1995

S-to-P Broadcasting on Coarse-Grained Machines

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Ashfaq A. Khokhar

Yi Liu

Report Number:
95-060

S-TO-P BROADCASTING ON COARSE-GRAINED MACHINES

Susanne E. Hambrusch
Ashfaq A. Khokhar
Yi Liu

CSD-TR-95-060
September 1995

S-to-P Broadcasting on Coarse-Grained Machines *

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
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
liuy@cs.purdue.edu

September 22, 1995

*Research supported in part by ARPA 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.

1 Introduction

The broadcasting of messages is a basic communication operation on coarse-grained, message passing massively parallel processors (MPPs). In the standard broadcast operation, one processor broadcasts a message to every other processor. Various implementations of this operation for architectures with different machine characteristics have been proposed [5, 9, 12, 13, 14]. Another well-studied broadcasting operation is the all-to-all broadcast in which every processor broadcasts a message to every other processor [3, 7, 8, 15]. Let p be the number of processors. Assume that s of the p processors, which we call *source processors*, contain a message to be broadcast to every other processor, $1 \leq s \leq p$. In this paper we present broadcasting algorithms that handle all ranges of s. We report experimental results for s-to-p broadcasting algorithms on the Intel Paragon and discuss their scalability and performance.

In general, quantities influencing scalability, and thus the choice of which algorithm gives the best performance, include the number of processors, the message sizes, and the number of source processors [10]. Our algorithms are scalable with respect to p, s, and the message sizes; i.e., they maintain their speedup as these parameters change. For s-to-p broadcasting, other factors influence scalability as well. For any fixed s, a particular algorithm exhibits a different behavior depending where the s source processors are located. Each algorithm has ideal distribution patterns and distribution patterns giving poor performance. Poor distribution patterns for one algorithm can be ideal for another. Thus, the location of the source processors and the relationship of these locations to the size and dimensions of the architecture effect the scalability of an algorithm. In order to study these relationships to the fullest extent, we assume that every processor knows the position of the source processors and the size of the messages. This implies synchronization occurs before the broadcasting.

In this paper we describe a number of different broadcasting algorithms and investigate for each one its good or bad distribution patterns. We characterize features of s-to-p algorithms that perform well on a wide variety of source distributions. Some of our algorithms are tailored towards meshes, others are based on architecture-independent approaches. We show that algorithms that

- increase the number of processors actively involved in the broadcasting process as fast as

possible and

- increase the message length at processors as slowly as possible

give the best performance. For many algorithms, keeping the message size small implies a fast increase in the number of processors. However, we show that achieving these two goals can be 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. The performance obtained on ideal distributions can vary greatly from that obtained on poor distributions. We have observed a difference by almost a factor of 2. We conjecture that this holds in general, not just the algorithms we considered. We propose the approach of repositioning sources to guarantee a good performance. 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.

The paper is organized as follows. In Section 2 we describe the algorithms that do not reposition their sources. In Section 3 we discuss different repositioning approaches. Section 4 describes the different source distributions we considered and in Section 5 we discuss performance and scalability of the proposed algorithms. Section 6 outlines the conclusions of this work.

2 Algorithms without Repositioning

In this section we describe s-to-p broadcasting algorithms which do not reposition the sources. Our first class of broadcast algorithms generalizes an efficient 1-to-p broadcasting approach. S-to-p broadcasting could be done by having each one of the s source processor initiate a 1-to-p broadcast. However, having the s broadcasting processes take place without interaction is inefficient. Our approach is to let each processor initiate a broadcast, but whenever messages from different sources meet at a processor, messages are combined. Further broadcasting steps proceed thus with larger messages. We use a Binomial heap broadcasting tree [6, 9] to guide the broadcasts.

In **Algorithm** *Br_Lin* we view the processors of the mesh as forming a linear array (by using a snake-like row-major indexing). The existence of a linear array is not required and the approach is architecture-independent. 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 it to the other one. Then, Algorithm *Br_Lin* proceeds recursively on the first $p/2$ and the last $p/2$ processors.

Algorithms *Br_Lin* behaves differently for different machine sizes. 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 during a single iteration for arbitrary mesh sizes, we introduce Algorithm *Br_xy*. In **Algorithm** *Br_xy*, we first select either rows or columns. Assume the rows were selected. We then view each row as a linear array and invoke Algorithm *Br_Lin* within each row. After this, we invoke Algorithm *Br_Lin* within each column.

We consider two versions of Algorithm *Br_xy* which differ on how dimensions are selected. In **Algorithm** *Br_xy_source*, the number of sources in the rows and columns determine the order of the dimensions. Recall that every processor knows the positions of the sources. Every processor determines max_r, the maximum number of sources in a row, and max_c, the maximum number of sources in a column. If $max_r < max_c$, the rows are selected 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 elements, the broadcasting done within the rows is likely to generate messages of smaller size to be broadcast within the columns. Assume sources are located in a few, say α columns. Then, $max_r = \alpha$ and $max_c = r$, where r is the number of rows of the mesh. First broadcasting in the rows results in every processors

containing α messages at the time the column broadcast starts.

For the sake of comparison, we also consider a version of Algorithm *Br_xy* which compares the dimensions and broadcasts first along longer dimension. 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 $r < c$.

In the algorithms described so far processors issue sends and receives to facilitate communication. We do not make use of existing communication operations generally available in communication libraries [1, 2, 7]. S-to-p broadcasting can easily be stated in terms of known communication operations. We considered two such approaches. The first one, **Algorithm** *Xor*, invokes an all-to-all personalized exchange communication. In an all-to-all personalized exchange, every processor sends a unique message to every other processor. A processor containing a source message to be broadcast thus creates $p-1$ copies of this message.

The second such approach results in **Algorithm** *2-Step*. This algorithm performs the broadcast by invoking two regular communication operations, one s-to-one followed by an one-to-all operation. In the s-to-one communication, processor P_0, receives the s messages from the source processors. P_0 combines the s messages and initiates an one-to-all broadcast.

3 Algorithms with Repositioning

On coarse-grained machines like the Paragon, sending relatively short messages is cheap compared to the cost of an entire s-to-p operation. At the same time, experimental results show that the performance of our s-to-p algorithms can differ by a factor up to 2 for the same number of sources, depending on where the sources are positioned. Each algorithm has its own ideal source distribution. In this section we consider the approach of repositioning the sources and then invoking an s-to-p algorithm on its ideal input distribution.

Algorithm *Repos* is invoked with one of the algorithms described in the previous section. For the sake of an example, assume it is Algorithm *Br_Lin*. The first step generates *Br_Lin*'s ideal input distribution for s sources on the given machine size and machine dimension. This is achieved by each source processor sending 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.

We point out that our current implementation of Algorithm *Repos* does not check whether the initial distribution is actually close enough to an ideal distribution. We simply perform the repositioning.

Our second class of repositioning algorithms not only repositions the sources, but also makes use of the observation that the time for broadcasting $s/2$ sources on a $p/2$-processor machine is less than half of the time for broadcasting s sources on a p-processor machine. 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, and may depend on the choice of the broadcasting algorithm invoked on each processor group. The repositioning of the sources is done so that

- processor group G_1 contains s_1 sources, processor 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 is completed, every processor in G_1 (resp. G_2) exchanges its data with a processor in G_2 (resp. G_1). This communication step corresponds to a permutation between the processors in G_1 and G_2.

We refer to **Algorithm** *Part_Lin* as the algorithm based on this principle and using Algorithm *Br_Lin* within the submachines. We refer to **Algorithm** *Part_xy_source* as the algorithm based on this principle and using Algorithm *Br_xy-source* within the submachines.

4 Source Distributions

In this section we discuss different distributions 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. Assume the machine is a mesh of size $p = r \times c$ with $r \leq c$.

- Row and Column Distributions.

  In row distribution R_i, i rows contain source processors. The rows are spaced evenly, with

row 1 being the first row. The number of source processors in two rows differs by at most 1. For $s = 30$ and $p = 10 \times 10$, distribution R_i has the source processors positioned on 3 rows, as shown in Figure 1. For $s = 27$, two rows contain 10 sources and one row contains 7 sources. Column distribution C_i is defined analogously.

- Right and Left Diagonal Distributions.

  In the right diagonal distribution DR_i, i diagonals contain source processors. We always include the diagonal from $(1,1)$ to (r,r), which we view as the 0-th diagonal. We space the remaining $i-1$ diagonals evenly from diagonal 0 to the right to diagonal $c-1$ and to the left to diagonal $-r$. Left diagonal distribution DL_i has the diagonal from $(1,c)$ to $(r,c-r)$ as the 0-th diagonal and spaces the remaining $i-1$ diagonals accordingly.

- Equal Distribution.

  In equal distribution E_i, we make every i-th processor of the mesh, when indexing the processors in row-major order, a source processor. Processor $(1,1)$ is the first source processor. Distribution E_i contains $\lceil p/i \rceil$ sources. The location of the sources varies depending on i, r, and c. For particular values of i, r, and c, E_i can turn into a row, column, or diagonal distribution, or exhibit a rather irregular position of sources.

- Cross Distribution.

  In cross distribution C_i, i rows and i columns contain source processors. Rows and columns containing sources are again spaced out evenly within the mesh.

- Block Distribution.

  In block distribution $B_{i,j}$, the source processors form a submesh of size $i \times j$. If not stated otherwise, we assume that processor $(0,0)$ is the top-left corner of the submesh.

Figure 1 shows three of the above distributions for $s = 30$. We briefly describe how the algorithms handle different distributions and point out the salient features of each distribution. The performance of the algorithms on these distributions is discussed in the next section.

Consider first Algorithm *Br_xy_source*. Clearly, both a row and a column distribution are ideal distributions. Algorithm *Br_xy_source* will choose the first dimension so that the number of

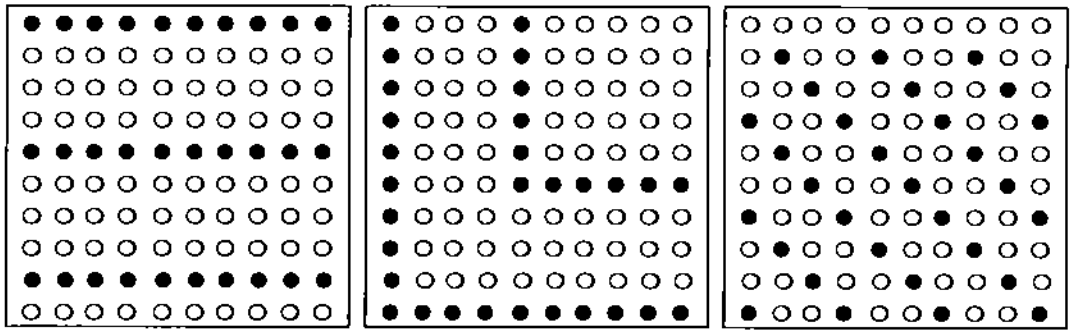

Figure 1: Placement of 30 source processors in row, cross, and diagonal distributions on a 10×10 machine.

source processors is increased as fast as possible while the message length increases as slowly as possible. However, not all distributions consisting of the same number of rows are equally good. For example, in R_2 on a mesh of size 10×10, the first iteration does not increase the number of source processors (since the first and the sixth row contain the sources). Having sources in the first and the seventh row eliminates this. The diagonal distribution places the same number of sources in each row and column. Since the sources are distributed evenly in rows and columns, one can expect Algorithm *Br_xy_source* to perform quite well. The performance of Algorithm *Br_xy_source* on the equal distribution will vary. The cross and the block distribution should be considerably more expensive than all the others on the same number of sources. The position of the sources is such that no fast increase in the number of sources takes place.

The behavior of Algorithm *Br_Lin* on the same set of input distributions is quite different. Given a particular distribution and number of sources, Algorithm *Bi_Lin* can exhibit all desired properties on one machine size, but can fail to do so for another machine size. The column distribution is clearly not an ideal distribution. For an even number of rows, an iteration achieves no increase in the number of sources. For an odd number of rows there can be an increase, depending on the number of columns. The performance of Algorithm *Br_Lin* on the row distribution should be identical to that of Algorithm *Br_xy_source* when the number of rows is a power of 2. For an odd number of rows, an iteration behaves differently. Communication occurs now between processors not in the same column and congestion will increase. Hence, neither row or column distribution are ideal distributions for Algorithm *Br_Lin*. The equal

distribution can turn into a row or a column distribution and will thus not be ideal either. The behavior on the left and the right diagonal distribution can differ (no such difference exists for Algorithm *Br_xy_source*). On a machine of size 10×10, DR_1 experiences no increase in the number of sources in the first iteration (since processor P_{50} lies on the 0-th diagonal). For other machine sizes, the right diagonal distribution may not experience such disadvantage. The left diagonal distribution is least sensitive towards the size of the machine and it achieves the desired properties of an efficient broadcasting algorithm. The block and the cross distribution will not be ideal distributions.

Finally, Algorithm *Br_xy_dim* suffers the obvious drawbacks when the selection of the dimension is done according to the size of the dimensions and not according to the number of sources. The ideal distribution for Algorithm *Br_xy_dim* will either be a row or a column distribution, depending on the dimensions.

5 Experimental Results

In this section we report performance results for the s-to-p broadcasting algorithms on the Intel Paragon. We consider machine sizes from 4 to 256 processors and message sizes from 32 bytes to 16K bytes. We study the performance over a whole spectrum of source numbers ranging from 1 to p and a representative selection of source distributions. In this paper we report only the performance for the case when all source processors broadcast messages of the same length. The performance we observed for broadcasting instances with different length messages does not impact the choice of algorithms. In particular, good distributions remain good distributions when the length of messages varies. Throughout this section, we use L to denote the size of the messages at source processors.

Most implementation issues follow in a straightforward way from the descriptions given in the previous sections. We point out that we do not synchronize globally after each iteration or after one dimension has been handled. In all our algorithms, as soon as a processor has all relevant data, it continues.

The communication operations invoked in Algorithms *Xor* and *2-Step* use the implementations described in [7]. We choose the best algorithms for the given situation. In particular, the

all-to-all exchange algorithm views the exchanges as consisting of p permutations and it uses the exclusive-or on processor indices to generate the permutations. The most efficient Paragon implementation of an one-to-all communication views the mesh as a linear array and applies the 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 expect Algorithms *Xor* and *2-Step* to give good performance only for special cases. *Xor* simply exchanges too many messages and Algorithm *2-Step* creates unnecessary communication bottlenecks. However, we did want to see their performance against the other proposed algorithms to show the disadvantage of using existing communication routines in a brute-force way.

In the following we first study the scalability of the algorithms for 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 that these parameters have a significant impact on the performance.

Figure 2 shows the performance of all five algorithms described in Section 2. From this figure it is apparent that Algorithms *2-Step* and *Xor* are not efficient. In particular, for more than 4 sources, Algorithm *2-Step* suffers congestions at the node which receives all the messages. Algorithm *Xor* is inherently inefficient because of the large number of sends issued by the source processors. For Algorithm *2-Step*, the rate of increase in the execution time is steeper than the increase in number of sources. This is due to the fact that as the number of source processors increases, the bottleneck processor in Algorithm *2-Step* receives more messages in the first step and sends out more data in the second step. However, in the case of Algorithm *Xor*, with the increase in number of source processors, the increase in the number of sources is more distributed among all processors. The bandwidth of the network is high enough to handle this type of increased communication volume better. 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

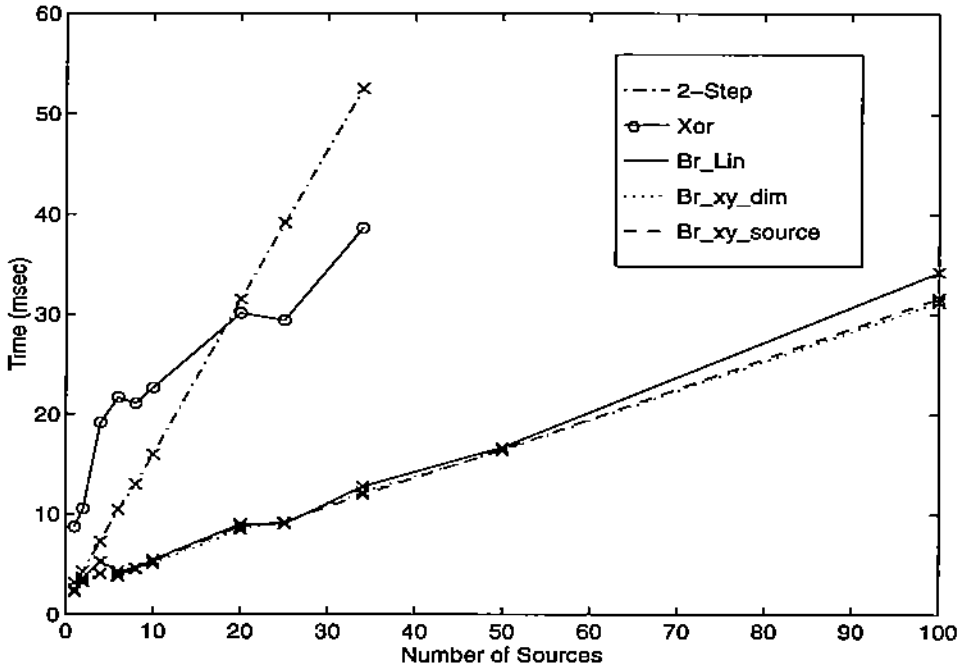

Figure 2: Performance of algorithms when the number of sources varies from 1 to 100, assuming $L = 4K$ and equal distribution on a 10×10 machine.

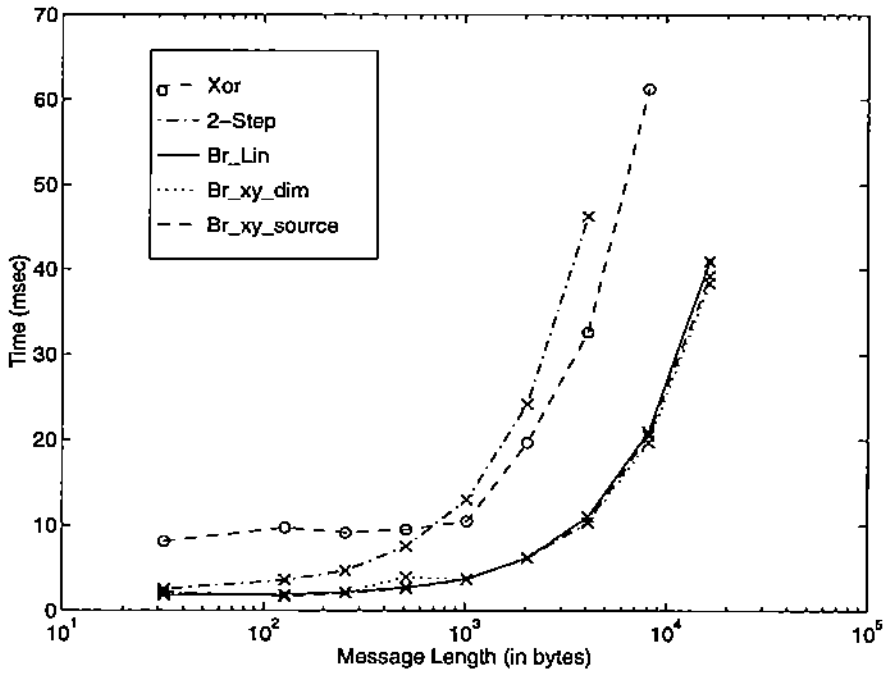

Figure 3: Performance of algorithms when L varies from 32 bytes to 16K keeping s =30 on a 10×10 machine with right diagonal distribution.

the machine, the performance of these algorithms differs slightly.

Figure 3 shows the performance for a right diagonal distribution with $s = 30$ when the message size changes. As already stated, diagonal distribution places the same number of sources in the rows and columns. Once again, regardless of how small a message size, Algorithms *2-Step* and *Xor* perform poorly. The almost flat curve up to a message size of 1K for Algorithm *Xor* further supports our observation related to Figure 2. The other three algorithms experience little increase in the time until $L = 512$ bytes. Then we see a linear increase.

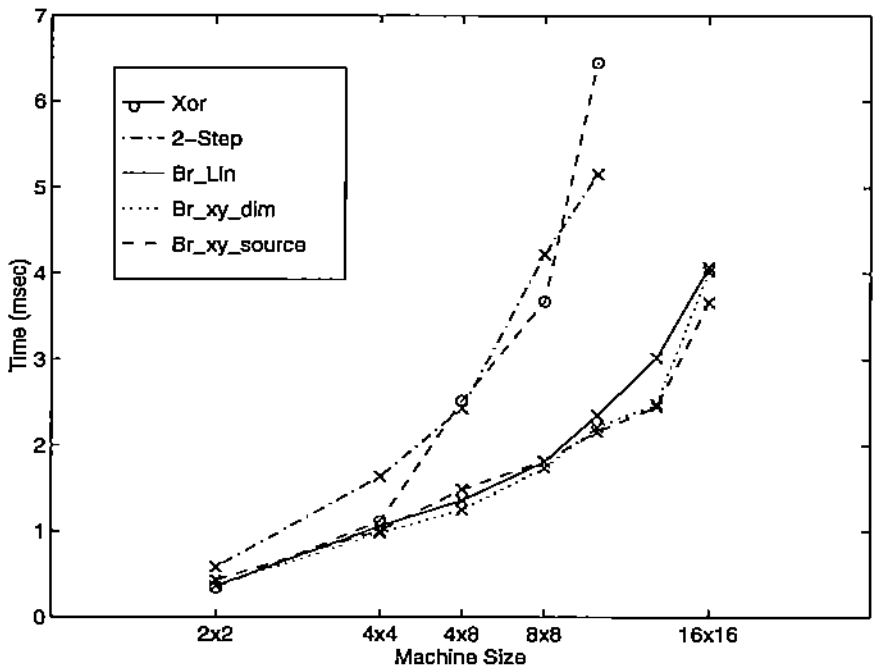

Figure 4: Performance of algorithms when the machine size varies, assuming $L = 1$K and having approximately $\sqrt{p}$ sources in a right diagonal distribution.

Figure 4 shows the behavior of all five algorithms when the machine size varies from 4 to 256 processor. Algorithm *Xor* 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.

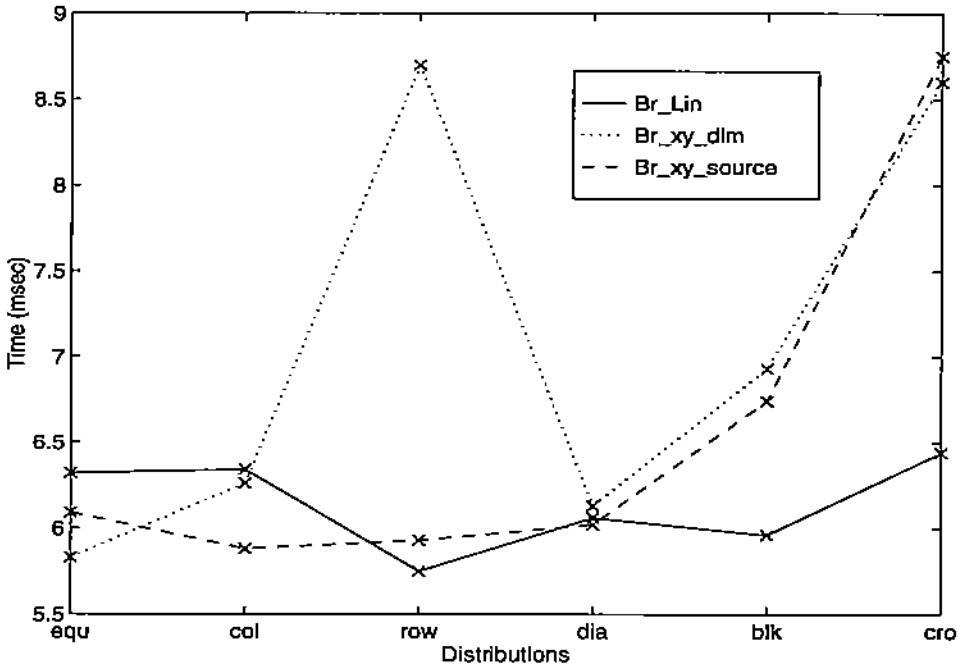

Figure 5: Performance of three algorithms on a 10x10 machine with $L = 2K$ assuming different source distributions with $s = 30$.

Figure 5 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 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. The 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 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 6 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. An interesting aspect of the performance curves is that if the data is spread among a larger number of sources, the broadcast

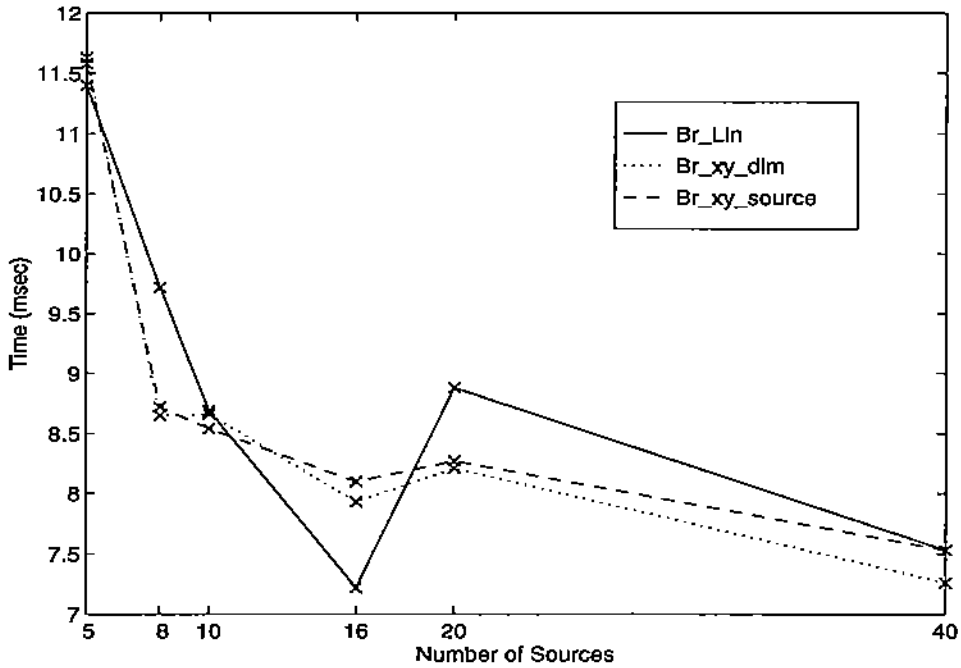

Figure 6: Performance of three algorithms on a 10x10 machine with a right diagonal distribution. The total message size is kept at 80K and the number of sources varies.

operation is accomplished faster. For example the 80K size, data spread among 5 sources takes approximately 11.4 ms using Algorithm *Br_xy_source*. However, the same amount of data spread among 40 sources to begin with takes only 7.3 ms. This plot highlights our claim made earlier that for a given amount of data more number of sources involved in broadcasting yield faster execution times.

Figure 7 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 source processors are likely to be positioned within columns. This does not allow a fast increase in the

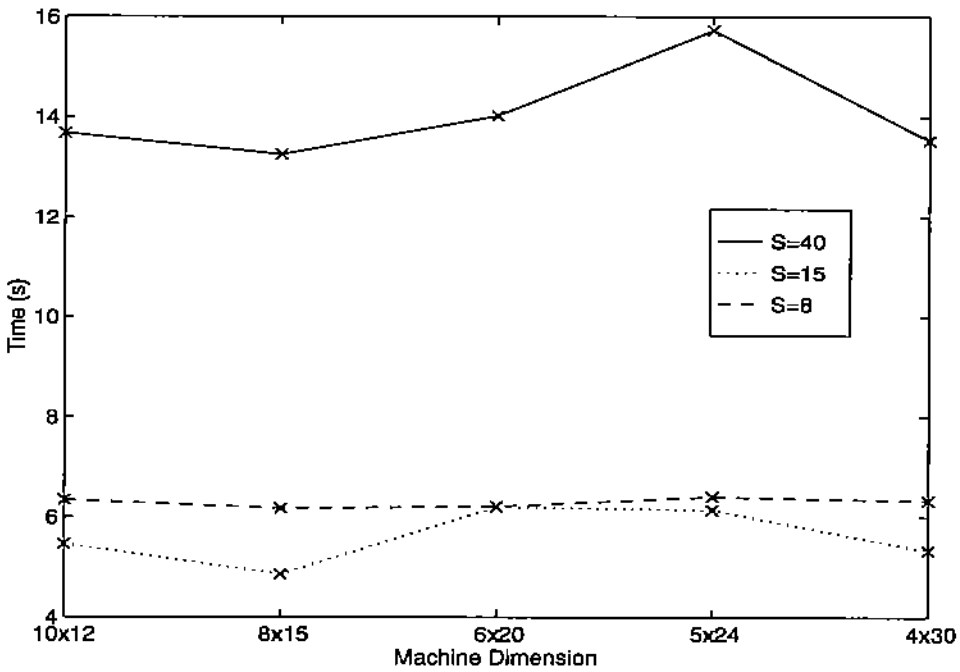

Figure 7: Performance of Algorithm *Br_Lin* when $p = 120$ and the dimensions vary, assuming equal distribution. Three source sizes are shown and $L = 4K$ in all the cases.

number of sources. On the other hand, for $s = 15$, the source processors are, with the exception of size 4×30, positioned along diagonals.

We conclude this section with a brief description of the performance of Algorithms *Repos* and *Part_xy_source*. Consider the version of *Repos* that invokes *Br_xy_source*. In this algorithm we first perform a permutation to redistribute source processors according to an ideal distribution, say a row distribution. We generate a row distribution that 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 depends on s and where the s source processors are located. For messages of size up to 2K and $s < p/2$, 0.5 msec is a good general estimate. Figure 8 compares the performance of Algorithm *Repos* to the performance achieved for other distributions, keeping all other parameters fixed. The input distribution given to Algorithm *Repos* was the equal distribution. For a given s, the performance of *Repos* has proven to be almost independent of the input distribution. Our conclusion is that unless the initial distribution of sources is close to an ideal distribution, it pays to perform the

redistribution.

	10×10, $L = 2K$, $s = 20$	15×16, $L = 2K$, $s = 58$
Repos	5.0	8.5
Row	4.5	7.5
Column	4.5	9.3
Equal	5.2	9.1
Right Diagonal	5.1	8.3
Cross	7.3	12.0

Figure 8: Comparing the performance of *Repos* invoking *Br_xy_source* to *Br_xy_source* with other source distributions.

At this point we cannot draw the same conclusion for Algorithm *Part_xy_source*. Preliminary results indicate that the partitioning approach gives a better performance only for difficult input distributions. The reason lies in the cost of the final permutation. The exchange of long messages done in the final step dominates the performance. Figure 9 shows a data point

	redistribution	*Br_xy_source*	final permutation	Total
Repos	0.4	3.7	--	4.1
Part_xy_source	0.4	1.9	2.5	4.8

Figure 9: Performance of *Repos* and *Part_xy_source* on a 8×8 machine for the cross distribution with $s = 15$ and $L = 2K$.

comparing Algorithms *Repos* and *Part_xy_source*. The input to both algorithms is the cross distribution with $s = 15$. The third column in the figure lists the time for invoking Algorithm *Br_xy_source*. For *Repos* we do so with an 8×8 machine, $s = 15$, and the row distribution (which costs 3.7 msec). For *Part_xy_source* we invoke two instances, each one a 4×8 machine, one with $s = 8$ and one with $s = 7$. Recall that Algorithm *Part_xy_source* creates two broadcasting problems, one for each half of the mesh. The row distribution for $s = 8$ costs 1.9 msec. For comparison, on an 8×8 machine, the cross distribution with $s = 15$ costs 4.5 msec. We are currently exploring different partitioning approaches that may lead to a better overall performance of Algorithm *Part_xy_source*.

6 Conclusions

We described different s-to-p broadcasting algorithms and analyzed their scalability and performance on the Intel Paragon. We showed that the performance of each algorithm is influenced by the distribution of the source processors and a relationship between the distribution and the dimension of the machine. Each algorithm has ideal distributions and distributions on which the performance degrades. To reduce the dependence of the performance on the input distribution we proposed a repositioning approach. In this approach the given distribution is changed turned into an ideal distribution of a particular broadcasting algorithm which is then invoked.

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. 8-th International Parallel Processing Symposium*, 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. Scalable High-Performance Computing Conference*, pp. 357-364, 1994.

[3] S.H. Bokhari, "Multiphase Complete Exchange on a Circuit Switched Hypercube," *Proceedings of 1991 International Conference on Parallel Processing*, pp. 525-529, 1991.

[4] Z. Bozkus, S. Ranka, G. Fox "Benchmarking the CM-5 Multicomputer," *Proc. Symposium on the Frontiers of Massively Parallel Computation*, pp. 100-107, 1992.

[5] 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. International Symposium on Parallel and Distributed Processing*, pp. 594-602, 1994.

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

[7] 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.

[8] S. Hinrich, C. Kosak, D. O'Halloron, T. Stricker, R. Take, "An Architecure for Optimal All-to-All Personalized Communication," *Proc. of 6-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 310-319, 1994.

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

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

[11] Y. Lan, A. H Esfhanian, and L. M. Ni, "Multicast in Hypercube Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 30-41, 1990.

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

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

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

[15] R. Thakur, A. Choudhary, "All-to-all Communication on Meshes with Wormhole Routing," *Proc. 8-th International Parallel Processing Symposium*, pp. 561-565, 1994.