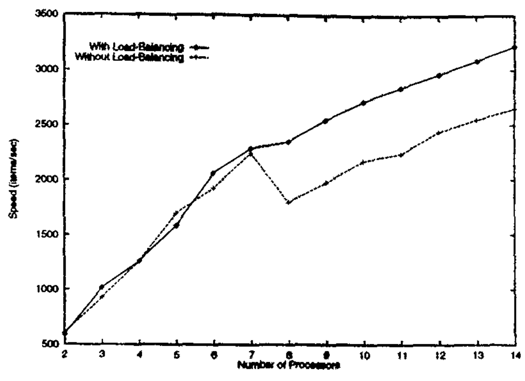

Figure 2: Performance of CGM and HCGM versions of Floyd-Warshall algorithm

Timing results are presented for a sorting algorithm, which uses the Random-Sample and Linear-Partition patterns, and for the Floyd-Warshall all pairs shortest path algorithm, which is based on the Circulate pattern. These results were obtained on a dedicated cluster of workstations consisting of 16 166MHz Pentium processors interconnected by a 100MHz Ethernet switch, running Linux, and using the LAM MPI implementation. In order to simulate slow processors, a crippling process was launched on those processors in order to reduce their effective speed. Crippling processes do nothing but spin in a tight loop performing useless calculations, effectively reducing the speed of the processor to $\frac{1}{2}$ its usual speed. For these tests up to 14 processors were used.[2] P_0 through P_6 were run at the regular speed, while P_7 through P_{13} were crippled.

Figure 1 compares the results of using the HCGM Linear-Partition algorithm and then sorting locally against the results obtained by standard Sample Sort [13]. The test sorts a list of $2.5 \cdot 10^6$ integers, using the LEDA implementation of quicksort as the local sorting function. In both cases, the input is initially distributed in a load balanced manner. It is clear from Figure 1 that the HCGM version (labelled "With Load-Balancing") of the algorithm performs much better than the standard version (labelled "Without Load-Balancing") when slow processors are introduced into the system.

In order to measure the performance of another class of HCGM algorithms we implemented a CGP version of the Floyd-Warshall all pairs shortest path algorithm which uses the Circulate pattern on the columns of the adjacency matrix. The results of running this test with $n = 1.0 \cdot 10^3$ are shown in Figure 2. As we would expect, the HCGM version of the algorithm performs much better. With the CGM version it is faster to run the application with 7 fast processors than it is to run it with 7 fast processors and 4 slow processors, while with the HCGM version the performance improves each time a processor is added to the cluster.

[2]One processor was needed for other purposes, and hardware problems prevented the use of the remaining processor.

8 CONCLUSIONS

The HCGM model of parallel computing has been introduced. This is the first CGP model which takes into account the effects of differing processor speeds. The HCGM model is simple enough that the expressions derived when analyzing algorithms under the model are immediately meaningful, yet the model is powerful enough that it accurately reflects current hardware.

Even though the HCGM model is new, Section 6 shows that there are many algorithms already available for it. The empirical results of Section 7 show that algorithms under this model perform as well as CGM algorithms on homogeneous networks and have the advantage of also working on heterogeneous networks. It is worth noting that the modifications introduced to the CGM model to produce the HCGM model could also be incorporated into the BSP or LogP models.

The high level communication patterns of Section 4 are also of independent interest, as they form the basis of a high-level library for implementing coarse grained parallel algorithms. This library should help speed up the implementation and testing of coarse grained parallel algorithms as well as applications which use these algorithms. Future work in this area includes continuing work on the PLEDA library, including keeping the list of communication patterns up to date as new CGP algorithms are developped which use new communication patterns.

ACKNOWLEDGEMENTS

The author would like to thank Silvia Götz, Anil Maheshwari, Ben Juurlink and Jörg Sack for several helpful discussions, and for having read and commented on earlier versions of this paper.

References

[1] D. Bader, D. Hellman, and J. JáJá. Parallel algorithms for personalized communication and sort-

ing with an experimental study. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, 1996.

[2] A. Bäumker, W. Dittrich, F. Meyer auf def Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of Euro-Par '96*, pages 27–29, 1996.

[3] A. Beguelin, J. Dongarra, A. Geist, B. Manchek, and V. Sunderam. Solving computational grand challenges using a network of heterogeneous supercomputers. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 596–601, 1991.

[4] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. In *Proceedings of 1st International Symposium on High Performance Distributed Computing*, pages 152–160, 1992.

[5] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Proceedings of 4th International Symposium on High Performance Distributed Computing*, 1995.

[6] P. E. Crandall and M. J. Quinn. A decomposition advisory system for heterogeneous data-parallel processing. In *Proceedings of 3rd International Symposium on High Performance Distributed Computing*, pages 114–121, 1994.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Snatos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM Symposium on Principles and Practices of Parallel Programming*, pages 1–12, 1993.

[8] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Kokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, 1995.

[9] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[10] A. Ferreira and S. Ubéda. Computing the medial axis transform with 8 scan operations. In *IEEE International Conference on Image Processing*, 1995.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[12] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, 1996.

[13] A. V. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. In *3rd Scandinavian Workshop on Algorithm Theory*, pages 1–18, 1992.

[14] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[15] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1995.

[16] P. Morin. *The PLEDA User's Guide*. Carleton University, 1.0 edition, 1997.

[17] S. Näher. The LEDA manual. Technical Report MPI-I-93-109, Max-Planck Institut für Informatik, 1993.

[18] N. Nedeljković and M. J. Quinn. Data parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience*, 5(4):257–268, June 1993.

[19] M. V. Nibhanupudi, C. D. Norton, and B. K. Szymanski. Plasma simulation on networks of workstations using the bulk-synchronous parallel model. In *International Conference on Parallel and Distributed Techniques and Applications*, 1995.

[20] S. Orlando and R. Perego. A template for non-uniform parallel loops based on dynamic scheduling and prefetching techniques. In *Proceedings of the 10th ACM International Conference on Supercomputing*, 1996.

[21] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 3rd edition, 1997.

[22] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[23] M. H. Willebeck-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[24] Y. Yan, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[25] X. Zhang and Y. Yan. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 25–34, 1995.