1-1-2001

Practical experience using a computational model for the design of heterogeneous distributed software

T. L. Williams
University of Central Florida

R. J. Parsons
University of Central Florida

Practical Experience Using a Computational Model for the Design of Heterogeneous Distributed Software

T. L. Williams and R. J. Parsons
University of Central Florida
School of EECS
Orlando, FL USA 32816-2362
{williams,rebecca}@cs.ucf.edu

April 15, 2001

KEYWORDS: distributed systems, heterogeneous computing, performance evaluation, BSP, cluster computing, collective communication.

Abstract

Heterogeneous cluster environments are becoming an increasing popular platform for executing parallel applications. Efficient heterogeneous programs must account for the differences inherent in such an environment. We propose the HBSP^1 model of computation as a framework for developing applications for heterogeneous clusters of workstations. The utility of the model is demonstrated through the design and analysis of the scatter and one-to-all broadcast algorithms. Extensive experimentation illustrates the benefits of using the model for heterogeneous program development. By hiding the non-uniformity of the underlying system, the HBSP^1 model provides a framework that embraces the heterogeneity of the underlying system.

1 Introduction

The growth of the Internet has contributed to an increased interest in distributed software. In fact, it is not uncommon for distributed applications to execute on a collection of machines with myriad differences such as computational speeds, memory sizes, and data formats. Such platforms are considered to be heterogeneous distributed environments. One example is the SETI@home project, which exploits the enormous amounts of idle time going to waste on PCs to crack encryption challenges. Performance gains in heterogeneous environments result from effectively exploiting the speeds of the underlying components. Executing standard (homogeneous) distributed applications on heterogeneous platforms leads to low-end systems becoming a bottleneck, which reduces overall system performance. Thus, a new approach is necessary for the design of efficient heterogeneous distributed applications.

The k-Heterogeneous Bulk Synchronous Parallel model (HBSP^k) is the model we propose for the development of general-purpose heterogenous applications (Williams, 2000). It is an extension of the BSP model of parallel computation (Valiant, 1990). The superscript k refers to the number of network layers present in the heterogeneous environment. Unlike BSP, the HBSP^k model describes multiple heterogeneous platforms connected by some combination of internal buses, local-, campus, and wide-area networks. Applicable environments include workstation clusters, the Internet, and computational grids (Foster and Kesselman, 1998). In this paper, we focus on the development of programs for a heterogeneous cluster of workstations. Since these systems are connected by a single communications network, we concentrate on the HBSP^1 model, which is a specific instantiation of the generalized HBSP^k model.

Collective communication algorithms are used frequently as building blocks in a variety of distributed algorithms. Proper implementation of these operations is vital to the efficient execution of the distributed algorithms that use them. Collective operations designed for homogeneous distributed systems are not adequate for heterogeneous environments. As a result, we present two collective communication algorithms—scatter and one-to-all broadcast—for a heterogeneous cluster of workstations. Our HBSP^1 algorithms are based on BSP communication routines (Hill, Donaldson and Skillicorn, 1997; Juurlink and Wijshoff, 1996). Our design strategy, which is guided by the HBSP^1 model, for these algorithms is two-fold. Faster workstations should be involved more in the computation than slower machines. Secondly, faster nodes should receive more data items than slower nodes. HBSP^1 predicts that increased performance will result if these guidelines are taken into consideration when designing heterogeneous applications.

We perform extensive experiments to validate the predictions of the model. Our experimental testbed consists of a non-dedicated, heterogeneous cluster of workstations. Experimental results demonstrate that our collective algorithms have increased performance on heterogeneous platforms. Moreover, the model accurately predicts the performance trends of the communication algorithms. Improved performance is not a result of programmers having to account for myriad differences in a heterogeneous environment. By hiding the non-uniformity of the underlying system from the application developer, the HBSP^1 model offers a framework that encourages the design of software for heterogeneous clusters in an architecture-independent manner.

The rest of this paper is organized as follows. Section 2 gives a brief overview of related work. The HBSP^1 model is described in Section 3. Sections 4 and 5 present our experimental approach and the experimental results, respectively. Conclusions are given in Section 6.

2 Related Work

The Bulk Synchronous Parallel (BSP) (Valiant, 1990) model provides the foundation for the HBSP^k model. The BSP model provides guidance on designing applications for good performance on homogeneous parallel machines. Support for BSP includes theoretical results, empirical results, and experimental parameterization of BSP programs (Gerbessiotis and Valiant, 1994; Goudreau, Lang, Rao, Suel and Tsantilas, 1999).

Two models that address heterogeneous clusters of workstations are the Heterogeneous Coarse-Grained Multicomputer (HCGM) model (Morin, 1998) and the Heterogeneous Bulk Synchronous Parallel (HBSP) (Williams and Parsons, 2000), which is synonymous with HBSP^1. Both of these models take into account varying processor speeds to develop parallel algorithms for heterogeneous systems. The main difference between the two models is that HCGM is not intended to be an accurate predictor of execution times whereas HBSP attempts to provide the developer with predictable algorithmic performance.

Additional research has studied the performance of collective algorithms for heterogeneous workstation clusters. The ECO package (Lowekamp and Beguelin, 1996), built on top of PVM, automatically analyzes characteristics of heterogeneous networks to develop optimized communication patterns. Bhat, Raghavendra and Prasanna (1999) extend the FNF algorithm (Banikazemi, Moorthy and Panda, 1998) and propose several new heuristics for collective operations. Their heuristics consider the effect communication links with different latencies have on a system. Banikazemi, Sampathkumar, Prabhu, Panda and Sadayappan (1999) present a model for point-to-point communications in heterogeneous networks of workstations and use it to study the effect of heterogeneity on the performance of collective operations.

3 The HBSP^1 Model

HBSP^1 is a synchronous model of computation that provides a framework for the design of software for a heterogeneous cluster of workstations. The HBSP^1 model consists of a *cost model* that provides predictable costs of algorithm execution. HBSP^1 captures the essential characteristics of heterogeneous clusters with only a few parameters. More complex models tend to use more parameters that render them too tedious for practical use. Moreover, the HBSP^1 model can be viewed as a kind of *programming methodology*. The essence of the HBSP^1 approach is the notion of the superstep and the idea that the input/output (i.e., sends and receives) associated with a superstep is performed as a global operation. Viewed in this way, an HBSP^1 program is simply one that proceeds in phases, with the necessary global communications taking place between the phases.

In this section, we formally define the HBSP^1 model as well as describe the associated programming methodology. Afterwards, we use the model to guide the design and analysis of the scatter and one-to-all broadcast operations.

3.1 Model description

An HBSP^1 computer is characterized by the following parameters:

- p, the number of processors or workstations labeled $P_0, \ldots, P_{p-1}$;
- g, a bandwidth indicator that reflects the speed with which the *fastest* machine can inject messages into the communications network;
- r_j, the speed relative to the *fastest* processor for P_j to inject a packet into the network;
- L, the overhead to perform a barrier synchronization of the p processors; and
- c_j, the fraction of the problem size that P_j receives.

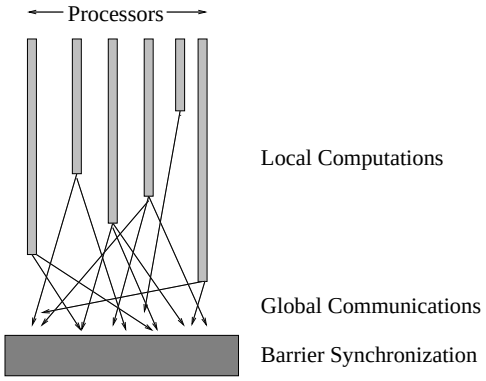

Figure 1: A superstep in the HBSP^1 model.

For notational convenience, the indices f and s identify the fastest and slowest nodes, respectively. We assume that the r_f value of the fastest machine is normalized to 1. If $r_j = t$, then P_j communicates t times slower than the fastest workstation. The c_j parameter adds a load-balancing feature into the model. It's value is in the range from 0 to 1. Specifically, it attempts to provide P_j with a problem size that is proportional to its computational and communication abilities. Intuitively, c_j is inversely proportional to r_j.

Computation in an HBSP^1 machine consists of a sequence of supersteps. During a superstep, each processor performs asynchronously some combination of local computation, message transmissions, and message arrivals. A message sent in one superstep is guaranteed to be available to the destination processor at the beginning of the next superstep. Each superstep is followed by a global synchronization of all the processors. Figure 1 shows an example of a superstep.

Since HBSP^1 defines a specific programming style, the formal parameters of the model allow for the cost analysis of HBSP^1 programs. Again, the basic notion of an HBSP^1 computation is the superstep, which consists of local computation, communication, and synchronization. Let w represent the largest amount of local computation performed by a workstation. Let h_j be the largest number of messages sent or received by processor j. The size of the *heterogeneous h-relation* is $h = \max\{r_j \cdot h_j\}$ requiring a communication cost of gh. Thus, the cost of a superstep is

$$w + gh + L. \tag{1}$$

The overall cost of the program is the sum of the superstep times.

The above cost model demonstrates what factors are important when designing HBSP^1 applications. To minimize execution time, the programmer must attempt to (i) balance the local computation in each superstep, (ii) balance the communication between the machines, and (iii) minimize the number of supersteps. Balancing these objectives is a nontrivial task. Nevertheless, HBSP^1 provides assistance with making the tradeoffs necessary for the design of efficient heterogeneous programs.

3.2 Heterogeneous Algorithm Design

The HBSP^1 model provides parameters that allow application developers to exploit the heterogeneity of the underlying system. The model promotes a two-fold strat-

egy for designing heterogeneous collective operations. First, faster machines should be involved in the computation more often than their slower counterparts. Collective operations use specific nodes to collect or distribute data to the other nodes in the system. For faster algorithmic performance, these nodes should be the fastest machines in the system. Secondly, faster machines should receive more data items than slower machines. This principle encourages the use of *balanced* workloads, where machines receive problem sizes relative to their communication and computational abilities. Partitioning the workload so that nodes receive an equal number of elements works quite well for homogeneous distributed environments. However, this strategy encourages *unbalanced* workloads in heterogeneous environments since faster machines typically sit idle waiting for slower nodes to finish a computation.

Throughout the rest of this section, let n represent the total number of items of interest. Balanced workloads assume P_j possesses $c_j n$ elements.

3.2.1 Scatter

The scatter operation uses a single root node to distribute a unique message to each of the other nodes. Here, each processor j will receive $c_j n$ unique data items from P_f. In the homogeneous version, each node receives $\frac{n}{p}$ elements. The HBSP^1 scatter algorithm requires a single superstep. Therefore, the size of the single, heterogeneous h-relation is $\max\{r_j \cdot c_j n, r_f \cdot n\}$. Each processor's r_j value is relative to the fastest processor. Hence, $r_f = 1$ and $r_j \geq r_f$. Recall that c_j is inversely proportional to the speed of P_j. Consequently, $r_j c_j < 1$. Thus, the HBSP^1 scatterr cost is $gn + L$.

The above cost of the scatter operation is efficient since the fastest processor is performing most of the work. If $r_j c_j > 1, P_j$ has a problem size that is too large. Its communication time will dominate the cost of the scatter operation. Whenever possible, the fastest processor should handle the most data items. Our analysis demonstrates the importance of balanced workloads. Thus, the HBSP^k model rewards programs with balanced design.

3.2.2 One-to-all broadcast

In the one-to-all broadcast, only the source processor has the data that needs to be broadcast. At the termination of the procedure, each node has a copy of the data. The HBSP^1 broadcasts executes similarly to the two-phase BSP algorithm (Hill et al., 1997). During the first phase, the root node distributes $\frac{n}{p}$ items to each processor. Afterwards, processor j is responsible for sending its share of the data to its peers.

During the first phase of the algorithm, P_j receives $\frac{n}{p}$ items from P_f. This phase requires a heterogeneous h-relation of size $\max\{r_f \cdot n, r_j \cdot \frac{n}{p}\}$. In a typical environment, it is reasonable to assume that p ranges from the tens to the hundreds. It is quite unlikely that a machine would communicate p times slower than the fastest machine. If this is the case, it may be more appropriate not to include that machine in the computation. As a result, the communication time of the first phase reduces to gn. During the second phase, each processor must receive the same number of items. Thus, the slowest processor will cause a bottleneck. Let r_s represent the communication time of the slowest node. This results in a communication time of $gr_s n$. Actually, P_s will receive $n - \frac{n}{p}$ elements. We use n to simplify the notation. Thus, the complexity of a

two-phase broadcast on an HBSP^1 machine is $gn(1 + r_s) + 2L$.

As a point of comparison, the one-phase broadcast (P_f sends n items to each of its children) costs $gnp+L$. For reasonable values of r_s, the two-phase approach is the better overall performer. An interesting conclusion concerning the broadcast operation is that it effectively cannot exploit heterogeneity. Since the slowest processor must receive n items, its cost will dictate the complexity of the algorithm. Moreover, partitioning the problem size based on the c_j parameter is ineffective. Although wall clock performance may improve, theoretically, the resulting speedup is negligible.

4 Experimental Approach

Using the HBSP^1 as a guide, we have designed and analyzed two collective communication operations—scatter and one-to-all broadcast. According to the model, the algorithms should demonstrate good performance on a heterogeneous cluster of workstations. We are now ready to investigate the behavior of these algorithms on an actual heterogeneous platform. In this section, we describe our experimental methodology and Section 5 provides the experimental results.

4.1 HBSP*lib*

Our collective communication algorithms are implemented using the HBSP Programming Library (HBSP*lib*). Table 1 lists the functions that constitute the HBSP*lib* interface. The design of HBSP*lib* incorporates many of the functions contained in BSP*lib* (Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas and Bisseling, 1998). HBSP*lib* is written on top of PVM (Sunderam, 1990), a software package that allows a heterogeneous network of computers to appear as a single, concurrent, computational resource. The computers compose a *virtual machine* and communicate by sending messages to each other. We use PVM's `pvm_send()` function for asynchronous communication to directly send messages between heterogeneous processors. To receive a message, we take advantage of the PVM function `pvm_recv()`. The `pvm_barrier()` primitive provided by PVM assisted with the development of `hbsp_sync()`. However, our implementation of global synchronization is somewhat complex since we needed to guarantee that all messages arrived at their destination before the beginning of the next superstep.

HBSP*lib* incorporates functions that allow the programmer to take advantage of the heterogeneity of the underlying system. Under HBSP^1, faster machines should perform the most work. The primitive `hbsp_get_rank(1)` returns the identity of the fastest processor. `hbsp_get_rank(p)` returns the slowest machine's identity, where p is the number of processors. HBSP*lib* also includes functions to help the programmer distribute the workload based on a machine's ability. The HBSP*lib* primitive `hbsp_get_speed(j)` provides the speed of processor j. `hbsp_cluster_speed` returns the speed of the entire cluster. When combined together, these two functions allow for finding the value of processor j's c_j parameter. We discuss in more detail in Section 4.4.

Figure 2 shows the implementation of the scatter algorithm using HBSP*lib*. The algorithm requires 3 parameters: `sendbuf`, which contains the data items the root node sends to the other processors; `sendcounts`, which is an array that tells the root

Function	*Semantics*
hbsp_begin	Starts the program with the number of processors requested.
hbsp_end	Called by all processors at the end of the program.
hbsp_abort	One process halts the entire HBSP computation.
hbsp_pid	Returns the processor id in the range of 0 to one less than the number of processors.
hbsp_time	Returns the time (in seconds) since `hbsp_begin` was called. The timers on each of the processors are not synchronized.
hbsp_nprocs	Returns the number of processors.
hbsp_sync	The barrier synchronization function call. After the call, all outstanding requests are satisfied.
hbsp_send	Sends a message to a designated processor.
hbsp_get_tag	Returns the tag of the first message in the system queue.
hbsp_qsize	Returns the number of messages in the system queue.
hbsp_move	Retrieves the first message from the processor's receive buffer
hbsp_get_rank	Returns the identity of the processor with the requested rank.
hbsp_get_speed	Returns the speed of the processor of interest.
hbsp_cluster_speed	Returns the total speed of the heterogeneous cluster.

Table 1: The functions that constitute HBSP*lib* interface.

node the number of elements that each processor should receive (i.e., the root will send `sendcounts[j]` elements to P_j); `recvbuf`, where the nodes store the items received from the root node; and `root`, the identity of the source node. The algorithm first requires the root node to send the data to all of the other processors. In order to send the data, the `hbsp_send` requires the destination, a tag to identify the message (if relevant), the beginning address of the data buffer, and the size of the data to be communicated. In the second superstep, each processor puts the data sent to it from the root into its `recvbuf`.

4.2 Experimental platform

Our experimental testbed consists of a non-dedicated, heterogeneous cluster of SUN and SGI workstations at the University of Central Florida. Table 2 lists the specifications of each machine. Our platform is quite heterogeneous. CPU speeds range from 85 MHz to 360 MHz and memory sizes vary between 64 MB to 256 MB. Each node is connected by a 100Mbit/s Ethernet connection.

4.3 Machine ranking

The ranking of the heterogeneous nodes is determined by the BYTEmark benchmark (Magazine, 1995), which consists of a variety of different tests that extensively exercise a machine's capabilities. A sampling of programs in the benchmark suite include numeric and string sorting, an IDEAL encryption algorithm, Huffman compression, a floating-point package, a back-propagation network simulator, and a LU Decomposition solver.

```
void bsp_scatter(int *sendbuf, int *sendcounts, int *recvbuf, int root)
{
   int bytes, i, j, offset, size, temp;

   /* root sends data to the processors */
   if (hbsp_pid() == root) {
      offset = 0;
      for (i = 0; i < p; i++) {
         if (root != i)
             hbsp_send(i, NULL, sendbuf+offset, sendcounts[i] *sizeof(int));
         else
             temp = offset;
         offset += sendcounts[i];
       }

       /* root copies its data into recvbuf */
       size = sendcounts[root];
       for (i = 0; i < size; i++)
          recvbuf[i] = sendbuf[i+temp];
   }
   hbsp_sync();

   /* processors receive data from root */
   if (hbsp_pid() != root) {
      hbsp_get_tag (&bytes, NULL);
      bsp_move(recvbuf, bytes);
   }
}
```

Figure 2: The scatter algorithm written using HBSP*lib*.

Host	CPU type	CPU speed (MHz)	Memory (MB)	Data cache (KB)
aditi‡	UltraSPARC II	360	256	16
chromus	microSPARC II	85	64	8
dcn_sgi1	MIPS R5000	180	128	32
dcn_sgi3	MIPS R5000	180	128	32
gradsun1	TurboSPARC	170	64	16
gradsun3	TurboSPARC	170	64	16
gromit	UltraSPARC IIi	333	128	16
sgi1	MIPS R5000	180	96	32
sgi3	MIPS R5000	180	96	32
sgi7	MIPS R5000	200	64	32

Table 2: Specification of the nodes in our heterogeneous cluster. ‡ A 2 processor system, where each number is for a single CPU.

Machine	Integer Index	Floating-point Index
aditi	4.45	3.77
chromus	0.75	0.59
dcn_sgi1	2.80	3.73
dcn_sgi3	2.79	3.67
gradsun1	1.80	1.41
gradsun3	1.81	1.42
gromit	4.89	3.33
sgi1	2.81	3.60
sgi3	2.77	3.30
sgi7	3.13	4.11

Table 3: BYTEmark benchmark scores.

After running all of the tests, BYTEmark produces two overall figures, an Integer and a Floating-point index. The Integer index is the geometric mean of those tests that involve only integer processing. The remaining tests comprise the Floating-point index. Since the benchmark is a few years old, the index score calculation is based on the performance of a 90 MhZ Pentium. If a machine has an index score of 2.0, it is twice as fast a 90 MhZ Pentium computer.

Table 3 presents the Integer and Floating-point index scores for each machine in the heterogeneous cluster. Since we consider integer data only, the Integer index scores were used to rank the processors. According to the results, `chromus` is the slowest node. `gromit` is the fastest machine in the cluster. This result is surprising considering `aditi` appears faster on paper. Interestingly, `aditi` narrowly edges out `gromit` in every test, except string sort, where `gromit` outperforms `aditi` with a score of 7.63 to 2.40. Since BYTEmark uses only a single execution thread, it cannot take advantage of `aditi`'s additional processor. This does not present a problem for our experiments since our HBSP*lib* implementation does not use threads. We ran our experiments with both `aditi` and `gromit` as the fastest processor. There was no major difference in the execution times. Therefore, we consider `gromit` to be the fastest processor in the cluster.

4.4 Parameter estimation

In order to compare the actual and predicted (theoretical) execution times of the algorithms, we must determine the values of the HBSP^1 parameters on an actual heterogeneous platform. Below, we describe our method for finding the values of the c_j, r_j, and L parameters of the HBSP^1 model. It is important to note that these are architecture-dependent parameters. If we were to change the underlying platform, we would need to recalculate the parameter values for that environment.

Unlike a homogeneous environment, the ordering of the processors can have a dramatic effect on the performance results. To ensure consistent results, we apply the same processor ordering for each experiment. Table 4 shows the ordering. When $p = 2$, the experiments utilize `gromit` and `chromus`. The speed of this configuration is 5.64, which is the sum of each machine's Integer index score. Each machine's c_j value is

p	Machine	Speed	$L(\mu s)$
2	gromit, chromus	5.64	9,000
4	aditi, dcn_sgi1	12.89	15,000
6	dcn_sgi3, gradsun1	17.48	23,000
8	gradsun3, sgi1	22.10	30,000
10	sgi3, sgi7	28.00	37,000

Table 4: Cluster speed and synchronization costs.

machine	r_j
aditi	1.03
chromus	4.08
dcn_sgi1	2.12
dcn_sgi3	1.95
gradsun1	2.00
gradsun3	2.46
gromit	1.00
sgi1	1.68
sgi3	1.20
sgi7	1.16

Table 5: r_j values.

based on its Integer index score and the cluster speed. In general, $\sum_{j=0}^{p} c_j = 1$. When $p = 2$, `gromit`'s c_j value is $\frac{4.89}{5.64}$ (or .867). The c_j value of `chromus` is .133. Therefore, `gromit` receives 86.7% of the data elements and `chromus` acquires the remaining 13.3%. When $p = 4$, the cluster speed is 12.89. The workstations that comprise the cluster are `gromit, chromus, aditi,` and `dcn_sgi1`, which receive 37.9%, 5.8%, 34.5%, and 21.7% of the input, respectively.

Table 4 also presents the synchronizing costs of the clusters comprised of 2, 4, 6, 8, and 10 workstations. For example, synchronizing two processors (i.e, `gromit` and `chromus`) requires 9,000 μs. The value of L corresponds to the time for an empty superstep (i.e., no computation or communication). When $p = 4$, 15,000 μs are needed in order to synchronize the processors. Several factors contribute to the high synchronization costs. Since the cluster is non-dedicated, many other nodes share the network link, which effectively degrades communication performance. Secondly, our implementation of barrier synchronization is not necessarily efficient. Despite the high L values, our collective algorithms outperformed their PVM counterparts. Additional work will focus on the development of a more efficient barrier synchronization primitive.

Table 5 shows the r_j values achieved on our heterogeneous cluster. To obtain these values, we measure the time needed for each machine to inject a sufficiently large packet into the network. `gromit` performed the best with a score of 0.196 $\frac{\mu s}{byte}$. Processor j's r_j value is relative to this score.

5 Experimental Results

The input data for each experiment consists of 100 KBytes to 1000 KBytes of uniformly distributed integers. The problem size, n, refers to the largest number of integers possessed by the root. Experimental results are given in terms of an improvement factor. Let T_A and T_B represent the execution time of algorithm A and algorithm B, respectively. The improvement factor of using algorithm B over algorithm A is $\frac{T_A}{T_B}$.

The HBSP^k model encourages the use of fast processors and balanced workloads. According to the model, applications that embody both of these principles will result in good performance. We designed two types of experiments to validate the predictions of the model. The first experiment tests whether processor speed has an impact on algorithmic performance. Let T_s represent the execution time of a collective routine assuming the root node is the slowest processor, P_s. T_f denotes the algorithmic cost of using P_f as the root. For these experiments, each processor has an equal number of data items since our objective is to monitor the performance of slow versus fast root nodes. Hence, $c_j = \frac{1}{p}$. The results demonstrate that often times using the fastest node as the root results in significant performance improvement.

Our second experiment studies the benefit of using the fastest processor as the root and balanced workloads. Let T_u be the execution time when the workload is unbalanced. Note that $T_u = T_f$. Each processor j's c_j value is $\frac{1}{p}$. T_b denotes the execution time when the workload is balanced. Here, c_j is computed as described in the previous section. In most cases, the results demonstrate that balanced workloads improve the performance of the algorithm.

We also investigate the accuracy of the HBSP^1 cost function in predicting execution times. Similarly to BSP, we consider HBSP^k to model only communication and synchronization (Goudreau et al., 1999). I/O and local computation are not modeled. As a result, none of our experiments include I/O. Furthermore, the work component (w) of our algorithms is neglible. As a result, the cost model that we use to predict the cost of a superstep is $gh + L$. Our results show that the model is able to predict performance trends, but not specific execution times. The inability of HBSP^k to predict specific execution times does not reflect negatively toward the model. The accuracy of the cost function depends on the choices made in the implementation of the HBSPlib library. Thus, one source for inaccurate predictions may result from the shortcomings of the library implementation.

The remainder of this section provides experimental results for the scatter and one-to-all broadcast operations. Complete experimental results can be found in Williams (2000). Each data point is the average of 10 runs. For each of the experiments, the logic of the algorithms is not changed. Instead, the modifications occur in either root node selection or problem size distribution. In both cases, performance increase is substantial.

5.1 Scatter

Figure 3 (a) plots the increase in performance if the root node is the fastest processor. The improvement factor is steady as the problem size increases. The best improvement occurs when $p = 6$ and $n = 500$KB. When $p = 2$, $\frac{T_s}{T_f} < 1$. Figure 3 (b) compares the performance of unbalanced and balanced workloads. The results indicate that there

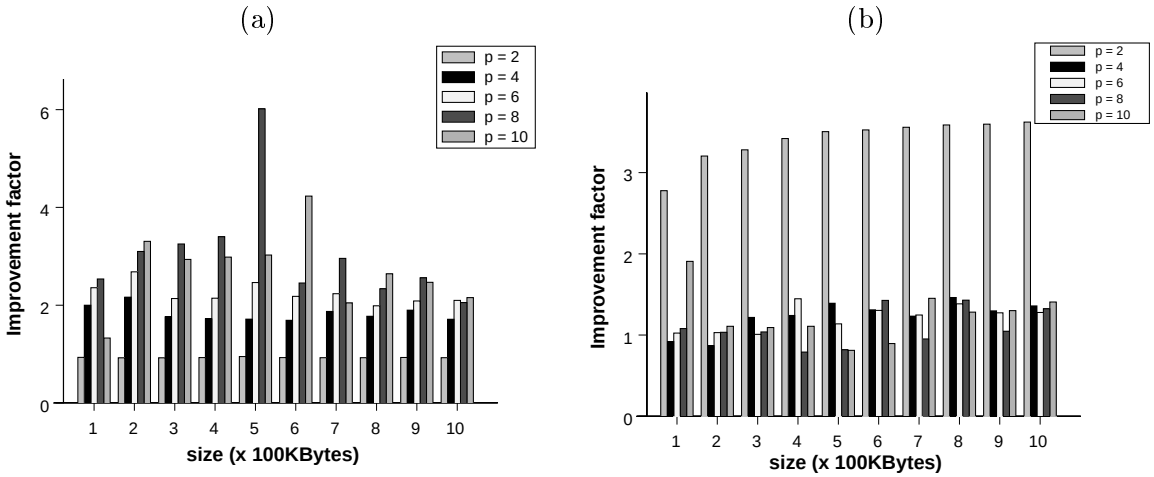

Figure 3: Scatter actual performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the average of 10 runs on a cluster comprised of 2, 4, 6, 8, and 10 heterogeneous processors.

is a benefit to distributing the problem size based upon a processor's computational abilities. Here, $p = 2$ had the best performance with a maximum improvement of 3.62. Figure 4 shows predicted performance for the scatter operation.

For both experiments, the results at $p = 2$ are interesting. First, Figure 3 (a) shows that it is better for the root node to be the slowest workstation. This seems counterintuitive. In our implementation of scatter (as well as the other collective operations), a processor does not send data to itself. When P_s is the root, P_f receives $\frac{n}{p}$ items from it. Similarly, if the fastest processor is the root, P_s receives $\frac{n}{p}$ elements from P_f. $T_s < T_f$ implies that it is more beneficial to have P_f waiting on data from P_s. Clearly, the root node should be P_f as the number of processors increase.

Secondly, at $p = 2$, balanced workloads contribute to increased performance. T_u is the execution time of P_s receiving $\frac{n}{p}$ data elements from the fastest processor. T_b is the cost of P_s receiving $c_s n$ integers from P_f, where c_s is calculated as described in Section 4.4. Note that $c_s n < \frac{n}{p}$. In this setting, balanced workloads make a difference (i.e., $T_b < T_u$) since P_f sends a smaller number of elements to P_s than in the unbalanced case.

5.2 One-to-all broadcast

Figure 5 (a) compares the execution time of the algorithm assuming the root node is either P_s or P_f. The plot demonstrates that their is neglible improvement in performance. The HBSP^k model predicted this behavior. The broadcast operation takes small advantage of the heterogeneity since each processor must receive all of the data. In fact, the improvement in performance is a result of P_f distributing $\frac{n}{p}$ integers to each processor during the first phase of the algorithm. Our analysis also applies if processor j receives $c_j n$ elements during phase one of the algorithm. Figure 5 (b) corroborates the theoretical results. Figure 6 plots the predictions of the cost model,

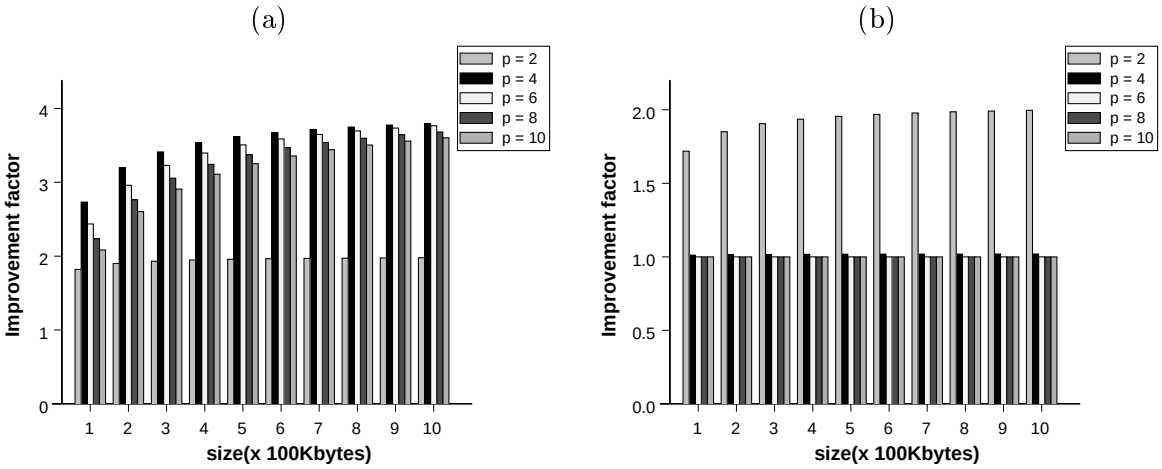

Figure 4: Scatter predicted performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8, and 10 heterogeneous processors.

which over-predicts the benefit of using the fastest processor.

6 Conclusions

The HBSP^1 model offers a framework that promotes the development of distributed applications for heterogeneous clusters of workstations. HBSP^1 incorporates a small set of parameters that characterize the underlying heterogeneous platform. Efficient algorithmic execution results from nodes receiving a workload proportional to their computational and communication abilities, if applicable. For example, a close examination of the one-to-all broadcast operation demonstrates that it is impossible to avoid unbalanced workloads since the slowest machine must receive n items. The performance of our collective operations is quite impressive. Complete results are shown in Williams (2000). Fundamental changes to the algorithms are not necessary in order to attain an increase in performance. Besides good performance, the model predicts the behavior of our collective routines within a reasonable margin of error.

In conclusion, HBSP^1 offers a single-system image of a heterogeneous platform to the application developer. Under HBSP^1, improved performance is not a result of programmers having to account for myriad differences in a heterogeneous environment. By hiding the non-uniformity of the underlying system from the application developer, the HBSP^1 model offers an environment that encourages the design of heterogeneous distributed software in an architecture-independent manner. Extensions to this work include designing HBSP^1 applications that can take advantage of our heterogeneous collective routines. We also intend to perform additional experiments on a heterogeneous cluster with a larger set of workstations.

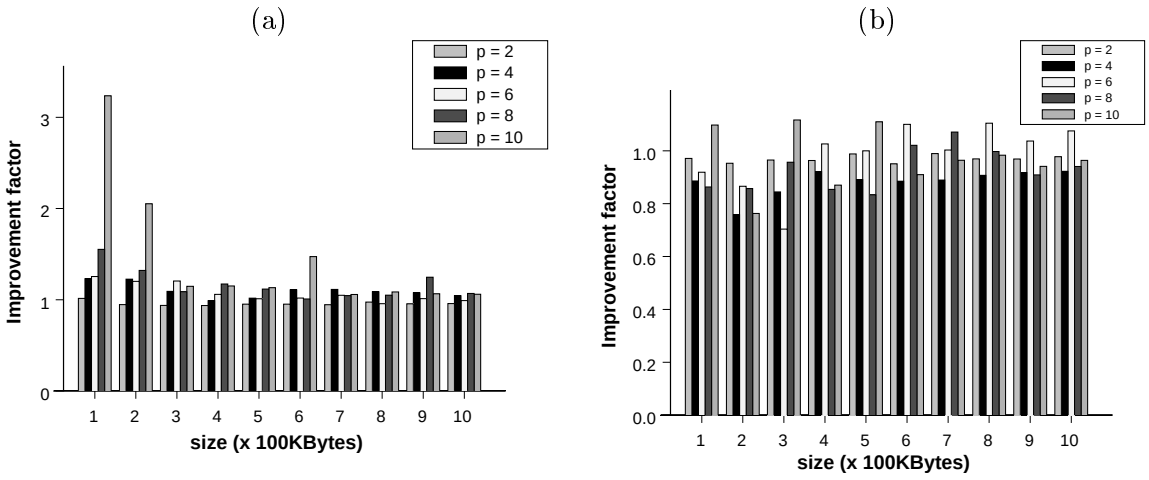

Figure 5: One-to-all broadcast actual performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the average of 10 runs on a cluster comprised of 2, 4, 6, 8, and 10 heterogeneous processors.

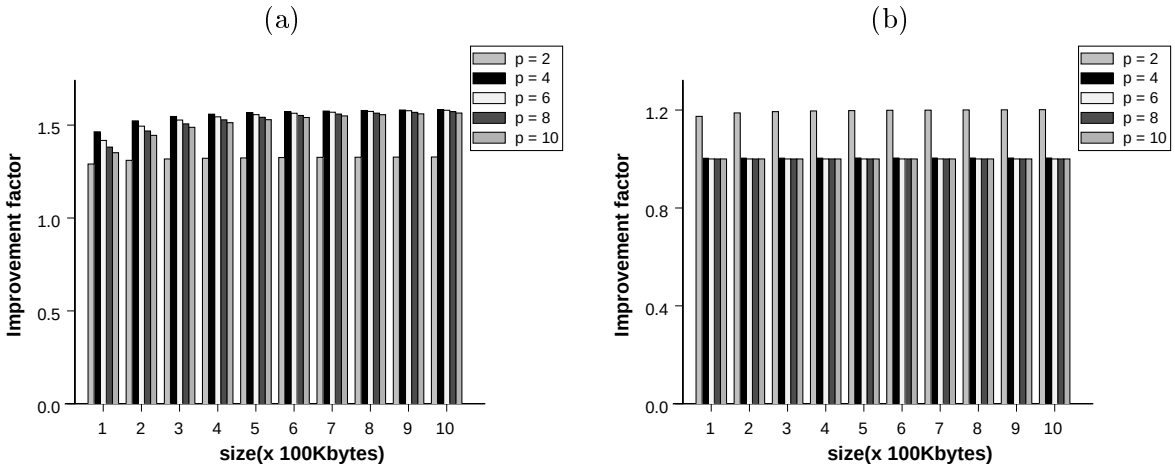

Figure 6: One-to-all broadcast predicted performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8, and 10 heterogeneous processors.

References

Banikazemi, M., Moorthy, V. and Panda, D. (1998). Efficient collective communication on heterogeneous networks workstations, *International Conference on Parallel Processing*, pp. 460–467.

Banikazemi, M., Sampathkumar, J., Prabhu, S., Panda, D. and Sadayappan, P. (1999). Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations, *Heterogeneous Computing Workshop (HCW '99)*, pp. 125–133.

Bhat, P., Raghavendra, C. and Prasanna, V. (1999). Efficient collective communication in distributed heterogeneous systems, *International Conference on Distributed Computing Systems*.

Foster, I. and Kesselman, C. (eds) (1998). *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann.

Gerbessiotis, A. V. and Valiant, L. G. (1994). Direct bulk-synchronous parallel algorithms, *Journal of Parallel and Distributed Computing* **22**(2): 251–267.

Goudreau, M. W., Lang, K., Rao, S. B., Suel, T. and Tsantilas, T. (1999). Portable and efficient parallel computing using the BSP model, *IEEE Transactions on Computers* **48**(7): 670–689.

Hill, J. M. D., Donaldson, S. R. and Skillicorn, D. (1997). Portability of performance with the BSPlib communications library, *Programming Models for Massively Parallel Computers (MPPM '97)*.

Hill, J. M. D., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. and Bisseling, R. (1998). BSPlib: The BSP programming library, *Parallel Computing* **24**(14): 1947–1980.

Juurlink, B. H. H. and Wijshoff, H. A. G. (1996). A quantitative comparison of parallel computation models, *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 13–24.

Lowekamp, B. B. and Beguelin, A. (1996). Eco: Efficient collective operations for communication on heterogeneous networks, *International Parallel Processing Symposium*, Honolulu, HI, pp. 399–405.

Magazine, B. (1995). The BYTEmark benchmark, URL http://www.byte.com/bmark/bmark.htm.

Morin, P. (1998). Coarse-grained parallel computing on heterogeneous systems, *Proceedings of the 1998 ACM Symposium on Applied Computing*, pp. 629–634.

Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing, *Concurency: Practice and Experience* **2**(4): 315–349.

Valiant, L. G. (1990). A bridging model for parallel computation, *Communications of the ACM* **33**(8): 103–111.

Williams, T. L. (2000). *A General-Purpose Model for Heterogeneous Computation*, Ph.D. dissertation, University of Central Florida, Orlando.

Williams, T. L. and Parsons, R. J. (2000). The heterogeneous bulk synchronous parallel model, *Parallel and Distributed Processing*, Vol. 1800 of *Lecture Notes in Computer Science*, Springer-Verlag, Cancun, Mexico, pp. 102–108.