Coarse-Grained Parallel Computing on Heterogeneous Systems*

Pat Morin

School of Computer Science
Carleton University
Ottawa, ON
Canada, K1S 5B6
morin@scs.carleton.ca

*This work was partially funded by a scholarship from the Natural Sciences and Engineering Research Council of Canada.

Heterogeneous Coarse-Grained Parallel Computing

Pat Morin

School of Computer Science
Carleton University
Ottawa, ON
Canada, K1S 5B6
morin@scs.carleton.ca

Abstract

We consider the problem of finding efficient parallel algorithms for *heterogeneous* parallel computers, i.e., parallel computers in which different processors have different computational potential. To this end, we define a formal computational model for heterogeneous systems and develop algorithms for commonly used communication operations. The result is that many existing parallel algorithms which use these communication operations can be adapted to our model with little or no modifications. Experimental results are give which show that our algorithms are of considerable practical relevance.

Keywords: Parallel algorithms, heterogeneous systems, bulk synchronous parallel, coarse grained multicomputer.

1 Introduction

In recent years, parallel computing has been increasing in popularity. Individuals with limited budgets can now build workstation clusters from off-the-shelf processing components and interconnection networks [10, 28, 31]. High speed networks are being used to interconnect traditional supercomputers in order to direct large amounts of computing power at Grand Challenge problems [8]. Even traditional supercomputers usually consist of a very fast workstation host connected to a number of slower in-the-box processors.

The three situations above, which cover nearly all modern parallel computing systems, are all potential examples of *heterogeneous systems*, i.e., systems in which different processors have different computational potential. In the case of workstation clusters, the processing components may be different because the system was grown incrementally and newly added processors are more modern than the originals. The same may be true in the case of supercomputer clusters, or the supercomputers may have even come from different manufacturers. Finally, even in the case of traditional supercomputers, it may be beneficial 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.

Figure 1 shows an example of a sorting algorithm in which the overall performance of a system decreases with the addition of slow processors. The first seven processors are fast processors, while all other processors are slow processors. Important to note is the decrease in performance when the first slow processor is added to the 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, either by having a master processor assign all subproblems, or by having processors request subproblems from other processors when 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

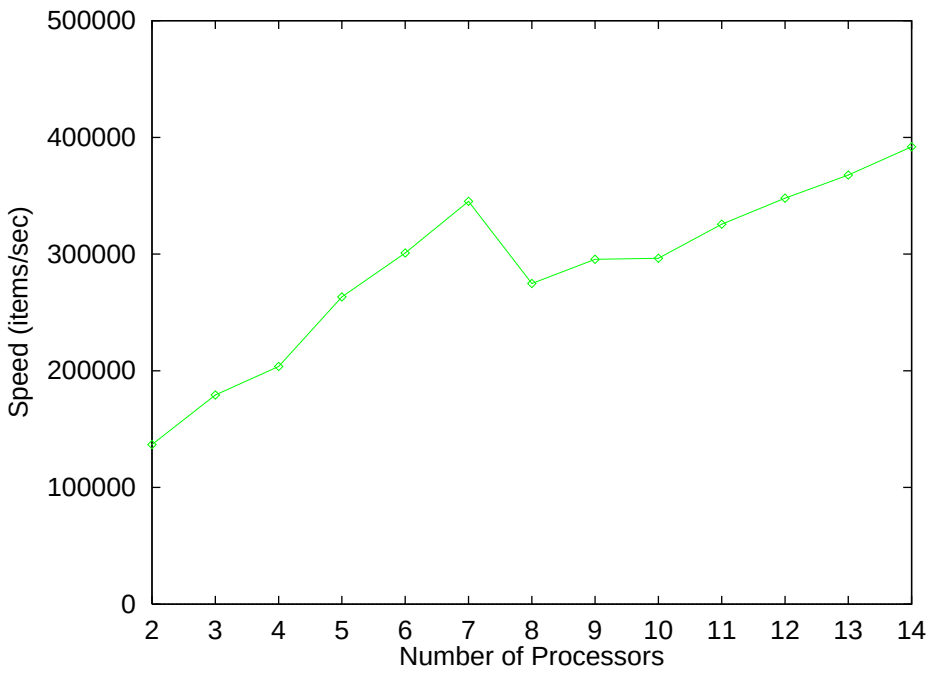

Figure 1: Performance of sorting algorithm with slow processors

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.

In this paper, we take a third approach. Namely that of modifying 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 include algorithms for the *bulk synchronous parallel* (BSP) [32], *coarse grained multicomputer* (CGM) [15], and *LogP* [14] 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 standard 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 4 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 4 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 part of the paper are the following:

1. The definition of a parallel computation model called the *heterogeneous coarse grained multicomputer* (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 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.

1.1 Comparisons with Related Work

In order to differentiate this work from other research on heterogeneous parallel computing, we compare it with previous work in this area.

The topic of data partitioning in heterogeneous systems with simple fixed communication patterns is addressed in [13, 27], 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 [11]. The results in this paper go beyond these in that the problems addressed have much less structure than simple stenciling operations on 2D grids or uniform parallel loops whose communication patterns can be analyzed at compile time. In Section 4 algorithms are presented for sorting, median finding, and a number of computational geometry problems.

Methods for dynamic load balancing such as those described in [24, 29, 34] can also be applied to heterogeneous systems. All these methods fall into the category of overpatitioning strategies. The advantages of our strategy over such overpartitioning strategies have been described above. These are the minimization of the effects latency and simplicity of the algorithm parameters.

In [36] a mathematical model of a network of workstations is described. In [35], 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 [35], a 12 processor configuration actually performs worse than a 2 processor configuration (although the model correctly predicted this).

The main difference between the model in [36, 35] 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.

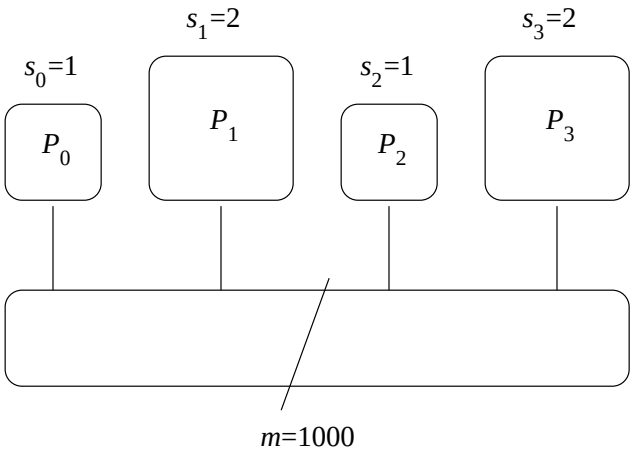

Figure 2: An example of an HCGM(m, p, s) with $m = 1000$, $p = 4$, $s = 6$, $P^{max} = P_1$, $s^{max} = 2$, and $P^{min} = P_0$, and $s^{min} = 1$.

2 A Heterogeneous Computing Model

This section defines a generalization of the CGM model of Dehne *et al.* [15] which takes the presence of heterogeneous processors into account. This model, called the heterogeneous coarse grained multicomputer (HCGM) model maintains the simplicity of the CGM model while providing a means of modelling the effects of heterogeneous processors such as the anomaly described in Section 1.

2.1 The Heterogeneous Coarse Grained Multicomputer

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. Each processor knows the values of $s_0, \ldots, s_{p-1}$ as well as the value of s.

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, we define $P^{max} = P_{\min\{i:s_i=s^{max}\}}$ and $P^{min} = P_{\min\{i:s_i=s^{min}\}}$. That is, P^{max} is a representative fastest processor, and P^{min} is a representative slowest processor. An example of an HCGM(m, p, s) is shown in Figure 2.

Each processor, P_i, in an HCGM(m, p, s) has $\Theta(\max\{\frac{m}{p}, \frac{s_i}{s}m\})$ local memory. 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 b words of information, then P_i incurs a penalty in computation time of $\frac{b}{s_i}$. 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 $\max\{\frac{h}{s_i} : 0 \leq i \leq p-1\} = \frac{h}{s^{min}}$.

Because the memory and communication speeds of the processors are proportional to the processor speeds, HCGM algorithms can take advantage of faster processors by having them process and communicate more data. If, as in the CGM(m, p) model, each processor has only $O(\frac{m}{p})$ memory, it may not be possible to improve the performance of algorithms and avoid the anomaly described in Section 1. Such is the case when the problem size, n, is equal to the size of the total memory, m.

We assume that the input to an 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 $s_0 = s_1 = \cdots = s_{p-1} = 1$.

Like a CGM algorithm, the performance of an HCGM algorithm is measured in terms of computation time and the number of supersteps. Both of these quantities can be functions of n, p, s, and $s_0, \ldots, s_{p-1}$. However, rather than measuring the amount of local computation, w, it is the local computation *time*, $\frac{w}{s_i}$, that is measured. 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. If at all possible, this speedup should be independent of the values of $s_0, \ldots, s_{p-1}$.

One possible way of obtaining 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 $\gcd(s_0, \ldots, s_{p-1})$ is the greatest common divisor of $s_0, \ldots, s_{p-1}$. There are at least three problems with this approach.

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 creating virtual processors increases the number of supersteps.

3. Many coarse grained parallel algorithms rely on some version of the coarse grained assumption $p \ll n$, and introducing virtual processors may violate this assumption.

2.2 A Simple Example: Prefix Sum

We consider the problem of computing the prefix sum of n elements using the following algorithm.

HCGM-PREFIX-SUM()

1. Each processor locally computes the prefix sum of its $\frac{s_i}{s}n$ input elements.

2. Each processor, P_i, sends the total sum of its input elements to P^{max} in a single communication superstep.

3. P^{max} computes the prefix sum of the p elements received in Step 2.

4. For $1 \leq i \leq p-1$, P^{max} sends the $(i-1)$st element computed in Step 3 to P_i in a single communication superstep.

5. Each processor computes its final portion of the prefix sum by adding the value received in Step 4 to each of the values computed in Step 1.

This algorithm provides our first result for the HCGM model.

Theorem 1. *The prefix sum of n elements can be computed on an HCGM(n, p, s), using $O(\frac{n}{s})$ computation time and $O(1)$ supersteps, provided that $\frac{n}{p} \geq p$.*

Proof. In Step 1 and Step 5, each processor P_i does $O(\frac{s_i}{s}n)$ work and this work can be done in $O(\frac{n}{s})$ time. Steps 2 and 4 each processor uses $O(1)$ computation time except P^{max} which uses $O(\frac{p}{s^{max}})$ computation time. Step 3 takes $O(\frac{p}{s^{max}})$ computation time. Since $p \leq \frac{n}{p}$, and by pigeonhole principle $s^{max} \geq \frac{s}{p}$, all steps can be performed in $O(\frac{n}{s})$ time. □

In [16], Ferreira and Ubéda describe an algorithm for computing the medial axis transform of a digital image using 8 prefix sum operations and $O(\frac{n}{p})$ local computation on a CGM(n,p). By using the HCGM(n,p,s) version of the prefix sum algorithm described above, this algorithm is readily adapted to an HCGM(n,p,s), yielding the following result.

Corollary 1. *The medial axis transform of a $\sqrt{n} \times \sqrt{n}$ image can be computed using $O(\frac{n}{s})$ computation time and $O(1)$ supersteps on a* HCGM(n,p,s) *with $p \leq \frac{n}{p}$.*

Corollary 1 is the first example of a technique which will be used repeatedly in this part of the paper. Namely, to obtain HCGM algorithms from existing CGM algorithms, one need only find HCGM algorithms for the primitive communication operations performed by the CGM algorithm. When these operations are replaced in the CGM algorithm by their HCGM counterparts, the resulting algorithm is an efficient HCGM algorithm.

3 Communication Patterns

This section discusses common communication patterns used in coarse grained parallel algorithms and gives their implementations both in the CGM and HCGM models. The motivation for this is that by implementing HCGM versions of these patterns, we obtain a number of HCGM algorithms directly from CGM algorithms which use these patterns.

This work also has applications outside of heterogeneous parallel computing. As is well known in the field of software engineering, the study of software patterns is a field in itself (see e.g., [17]). By identifying common patterns used in coarse grained parallel algorithms we provide a good starting point for the development of libraries and frameworks supporting the implementation of such algorithms.

3.1 Random-Sample

The technique of random sampling is one of the most useful tools used in the design of randomized parallel algorithms. In random sampling, a random subset of size $O(r)$ is chosen from the n input values.

In the context of coarse grained parallel computing, random sampling involves choosing $O(r)$ samples from the input and routing them to a designated processor, usually P_0. This processor then typically performs some computation on these elements and broadcasts the

results of this computation to all processors. The algorithm for the Random-Sample pattern proceeds as follows:

CGM-Random-Sample(r)

1. Each processor, P_i, tosses a biased coin with success probability $\frac{r}{n}$ for each of its input elements.

2. The elements for which the coin toss was successful are routed to P_0.

Using Chernoff bounds (see Appendix A), it is easily shown (see, e.g., [2]) that the number of elements which arrive at P_0 is $\tilde{O}(r)$, for $r \geq \log n$.[1] Thus, if $r \in O(\frac{n}{p})$, the computation time used by the Random-Sample pattern is $\tilde{O}(\frac{n}{p})$ and the number of supersteps is $\tilde{O}(1)$ on a CGM(n, p).

In order to modify the random sample pattern for the HCGM(n, p, s) model we need only change the designated processor to which the sample are routed. Rather than routing the samples to P_0, we route the samples to P^{max}. This results in the following implementation:

HCGM-Random-Sample(r)

1. Each processor, P_i, tosses a biased coin with success probability $\frac{r}{n}$ for each of its input elements.

2. The elements for which the coin toss was successful are routed to P^{max}.

Theorem 2. *The* HCGM-Random-Sample(r) *algorithm uses* $\tilde{O}(\frac{n}{s})$ *computation time and* $\tilde{O}(1)$ *supersteps on an* HCGM(n, p, s), *provided that* $\frac{s^{max}}{s} n \geq r \geq 3 \ln n$.

Proof. In Step 1, each processor, P_i must perform $\frac{s_i}{s} n$ coin tosses and can do these in $O(\frac{n}{s})$ time. In Step 2, each processor P_i sends at most $\frac{s_i}{s} n$ elements and can do this in $O(\frac{n}{s})$ time. Let r' be the number of elements received by P^{max} in Step 2. Then r' is a random variable following the binomial distribution $b(n, \frac{r}{n})$. Applying Theorem 14, Equation 1 we get that

$$\begin{aligned} \Pr[r' \geq cr] &= \left(\frac{1}{e}\right)^{(c-1)^2 r/3} \\ &\leq \frac{1}{n^{(c-1)^2}}, \text{ for } r \geq 3 \ln n. \end{aligned}$$

[1] See Appendix A for a definition of the $\tilde{O}$ notation.

Therefore, during Step 2 P^{max} receives $\tilde{O}(r)$ elements and can do this in $\tilde{O}(\frac{r}{s^{max}}) \subseteq \tilde{O}(\frac{n}{s})$ time. □

3.2 Random-Assign

Random assignment is a tool used in a number of CGP algorithms to achieve load balancing. The idea behind random assignment is to assign elements of the input to processors in a random fashion. In this way, if the work performed on each element is variable, then one expects that all processors will be assigned roughly the same amount of work. This idea is realized in the following procedure.

CGM-Random-Assign()

1. Each processor, P_i, randomly assigns each of its elements to one of p buckets, $b_{i,0}, \ldots, b_{i,p-1}$ with equal probability.

2. Each processor, P_i, routes the contents of each bucket, $b_{i,j}$, to P_j.

Once again, using Chernoff bounds it is not difficult to show that the number of elements routed to any processor is $\tilde{O}(\frac{n}{p})$ (see, e.g., [2]). Thus, the algorithm uses $\tilde{O}(\frac{n}{p})$ computation time and $\tilde{O}(1)$ communication rounds.

The HCGM version of the Random-Assign pattern is similar to the CGM version except that each processor, P_i, should receive $O(\frac{s_i}{s}n)$ elements in Step 2. To achieve this, we change the probability to which elements are assigned to buckets in Step 1. The modified algorithm works as follows.

HCGM-RANDOM-ASSIGN()

1. Each processor, P_i, randomly assigns each of its elements to one of p buckets, $b_{i,0}, \ldots, b_{i,p-1}$. P_i assigns an element to bucket $b_{i,j}$ with probability $\frac{s_j}{s}$.

2. Each processor, P_i, routes the contents of each bucket, $b_{i,j}$, to P_j.

Theorem 3. *The* HCGM-RANDOM-ASSIGN() *algorithm uses* $\tilde{O}(\frac{n}{s}\log p)$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s), *with* $p \leq n$ *and* $\frac{s^{min}}{s} n \geq 3 \ln n$.

Proof. Step 1 can be accomplished by performing a binary search on the p buckets for each of the input elements, and can therefore be done in $O(\frac{n}{s}\log p)$ time.

Next we show that the number of elements received by P_i in Step 2 is $\tilde{O}(\frac{s_i}{s}n)$. Let n_i be the number of elements received by P_i in Step 2. Then clearly n_i is a random variable which follows the binomial distribution $b(\frac{s_i}{s}, n)$. Applying Theorem 14, Equation 1, we get

$$\begin{aligned} \Pr\left[n_i \geq c\frac{s_i}{s}n\right] &\leq \left(\frac{1}{e}\right)^{(c-1)^2 \frac{s_i}{s} n/3} \\ &\leq \frac{1}{n^{(c-1)^2}}, \text{ for } \tfrac{s_i}{s}n \geq 3\ln n. \end{aligned}$$

Therefore, the probability that *any* processor, P_i, receives more than $c\frac{s_i}{s}n$ elements is bounded by

$$\begin{aligned} \Pr\left[\exists i \text{ s.t. } n_i \geq c\frac{s_i}{s}n\right] &\leq \frac{p}{n^{(c-1)^2}} \\ &\leq \frac{1}{n^{(c-1)^2-1}}, \text{ for } p \leq n. \end{aligned}$$

Therefore Step 2 can be done using $\tilde{O}(\frac{n}{s})$ computation time and the entire algorithm uses $\tilde{O}(\frac{n}{s}\log p)$ computation time. □

3.3 Linear-Partition

Let S be a be a set of keys and $\leq$ be a relation that defines a total order on S. A linear partition of S is a partitioning of S into p disjoint subsets $S_0, \ldots, S_{p-1}$ such that $x \leq y$ for all $x \in S_i$, $y \in S_j$ and $i < j$. Linear partitioning is one of the most commonly used communication patterns in parallel computing. This is due simply to the fact that sorting is a special case of linear partitioning in which the keys are sorted locally after being partitioned.

Here we describe a randomized linear partitioning algorithm based on the sample sort algorithm described in [21]. We assume that the n keys are all distinct since if they are not, they can be made so by, e.g., concatenating their value with their processor number and memory location. The algorithm proceeds as follows.

CGM-Linear-Partition()

1. All processors take a random sample of size $O(r)$ using the CGM-Random-Sample algorithm and route the sample keys to P_0.

2. P_0 sorts the sample keys. Denote these keys by $sample_0, \ldots, sample_{pr-1}$ where $sample_i$ is the sample with rank i in the sorted order.

3. P_0 defines $p+1$ splitters, $splitter_0, \ldots, splitter_p$, where

$$splitter_i = \begin{cases} -\infty & \text{if } i = 0 \\ sample_{ir/p} & \text{if } 0 < i < p \\ \infty & \text{if } i = p \end{cases}$$

4. P_0 broadcasts $splitter_0, \ldots, splitter_p$ to all processors.

5. Each processor, P_i, places each of its keys into one of p buckets, where a key x is placed in bucket b_{ij} if and only if $splitter_j \leq x < splitter_{j+1}$.

6. Each processor, P_i, routes the contents of bucket b_{ij} to P_j for all i, j.

That this algorithm produces a valid linear partition is clear since (1) all keys are assigned to exactly one bucket, and hence one processor, (2) all keys in bucket i are strictly less than all keys in bucket j for all $i < j$. Less clear is the running time of the algorithm, since it is conceivable that some processor receives significantly more than $O(\frac{n}{p})$ elements in Step 6. Gerbessiotis and Valiant [21] showed that for properly chosen values of s, n, and p such a situation does not occur.

When adapting this algorithm to the HCGM model, we change the way in which the splitters are chosen. In order to balance the work according to $s_0, \ldots, s_{p-1}$ it is necessary that $O(\frac{s_i}{s}n)$ *input keys* fall between $splitter_i$ and $splitter_{i+1}$. In order to achieve this, we choose the splitters so that $O(\frac{s_i}{s}r)$ *sample keys* fall between $splitter_i$ and $splitter_{i+1}$. This leads to the following algorithm.

HCGM-Linear-Partition()

1. All processors take a random sample of size $O(r)$, r to be defined later, using the HCGM-Random-Sample algorithm and route the sample keys to P^{max}.

2. P^{max} sorts the sample keys. Denote these keys by $sample_0, \ldots, sample_{r-1}$ where $sample_i$ is the sample with rank i in the sorted order.

3. P^{max} defines $p+1$ splitters, $splitter_0, \ldots, splitter_p$, where

$$splitter_i = \begin{cases} -\infty & \text{if } i = 0 \\ sample_{\left\lceil \left(\sum_{j=0}^{i} \frac{s_j}{s} \right) r \right\rceil} & \text{if } 0 < i < p \\ \infty & \text{if } i = p \end{cases}$$

4. P^{max} broadcasts $splitter_0, \ldots, splitter_p$ to all processors.

5. Each processor, P_i, places each of its keys into one of p buckets, where a key x is placed in bucket b_{ij} if and only if $splitter_j \leq x < splitter_{j+1}$.

6. Each processor, P_i, routes the contents of bucket b_{ij} to P_j for all i, j.

Theorem 4. *The* HCGM-Linear-Partition() *algorithm uses* $\tilde{O}(\frac{n}{s} \log p)$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s), *with* $r = \frac{s^{max}}{s} n$, $\frac{s^{min}}{s} r \geq 2 \ln n$, *and* $p \leq n$.

Proof. By Theorem 2, Step 1 of the algorithm uses $\tilde{O}(\frac{n}{s})$ computation time. Step 2 of the algorithm uses $\tilde{O}(\frac{r}{s^{max}} \log r) \subseteq \tilde{O}(\frac{n}{s} \log n)$ computation time. Steps 3 and 4 use $\tilde{O}(\frac{n}{s})$ computation time. Step 5 uses $O(\frac{n}{s} \log p)$ computation time.

Next we consider the possibility that some processor receives too many keys in Step 6. Let n_i be the number of keys received by P_i in Step 6. Let r_i be the number of samples chosen from the $c\frac{s_i}{s} n$ keys following $splitter_i$ in the overall sorted order. Now note that $n_i > c\frac{s_i}{s} n$ only if $r_i < \frac{s_i}{s} r$. That is, the number of keys between $splitter_i$ and $splitter_{i+1}$ can only exceed $c\frac{s_i}{s} n$ if less than $\frac{s_i}{s} r$ samples are chosen from these keys.

Since r_i is a random variable that follows the binomial distribution $b(c\frac{s_i}{s} n, \frac{r}{n})$, Theorem 14, Equation 2 can be applied to get

$$\Pr\left[q_i \leq \frac{s_i}{s} r \right] \leq \left(\frac{1}{e} \right)^{(1-\frac{1}{c})^2 (\frac{r}{n})(c\frac{s_i}{s} n)/2}$$

$$= \left(\frac{1}{e}\right)^{(1-\frac{1}{c})^2 c(\frac{s_i}{s}r)/2}$$
$$= \frac{1}{n^{(1-\frac{1}{c})^2 c}}, \text{ for } \frac{s_i}{s}r \geq 2\ln n.$$

Therefore, the probability that *any* processor, P_i, receives more than $c\frac{s_i}{s}n$ elements is bounded by

$$\Pr\left[\exists i \text{ s.t. } q_i \leq \frac{s_i}{s}r\right] \leq \frac{p}{n^{(1-\frac{1}{c})^2 c}} \leq \frac{1}{n^{(1-\frac{1}{c})^2 c-1}}, \text{ for } p \leq n,$$

and Step 6 can be done using $\tilde{O}(\frac{n}{s})$ computation time. □

Since sorting is a special case of Linear-Partition in which elements are sorted locally after partitioning, we obtain the following corollary.

Corollary 2. *Sorting n keys can be done using $\tilde{O}(\frac{n}{s}\log n)$ computation time and $O(1)$ supersteps on an* HCGM(n,p,s)*, with $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $p \leq n$.*

3.4 PRAM-Simulation

PRAM simulations on the BSP model were introduced by Valiant [32], and by Gerbessiotis and Valiant [21] as a means of obtaining BSP algorithms from PRAM algorithms , and it was shown that if the BSP parameter g is close to unity, the resulting BSP algorithms would be optimal. Unfortunately, this condition is not usually met in practice and the performance of the resulting algorithms is often disappointing. More recently, PRAM simulations have been revived in the form of *clipping* [9]. Clipping involves simulating a PRAM algorithm for $O(\log p)$ rounds, stopping the algorithm (clipping it), and completing the computation with a specialized CGP algorithm.

In doing PRAM simulations on a CGM, each processor simulates $\frac{n}{p}$ EREW-PRAM processors and stores $\frac{n}{p}$ data elements. Each round of the simulation consists of a read phase and a write phase. The following algorithm performs 1 step of an EREW-PRAM simulation on a CGM(n,p):

CGM-PRAM-Simulation()

1. Each processor, P_i, formulates $O(\frac{n}{p})$ read requests and sends each request to the processor holding the element to be read.

2. Each processor, P_i, responds to the $O(\frac{n}{p})$ read requests received in Step 1.

3. Each processor, P_i, formulates $O(\frac{n}{p})$ write requests and sends each request to the processor holding the element to be written.

4. Each processor, P_i, responds to the $O(\frac{n}{p})$ write requests received in Step 3.

Theorem 5. *The* CGM-PRAM-SIMULATION() *algorithm uses* $O(\frac{n}{p})$ *computation time and* $O(1)$ *supersteps on a* CGM(n, p).

Proof. Determining which processor, P_i, services a read or write request for memory location j can be done in constant time using the formula $i = \lfloor j/(n/p) \rfloor$. Therefore Step 1 and Step 3 take $O(\frac{n}{p})$ time. Since an EREW-PRAM is being simulated, no processor receives more than $O(\frac{n}{p})$ requests in Step 2 and Step 4. Therefore these steps can be done in $O(\frac{n}{p})$ time, yielding the stated time bound. □

The HCGM(n, p, s) version of the PRAM-SIMULATION procedure is nearly identical to the CGM version, though it only holds for a restricted range of parameters. The algorithm proceeds as follows.

HCGM-PRAM-SIMULATION()

1. Each processor, P_i, formulates $O(\frac{s_i}{s}n)$ read requests and sends each request to the processor holding the element to be read.

2. Each processor, P_i, responds to the $O(\frac{s_i}{s}n)$ read requests received in Step 1.

3. Each processor, P_i, formulates $O(\frac{s_i}{s}n)$ write requests and sends each request to the processor holding the element to be written.

4. Each processor, P_i, responds to the $O(\frac{s_i}{s}n)$ write requests received in Step 3.

The extra restriction on the range of parameters comes from the fact that the processor, P_i, which corresponds to PRAM memory location j can not be determined using a simple

formula as it is in the CGM procedure. A simple workaround to this is to use binary search to find the correct processor but this leads to an $O(\log p)$ slowdown. A more efficient method can be obtained using integer sorting.

Theorem 6. *The* HCGM-PRAM-SIMULATION() *algorithm uses* $O(\frac{n}{s})$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s)*, provided that* $\frac{s}{s^{min}} \leq c_1 p^{c_2}$ *for some constants* $c_1, c_2 > 0$*, and* $\frac{s^{min}}{s} n \geq p$*.*

Proof. Clearly the proof of Theorem 5 extends to this theorem with the exception of finding the processors which service requests. Thus we need only show how this is done. By first sorting the requests locally at each processor, the relevant processors can be determined by (sequentially) scanning the sorted lists using $O(\frac{s_i}{s} n + p) = O(\frac{s_i}{s} n)$ work at each processor P_i, and can therefore be done using $O(\frac{n}{s})$ computation time. Thus we need only consider how to sort the requests.

Fact 1 (Radix Sort [23]). *It is possible to sort (sequentially)* k *integers in the range* $[0, l-1]$ *using* $O(\frac{\log l}{\log k} k)$ *computation and* $O(k)$ *memory.*

Each processor P_i must sort $\frac{s_i}{s} n$ elements in the range $[0, n-1]$. By substituting $k = \frac{s_i}{s} n$ and $l = n$ in Fact 1, we see that the sorting can be done using

$$\begin{aligned}
W_{sort} &\in O\left(\left(\frac{\log n}{\log \frac{s_i}{s} n}\right)\left(\frac{s_i}{s} n\right)\right) \\
&= O\left(\left(\frac{\log \frac{s_i}{s} n + \log \frac{s}{s_i}}{\log \frac{s_i}{s} n}\right)\left(\frac{s_i}{s} n\right)\right) \\
&= O\left(\left(1 + \frac{\log \frac{s}{s_i}}{\log \frac{s_i}{s} n}\right)\left(\frac{s_i}{s} n\right)\right) \\
&\subseteq O\left(\left(1 + \frac{\log c_1 + c_2 \log p}{\frac{s_i}{s} n}\right)\left(\frac{s_i}{s} n\right)\right), \text{ for } \tfrac{s}{s_i} \leq c_1 p^{c_2} \\
&\subseteq O\left(\left(1 + \frac{\log c_1 + c_2 \log p}{\log p}\right)\left(\frac{s_i}{s} n\right)\right), \text{ for } \tfrac{s_i}{s} n \geq p \\
&= O\left(\frac{s_i}{s} n\right)
\end{aligned}$$

work at each processor, P_i, and $O(\frac{n}{s})$ time. □

For an algorithm which uses PRAM simulation to be useful in practice, the simulation must be done as efficiently as possible. If we have the extra restriction that $s / \gcd(s_0, \ldots, s_{p-1}) \in$

$O(\frac{s^{min}}{s}n)$ we can simulate the CGM-PRAM-Simulation algorithm by having each processor, P_i, simulate $s_i/\gcd(s_0, \ldots, s_{p-1})$ CGM processors thereby reducing the constants in the running time of the HCGM-PRAM-Simulation algorithm.

If we have the weaker restriction that $\frac{s}{s^{min}} \in O(\frac{s^{min}}{s}n)$ then the same game can be played by having each processor, P_i, simulate $O(\frac{s_i}{s^{min}})$ CGM processors. Although this leads to an algorithm with running time $O(\frac{n}{s})$, the big-Oh notation hides the fact that the processors are not doing work which is exactly proportional to their speeds. This may or may not be acceptable in practice.

In many cases, PRAM algorithms operate on pointer-based data structures such as lists or graphs and the memory locations accessed by the PRAM processors are dictated by the values of pointers in the data structure. In such cases the data structure can be preprocessed so that the pointers are modified to contain a processor/address pair to allow addressing in constant time. This preprocessing can easily be done in time $O(\frac{n}{s}\log p)$, and will most likely yield the most efficient simulations in practice.

Using the more sophisticated techniques in [21, 32], it is possible to simulate other types of PRAMs on the HCGM model. In particular, extensions of the randomized EREW-PRAM and CRCW-PRAM simulations described by Gerbessiotis and Valiant [21] to the HCGM model are possible.

3.5 Circulate

Scientific computations often use a very regular communication pattern in which a set of items is "rotated" or "circulated" through the processors in rounds, so that after p rounds, every processor has seen every item. Examples include dense matrix multiplication, in which the rows of the matrix are rotated, and solutions to the so-called n-body problem, in which the n bodies in question are rotated.

The Circulate 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 $O(|A_i| \cdot |B_i| \cdot n^c)$, where

A_i (resp. B_i) is the sublist of A (resp. B) stored at P_i. This is captured by the following algorithm.

CGM-CIRCULATE(A, B)

1. Repeats Steps 2 and 3 p times.
2. Each processor P_i performs computation on A_i and B_i.
3. Each processor P_i send B_i to $P_{(i+1) \bmod p}$.

The number of supersteps used by the CGM-PARTITION algorithm is clearly $O(p)$. Initially, A and B are distributed evenly among the processors, and so the amount of computation done during each of the p computation supersteps is

$$O\left(\frac{|A|}{p} \cdot \frac{|B|}{p} \cdot n^c\right) = O\left(\frac{n^{c+2}}{p^2}\right)$$

and the overall computation time is $O(\frac{n^{c+2}}{p})$.

To implement an HCGM version of the Circulate pattern, we need only change the way in which A and B are distributed among the processors. This leads to the following algorithm:

HCGM-CIRCULATE(A, B)

1. Distribute A and B so that P_i stores $\frac{s_i}{s}|A|$ elements of A and $\frac{|B|}{p}$ elements of B.
2. Repeats Steps 2 and 3 p times.
3. Each processor P_i performs computation on A_i and B_i.
4. Each processor P_i send B_i to $P_{(i+1) \bmod p}$.

Theorem 7. *The* HCGM-CIRCULATE(A, B) *algorithm uses* $O(\frac{n^{c+2}}{s})$ *computation time and* $O(p)$ *supersteps on an* HCGM(n, p, s), *provided that* $\frac{n}{p} \geq p$, *and* $\lfloor \frac{s^{min}}{s} n \rfloor \geq 1$.

Proof. Redistributing A and B in Step 1 is a straightforward matter using the prefix sum algorithm of Theorem 1, and takes $O(\frac{n}{ps^{min}})$ computation time and $O(1)$ supersteps.

During each execution of Step 2, the work done by P_i is given by

$$O\left(\frac{s_i}{s}|A| \cdot \frac{|B|}{p} \cdot n^c\right) = O\left(\frac{s_i n^{c+2}}{ps}\right)$$

and this work can be done in time $O(\frac{n^{c+2}}{ps})$ time. Therefore, over the p rounds, the total computation time is $O(\frac{n^{c+2}}{s})$, and this dominates the overall computation time. □

4 HCGM Algorithms

This section provides a sampling of algorithms for the HCGM model, and describes some empirical results which show that these algorithms work well. These algorithm are arrived at by expressing existing CGM and BSP algorithms in terms of the communication patterns described in Section 3. We present algorithms for the following problems:

1. Parallel insertion and deletion operations on a priority queue illustrate the RANDOM-ASSIGN and LINEAR-PARTITION patterns.
2. Computing the lower envelope of non-intersecting line segments illustrates the LINEAR-PARTITION pattern.
3. List ranking illustrates the PRAM-SIMULATION pattern.
4. Matrix multiplication illustrates the CIRCULATE pattern.

Rather than give a superficial treatment of a large number of algorithms, we have chosen to examine a few algorithms in detail. A consequence of this approach is that we do not provide an exhaustive list of possible HCGM algorithms based on the communication patterns in Section 3. However, after presenting each algorithm, we make note of other BSP and CGM algorithms which could be converted to HCGM algorithms in a similar manner.

4.1 Priority Queue Operations

Priority queues are a fundamental data structure used in a large number of graph and optimization algorithms. A common example is the class of "Branch and Bound" algorithms used in combinatorial optimization.

In this section, we consider the problem of performing a batch of m priority queue operations on a priority queue, Q, of size n. Throughout the following discussion, we will assume that $n > m$. The two operations we wish to support are:

1. MULTIINSERT$(k_0, \ldots, k_{m-1}, Q)$. Insert the keys $k_0, \ldots, k_{m-1}$ into Q.

2. MULTIDELETE(m, Q). Delete the m smallest keys from Q.

The algorithms presented in this section are a generalization of the algorithms discovered by Bäumker *et. al.* [4] and by Gerbessiotis and Siniolakis [20]. In these schemes, each processor, P_i maintains a local priority queue Q_i which contains $\tilde{O}(\frac{n}{p})$ of the keys in the overall priority queue. When inserting keys into the priority queues, the keys are assigned to processors at random. The MultiInsert algorithm is given below:

MULTIINSERT$(k_0, \ldots, k_{m-1}, Q)$

1. All processors use the Random-Assign algorithm to randomly assign the keys to processors.

2. Each processor, P_i, inserts the keys received in Step 1 into Q_i.

Theorem 8. *The algorithm* MULTIINSERT$(k_0, \ldots, k_{m-1}, Q)$ *runs uses* $\tilde{O}(\frac{m}{s} \log n)$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s), *provided that* $\frac{s^{min}}{s} m \geq 3 \ln m$.

Proof. By Theorem 3, Step 1 takes $\tilde{O}(\frac{m}{s} \log p)$ computation time and $\tilde{O}(1)$ supersteps. Furthermore, each processor, P_i, receives $\tilde{O}(\frac{s_i}{s} m)$ keys. Therefore, Step 2 takes $O(\frac{m}{s} \log n)$ computation time and dominates the computation time. □

Assigning the keys to processors using the Random-Assign pattern not only ensures load balancing during insertion, but also ensures load balancing during deletion. This is because each local queue, Q_i, contains no more than $c\frac{s_i}{s} m$ of the m smallest keys, with high probability. Thus, the strategy used when deleting keys is to delete too many keys from Q_i and then reinsert those keys which should not have been deleted. This leads to the following algorithm.

MULTIDELETE(m, Q)

1. Each processor P_i removes the minimum $c\frac{s_i}{s} m$ keys from Q_i.

2. All processors globally sort the keys removed in Step 1.

3. The keys with rank less than m in the sorted order are deleted, while the keys with rank greater than m in the sorted order are reinserted using the MultiInsert algorithm.

Note that the MULTIDELETE(m, Q) algorithm is a Monte Carlo algorithm as well as a Las Vegas algorithm. It is possible that some Q_i contains more than $c\frac{s_i}{s}m$ of the m smallest keys, in which case the algorithm produces an incorrect result. However, as Theorem 9 shows, this is highly unlikely.

Theorem 9. *Let m_i be the number of the m smallest keys which are stored in Q_i. Then*

$$\Pr\left[m_i \geq c\frac{s_i}{s}m\right] \leq \frac{1}{m^{(c-1)^2}}$$

provided that $\frac{s_i}{s}m \geq 3\ln m$.

Proof. Note that any key is assigned to Q_i with probability $\frac{s_i}{s}$ and that this probability is independent of any other key being assigned to Q_i. Therefore the value of m_i follows the binomial distribution $b(m, \frac{s_i}{s})$. Applying Equation 1, we get that

$$\Pr\left[m_i \geq c\frac{s_i}{s}m\right] \leq \left(\frac{1}{e}\right)^{(c-1)^2\frac{s_i}{s}m/3} \leq \frac{1}{m^{(c-1)^2}}$$

□

Theorem 10. *The algorithm* MULTIDELETE(m, Q) *uses $\tilde{O}(\frac{m}{s}\log n)$ computation time and $O(1)$ supersteps on an* HCGM(n, p, s)*, provided that $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $p \leq n$.*

Proof. In Step 1, each processor P_i deletes $\frac{s_i}{s}m$ items from Q_i, and can do this using $O(\frac{m}{s}\log n)$ computation time. By Corollary 2, Step 2 can be done using $O(1)$ supersteps and $O(\frac{m}{s}\log m)$ computation time. By Theorem 8, Step 3 can be done using $O(1)$ supersteps and $O(\frac{m}{s}\log n)$ computation time. Thus, all steps can be completed in the stated resource bounds. □

A number of BSP and CGM algorithms exist which use the Random-Assign pattern. These include the randomized sorting algorithm of Bader *et. al.* [2], the tree multisearch algorithms of Bäumker *et. al.* [3, 5, 6, 7], and the DAG multisearch algorithms of Gerbessiotis and Siniolakis [18, 19].

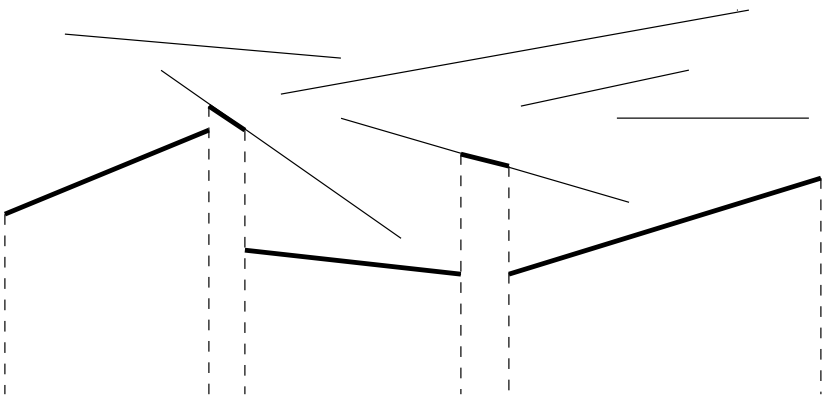

Figure 3: The lower envelope of a set of line segments. The portions of the line segments which form the lower envelope are shown in bold.

4.2 Lower Envelopes

Computing the lower envelope of non-intersecting line segments is a classic problem in computational geometry. Given a set of non-intersecting line segments in the plane, the lower envelope problem is to determine which portions of these segments are visible to a viewer standing at $(0, -\infty)$ (see Figure 3). A number of (more realistic) visibility problems can be reduced to the lower envelope problem through standard geometric transformations. The sequential complexity of the lower envelope problem is $O(n \log n)$.

The algorithm we describe relies on the following property of the lower envelope of line segments.

Observation 1 (x-Monotonicity). *Any lower envelope, L, is x-monotone, i.e., any vertical line intersects L at most once.*

Dehne, Fabri, and Rau-Chaplin [15] describe a CGM algorithm for the lower envelope problem which uses $O(1)$ supersteps and $O(\frac{n \log n}{p})$ computation time. The algorithm we describe is arrived at by simply replacing the global sort operation used in [15] with the Linear-Partition algorithm in Section 3.

The algorithm works by first computing p individual lower envelopes. Next, the plane is partitioned into p vertical *slabs*, where each slab intersects at most $O(\frac{n}{p})$ of the segments of the lower envelopes computed above. Finally, the lower envelope of the segments in each slab is computed, and the overall lower envelope is the union of the lower envelopes of all slabs.

LowerEnvelope(S)

1. Each processor P_i computes the lower envelope of it's locally stored segments, call this L_i.
2. All processors globally perform a linear partition of the L_i computed in Step 1, using the x-coordinate of the right endpoint of each segment as the key.
3. Each processor, P_i, determines the vertical line, l_i, through the rightmost segment received in Step 2, and broadcasts this line to all other processors.
4. Each processor, P_i, sends the segments $s \in L_i$ to P_j if and only if s intersects l_j.
5. Each processor, P_i, computes the lower envelope of the segments received in Steps 2 and 4.

Theorem 11. *The lower envelope of n non-intersecting line segments can be computed using $\tilde{O}(1)$ supersteps and $\tilde{O}(\frac{n}{s}\log n)$ computation time on an* HCGM(n,p,s) *provided that $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $\frac{s^{min}}{s}n \geq p$.*

Proof. The correctness of the algorithm follows from the correctness of the algorithm in [15]. At the beginning of Step 1 each processor, P_i, contains $O(\frac{s_i}{s}n)$ segments and can therefore compute the lower envelope of these segments in $O(\frac{n}{s}\log n)$ time. By Theorem 4, Step 2 can be done in $\tilde{O}(1)$ supersteps and $\tilde{O}(\frac{n}{s}\log p)$ computation time. Step 3 consists of routing an h-relation with $h = p$. As noted in [15], during Step 4, each processor sends and receives at most $p \leq \frac{s^{min}}{s}n$ segments. This is due to Observation 1, since each L_i intersects each l_j at most once. At the beginning of Step 5 each processor, P_i, contains $\tilde{O}(\frac{s_i}{s}n)$ segments and can therefore compute the lower envelope of these segments in $\tilde{O}(\frac{n}{s}\log n)$ time. □

The Linear-Partition algorithm is extremely useful in adapting CGM and BSP algorithms to the HCGM model. The only communication operation used by the geometric algorithms in [15] is global sorting. A first step in generalizing almost any CGM and BSP algorithms to the HCGM model is to replace all calls to global sort with calls to Linear-Partition. In some cases, the resulting algorithm is in fact more efficient than the original since global sort is often used when a linear partition will suffice. (This is the case with the LowerEnvelope algorithm if a sequential algorithm is used in Step 5 that does not require sorting.)

4.3 List Ranking

The list ranking problem takes as input a linked list and returns as output the distance of each list element to the last element of the list. This list ranking problem has been studied extensively for the PRAM model and optimal randomized and deterministic algorithms have been devised [30].

The simplest (near-optimal) solution to the list ranking problem uses a recursive doubling technique known as pointer jumping (see, e.g., [22]. In pointer jumping, each element is initially assigned a rank of 1, with the exception of the last list element which is assigned a rank of 0. Each element then gets assigned the pointer of its successor, and adds to its rank, the rank of its successor. After repeating this procedure $\log n$ times, each element has the last list element as its successor and is correctly ranked.

In [9], Cáceres *et. al.* describe a CGM algorithm for the list ranking problem. The algorithm begins by finding a p^2-*ruling set* of size $O(\frac{n}{p})$. This is a subset of the original list elements such that no two consecutive elements are further than distance p^2 apart. At the same time, the algorithm determines for each list element x, $nextrs(x)$ the first ruling set element which occurs after x in the list, and $dist(x)$ the distance between x and $nextrs(x)$. The ruling set elements are then broadcast to all processors and are ranked sequentially. Elements not in the ruling set are then ranked using the formula $rank(x) = rank(nextrs(x)) + dist(x)$. For the HCGM model we obtain the following algorithm.

HCGM-List-Rank()

1. All processors compute a p^2 ruling set of size $O(\frac{n}{p})$ using the EREW-PRAM simulation procedure described in [9].

2. All processors gather the ruling set elements into P^{max}.

3. P^{max} ranks the ruling set elements.

4. P^{max} sends each (now ranked) ruling set element back to the processor from which it was received.

5. All processors simulate pointer jumping to determine, for each list element x, the values $nextrs(x)$, $dist(x)$, and $rank(nextrs(x))$.

6. Each processor, P_i, ranks the elements it contains which are not in the ruling set using the formula $rank(x) = rank(nextrs(x)) + dist(x)$.

Theorem 12. *The list ranking problem can be solved on an* HCGM(n, p, s) *using* $O(\frac{n}{s} \log p)$ *computation time and* $O(\log p)$ *supersteps.*

Proof. In a preprocessing step, each processor, P_i, can preprocess its subarray in $O(\frac{n}{s} \log p)$ time, so that rather than containing indices, the subarray contains processor/index pairs. In this way, the PRAM simulation in Step 1 and the pointer jumping in Step 5 can be done in $O(\frac{n}{s} \log p)$ time. Steps 2–4 can clearly be done using $O(1)$ supersteps and $O(\frac{n}{s})$ computation time. □

List ranking is used as a subroutine in a number of parallel tree and graph algorithms. In [9], Cáceres *et. al.* describe a number of CGM graph algorithms based on existing PRAM algorithms. Using the PRAM simulation technique of Section 3, most of these algorithms can be adapted to the HCGM model.

4.4 Matrix Multiplication

Matrix multiplication is perhaps one of the most common operations used in large-scale scientific computing. Given two $n \times n$ matrices A and B, we define the matrix $C = A \times B$ as

$$C_{i,j} = \sum_{k=0}^{n-1} A_{k,i} B_{j,k}.$$

In this section, we show how to implement matrix multiplication using the CIRCULATE pattern. We assume that the matrix A is partitioned among the processors so that each processor, P_i holds $\frac{s_i}{s} n$ rows of A and $\frac{n}{p}$ columns of B. At the completion of the computation, P_i will hold $\frac{s_i}{s} n$ rows of C (see Figure 4). We denote the parts of A, B, and C held by P_i as A_i, B_i, and C_i respectively.

The matrix multiplication algorithm consists of circulating the columns of B among the processors. Note that when P_i receives column j of B, it can compute column j of C_i. Thus, once P_i has seen all columns of B, it will have computed all of C_i. Although, by now the operation of the algorithm should be obvious, we include it here for the sake of completeness.

HCGM-MATRIX-MULTIPLY(A, B)

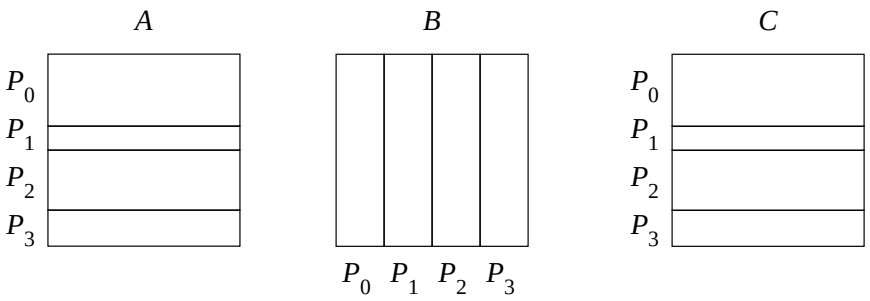

Figure 4: Partitioning matrices A, B, and C in a 4 processor system.

1. Apply the HCGM-Circulate algorithm, where the A set consists of the rows of A, and the B set consists of the columns of B. When processor P_i receives some columns of B, it computes the corresponding columns of C_i.

Theorem 13. *The* HCGM-Matrix-Multiply(A, B) *algorithm computes the matrix* C *using* $O(\frac{n^3}{s})$ *computation time and* $O(p)$ *supersteps on an* hcgm(n^2, p, s)*, provided that* $\frac{n}{p} \geq p$*, and* $\lfloor \frac{s^{min}}{s} n \rfloor \geq 1$*.*

Proof. The correctness of the algorithm follows from the fact that the Circulate pattern ensures that every processor P_i, sees every column of B exactly once, thereby enabling it to correctly compute C_i. The running time follows from Theorem 7 and from the fact that the work done by P_i during each round is of the form $O(|A_i| \cdot |B_i| \cdot n)$ □

4.5 Empirical Results

The algorithms for the communication patterns described in Section 3 and some of the algorithms in this section 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 [25]. This work builds on the LEDA library of sequential data structures and algorithms [26]. The library is written in C++ and uses MPI for message passing.

Timing results are presented for a sorting algorithm, which uses the Random-Sample and Linear-Partition patterns (see Corollary 2), and for a parallel version of the Floyd-Warshall all pairs shortest path algorithm (see, e.g., [12]), which is based on the Circulate pattern. These results were obtained on a dedicated cluster of workstations consisting of 14 166MHz Pentium processors interconnected by a 100MHz Ethernet switch, running Linux,

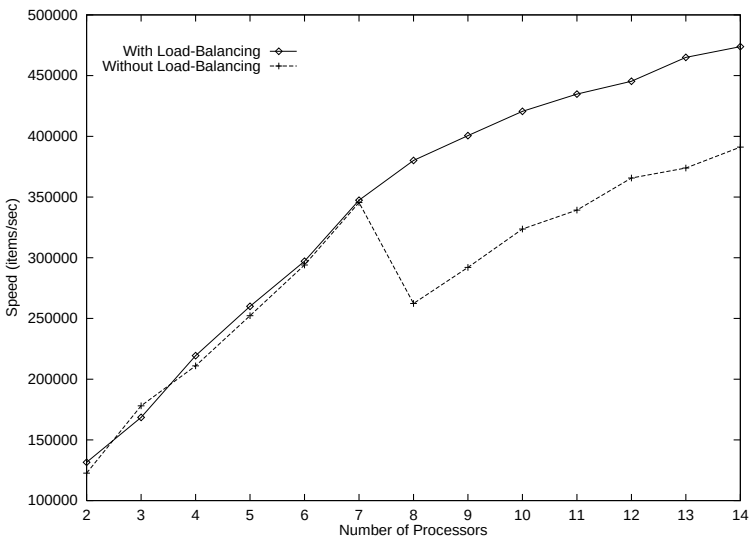

Figure 5: Performance of CGM and HCGM versions of Sample Sort.

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. P_0 through P_6 were run at the regular speed, while P_7 through P_{13} were crippled.

Figure 5 compares the results of using the HCGM Linear-Partition algorithm and then sorting locally against the results obtained by standard Sample Sort [21]. 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 5 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 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 6. 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.

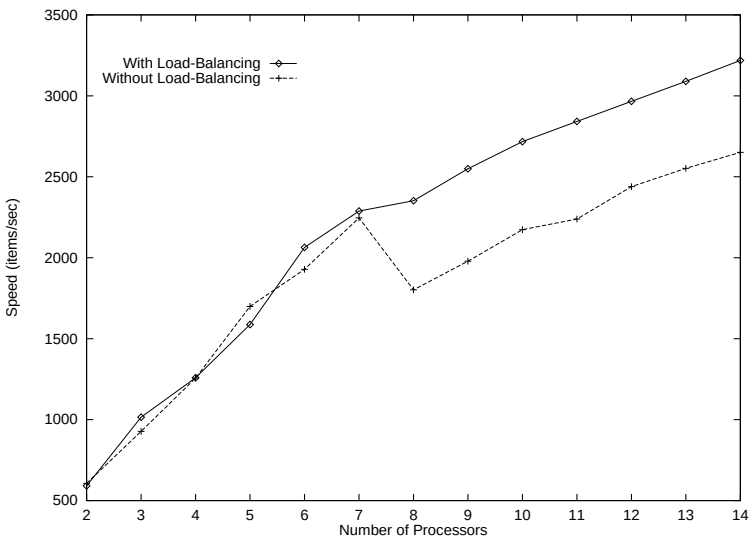

Figure 6: Performance of CGM and HCGM versions of Floyd-Warshall algorithm

5 Conclusions

We have defined a model for parallel computing on heterogeneous systems, and described a number of algorithms for this model. Although this model is very simple, the empirical results in Section 4 suggest that it captures the most important aspects of actual systems, and algorithms which the model predicts as being fast tend to work well in practice.

There are several directions in which this work could be extended. Although the HCGM model appears to be a fairly good model for developing algorithms, it is not (nor is it intended to be) an accurate predictor of exact running times. One could argue that the asymptotic analyses used in this thesis are not strong enough, since in practice processor speeds do not vary by more than constant factors. To this end, it might be useful to define an HBSP model which incorporates the BSP parameters g and L. Tests like those done by Wijshoff and Juurlink [33] could then be performed to determine whether the resulting model is an accurate predictor of performance.

We note that such extensions to the BSP model are not entirely trivial. The question of whether to apply a bandwith limitation locally, by e.g., charging a cost of $\frac{g}{s_i}$ for each word sent or received by P_i, or globally as is done in the HCGM model is not an easy one. In a recent paper, Adler *et. al.* [1] have shown that a model in which bandwidth is restricted globally is significantly more powerful than a model in which bandwidth is restricted locally.

Empirical testing is necessary to determine which is more appropriate, and it may be the case that the choice of global versus local bandwidth restriction depends on the actual parallel machine being modelled.

Another direction for future work is a direct comparison of the algorithms described in this thesis with algorithms based on the overparitioning approach described in Section 1. The difficulty with this is that although algorithms which use the overpartitioning approach exist for some problems, most of the algorithms described in Section 4 do not have counterparts based on overpartitioning. Thus, regardless of the results of the outcome of such tests, most of the algorithms in Section 4 are the only algorithms currently available for such problems.

One advantage which algorithms based on overpartitioning may have over HCGM algorithms is that they can adjust to changing load in systems in which processors are shared among users. Although the HCGM model could be extended to incorporate dynamically changing processor speeds, the algorithms for this model would have to be very different than those presented in this thesis. This is due to the fact that HCGM algorithms attempt to minimize the number of supersteps, which means maximizing (within reason) the amount of computation performed between communication operations. Therefore, if processor speeds change frequently and drastically, most of the computation will be non-optimal. It seems that the CGP paradigm of algorithm design is simply not suited for a scenario in which processor speeds change dynamically.

The communication patterns of Section 3 are also interesting in their own right. Since most existing CGP algorithms can be expressed in terms of these patterns, they are an excellent starting point for a software framework which is to support the development of CGP algorithms. This is the direction being pursued in the PLEDA project [25]. Ongoing work in this area includes the implementation and testing of more algorithms, as well as keeping the list of communication patterns up-to-date as new algorithms are developed which use different patterns. On a more theoretical note, it may be interesting to develop algorithms for the patterns of Section 3 which do not rely on randomization.

Acknowledgements

The author would like to thank Silvia Götz, Anil Maheshwari, Ben Juurlink, Jörg Sack, and Frank Dehne for several helpful discussions, and for having read and commented on earlier

versions of this paper.

References

[1] M. Adler, P. B. Gibbons, Y. Matias, and V. Ramachandran. Modelling parallel bandwidth: Local vs. global restrictions. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–105, 1997.

[2] D. Bader, D. Hellman, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, 1996.

[3] A. Bäumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[4] 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.

[5] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the European Symposium on Algorithms*, pages 17–30, 1995.

[6] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. Technical report, University of Paderborn, 1996.

[7] A. Bäumker, W. Dittrich, and A. Pietracaprina. The deterministic complexity of parallel multisearch. In *Proc. 5th SWAT*, 1996.

[8] 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.

[9] E. Cáceres, 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 *Proceedings of International conference on Automata, Languages, and Programming*, 1997.

[10] 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.

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

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[13] 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.

[14] 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.

[15] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International journal on Computational Geometry*, 6(3):379–400, 1996.

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

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications to computational geometry. In *Proceedings of EuroPar'96*, 1996.

[19] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications to computational geometry. Technical report, Oxford Computing Laboratory, 1996.

[20] 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.

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

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

[23] D. Knuth. *The Art of Computer Programming – Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] J. Reif, editor. *Synthesis of parallel algorithms.* Morgan Kaufmann, 1993.

[31] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of IEEE Aerospace*, 1997.

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

[33] H. A. G. Wijshoff and B. H. H. Juurlink. A quantitative comparison of parallel computation models. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–24, 1996.

[34] 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.

[35] 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.

[36] 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.

Biographical Note

Pat Morin received his B.C.S. degree from Carleton University in 1996, and his M.C.S. degree in 1998. His research interests include parallel algorithms and computational geometry. He is currently enrolled in the PhD program at Carleton.

A High Probability Bounds

When discussing randomized algorithms of any type it is usually not enough to give the expected running time of the algorithm. The reason for this is that, while the expected running time may be good, any particular execution of the algorithm may take significantly longer.

This is where high probability bounds are used. The idea behind such bounds is to show that the execution time of an algorithm rarely differs significantly from its expected running time, at least in the case where the input size is large. The definition of high probability is usually applied to a random variable. However, in order to emphasize its application, we describe it in terms of the running time of a randomized algorithm.

Definition 1. *We say that a randomized algorithm $\mathcal{A}$ has running time $O(f(n))$* with high probability*, denoted $\tilde{O}(f(n))$ if*

$$\Pr\left[\textit{Execution time of } \mathcal{A} \geq cf(n)\right] \leq \frac{1}{n^{\Omega(c)}} \ .$$

It is worth noting that no assumption is made about the input to $\mathcal{A}$, and so $\mathcal{A}$ must rely strictly on the randomization to achieve this running time.

A.1 The Binomial Distribution

A *Bernoulli trial* is any experiment with only two possible outcomes, success or failure. In such a trial we define q as the probability of success and $1-q$ as the probability of failure. If we denote success by 1 and failure by 0 we see that a Bernoulli trial X is defined by

$$\Pr\left[X = 1\right] = q \text{ and } \Pr\left[X = 0\right] = 1 - q.$$

If we define $X(n)$ as the number of successes during n Bernoulli trials with success probability q then we have

$$\Pr\left[b(n,q) = k\right] = C(n,k)q^k(1-q)^{n-k}.$$

We say that $b(n,q)$ follows a *binomial distribution* with success probability q and number of trials equal to n.

Binomial distributions occur frequently when giving high probability bounds for randomized algorithms. The nice property of the binomial distribution is that if the running time of an algorithm follows the distribution $b(n,q)$, then for certain values of n and q, the running time of the algorithm is $\tilde{O}(nq)$. The following theorem of Chernoff is useful in proving such results.

Theorem 14 (Chernoff Bounds). *Let X be a random variable which follows the distribution $b(n, q)$, then*

$$\Pr[X \geq cqn] \leq \left(\frac{1}{e}\right)^{(c-1)^2 qn/3}, \text{ for } c > 1 \tag{1}$$

and

$$\Pr[X \leq cqn] \leq \left(\frac{1}{e}\right)^{(1-c)^2 qn/2}, \text{ for } c < 1. \tag{2}$$