Scalable S-to-P Broadcasting on Message-Passing MPPs*

Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
seh@cs.purdue.edu

Ashfaq A. Khokhar
Department of Electrical Engineering and
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
ashfaq@eecis.udel.edu

Yi Liu
Software Technology Group
Hughes Network Systems
yliu@hns.com

March 17, 1998

Abstract: In s-to-p broadcasting, s processors in a p-processor machine contain a message to be broadcast to all the processors, $1 \leq s \leq p$. We present a number of different broadcasting algorithms that handle all ranges of s. We show how the performance of each algorithm is influenced by the distribution of the s source processors and by the relationships between the distribution and the characteristics of the interconnection network. For the Intel Paragon we show that for each algorithm and machine dimension there exist ideal distributions and distributions on which the performance degrades. For the Cray T3D we also demonstrate dependencies between distributions and machine sizes. To reduce the dependence of the performance on the distribution of sources we propose a repositioning approach. In this approach the initial distribution is turned into an ideal distribution of the target broadcasting algorithm. We report experimental results for the Intel Paragon and Cray T3D and discuss scalability and performance.

Key Words: Broadcasting, communication operations, message-passing MPPs, scalability.

*Research supported in part by DARPA under contract DABT63-92-C-0022ONR. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing official policies, expressed or implied, of the U.S. government. A preliminary version of this paper appeared in the *25-th International Conference on Parallel Processing, August 1996.*

1 Introduction

The broadcasting of messages is a basic communication operation in message-passing massively parallel processors (MPPs). Two variants of the broadcasting operation, the one-to-all and all-to-all broadcast, have been studied extensively and are generally included in communication libraries [1, 2, 6, 7, 12, 17]. In the one-to-all broadcast, one processor broadcasts a message to every other processor [4, 11, 15, 16, 18]. In the all-to-all broadcast, every processor broadcasts a message to every other processor [3, 8, 10, 13, 20]. Broadcasting problems arising in parallel applications are not limited to these two forms. The number and positions of the processors initiating a broadcast can vary and may not be known in advance. In this paper we present a unified framework for s-to-p broadcasting; i.e., when s of the p processor of a machine simultaneously initiate a broadcast, $1 \leq s \leq p$. We refer to the s processors as the *source processors.* We describe different algorithms for s-to-p broadcasting and discuss experimental results for the Intel Paragon and Cray T3D.

S-to-p broadcasting, which is also known as many-to-all broadcasting [19], arises, for example, when processors generate broadcast messages in a dynamic fashion. In iterative algorithms, processors may initiate a broadcast when their own computations have led to a significant change in data values stored at other processors. This scenario is also known as dynamic broadcasting [21]. In dynamic broadcasting the distribution of the sources is often random. An application in which the number of source processors is not known in advance, but the positions of the processors tend to follow regular patterns, is dynamic load balancing for distributed data structures as described in [9].

Parameters influencing scalability (i.e., the ability to maintain speedup) of s-to-p algorithms include the number of processors, the message sizes, and the number of source processors. Our results demonstrates that other architecture-dependent factors can have considerable influence on scalability and thus on the choice of the algorithm that gives the best performance. For any fixed s, an s-to-p algorithm can exhibit a different behavior depending where the source processors are located. An algorithm-architecture pair is likely to have ideal distribution patterns and distribution patterns that give poor performance. Poor distribution patterns for one pair can be ideal for another.

In this paper we describe a number of different s-to-p broadcasting algorithms and investigate for each algorithm good and poor distribution patterns. We characterize how parameters of s-to-p algorithms impact different source distributions. Some of our algorithms are tailored towards mesh architectures and others are based on architecture-independent approaches. Our goal is to design algorithms whose behavior can be characterized as follows:

- the number of processors actively involved in the broadcasting process increases as fast as possible and
- the message length at the processors increases as slowly as possible.

We show that achieving these two objectives can be more difficult for regular machine sizes (i.e., machines whose dimensions are a power of 2). This, in turn, implies that good or bad input distributions cannot be characterized by the pattern alone. The dimension of the machine plays a crucial role as well. We show that for the Paragon algorithms based on the above stated goal give the best performance. For the Paragon, the performance obtained on ideal distributions can differ by a factor of 2 from that obtained on poor distributions. To eliminate the dependence of an algorithm on source distribution and machine dimension, we propose the approach of repositioning sources. The basic idea is to perform a permutation to transform the given distribution into an ideal distribution for a particular algorithm which is then invoked to perform the actual broadcast. For the Cray T3D, we also show that some distributions performs better than the others, providing an argument for repositioning algorithms.

In order to study the relationships between algorithms, source distributions, and machine architecture, we assume that every processor knows the position of the source processors and the size of the messages when s-to-p broadcasting starts. If this does not hold, synchronization and possible communication is needed before our algorithms can be used. The paper is organized as follows. In Section 2 we describe the algorithms that do not reposition sources. In Section 3 we discuss different repositioning approaches. Section 4 describes the different source distributions we consider. In Section 5 we discuss performance and scalability of the proposed algorithms on the Intel Paragon and Cray T3D. Section 6 concludes.

2 Algorithms without Repositioning

A natural first approach for implementing communication operations is to make use of operations available in communication libraries [1, 2, 6, 7]. Not surprisingly, s-to-p broadcasting can be phrased in terms of such operations. We start this section by describing three such solutions. The broadcasting of the s messages can be performed by every source processor sending its message to a designated processor, say processor P_0. P_0 combines the s messages into one large message which is then broadcast to all processors. In terms of existing operations, this corresponds to an s-to-one gather operation followed by a 1-to-p broadcast. An alternative approach is to view the s-to-p broadcast as an all-to-all broadcast in which the s source processors broadcast their message and $p-s$ processors broadcast a null message. A third solution is to have each processor view its message as $p-1$ distinct messages and perform an all-to-all personalized exchange (i.e., an operation in which every processor sends a unique message to every other processor). Our work has shown that the above three approaches can have serious drawbacks resulting in poor performance. In our experimental study we include two of the approaches for the sake of comparison: **Algorithm 2-Step** which performs an s-to-one gather followed by a 1-to-p broadcast and **Algorithm PersAlltoAll** which performs a personalized message exchange.

Another possible implementation based on available operations is to allow each source processor to initiate its own 1-to-p broadcast, independent of the location and number of source processors. Such a solution seems attractive for dynamic broadcasting situations since it does not require synchronization before the broadcasting. However, having the s broadcasting processes take place without interaction and coordination leads to poor performance due to arising congestion and the large number of messages in the system.

In the algorithms we consider every source processor initiates broadcast, after a global synchronization. Whenever messages from different sources meet at a processor, messages are combined. Thus, subsequent steps proceed with fewer messages having larger size. Our algorithms differ on the patterns underlying the merging of messages. The merging patterns are chosen to satisfy the objectives given in Section 1. The remainder of this section describes three algorithms based on this principle: Algorithms *Br_Lin*, *Br_xy_source*, and *Br_xy_dim*.

In **Algorithm Br_Lin**, we view the processors as forming a linear array. When the underlying architecture is a mesh, the indexing may correspond to a snake-like row-major indexing. However, the linear array does not have to be a physical one, it can be a logical one. If processors P_i and $P_{i+p/2}$, $1 \leq i \leq p/2$, both contain a message to be broadcast, they exchange their messages and form a larger message consisting of the original and the received message. If only one of the processors contains a message, it sends the message to the other processor. Then, Algorithm *Br_Lin* proceeds recursively on the first $p/2$ and the last $p/2$ processors. Algorithm *Br_Lin* can thus be viewed as consisting of $\lceil \log p \rceil$ iterations, with each iteration having pairs of processors communicating and possibly exchanging messages.

Consider Algorithms *Br_Lin* on a mesh architecture. When $p = 2^k$ and the mesh is square, the first $\log p/2$ iterations use only column links, while the remaining iterations use only row links. Whether the number of processors actively involved in the broadcasting process increases, depends on where the source processors are located. For example, when the input distribution consists of columns, the first $\log p/2$ iterations introduce no new sources. For meshes with an odd number of rows, new sources are introduced in the case of column distribution. In order to study the use of only column links or row links for arbitrary mesh sizes, we consider broadcasting algorithms which operate on one dimension at a time. Within each dimension (e.g., within a row or within a column), these algorithms invoke Algorithm *Br_Lin*. We describe two such algorithms which differ on how dimensions are selected.

In **Algorithm Br_xy_source** the maximum number of sources in the rows and columns determines in which order the dimensions are processed. Recall that the processors know the positions of the sources. Let max_r be the maximum number of sources in a row and max_c be the maximum number of sources in a column. If $max_r < max_c$, rows are selected first and Algorithm *Br_Lin* is invoked on the rows. Otherwise, the columns are selected first. A reason for choosing the dimensions in this order is the following. When the rows contain fewer source processors, invoking *Br_Lin* within the rows is likely to generate messages of smaller size to be broadcast within the columns (i.e., at the time when *Br_Lin* is invoked on the columns we expect the message sizes to be smaller). As an example, assume all sources are located in α columns and each such column contains r sources, where r is the number of rows of the mesh. Then, $max_r = \alpha$ and $max_c = r$. First broadcasting in the rows results in every processor containing α messages at the time the column broadcast starts. If we were to first broadcast within the columns, $c - \alpha$ columns would be empty columns at the time the row broadcast starts.

For the sake of comparison, we also consider broadcasting without consideration as to where the source processors are located, only considering the size of the dimensions. Assume the mesh consists of r rows and c columns. **Algorithm Br_xy_dim** selects the rows if $r \geq c$ and the columns if $r < c$.

3 Algorithms with Repositioning

We will show in Section 5 that the performance of s-to-p algorithms depends on where the source processors are positioned. For the Intel Paragon, we have observed that performance can differ by a factor of 2. Further, each algorithm has its own ideal as well as poor source distributions. To break the dependence on the position of the sources we next describe algorithms which reposition the sources and then invoke the s-to-p algorithm on an ideal input distribution. Observe that the repositioning of processors containing messages has been applied successfully to obtain good routing algorithms [14]. We consider two such algorithms, a repositioning and a partitioning algorithm.

A repositioning algorithm for s-to-p broadcasting is composed from a non-repositioning algorithm and an ideal input distribution for this algorithm on the selected machine. We consider three repositioning algorithms: **Repos_Lin**, **Repos_xy_source**, and **Br_xy_dim**. We use *Repos_Lin* in our explanation. The first step of Algorithm *Repos_Lin* generates an ideal distribution of s sources for *Br_Lin* on the given machine. This is achieved by performing a partial permutation in which each source processor sends its message to a processor determined by the ideal distribution. We refer to the next section for a discussion on ideal distributions. Whether it pays to perform the redistribution depends on the quality of the initial distribution of sources. Our current implementations do not check whether the initial distribution is close to an ideal distribution and always reposition.

Our second class of algorithms makes use of the observation that the time for broadcasting $s/2$ sources on a $p/2$-processor machine is often less than half of the time needed for broadcasting s sources on a p-processor machine. In addition to repositioning the sources, these algorithms also partition the processors and perform two independent broadcasting problems simultaneously. Assume we partition the p processors into a group G_1 consisting of p_1 processors and into a group G_2 consisting of p_2 processors. The partition of the processors into two groups is independent of the position of the sources. However, it may depend on the choice of the broadcasting algorithm invoked on each processor group. The repositioning of the sources is done so that

- group G_1 contains s_1 sources, group G_2 contains s_2 sources, and $\frac{p_1}{p_2} = \frac{s_1}{s_2}$, and
- the new source distribution in G_1 (resp. G_2) is an ideal one for the broadcasting algorithm invoked in G_1 (resp. G_2).

After the broadcasting within G_1 and G_2 has been completed, every processor in G_1 (resp. G_2) exchanges its data with an assigned processor in G_2 (resp. G_1). This communication step corresponds to a permutation between the processors in G_1 and G_2. Depending on what s-to-p algorithm is used within the groups, we consider three partitioning algorithms: *Part_Lin*, *Part_xy_source*, and *Part_xy_dim*.

4 Source Distributions

In this section we discuss different source distribution patterns used in our experiments. Some of these distributions exploit the strengths while other highlight the weaknesses of the proposed algorithms. Some are chosen because we expect them to be difficult distributions for all algorithms. To define the distributions, assume a mesh of size $p = r \times c$ with $r \leq c$ and that processors are indexed in row-major order. Let $i = \lceil \frac{s}{c} \rceil$.

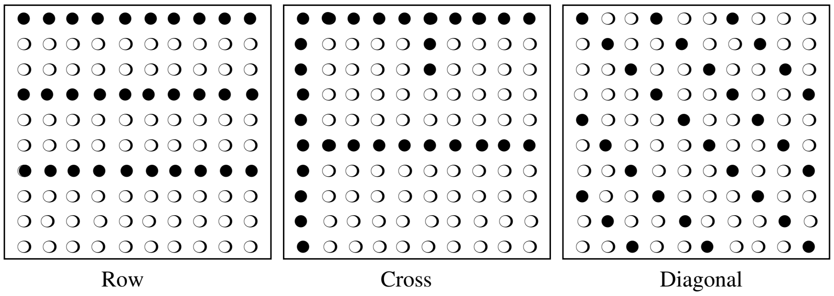


Figure 1: Placement of 30 sources in row, cross, and right diagonal distributions.

- *Row and Column Distributions:* $R(s)$ and $C(s)$
  In $R(s)$, i evenly spaced rows contain source processors. Every row, with the exception of the last one, contains c source processors. For a 10×10 mesh, $R(30)$ has the source processors positioned as shown in Figure 1. $C(s)$ is defined analogously for the columns.
- *Equal Distribution*: $E(s)$
  Processor $(1, 1)$ is a source processor and every $\lceil p/s \rceil$-th or $\lfloor p/s \rfloor$-th processor is a source processor. For particular values of s, r, and c, $E(s)$ can turn into a row, column, or diagonal distribution, or exhibit a rather irregular position of sources.

- *Right and Left Diagonal Distributions*: $Dr(s)$ and $Dl(s)$
  $Dr(s)$ has the s source processors positioned on i diagonals. We include the diagonal from $(1,1)$ to (r,r). The remaining $i-1$ diagonals are spaced evenly (assume wrap-around connections when placing sources) and the last diagonal not necessarily filled with sources. $Dl(s)$ has source processors on the diagonal from $(1,c)$ to $(r,1)$ and spaces the remaining $i-1$ diagonals accordingly.

- *Band Distribution*: $B(s)$
  $B(s)$ generalizes the right diagonal distribution which contains i diagonals, each having width 1. The band distribution $B(s)$ contains $b = \lceil \frac{c}{r} \rceil$ evenly distributed bands, each having width $\lceil \frac{s}{br} \rceil$.

- *Cross Distribution*: $Cr(s)$
  $Cr(s)$ corresponds to a union of a row and a column distribution, with the number of source processors in the row distribution being roughly equal to the number of processors in the column distribution. Figure 1 shows $Cr(30)$ for a 10×10 mesh. It consists of two equally spaced rows, each containing 10 sources, and two equally spaced columns. The second column contains only 4 sources.

- *Square Block Distribution*: $Sq(s)$
  In $Sq(s)$, the source processors are contained in a mesh of size $\lceil \sqrt{s} \rceil \times \lceil \sqrt{s} \rceil$. If not stated otherwise, we assume that processor $(1,1)$ is the top-left corner of the square. Within the square, the s sources are placed column by column.

Figure 1 shows three of the above distributions for $s = 30$ on a 10×10 mesh. The remainder of this section describes how the algorithms handle different distributions on a mesh architecture.

For Algorithm *Br_xy_source* one expects row and column distributions to be ideal source distributions. Algorithm *Br_xy_source* will choose the first dimension so that the number of source processors is increased as fast as possible, while the message length increases as slowly as possible. However, not all row and column distributions are equally good. For example, in $R(20)$ on a mesh of size 10×10, the first and the sixth row contain the source processors and thus the first iteration does not increase the number of source processors. Having 20 sources positioned in the first and the seventh row eliminates this. This is an important observation for the algorithms generating ideal distributions. It shows that the machine dimension effects the ideal distribution of sources. The diagonal distribution places the same number of sources in each row and column. One would expect Algorithm *Br_xy_source* to perform well on diagonal distributions. The performance of Algorithm *Br_xy_source* on the equal distribution will vary.

	algorithm dependent			distribution dependent	
	congestion	wait	# send/rec	av_msg_lgth	av_act_proc
2-step	$O(s)$	$O(1)$	O(p)	$O(sL)$	$O(\frac{p}{\log p})$
PersAlltoAll	$O(1)$	$O(1)$	O(p)	$O(L)$	$O(p)$
Br_Lin, $s = 2^l$	$O(1)$	$O(\log p)$	$O(\log p)$	$O(sL)$	$O(\frac{p}{\log p} + \frac{s \log s}{\log p})$
Br_Lin, $s \neq 2^l$	$O(1)$	$O(\log p)$	$O(\log p)$	$O(\frac{sL}{\log p})$	$O(\frac{p}{\log p} \log s)$

Figure 2: Comparing algorithm and data distribution characteristics for the equal distribution

Cross, square block, and band distributions should be considerably more expensive since the source positions may not allow a fast increase in the number of sources.

The behavior of Algorithm *Br_Lin* on the distributions is different. First, neither row or column distribution are ideal distributions for *Br_Lin*. For the column distribution and certain machine sizes, a fraction of the iterations may not increase the number of sources. When the number of rows is a power of 2, *Br_Lin* on the row distribution behaves like *Br_xy_source*. Since, depending on the ratio of s and p, the equal distribution can turn into a row or a column distribution, we do not consider it ideal either. The behavior of *Br_Lin* on the left and the right diagonal distribution can differ (no such difference exists for *Br_xy_source*). On a machine of size 10×10, $Dr(10)$ experiences no increase in the number of sources in the first iteration. For other machine sizes, the right diagonal distribution may not experience such a disadvantage. The left diagonal distribution is least sensitive towards the size of the machine and it is one of the ideal distributions for *Br_Lin*. The remaining distributions appear to be difficult.

We conclude this section by contrasting algorithm and distribution dependent parameters for selected algorithms. Figure 2 shows five paramters for Algorithms *2-Step*, *PersAlltoAll*, and *Br_Lin* on the equal distribution. We assume $p = 2^k$ and that every source processor contains a message of size L. For Algorithm *Br_Lin* we distinguish whether s is a power of 2 or not. The three algorithm dependent parameters are *congestion*, which measures the maximum number of sends and receives a processor handles in one iteration, *wait*, which bounds the number of times a processor waits for data before proceeding with the next send operation, and *#send/rec* which measures the total number of send and receive operations per processor during the entire algorithm. For *2-Step* and *PersAlltoAll* the values are given for implementations in terms of send/receive operations, with *2-Step* consisting of $\log p$ and *PersAlltoAll* consisting of p iterations, respectively. The two distribution dependent parameters are *av_msg_lgth* and *av_act_proc*. Parameter *av_msg_lgth* bounds the maximum average length of the messages sent and received by a processor over all iterations. More precisely, if the length of the messages for a processor is $l_1, l_2, \ldots, l_t$ over t iterations, then $(\sum_{i=1}^{t} l_i)/t \leq av\_msg\_lgth \leq sL$. Parameter *av_act_proc* measures the average number of active processors in the machine over all iterations (it is bounded by p).

The table shows a number of relationships influencing performance and which are discussed in more detail in the next section. Within the algorithm dependent parameters, *2-Step* and *PersAlltoAll* employ a low wait cost at the expense of a large number of total send and receives. Algorithm *Br_Lin* keeps *wait* and *#send/rec* balanced (in *Br_Lin* every iteration experiences a wait cost). For algorithms consisting of $\log p$ iterations, the ideal average number of active processors, ignoring the effect of s, is $\frac{p}{\log p}$. This is achieved by *2-Step*, but the other parameters are relatively high. For *Br_Lin* and the equal distribution, the distribution dependent parameters depend on whether s is a power of 2. For $s = 2^l$, the first $l/2$ iterations do not increase the number of active processors, they only increase the message length at the s source processors. For $s \neq 2^l$, the number of active processors increases faster and the message length grows slower. Our experimental results will show that the behavior for $s = 2^l$ occurs for other distributions and algorithms and generally results in poor performance.

5 Experimental Results

In this section we report performance results for the s-to-p broadcasting algorithms on the Intel Paragon and Cray T3D. We consider machine sizes from 4 to 256 processors and message sizes from 32 bytes to 16K bytes. We study the performance for source numbers ranging from 1 to p and the source distributions described in Section 4.

The Paragon has an underlying mesh architecture and applications can execute on submeshes of a specified dimension. The T3D's interconnection network forms a 3-dimensional mesh and applications execute on a specified number of virtual processors whose mapping to physical processors processors cannot be controlled by the user. The T3D has a larger communication bandwidth: every interconnect node has six outgoing channels which are able to simultaneously support hardware transfer rates of 300 MB/s. The Intel Paragon has 5 channels per interconnect node with 200 MB/sec per channel.

The results reported for the Paragon were obtained using the NX communication library, with the exception of two MPI implementations shown in Figure 3. We have compiled and run all algorithms on the Paragon under MPI environment. We have observed a performance loss of 2 to 5% in every MPI implementation. The results reported for the Cray T3D were obtained using MPI. The performance results reported have been obtained over multiple runs and are averaged over four best runs. Most implementation issues follow in a straightforward way from the descriptions given in the previous sections. We point out that we avoid global synchronization in our algorithms and use data parallelism to synchronize between steps and iterations.

In this paper we report only the performance for the case when all source processors broadcast messages of the same length. In our experiments, using different length messages did not influence the performance of the algorithms significantly. In particular, for a given algorithm, a good distribution remains a good distribution when the length of messages varies. Throughout this section, we use L to denote the size of the messages at source processors.

5.1 Algorithms without Repositioning on the Paragon

In this section we discuss the scalability of the five algorithms described in Section 2 on the Intel Paragon. We first consider standard scalability parameters such as machine size, number of source processors, and message length. We then consider other relevant parameters, including the distribution of the source processors, the dimension of the machine, and the interaction of the dimension of the machine and the source processor distribution with respect to a particular algorithm. We show how these parameters impact performance.

The communication operations invoked in Algorithms *2-Step* and *PersAlltoAll* use the implementations described in [8]. In particular, the personalized all-to-all operations makes message exchanges consisting of p permutations and uses the exclusive-or operation on processor indices to generate the permutations. Algorithm *2-Step* uses an one-to-all implementation which views the mesh as a linear array and applies the same communication pattern used in Algorithm *Br_Lin*; i.e., processor P_i exchanges a message with $P_{i+p/2}$ and then the one-to-all communication is performed within each machine half. We did not expect Algorithms *PersAlltoAll* and *2-Step* to give good performance on the Paragon. *PersAlltoAll* exchanges many messages and Algorithm *2-Step* creates significant communication bottlenecks. However, we did want to see

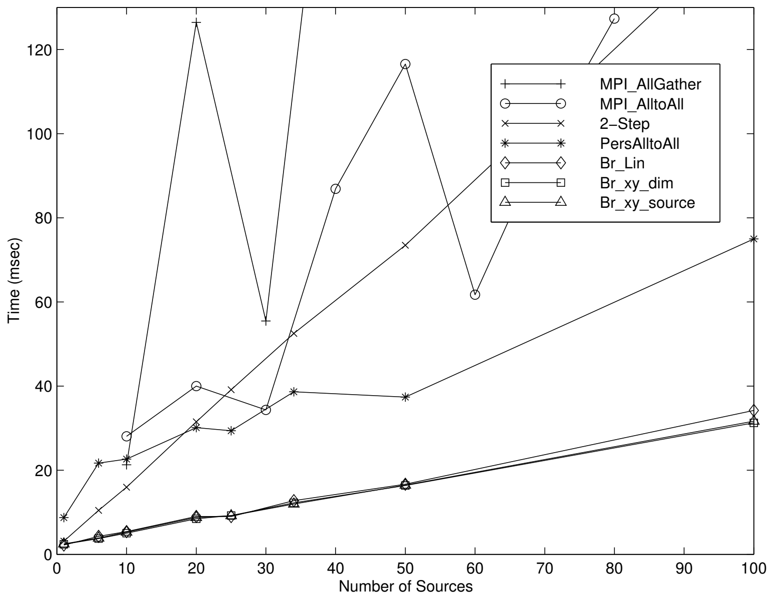


Figure 3: Performance of algorithms on a 10×10 Paragon; number of sources varies from 1 to 100, $L = 4K$, equal distribution.

their performance against the other proposed algorithms to show the potential disadvantage of using existing communication routines.

Figure 3 shows the performance of the different broadcasting algorithms on a 10×10 Paragon when the number of sources varies. For comparison sake, we also include the MPI version of Algorithms *2-Step* and *PersAlltoAll*: Algorithms MPI_AllGather and MPI_Alltoall. The three curves giving the best (and almost identical) performance correspond to Algorithms *Br_Lin*, *Br_xy_dim* and *Br_xy_source*. Algorithms *2-Step* and *PersAlltoAll* give poor performance and the MPI versions perform worse than the NX versions. The reason for the poor performance of these four algorithms lies in the communication patterns and the communication bandwidth. Algorithm *2-Step* suffers congestion at the node which first gathers all the messages. Algorithm *PersAlltoAll* suffers because of the large number of sends issued by the source processors. Both of these shortcomings are even more pronounced in the MPI versions. The performance of the other three algorithms, *Br_Lin*, *Br_xy_source*, and *Br_xy_dim* scales linearly with the increase in number of sources. Depending on the number of sources and how the equal distribution places sources in the machine, performances differ slightly.

Figure 4 shows the performance for a right diagonal distribution with $s = 30$ when the

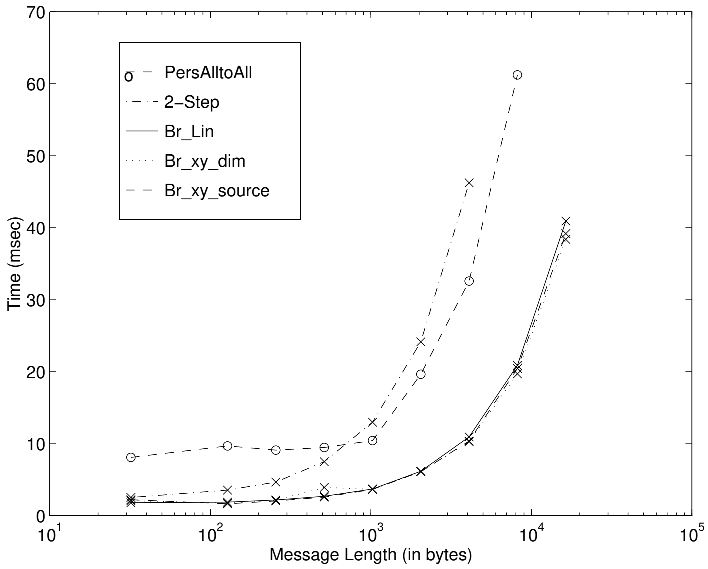


Figure 4: Performance of algorithms on a 10×10 Paragon; L varies from 32 bytes to 16K, s =30, right diagonal distribution.

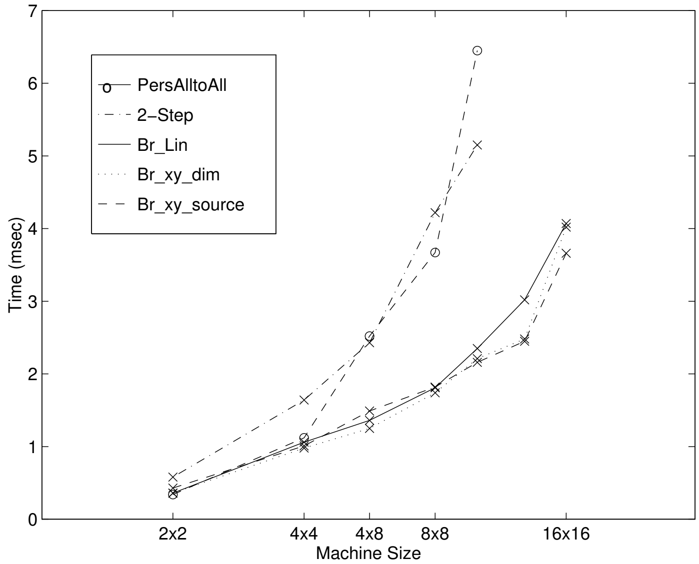


Figure 5: Performance of algorithms on a Paragon: L = 1K, approximately $\sqrt{p}$ sources, right diagonal distribution.

message size changes. As already stated, the diagonal distributions place the same number of sources in the rows and columns. Once again, regardless of how small a message size, Algorithms *2-Step* and *PersAlltoAll* perform poorly. The almost flat curve up to a message size of 1K for Algorithm *PersAlltoAll* further supports our observation related to Figure 3. The other three algorithms experience little increase in the time until $L = 512$ bytes and then we see a linear increase for larger message lengths.

Figure 5 shows the behavior of the five algorithms when the machine size varies from 4 to 256 processor. Algorithm *PersAlltoAll* is as good as any other algorithm for small machine sizes (4 to 16 processors). This feature is also observed when the number of sources is close to p for small machine sizes.

The first three figures give the impression that algorithms *Br_Lin*, *Br_xy_source*, and *Br_xy_dim* give the same performance. However, this is not true. In the following we show that different distributions and different machine sizes effect these algorithms in different ways.

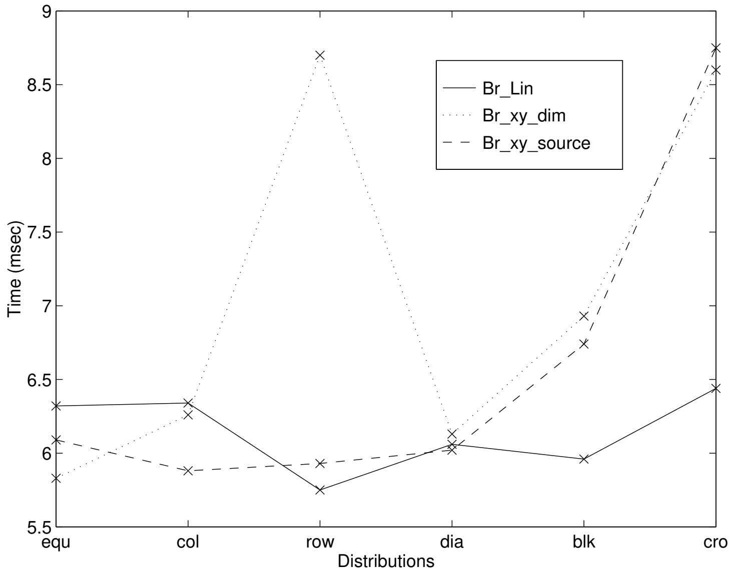


Figure 6: Performance of three algorithms on a 10x10 Paragon; $L = 2K$, different source distributions with $s = 30$.

Figure 6 shows the performance for $s = 30$ while using different distribution patterns. The figure confirms the discussion given in Section 4 with respect to ideal and difficult distributions. Algorithm *Br_xy_source* gives roughly the same performance on the first 4 distributions, but for the square block and cross distribution we see a considerable increase in time. We point out that the same performance on the first 4 distributions for *Br_xy_source* is not true in general. However, the row and the column distribution show up as ideal distributions. Square block and cross distributions require more time for all three algorithms. As expected, Algorithm *Br_Lin* performs best on them. This is due to the fact that in Algorithm *Br_Lin* sources can spread to different rows and columns in the first few iterations, thus utilizing the links more efficiently. On the other hand, for the square block distribution, Algorithms *Br_xy_source, Br_xy_dim* have only few columns and rows available to generate new sources. The big increase in Algorithm *Br_xy_dim* for the row distribution indicates the importance of choosing the right dimension first.

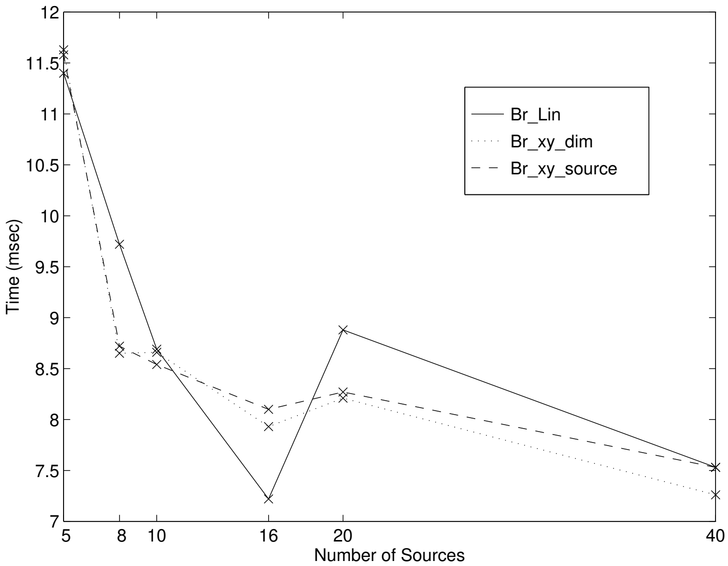


Figure 7: Performance of three algorithms on a 10x10 Paragon; right diagonal distribution, total message size is kept at 80K, the number of sources varies.

Figure 7 shows the performance of the three algorithms when the total message size (i.e., the sum of the message sizes in the source processors) is fixed. A goal for our algorithms is to increase the number of processors engaged in the broadcasting as fast as possible and to increase the message length as slowly as possible. Figure 7 indicates that this strategy works well on the Paragon: if the data is spread among a larger number of sources, the broadcast is faster. For example, for a total message size of 80K, data spread among 5 sources takes approximately 11.4 msec using Algorithm *Br_xy_source.* However, the same amount of data when spread among 40 sources takes only 7.3 msec.

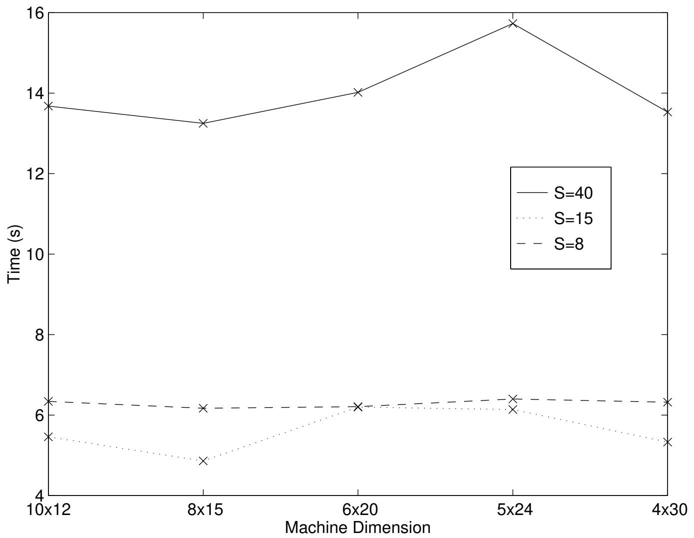


Figure 8: Performance of Algorithm *Br_Lin* on a 120-node Paragon; dimensions vary, equal distribution, $L = 4K$ three source sizes are shown.

Figure 8 shows the performance of three algorithms for $p = 120$ when the dimensions of the machine vary. It demonstrates that performance is related to the size the dimensions. For the same number of sources, message size, and number of processors, a distribution gives different performance (hence is considered good or bad) depending on the dimension of machine. For a small number of sources (for example $s = 8$) the machine dimensions may not affect the performance. For a large number of sources, machine dimensions impact the performance considerably more. It seems like an anomaly to have faster performance for $s = 15$ than for $s = 8$. The reason lies in the distribution and the number of rows. When $s = 8$, the equal distribution tends to place the source processors within columns. This does not allow a fast increase in the number of sources. On the other hand, for $s = 15$, the source processors are, with the exception of size 4×30, positioned along diagonals.

5.2 Algorithms with Repositioning on the Paragon

Algorithms *Br_xy_source* and *Br_Lin* exhibit good performance for a variety of source distributions and machine dimensions. However, each algorithm has source distributions which exhibit algorithm and machine weaknesses. The problems arising from the source distribution can be avoided by performing a repositioning of the sources. In Section 3 we described a repositioning and a partitioning approach. We next discuss the performance of the repositioning approach using Algorithm *Br_xy_source*. We use the row distribution as one of the ideal source distributions for *Br_xy_source*. Similar results hold for the repositioning algorithm using *Br_Lin* with the left diagonal distribution as an ideal source distribution.

Let Algorithm *Repos_xy_source* be the repositioning algorithm invoking *Br_xy_source*. In this algorithm we first perform a permutation to redistribute source processors according to the row distribution. The row distribution we generate positions the rows so that the number of new sources increases as fast as possible. The exact position of the rows depends on the number of rows of the mesh. The cost of the permutation achieving the repositioning depends on s, where the s source processors are located, and the message length.

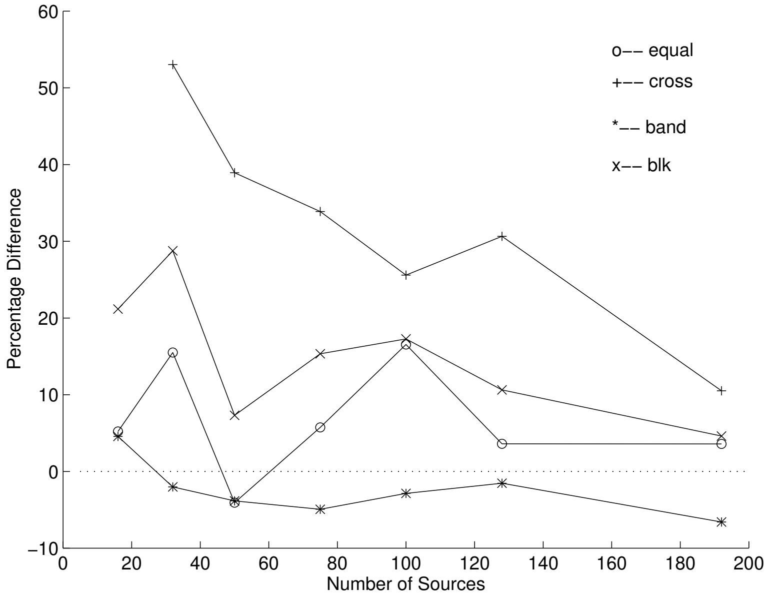


Figure 9: The difference between *Repos_xy_source* and *Br_xy_source* on a 16×16 Paragon; different input distributions, $L = 6K$, varying the number of sources.

Figure 9 shows the percentage difference between Algorithms *Repos_xy_source* and *Br_xy_source* on four input distributions when the number of sources increases from 16 to 192. The behavior shown is representative for other machine and message sizes and can be summarized as follows:

- Repositioning results in a significant gain for distributions like the cross and square block distributions. In terms of actual time, the gain for repositioning on the cross distribution lies between 13 and 31 msec. A gain of 13 msec is observed when $s = 192$, while for all other source numbers the gain lies between 20 and 31 msec.
- The somewhat erratic behavior for the equal distribution can be explained from the position of the sources in a 16×16 mesh. For example, for $s = 50$, the generated

distribution is similar to a row distribution and thus Algorithm *Repos_xy_source* costs more than *Br_xy_source*.

- Repositioning for the band distribution costs up to 6.5% more. Translating this into actual time, when s is less than 150, *Repos_xy_source* costs between 1 and 2 msec more. For $s = 192$, repositioning costs 7.5 msec more. The band distribution for a 16×16 mesh consists of a single diagonal band of width $s/16$. The communication characteristics for the band distribution are thus similar to those of an ideal distribution. Not surprisingly, nothing is gained by repositioning.

Our overall observations are that the gain of repositioning tapers off for any distribution when the number of sources gets large. When repositioning is done on an ideal or almost ideal distribution, we observed an increase of 1-2 msec when the message size is $< 16K$ and $s \leq p/2$. Clearly, if the input distribution is close to an ideal distribution, it does not pay to reposition. We point out that our algorithms do not analyze the input distribution.

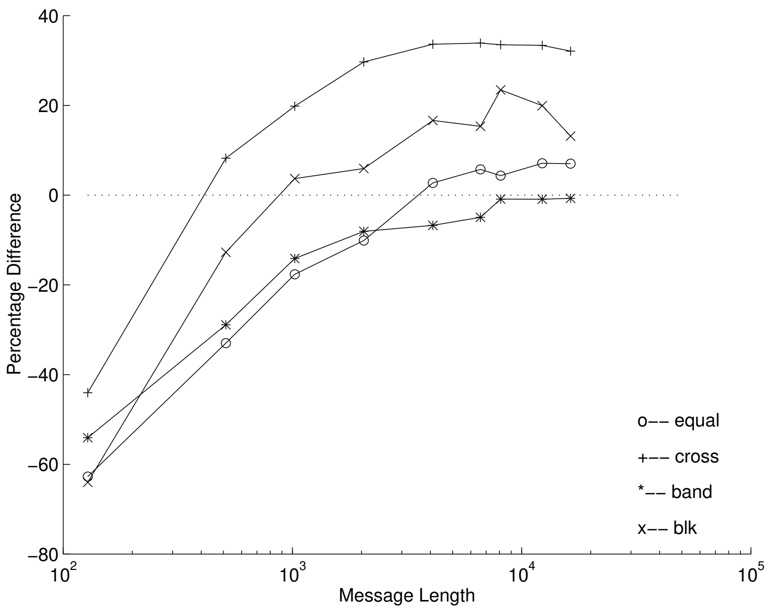


Figure 10: *Repos_xy_source* and *Br_xy_source* on a 16×16 Paragon; different input distributions, $s = 75$, varying the message length.

The effect of the message length on the repositioning is illustrated in Figure 10. The figure shows the percentage difference for the same four distributions on a 16×16 machine and 75 sources when the message length increases. For a message size of less than $1K$, repositioning pays only for the cross distribution. As the message size increases, the benefit of repositioning increases for all distributions. Not surprisingly, the gain tapers off for large message lengths.

In Section 3 we have proposed to combine the repositioning with a partitioning approach. We first generate an ideal source distribution and then create two broadcasting problems, each on one half of the machine. Let Algorithm *Part_xy_source* be such a partitioning algorithm using *Br_xy_source* within each machine half. We have compared *Part_xy_source* against the performance of *Repos_xy_source* and *Br_xy_source*. Our results showed that for the Intel Paragon the partitioning approach hardly ever gives a better performance than repositioning alone. The reason lies in the cost of the final permutation. The exchange of large messages done in the final step dominates the performance and eliminates the gain obtained from broadcasting on smaller machines.

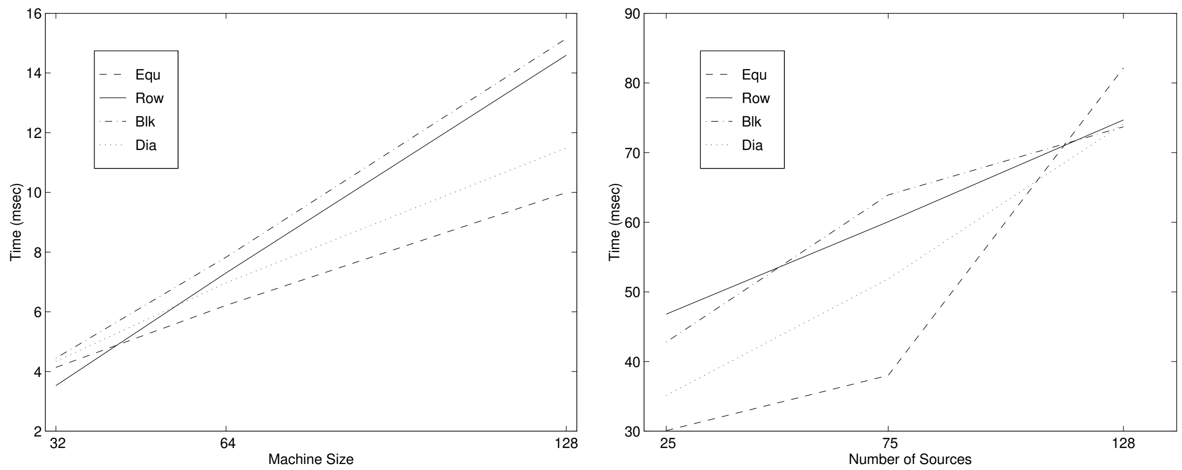
The conclusion of our experimental study on the Paragon is that repositioning pays more often than not. If the following three conditions hold, one can expect the repositioning algorithm to give a better and more predictable performance:

1. The number of sources processors is moderate; $s < p/2$ appears to be the breakpoint.
2. The number of processors is not too small. For $p \leq 16$, there is little difference between the algorithms and different source distributions.
3. The message length is at least $1K$ and at most $16K$.

When all three conditions are satisfied and the initial distribution of sources is close to an ideal distribution, the cost of repositioning is a small fraction of the overall broadcasting cost. When performing a repositioning as recommended above we observed a cost of 1-2 msec for generating an ideal distribution. Since the gain obtained from repositioning is significant for more difficult patterns, our results suggest that Algorithm *Repos_xy_source* should be used on the Paragon.

5.3 Performance on the T3D

In this section we present performance results on the Cray T3D for three of the s-to-p algorithms discussed in Section 2. The algorithms are MPI_AllGather, an MPI version of *2-Step*, MPI_AlltoAll, an MPI version of *PersAlltoAll*, and Algorithm *Br_Lin* implemented using MPI send and receive primitives. Since only production level access to the T3D was available, our implementations had no control over the mapping of virtual to physical processors. For this reason we did not consider the s-to-p algorithms whose performance is sensitive to the topology of the underlying communication network. We first present scalability results for Algorithm MPI_AllGather and then compare and contrast these results with those for the other two algorithms.



(a) (b)

Figure 11: Performance of MPI_AllGather on the T3D for different distributions; a) varying machine size while $s = 32$ while total length of all the messages $= 128K$, b) varying problem size, $p = 128$, $L = 16K$.

Figure 11 shows the scalability of the MPI_AllGather with respect to machine size and problem size assuming different source distributions. It shows that for small machine sizes the distribution of the source processors has little impact on the performance. For larger machine sizes, the equal distribution consistently performed better. For the data shown in Figure 11(a), the equal distribution takes 28% less time than the other distributions. Similarly, in 11(b), the equal distribution outperforms the other distributions. The convergence and deterioration of MPI_AllGather when s approaches p is as expected.

Figure 12 shows the performance of MPI_AllGather on different distributions when the problem size and machine size are fixed and the number of source processors varies. The data shown supports our earlier claim: for a given problem size, better performance is obtained when the broadcast data is initially distributed over a large number of source processors. The type of distribution has significant impact on the performance when $s \leq p/4$. The figure demonstrates again that the equal distribution tends to give better performance. The computing environment used to obtain results on the T3D does not allow us to conclude that the equal distribution is an ideal distribution for MPI_AllGather. Our results only indicate that the distribution of the

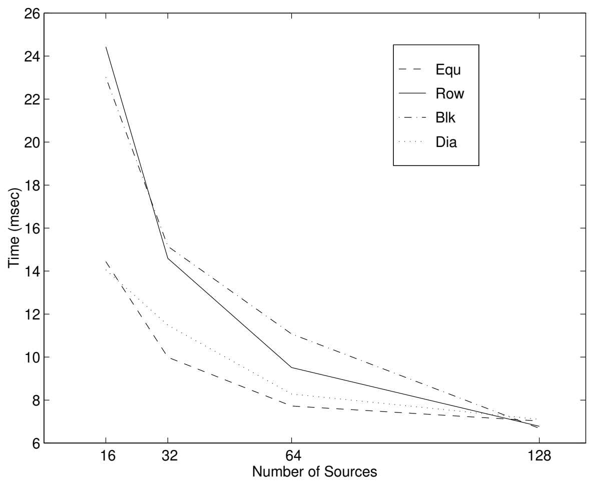
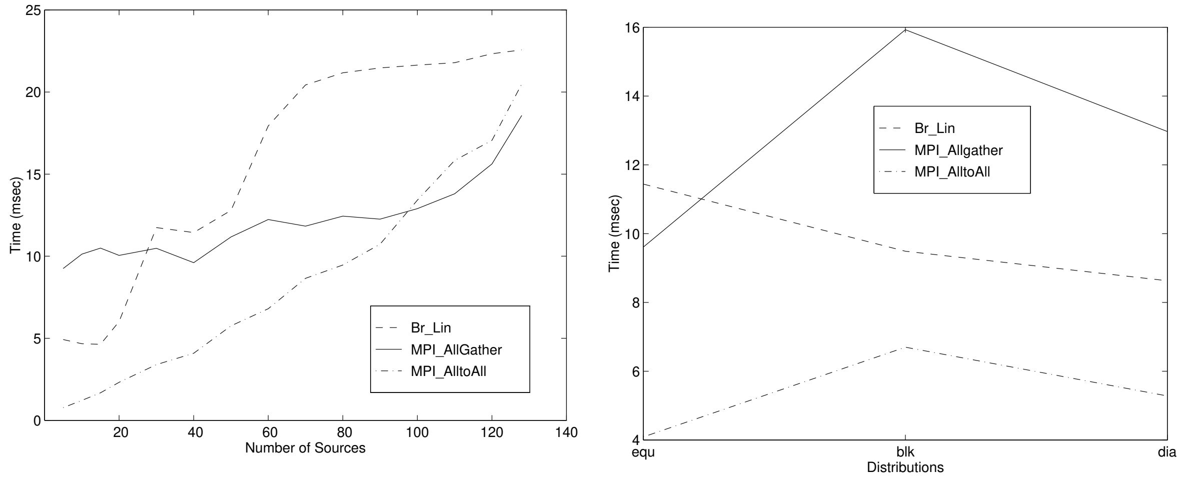


Figure 12: Performance of the MPI_AllGather on a 128-processor T3D assuming total length of all the messages fixed at $128K$ and the number of sources varies.

sources has an impact on the performance.



(a) (b)

Figure 13: Performance of three algorithms on a 128 processor T3D for $L = 4K$; (a) sources vary from 5 to 128, equal distribution; (b) different source distributions with $s = 40$.

Figure 13 compares the performance of the MPI_AllGather, MPI_AlltoAll, and *Br_Lin*. In 13(a), the number of sources varies. Contradictory to our results on the Paragon, MPI_AlltoAll gives the best performance on the T3D. This is primarily due to the large communication bandwidth available on T3D combined with the fact that MPI_AlltoAll does not combine messages and does not need to wait for messages. Using the parameters introduced in Figure 2, MPI_AlltoAll has a small *wait* cost. While MPI_AllGather has an equally small *wait* cost, the congestion arising at processor P_0 can be associated with the higher cost compared to MPI_AlltoAll. As the number of sources increases, the performance of MPI_AllGather and MPI_AlltoAll converges. The reason for the poor performance of Algorithm *Br_Lin* lies primarily in the higher *wait* cost and the cost of combining messages (and the resulting delay for starting the next iteration).

Figure 13(b) shows the performance of the three algorithms under different source distributions. MPI_AlltoAll performs well for all distribution patterns. For all three algorithms the performance varies for the different source distributions. For T3D we cannot identify ideal distributions for the algorithms considered. We conjecture that the equal distribution performs well for MPI_AlltoAll and MPI_AllGather since the placement of source processors tends to be uniform and may resemble a uniformly random distribution. Therefore, a random distribution appears to be a good choice for the T3D. However, generating a random distribution and communicating such a distribution to all processors may entail more overhead than what was needed in the repositioning algorithms on the Paragon. From the results obtained for the T3D we conclude that an algorithm which minimizes *wait* cost and thus tends to make good use of the large communication bandwidth can be expected to give a good performance.

6 Conclusions

We have described different s-to-p broadcasting algorithms and analyzed their scalability and performance on the Intel Paragon and Cray T3D. We showed that the performance of each algorithm is influenced by the distribution of the source processors as wells as by the relationship between the number of sources and machine size. Each algorithm has ideal distributions and distributions on which the performance degrades. Our work shows that the scalability of s-to-p implementations on the T3D is influenced by the same parameters that influence the scalability on the Paragon. Due to the larger communication bandwidth, the differences are not quite as striking on the T3D. While the use of standard communication primitives results in very poor performance for the Paragon, it gives a good performance for the T3D. On the Paragon, the best performance is achieved by using the repositioning approach which reduces the dependence of the input distribution on the performance.

7 Acknowledgments

We thank Shaogang Chen, Jianwei Xu, and Michael Becht for their programming help on the Intel Paragon and Cray T3D. We would also like to thank Pittsburgh Supercomputing Center for providing access to the 512 node Cray T3D through an educational grant.

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *Proc. of 8-th IPPS*, pp. 835-844, 1994.

[2] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts, "Interprocessor Collective Communication Library," *Proc. of Scalable High-Performance Computing Conference*, pp. 357-364, 1994.

[3] S.H. Bokhari, "Multiphase Complete Exchange on a Circuit Switched Hypercube," *Proc. of ICPP*, pp. 525-529, 1991.

[4] J. Bruck, L. de Coster, N. Dewulf, C.-T. Ho, and R. Lauwereins, "On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model," *Proc. of Intl. Symp. on Parallel and Dist. Processing*, pp. 594-602, 1994.

[5] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[6] J.J. Dongarra, R. Hempe, A.J.G. Hey, D.W. Walker, A Proposal for a User-level, Message Passing Interface in a Distributed Memory Environment, Technical Report, TM 12231, Oak Rigde Laboratory, 1993.

[7] W. Gropp, E. Lusk, The MPI Communication Library: Its Design and a Portable Implementation, *Proc. of Scalable Parallel Libraries Conference*, pp 160-165, 1993.

[8] S. E. Hambrusch and F. Hameed and A. A. Khokhar, "Communication Operations on Coarse-Grained Mesh Architectures," *Parallel Computing*, Vol. 21, pp. 731-751, 1995.

[9] S. E. Hambrusch and A. Khokhar. "Maintaining spatial data sets in distributed-memory machines," *Proc. of 11-th International Parallel Processing Symposium*, April 1997.

[10] S. Hinrich, C. Kosak, D. O'Halloron, T. Stricker, R. Take, "An Architecture for Optimal All-to-All Personalized Communication," *Proc. of 6-th ACM SPAA*, pp. 310-319, 1994.

[11] R. Karp, A. Sahay, E. Santos, K. Schauser, "Optimal Broadcast and Summation in the LogP Model," *Proc. of 5-th ACM SPAA*, pp. 142-153, 1993.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings Publishing, 1994.

[13] Y. Lan, A. H Esfhanian, and L. M. Ni, "Multicast in Hypercube Multiprocessors," *JPDC*, vol. 8, pp. 30-41, 1990.

[14] F.T. Leighton, *Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, 1992.

[15] X. Lin, P. K. McKinley, and L. M. Ni, "Performance Evaluation of Multicast Wormhole Routing in 2D-Mesh Multicomputers," *Proc. of ICPP*, pp. 1435-1442, 1991.

[16] S.L. Johnsson, C.-T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercubes," *IEEE Trans. on Comp.*, Vol. 38, pp. 1249-1268, 1989.

[17] J.Y. Park, H.-A. Choi, N. Nupairoj and L.M. Ni, "Construction of Optimal Multicast Trees Based on the Parameterized Communication Model", *Proc. of 1996 Internat. Conference on Parallel Processing*, Vol. I, pp 180 – 187, 1996.

[18] S. Ranka, R. Shankar and K. Alsabti, "Many-to-Many Personalized Communication With Bounded Traffic," *Proc. of Symp. on the Frontiers of Massively Parallel Computation*, 1995.

[19] S. Ranka, J. C. Wang, and G. C. Fox, "Static and Runtime Scheduling of All-to-Many Personalized Communication on Permutation Networks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, Nr. 11, 1994.

[20] R. Thakur, A. Choudhary, "All-to-all Communication on Meshes with Wormhole Routing," *Proc. of 8-th IPPS*, pp. 561-565, 1994.

[21] E.A. Varvarigos, D. Bertsekas, "Dynamic Broadcasting in Parallel Computing," *IEEE Trans. on Parallel and Distributed Systems*, pp 120-131, Vol. 6, Nr. 2, 1995.