Fast Collective Communication Libraries, Please*

Prasenjit Mitra
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188

David G. Payne
Scalable Systems Division
Intel Corporation
15201 N.W. Greenbrier Pkwy
Beaverton, Oregon 97006

Lance Shuler
Parallel Computing Sciences Department, 1424
Sandia National Laboratory
Albuquerque, New Mexico 87185-1109

Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188

Jerrell Watts
Scalable Concurrent Programming Laboratory
California Institute of Technology
Pasadena, California 91125

Abstract

It has been recognized that many parallel numerical algorithms can be effectively implemented by formulating the required communication as collective communications. Nonetheless, the efficiency of such communications has been suboptimal in many communication library implementations. In this paper, we give a brief overview of techniques that can be used to implement a high performance collective communication library, the iCC library, developed for the Intel family of parallel supercomputers as part of the InterCom project at the University of Texas at Austin. We compare the achieved performance on the Intel Paragon to those of three widely available libraries: Intel's NX collective communication library, the MPICH Message Passing Interface (MPI) implementation developed at Argonne and Mississippi State University and a Basic Linear Algebra Communication Subprograms (BLACS) implementation, developed at the University of Tennessee.

*This work was partially supported by the Intel Research Council and Intel Scalable Systems Division. Jerrell Watts is being supported by an NSF Graduate Research Fellowship.

1 Introduction

Efficient communication is crucial for obtaining good applications performance on distributed-memory multicomputers like the Intel Paragon, the Cray T3D, the IBM SP-2, and the Thinking Machines CM-5. For many applications, the required communication is collective in nature. By this we mean that a group of (possibly all) processing nodes cooperate in a communication. Examples of this include broadcasting of a message, collecting of messages from all nodes, scattering a vector from one node to other nodes, and forming a vector that is the result of the element-wise summation of vectors that reside on different nodes.

Over the last few years, we have written a number of papers that showed how to efficiently implement individual collective communications on hypercubes and multidimensional meshes. More recently, as part of the Interprocessor Collective Communication (InterCom) project, we showed how essentially all major collective communications can be implemented using a comprehensive approach, yielding highly efficient implementations of an entire library of such communications. This work was motivated by the fact that interfaces to such libraries had been standardized as part of the Message Passing Interface (MPI) and there was a need for developing strategies for implementations of this standard.

In this paper, we show that highly optimized implementations of collective communication libraries can be attained using simple methods, yielding considerable payoff.

2 Motivation

To understand how many algorithms can be formulated in terms of collective communications, we start by describing a typical algorithm that benefits from these decisions.

2.1 LAPACK LU factorization

The LU factorization is the well-known operation that factors a given matrix A into the product of lower and upper triangular matrices:

$$A = LU$$

where L is lower triangular and U is upper triangular. The LAPACK implementation of this operation can be formulated as follows: Partition

$$\begin{aligned} A &= \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \\ &= \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right) \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array} \right) = LU \end{aligned}$$

The above equation shows that the following equalities must hold:

$$\begin{aligned} \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} &= \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} \begin{pmatrix} U_{11} \\ 0 \end{pmatrix} && (1) \\ A_{12} &= L_{11} U_{12} && (2) \\ A_{22} &= L_{21} U_{12} + L_{22} U_{22} && (3) \end{aligned}$$

This formulation allows the bulk of the computation to be cast in terms of "matrix-matrix" operations, or Level-3 BLAS, which generally perform much better on architectures with hierarchical memories: First, the panel of the matrix consisting of A_{11} and A_{21} is factored, overwriting these submatrices. This itself requires a sequence of Level-1 and Level-2 BLAS calls, but is of lower order time. Next, submatrix A_{12} is overwritten with $U_{12} = L_{11}^{-1} A_{12}$, a call to the Level-3 BLAS routine `DTRSM`. After this, submatrix A_{22} is overwritten with $A_{22} - L_{21} U_{12}$, a call to the Level-3 BLAS routine `DGEMM`. Finally, this updated submatrix A_{22} itself is factored similarly, in a recursive manner.

2.2 ScaLAPACK LU factorization

To describe the approach that ScaLAPACK takes towards parallelizing algorithms like the LU factorization, consider the case of a parallel computer with six processing nodes, which are *logically* viewed as a 2×3 mesh of nodes. Nodes are now indexed by row and column indices as $\mathbf{P}_{ij}$. Next, the matrix is blocked into $nb \times nb$ submatrices, which are assigned to the logical mesh in a block-cyclic fashion as indicated below:

	$\mathbf{P}_{\star 0}$	$\mathbf{P}_{\star 1}$	$\mathbf{P}_{\star 2}$	$\mathbf{P}_{\star 0}$	$\mathbf{P}_{\star 1}$	$\cdots$
$\mathbf{P}_{0\star}$	A_{00}	A_{01}	A_{02}	A_{03}	A_{04}	$\cdots$
$\mathbf{P}_{1\star}$	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	$\cdots$
$\mathbf{P}_{0\star}$	A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
	$A_{(N-1)0}$	$A_{(N-1)1}$	$A_{(N-1)2}$	$A_{(N-1)3}$	$A_{(N-1)4}$	$\cdots$

Here $\mathbf{P}_{i\star}$ and $\mathbf{P}_{\star j}$ denote the ith row and jth column of the node mesh, respectively.

The LAPACK LU factorization can now proceed as follows: Factor the submatrix of A consisting of the first panel (column of blocks) in the above figure. This operation is performed cooperatively by the first column of nodes. Next, the resulting lower *trapezoidal* factor is distributed to all other nodes. Then, the first row of nodes update $A_{0j} \leftarrow L_{00}^{-1} A_{0j}$, $j = 1, \ldots, (N-1)$ in parallel. Finally, this updated submatrix of A is distributed to the other nodes, and the update of the remainder of the matrix is performed in parallel by all nodes, after which the process proceeds similarly in a recursive fashion.

A few important observations can be made from this simple example:

- If partial pivoting is added to this algorithm, the column of nodes that performs the factorization of the panel must be able to determine the pivot rows within that column of nodes. This suggest the need for a "MAX" operation to be performed with columns of nodes.

- Pivoting of the rows require messages to be exchanged between nodes that reside in the same column of nodes.

- L_{i0} is only needed in the row of nodes that holds the ith row of blocks. Hence the distribution of the factored panel becomes a broadcast within rows.

- Similarly, U_{0j} is only needed within columns of nodes that hold the jth column of blocks. This suggests a broadcast within rows.

- Both of the above mentioned broadcasts involve *submatrices* of the original matrix. In some cases, the shape of these submatrices is trapezoidal.

The above example clearly shows the need for collective communication, as well as the need to perform such communication within groups of nodes, motivating both the iCC library and the BLACS interface.

3 Interprocessor Collective Communication (iCC) Library

In this section, we briefly describe the approaches to implementing collective communications on mesh architectures developed as part of the InterCom project.

Our current implementation assumes a two-dimensional *physical* mesh of processing nodes, with bidirectional links between nodes and worm-hole (cut-through) routing. Furthermore, we assume that it is possible to model the time required for sending a message of length n bytes between any two nodes by $\alpha + n\beta$, where α is the latency for sending a message, and β is the communication time per item, in the absence of network conflicts. In our discussions below, we assume a processor can both send and receive at the same time. But it can only send to, or receive from, one other node at a given time. When two messages traverse the same physical link on the communication interconnect, we assume they share the bandwidth of that link. In addition, we assume that the time for performing an arithmetic operation is denoted by γ.

3.1 Building blocks

We start by presenting building blocks for the iCC library. All the building blocks have the property that they are simple to implement, do not require power-of-two size partitions, and incur no network conflicts. The resulting implementations of the short vector primitives can be shown to have optimal latency (minimal startup overhead). The implementation of the long vector primitives can be shown to be asymptotically optimal on linear arrays as vector size increases.

We discuss the implementation of the building blocks in the setting of linear arrays, which due to worm-hole routing can be considered unidirectional rings, when convenient. (For example, if all messages are sent to the right nearest neighbor, only the rightmost processor in the linear array sends to the left. Hence, there are no message conflicts.) We later discuss how these techniques are generalized to meshes.

3.1.1 Short vector primitives

Algorithms for implementing collective communications for short vectors must minimize startup cost, i.e. the number of messages sent. On hypercubes, this can be easily accomplished by staging the algorithms as $\log p$ steps during which communication is performed in each hypercube dimension. For meshes, this idea can be utilized as well, provided some care is taken at each stage [4].

All our target short vector collective communication operations can be built from four primitives. These are **broadcast**, **combine-to-one**, **scatter**, and **gather**.

Consider the **broadcast**. For short vectors, this operation can be implemented on a linear array of nodes in the following way: Start by assuming a given root node has the message of length n. The broadcast can proceed by dividing the linear array in two (approximately) equal parts and choosing a receiving node in the part that does not contain the root. The broadcast proceeds recursively by treating each of the involved nodes as a new root for a broadcast within its own half of the previous array. It is easy to see that no network conflicts occur and the total time required is $\lceil \log p \rceil (\alpha + n\beta)$.

The **combine-to-one** can be implemented similarly by running the broadcast communications in reverse order and interleaving communication with the combine operation. This requires a total time of $\lceil \log p \rceil (\alpha + n\beta + n\gamma)$. The **scatter** can be implemented like the broadcast, except at each stage only the data that eventually resides in the other part of the network is sent. If each node receives an equal share of the initial vector, the cost is approximately $\lceil \log p \rceil \alpha + [(p-1)/p] n\beta$. The **gather** can be implemented as the scatter in reverse and incurs the same cost.

3.1.2 Long vector primitives

For long vectors, a strategy that minimizes overhead due to vector length, in addition to avoiding network conflicts, is necessary. It should be noted that the above mentioned **scatter** and **gather** operations have this property, and they also act as long vector primitives. In addition, we propose two more long vector primitives, the **bucket collect** and **bucket distributed combine**. These four primitives constitute the set from which all our target long vector collective communication operations can be built.

The **bucket collect** is a special implementation of the collect, which views the linear array as a ring.

Buckets are passed between the nodes that move the subvectors to be collected, leaving the result on all nodes. Note that no network conflicts occur. Cost: $(p-1)\alpha + [(p-1)/p]n\beta$.

The **bucket distributed global combine** is similar to the bucket collect, executed in reverse, where the buckets are used to accumulate contributions. Cost: $(p-1)\alpha + [(p-1)/p]n\beta + [(p-1)/p]n\gamma$.

3.2 Using the building blocks

In this section, we describe how the short and long vector primitives can be used to generate short and long vector implementations for all collective communications.

3.2.1 Short vector algorithms

For short vectors, the broadcast, combine-to-one, scatter and gather primitives are, of course, implementations of the operations themselves. The other three collective communications can be generated using these primitives as follows:

Collect: Gather followed by broadcast. Cost:

$$2\lceil\log p\rceil\alpha + \left(\frac{p-1}{p} + \lceil\log p\rceil\right) n\beta.$$

Distributed global combine: Combine-to-one followed by scatter. Cost:

$$2\lceil\log p\rceil\alpha + \left(\frac{p-1}{p} + \lceil\log p\rceil\right) n\beta + \lceil\log p\rceil n\gamma.$$

Global combine-to-all: Combine-to-one followed by broadcast. Cost:

$$2\lceil\log p\rceil\alpha + 2\lceil\log p\rceil n\beta + \lceil\log p\rceil n\gamma.$$

For all these implementations, the startup cost is within a factor two of optimal. On both the Touchstone Delta and the Paragon, due to machine specific issues, the startup is actually optimal.

3.2.2 Long vector algorithms

For long vectors, the collect, distributed combine, scatter and gather primitives are once again the implementations themselves. The other three collective communications can be generated using these primitives as follows:

Broadcast: Scatter followed by collect. Cost:

$$(\lceil\log p\rceil + p - 1)\alpha + 2\frac{p-1}{p}n\beta.$$

Combine-to-one: Distributed combine followed by gather. Cost:

$$(\lceil\log p\rceil + p - 1)\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

Global combine-to-all: Distributed combine followed by collect. Cost:

$$2(p-1)\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

For the broadcast and combine-to-one, it can be argued that the β term is asymptotically within a factor two of optimal, while for the combine-to-all it can be argued that the β term is asymptotically optimal.

3.3 Extension to meshes

The above techniques can be easily extended to physical meshes. We will concentrate on the two dimensional case in our descriptions. All of the operations can be implemented on meshes by performing the communication first within rows and next within columns, or visa versa. Both rows and columns are simply linear arrays.

3.3.1 Short vector case

For short vectors, the minimum spanning tree broadcast can be implemented by broadcasting within the root's column first, followed by simultaneous broadcasts within all rows, with the node in the original column as roots of their respective rows. A scatter can be accomplished by scattering within the root's column, followed by simultaneous scatters within rows. The resulting costs are given by:

Broadcast/Combine-to-one:

$$(\lceil\log r\rceil + \lceil\log c\rceil)\,\alpha + (\lceil\log r\rceil + \lceil\log c\rceil)\,n\beta$$

which equals the familiar $\log p\alpha + \log pn\beta$ when p is a power of two. For the combine-to-one, the term $(\lceil\log r\rceil + \lceil\log c\rceil)\,n\gamma$ must be added.

Scatter/Gather:

$$\lceil\log r\rceil\alpha + \frac{r-1}{r}n\beta + \lceil\log c\rceil\alpha + \frac{c-1}{c}\frac{n}{r}\beta = (\lceil\log r\rceil + \lceil\log c\rceil)\,\alpha + \tfrac{p-1}{p}n\beta$$

which equals the familiar $\log p\alpha + \frac{p-1}{p}n\beta$ when p is a power of two.

Notice that the cost essentially does not change from the linear array case.

To implement the algorithms that are built from these primitives, we merely use the mesh implementations of the building blocks, given above. E.g. a collect is implemented as a gather within rows to a specified column, followed by a gather of the results within that column. Next, the result is broadcast within that column and broadcast within all rows. Costs for the different operations are given by

Collect/Distributed combine:

$$2\left(\lceil\log r\rceil + \lceil\log c\rceil\right)\alpha + \left(\tfrac{p-1}{p} + \lceil\log r\rceil + \lceil\log c\rceil\right) n\beta.$$

For the distribed combine the term

$$\left(\lceil\log r\rceil + \lceil\log c\rceil\right) n\beta$$

must be added.

Global combine-to-all:

$$2\left(\lceil\log r\rceil + \lceil\log c\rceil\right)\alpha + 2\left(\lceil\log r\rceil + \lceil\log c\rceil\right) n\beta + \left(\lceil\log r\rceil + \lceil\log c\rceil\right) n\gamma.$$

3.3.2 Long vector primitives

The same holds for long vector primitives. A collect can be implemented by simultaneous collects within rows, followed by simultaneous collects of the results within columns. The distributed combine is implemented by reversing this process. The approach for the scatter and gather is the same as for short messages. The new costs are given by

Collect/Distributed combine:

$$(c-1)\alpha + \frac{c-1}{c}\frac{n}{r}\beta + (r-1)\alpha + \frac{r-1}{r}n\beta = (c+r-2)\alpha + \tfrac{p-1}{p}n\beta$$

For the distributed combine, the term $\frac{p-1}{p}n\gamma$ must be added.

Notice that the net result is that the α term is *reduced* from $(p-1)$ for the linear array to $(c+r-2)$ on the mesh.

Again, to implement the algorithms that are built from these primitives, we merely use the mesh implementations of the building blocks, given above. E.g. a broadcast is implemented as a scatter within the column that owns the root, followed by simultaneous scatters within the rows. Next, collects of the results are performed within rows, followed by collects within all columns. Costs for the different operations are given by

Broadcast/Combine-to-one:

$$(\lceil\log r\rceil + \lceil\log c\rceil + c + r - 2)\alpha + 2\frac{p-1}{p}n\beta.$$

For the combine-to-one, the term $\frac{p-1}{p}n\gamma$ must be added.

Global combine-to-all:

$$2(c+r-2)\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

3.4 Hybrid algorithms

Since a range of vector lengths are encountered in communication required for parallel algorithms, it does not suffice to implement only the short vector or the long vector approach. Indeed, our research shows that high performance implementations must use *hybrid* collective communications algorithms: for short and long vectors, the short and long vector algorithms must be respectively employed. However, for the range of lengths in between, hybrids can be employed to achieve better performance then either the short or the long vector algorithm. The idea is to use a long vector approach in subgroups. This has the effect of shortening the vector length until it pays to perform a short vector approach to complete the collective communication. More details of this can be found in our paper [2].

4 Basic Linear Algebra Communication Subprograms

The BLACS (Basic Linear Algebra Communication Subprograms) project arose as part of a larger project called ScaLAPACK (Scalable Linear Algebra PACKage) [5]. The goal of the ScaLAPACK project is to implement a core set of the linear algebra routines provided in the sequential library LAPACK (Linear Algebra PACKage)[10] on distributed memory platforms. The BLACS are meant to be the communication kernels much like the BLAS are the computation kernels for LAPACK and ScaLAPACK.

A reference implementation of the BLACS developed at the University of Tennessee allowed the researchers involved in the ScaLAPACK project to *specify* a reasonable interface and start building the ScaLAPACK library. The implementation is essentially based on algorithms similar to the short vector algorithms discussed in the previous section. As a result, a considerable performance penalty is incurred.

The calling sequences for the collective communications incorporated into the BLACS include the ability to send submatrices and the ability to communicate within rows and columns, hence providing some ability to communicate within groups.

The approach chosen for the reference implementation was constrained by the fact that it needed to be reasonably efficient for a wide variety of parallel architectures, including the Intel iPSC, Touchstone Delta, and Paragon systems, the Cray T3D, the IBM SP1 and SP2, and the Thinking Machines CM-5. As a result, algorithms that are inherently tree based were chosen. These have the advantage that they are easy to implement and can be easily staged to avoid communication network conflicts. Moreover, they incur (near) minimal messages, thereby reducing the overhead due to message latency. However, the total volume of data communicated is non-optimal, which greatly reduces their effectiveness for long vector lengths. It should be noted that in [13] it is shown that this approach to implementation is competitive with the implementations of similar collective communication libraries by all the major vendors.

5 MPI Collective Communication Interface

We quote from *Using MPI* [6]:

> During 1993, a broadly based group of parallel computer vendors, software writers and application scientists collaborated on the development of a standard portable message-passing library definition called MPI, for Message-Passing Interface. As of mid-1994, a number of implementations are in progress, and applications are already being ported.

We fully endorse MPI as the interface that should be broadly adopted by application developers.

6 Performance Comparison

In this section, we compare the achieved performance on the Intel Paragon to those of three widely available libraries: Intel's NX collective communication library, the MPICH Message Passing Interface (MPI) implementation developed at Argonne and Mississippi State University and a Basic Linear Algebra Communication Subprograms (BLACS) implementation, developed at the University of Tennessee. Since the BLACS were favorably compared to those of most major vendors, we argue that a comparison to the BLACS is in effect a comparison to all major vendors' library implementations.

We concentrated on two operations: broadcast and global sum-to-all. The primary reason is that these are the only operations that all four libraries have in common. However, our description of how good implementations of all collective operations are based on the same simple ideas allows us to argue that comparisons of these two operations are actually representative. The calls used are summarized in Table 1.

	Broadcast	Sum-to-all
iCC	`iCC_bcast`	`iCC_gdsum`
NX	`csend(''-1'')`	`gdsum`
MPI	`MPI_Bcast`	`MPI_Allreduce`
BLACS	`dgebs2d`	`dgsum2d`

Table 1: Routines used for experiments.

Figures 1 and 2 report the results of our experiments. We also summarize our findings in Table 2. Notice that the iCC results are substantially superior to all the other library implementations, especially for very long messages. It should be noted that the NX timings for the broadcast did not use "forced message types". Had we done so, the results would be more in line with MPI and BLACS.

In Figure 3 we show a comparison of the different libraries when the operations are only within a row or column, as would be necessary for the example in Section 2. Notice that for long vectors, iCC continues to be superior, although the improvement is less. This is not supprising, since the ratio between the short vector approaches used by NX, BLACS, and MPI and the long vector approaches used by iCC is proportional to $\log p$. Somewhat disturbing is the fact that for short vectors, iCC is no longer the fastest implementation. We contribute this to the cost of the hybrid mechanism, which appears to need further tuning for the case where few nodes are involved in the communication.

> **It is important to realize that both the MPI and BLACS implementations are intended to be *reference* implementations of useful specifications, not high performance implementations. Our findings are intended to draw attention to what can be achieved, so that high**

performance implementations of these libraries will become available in the future.

	Broadcast		
bytes	NX/iCC	BLACS/iCC	MPI/iCC
16	1.4	1.0	1.6
1024	1.5	1.0	2.5
65536	5.5	2.9	2.8
1048576	11.3	6.1	7.5
	Sum-to-all		
bytes	NX/iCC	BLACS/iCC	MPI/iCC
16	1.0	1.2	2.1
1024	1.0	1.0	2.0
65536	21.1	4.1	6.9
1048576	34.6	5.9	11.8

Table 2: Comparison of the various library implementations on a 16x32 mesh Paragon

7 Status

A beta version of the iCC library was released in Spring 1994. The first release (R1.0) followed in Summer 1994. Release R2.0 was completed in March 1995. This current release includes a limited, MPI-like, group interface. A version of the library exists that automatically changes all NX collective communication calls, except broadcasts, to iCC calls. As of this date, no iCC bugs have been reported, despite wide use at all major Paragon sites.

Current information on the library can be found at the following web site:

`http://www.cs.utexas.edu/users/rvdg/intercom`

8 Conclusion

In this paper, we have described simple techniques that can be used to build highly efficient collective communication library. The experiments show them to be highly competitive, outperforming all libraries on the Intel Paragon that are widely available and used. Moreover, the techniques are such that the library is highly robust and can be easily maintained.

While we only perform comparisons on the Paragon, we mentioned that the BLACS were compared to vendor libraries on a number of different platforms, showing them to be highly competitive. Since our techniques can be easily extended to those platforms, we believe our techniques would provide a basis for efficient collective communication libraries on all platforms. As mentioned, we fully endorse MPI as the interface that should be broadly adopted by application developers. It is through efficient implementations that MPI will attain wide acceptance.

Acknowledgements

This research was performed in part using the Intel Paragon System and the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Intel Supercomputer Systems Division and the California Institute of Technology. Funding for this project was provided in part by the Intel Research Council, Intel Supercomputer Systems Division, and the University of Texas Center for High Performance Computing.

A number of other researchers need to be credited with having contributed to this work in the past. These include Mike Barnett (University of Idaho), Satya Gupta (Intel SSD), and Rik Littlefield (PNL).

Finally, we are grateful to the folks at Argonne and Mississippi State University, who make the reference MPI implementation available, and the folks at the University of Tennessee, who bring us the reference BLACS implementation.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. "LAPACK: A Portable Linear Algebra Library for High-Performance Computers," LAPACK Working Note 20, University of Tennessee, CS-90-105, May 1990.

[2] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts. "Interprocessor Collective Communication Library (InterCom)." *Proceedings of Supercomputing 94*, Nov. 1994.

[3] M. Barnett, R. Littlefield, D.G. Payne and R. van de Geijn. "On the Efficiency of Global Combine Algorithms for 2-D Meshes With Wormhole Routing," *Journal of Parallel and Distributed Computing*, **24**, pp. 191-201 (1995).

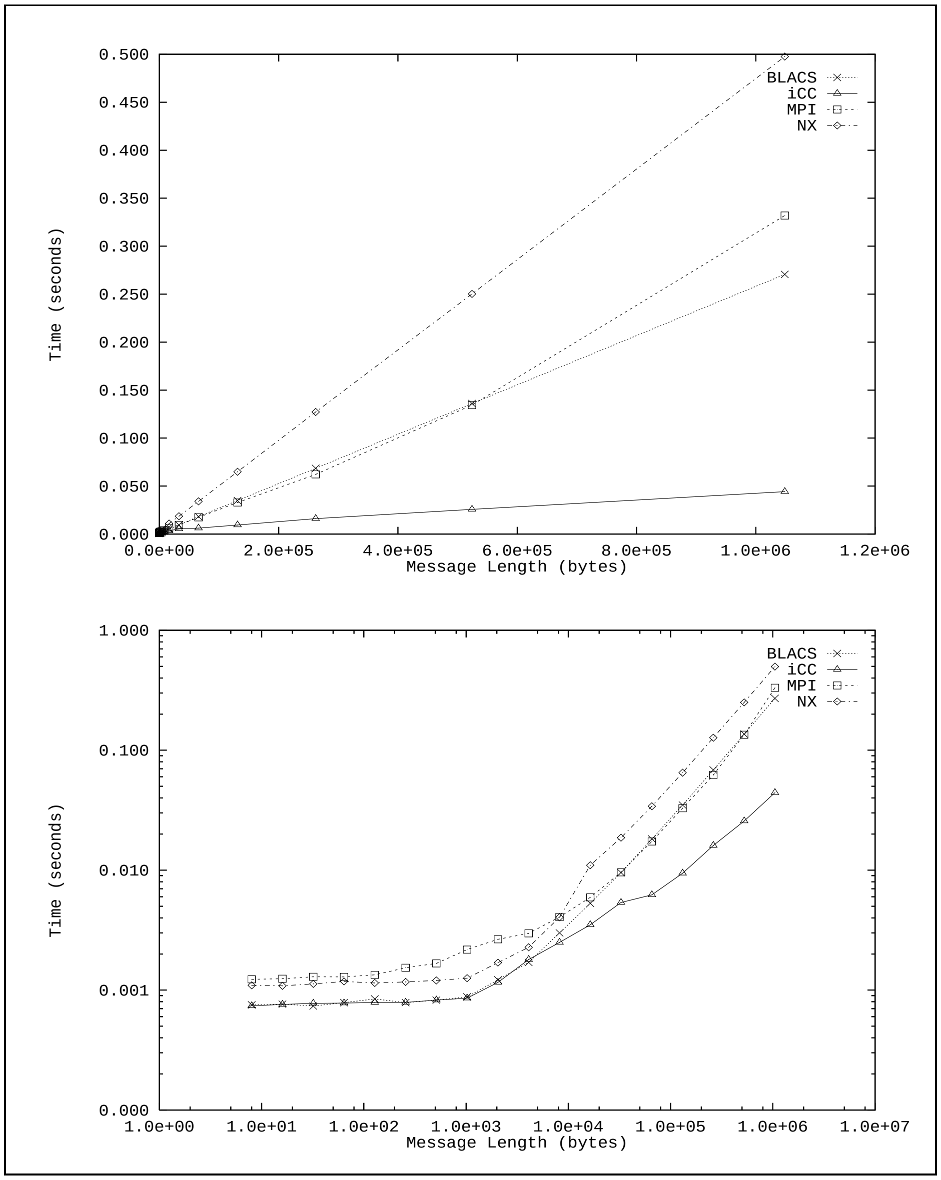

Figure 1: Performance of the various libraries for the broadcast on a 16x32 mesh Paragon (OSF R1.3).

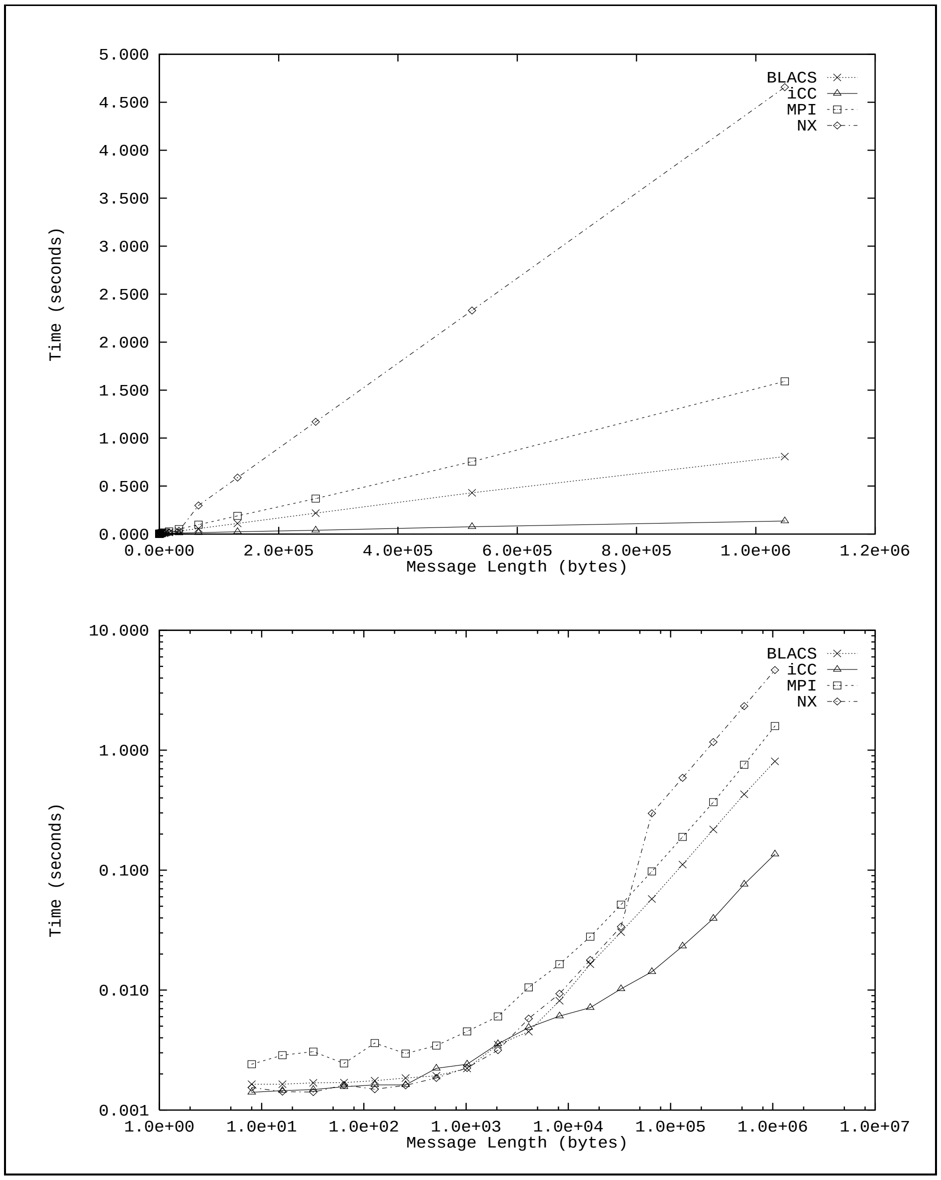

Figure 2: Performance of the various libraries for the combine-to-all on a 16x32 mesh Paragon (OSF R1.3).

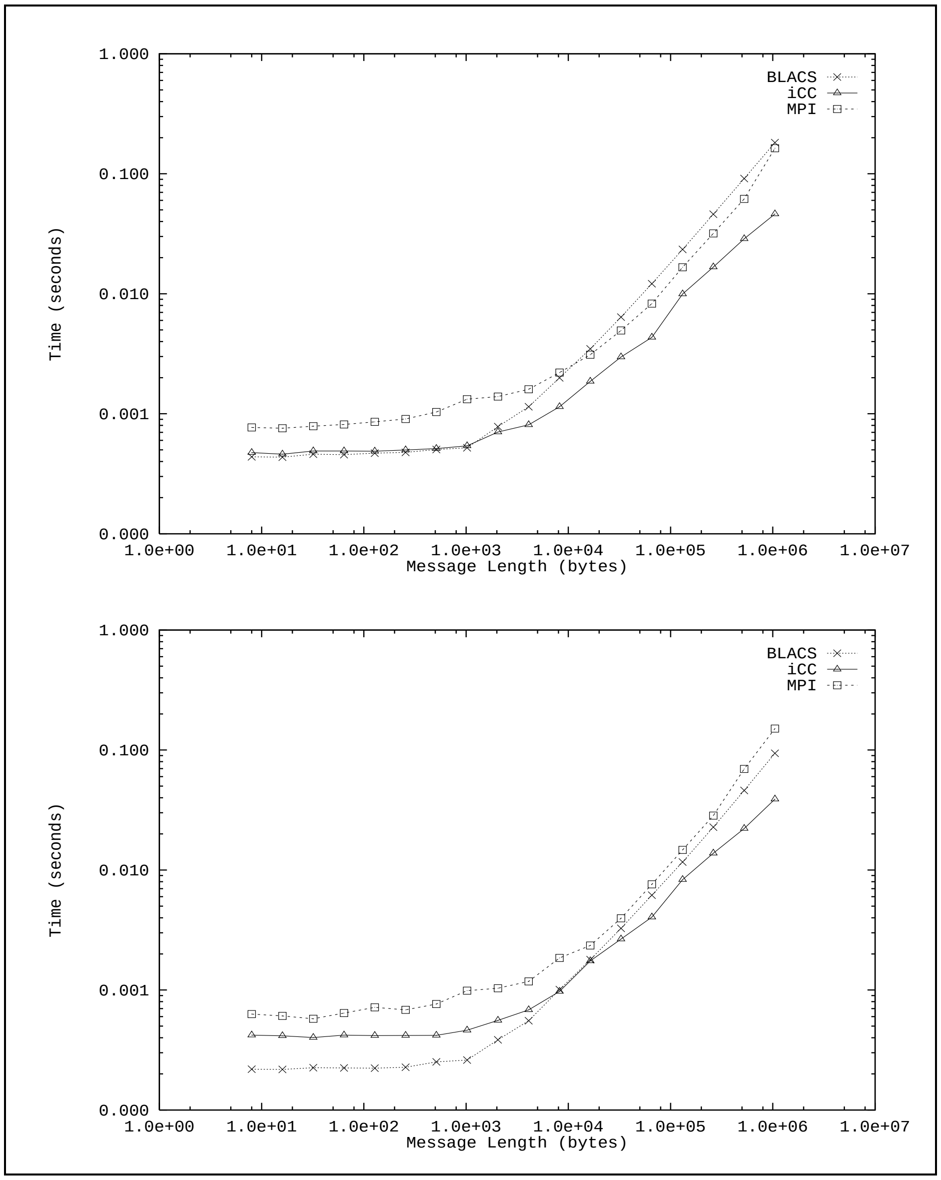

Figure 3: Performance of the various libraries for the broadcast on a 16x32 mesh Paragon (OSF R1.3) when broadcasting within a row of 32 nodes only (top) and a column of 16 nodes only (bottom).

[4] M. Barnett, D.G. Payne, R. van de Geijn and J. Watts. "Broadcasting on Meshes with Worm-Hole Routing," University of Texas, Department of Computer Sciences, TR-93-24 (1993).

[5] J. Choi, J. Dongarra, R. Pozo, and D. Walker. "ScaLAPACK: A Scalable Linear Algebra for Distributed Memory Concurrent Computers," LAPACK Working Note 55, University of Tennessee, CS-92-181, November 1992.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[7] C.-T. Ho and S. L. Johnsson. Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes. *Proceedings of the 1986 International Conference on Parallel Processing*, pg. 640–648, IEEE Computer Society Press, 1986.

[8] L. M. Ni and P. K McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer,* 26(2):62–76, Feb. 1993.

[9] Y. Saad and M. H. Schultz. Data Communication in Parallel Architectures. *Parallel Computing,* 11(2):131–150, Aug. 1989.

[10] Robert van de Geijn. "On Global Combine Operations," *Journal of Parallel and Distributed Computing*, **22** , pp. 324-328 (1994).

[11] R. van de Geijn and J. Watts. A Pipelined Broadcast for Multidimensional Meshes. *Parallel Processing Letters*, to appear.

[12] D. W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing,* Apr. 1994. (Up to date information about the MPI standard is available from `netlib`, directory `mpi`.)

[13] R. Clint Whaley, "Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures," LAPACK Working Note 73, University of Tennessee, CS-94-234, May 1994.