LAPACK Working Note 73
Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures. *

R. Clint Whaley †

June 10, 1994

Abstract

The BLACS (Basic Linear Algebra Communication Subprograms) project is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that is implemented efficiently and uniformly across a large range of distributed memory platforms.

The length of time required to implement efficient distributed memory algorithms makes it impractical to rewrite programs for every new parallel machine. The BLACS exist in order to make linear algebra applications both easier to program and more portable.

It is for this reason that the BLACS are used as the communication layer for the ScaLAPACK project, which involves implementing the LAPACK library on distributed memory MIMD machines.

*This work was supported in part by DARPA and ARO under contract number DAAL03-91-C-0047, and in part by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

†Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, rwhaley@cs.utk.edu

Contents

List of Tables

List of Figures

1 Introduction

The BLACS (Basic Linear Algebra Communication Subprograms) project arose as part of a larger project called ScaLAPACK (Scalable Linear Algebra PACKage) [10]. 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)[7] on distributed memory platforms.

LAPACK contains approximately 1000 routines, which are made up of around 650,000 lines of Fortran77. Since distributed memory computing is much more complex (and therefore requires many more lines of code) than sequential or shared-memory computing, the need for a standard, easy to use message passing interface was quickly identified.

ScaLAPACK is intended to run across a large range of parallel machines. Each platform has its own message passing library, and the number of routines in the ScaLAPACK library make supporting a version for each machine impractical. Thus the first goal of the BLACS is to present a standard interface that can be efficiently supported across a wide range of parallel platforms.

LAPACK already isolates much of its computation within a small library of routines, called the BLAS (Basic Linear Algebra Subprograms)[5, 4, 3]. An efficient port of the BLAS on a machine performs most needed optimization for a given platform. It was decided to extend this idea to communication. Then, an efficient BLAS implementation supplies the compute engine, and an optimized BLACS code satisfies our communication needs. Thus, the efficient porting of ScaLAPACK codes becomes largely a matter of porting these two, much smaller libraries.

There are various packages designed to provide a message passing interface that remains unchanged on several platforms, including PICL [14], and more recently, MPI [2]. These packages are not available on all of the platforms that ScaLAPACK targets. More importantly, they are attempts at general libraries, and are thus somewhat harder to use than a more restricted code.

The BLACS are written at a level where the manipulation of the matrices involved in linear algebra computations is both natural and convenient. Since the audience of the BLACS is known, the interface and methods of using the routines can be simpler than for those of more general message passing layers.

Therefore, the goals of the BLACS project include:

- *Ease of programming*: wherever possible, the BLACS will simplify message passing in order to reduce programming errors.
- *Ease of use* The interface to the BLACS will be at such as level as to be easily usable by linear algebra programmers.
- *Portability* The BLACS must supply an interface which can be supported across a large range of parallel computers.

Jack Dongarra and Robert van de Geijn proposed a BLACS standard in [9]. A prototype implementation was required to test the feasibility and programmability of the standard, at which point the author became involved in the project. The initial prototype was implemented on the Intel i860. After it was shown to be feasible, it was decided that the BLACS should not require the user to supply message identifiers (see section 6.1 for details), and the specifications were correspondingly changed.

Now that the standard had been shown to be both implementable and usable, the BLACS had to be written for several of the more important platforms so that ScaLAPACK would be portable. The Intel version of the BLACS allowed the ScaLAPACK authors to begin their work, and the BLACS could be implemented across other platforms while ScaLAPACK development was underway, so that portability was achieved as soon as a code was developed.

There are presently three versions of the BLACS available via netlib or anonymous ftp. The available versions are: PVM (Parallel Virtual Machine) [1], Thinking Machine's CM-5, and the Intel line of supercomputers. The Intel BLACS work on the following Intel machines: iPSC2, iPSC/860, Touchstone Delta, and Paragon XP/S. In addition, an initial IBM Scalable POWERparallel System 1 implementation has recently been finished, and will be released after optimization and further testing have been done.

Each of these implementations represent code in excess of 10,000 lines. It is therefore impractical to include them in this report, or indeed to cover in detail their development. If the reader wishes to see the software, appendix B provides information for down-loading the codes from netlib.

The first section of this report familiarizes the reader with some of the more important concepts and features of the BLACS. Then, we discuss the BLACS' three main categories of communication in turn. The first category consists of point to point message passing. Next, broadcasts, which take data from one process and send it to many processes, are examined. Finally, combines are discussed. Combines take data distributed over processes, and combine the data in some way to produce a result (at present, data can be combined by doing maximization, minimization, or summation).

After the BLACS interface and features have been explored, some of the more interesting implementation issues are discussed. We will then discuss future directions, possible optimizations, and draw some conclusions.

2 Features of the BLACS

The following discussion of the features of the BLACS are designed to be brief. More information can be obtained in the reports [9] [6]. An even better source of this type of information is the BLACS User's Guide, which is available on netlib (see appendix B for details).

In general, this paper refers to the basic unit of execution as a *process*. A process is a thread of execution which minimally includes a stack, registers, and memory. Multiple processes may share a *processor*. The term processor refers to the actual hardware.

The BLACS do not distinguish between a process and a processor. Each process is treated as if it were a processor: the process must exist for the lifetime of the BLACS run, and its execution should only affect other processes' execution through the use of message passing calls. With this in mind, we use the term process in all sections of this paper except those dealing with timings. When discussing timings, we specify processors as our unit of execution, since speedup will be largely determined by actual hardware resources.

2.1 Array-based Communication

Many communication packages can be classified as having operations based on one dimensional arrays, which are the machine representation for linear algebra's *vector* class. In programming linear algebra problems, however, it is more natural to express all operations in terms of matrices. Vectors and scalars are, of course, simply subclasses of matrices. On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays.

The BLACS recognize the two most common classes of matrices for dense linear algebra. The first of these classes consist of *general rectangular* matrices, which in machine storage are 2D arrays consisting of `M` rows and `N` columns, with a leading dimension, `LDA`, that determines the distance between successive columns in memory.

The second class of matrices recognized by the BLACS are *trapezoidal* matrices. Trapezoidal arrays are defined by `M`, `N`, and `LDA`, as above, but they also have the parameters `UPLO`, which indicates whether the matrix is upper or lower trapezoidal, and `DIAG`, which determines if the diagonal of the matrix need be communicated. Triangular matrices are a sub-class of trapezoidal, so these matrices are also handled by the BLACS. If the reader wishes a more detailed knowledge of the shapes which can be generated by trapezoidal matrices, the BLACS User's Guide, [9], or [6] are good sources.

The packing of arrays (if required) so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than how the data is organized in the machine's memory. A detailed discussion on when such buffering is necessary is given in section 6.2.

2.2 Process Grid and Scoped Operations

The processes of a parallel machine with N_p processes are often presented to the user as a linear array of process IDs, labeled $0, 1, \ldots, N_p - 1$. For reasons described below, it is often more convenient to map this 1-D array of N_p processes into a logical two dimensional process mesh, or grid. This grid will have $\mathcal{P}$ process rows and $\mathcal{Q}$ process columns, where $\mathcal{P} * \mathcal{Q} = N_g \le N_p$. A process can now be referenced by its coordinates within the grid (indicated by the notation $\{p, q\}$, where $0 \le p < \mathcal{P}$, and $0 \le q < \mathcal{Q}$), rather than a single number. An example of such a mapping is shown in figure 1.

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Figure 1: 8 processes mapped to a 2 x 4 process grid.

An operation which involves more than just a sender and a receiver is called a *scoped* operation. All processes that participate in a scoped operation are said to be within the operation's scope.

On a system using a linear array of processes, the only natural scope is all processes. Using a 2D grid, we have 3 natural scopes, as shown in table 1

SCOPE	MEANING
Row	All processes in a process row participate.
Column	All processes in a process column participate.
All	All processes in the process grid participate.

Table 1: Scopes provided by a 2D process grid

These groupings of processes are of particular interest to the linear algebra programmer, since distributed data decompositions of a 2D array (a linear algebra matrix) tend to follow this process mapping. For instance, all of a distributed matrix row can be found on a process row, etc.

Viewing the rows/columns of the process grid as essentially autonomous subsystems provides the programmer with additional levels of parallelism. Of course, how independent these rows and columns actually are will depend upon the underlying machine. For instance, if the grid's processors are connected via ethernet, we can see that the only gain will be in ease of programming. Speed is unlikely to increase, since if one processor is communicating, no others can. If this is the case, process rows or columns will not be able to perform different distributed tasks at the same time. Fortunately, most modern supercomputer interconnection networks are at least as rich as a 2D grid, so that these additional levels of parallelism can be exploited.

The LU factorization (used to solve a systems of linear equations) can be used to illustrate the usefulness of the process grid. Figure 2 shows the basic steps of a right-looking LU factorization as they affect the data matrix. The first action in the algorithm is to form the panel of L as shown. A process column will cooperate to do this. This process column will then broadcast its portion of L along process rows. A process row will use this information and cooperate to form U. U is then broadcast within process columns, and all processes will use the values of L and U to find $\tilde{A}$.

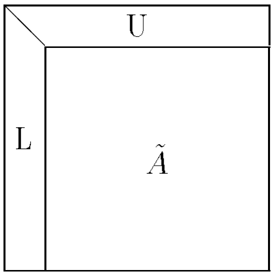

Figure 2: After first step of LU factorization

This is a very sketchy description of LU, and will likely confuse someone not familiar with the algorithm. A more complete analysis, which includes an examination of scalability, is given in [8].

A more detailed understanding of the process grid will be obtained as we discuss the various BLACS routines later in the paper. Also, in section 7.1.1, the extension of the scope idea to allow arbitrary groupings will be discussed.

2.3 ID-less Communication

One of the things that sets the BLACS apart from other message passing layers is that the user does not need to specify *message IDs*, (abbreviated msgid). A msgid (also referred to as a message type) is usually an integer which allows a receiving process to distinguish between incoming messages. The generation of these IDs can become problematic. A common mistake is to use a constant msgid within a loop, so that if one process takes longer than others to finish the loop, it may wind up receiving data from the next iteration as this iteration's data. This is just the most obvious way such msgid problems can happen. The same result can occur whenever non-unique IDs are used in any two sections of code not separated by an explicit barrier. These kinds of programming mistakes can lead to non-deterministic code which will finish correctly some of the time, give wrong results some of the time, and at other times simply crash.

Many parallel projects are too large for one person/team to write. This means that msgids must be coordinated between all routines and all writers of the package. If another routine is added at a later date, care must be taken to ensure that the new routine's IDs do not conflict with any other routine's.

Therefore, to add to the programmability of the BLACS, it was decided that the BLACS would generate the required msgids. These generated IDs had to have certain properties. First, it must never be the case that unrelated messages with the same destination would get the same ID. Second, in order to maintain performance, the ID generating algorithm had to use only local information: off-processor memory access could not be allowed. Further, it is necessary to allow for BLACS packages to be used alongside other communication platforms. An example that occurs regularly is linking a BLACS package with a machine specific package.

Therefore, the BLACS must allow the user to specify what range of IDs the BLACS can use. The user may do this by a call to the support routine `SHIFT_RANGE` (see the BLACS User's Guide for details).

By placing two restrictions on communication, these goals were achieved. First, a receiver must know the coordinates of the sending processor. Second, communication between two processes is strictly ordered. This means that if $\{0, 0\}$ sends two messages to $\{0, 1\}$, then $\{0, 1\}$ *must* receive them in the same order that they were sent. Section 6.1 discusses the BLACS msgid generating algorithms in detail.

2.4 Support Routines

The core BLACS routines are discussed later, in the specific section dealing with the various types of communication. There are a number of routines which do not deal directly with communication, however, that are required for programming in a parallel environment. The BLACS label these routines as *support* routines.

These routines return a process's grid coordinates, place barriers for rough synchronization, etc. Most of these routines are uninteresting as far as our discussion is concerned, and if the reader desires information about these routines, appendix C gives a quick reference to the BLACS. More detailed information about support routines can be obtained in the BLACS User's Guide.

There are two support routines important enough that they require discussion. They are the routines `GRIDMAP` and `BLACSINIT`, which are both used to create a process grid from a linear array of process IDs.

`BLACSINIT` creates a process grid with the first N_g process IDs distributed in the grid using a row-major natural ordering. Natural ordering implies that the order in which the IDs are dealt out is by increasing value. Row-major means that a row is consigned consecutive IDs. Column-major would similarly imply that a column is given IDs before going to the next column. Therefore, a row-major natural mapping results in the type of process grid shown in figure 1.

Most users will never need more flexibility than `BLACSINIT` provides. For users with more advanced needs, however, the function `GRIDMAP` exists.

`GRIDMAP` is a more general grid creation routine. It allows for arbitrary mappings of processes to the grid. This can be handy when row-major natural ordering does not supply nearest neighbor communication. If a machine has a hypercube interconnection network, for instance, a graycode mapping will be required to ensure grid neighbors correspond to physical network neighbors.

`GRIDMAP` does more than free one from row-major natural ordering. It also allows any of the available processes to be used for the grid, not just the first N_g processes. This paves the way for an important concept, referred to as *multigridding*.

Multigridding is the idea that within a program which has N_p available processes, there can be several grids performing separate tasks, at the completion of which the processes may become idle, join with another grid to make a larger grid, etc. None of our users have yet needed this feature, and it has not been tested. However, there are whole classes of problems where this kind of behavior is natural, and `GRIDMAP` is designed to support it.

If further information regarding these routines is desired, the BLACS User's Guide should be consulted. In addition, Appendix C provides a quick reference of all BLACS routines.

3 Point To Point Communication

3.1 Semantics

Point to point communication requires two complementary operations. The *send* operation produces a message, which is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer which holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking The return from the send or receive does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. I.e., a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking Return from the send or receive indicates that the resources may be re-used. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all (message will be lost).

Globally-blocking Return from a globally-blocking procedure indicates that the operation's resources may be reused, and that the operation's complement has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all machines support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel machines support locally-blocking sends, with buffering done on the receiving process. The CM-5 and SP1, however, possess globally-blocking sends.

This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Figure 3 shows a simple example of how this can occur.

```
IAM = MY_PROCESS_ID()

IF (IAM .EQ. 0) THEN
   SEND TO PROCESS 1
   RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
   SEND TO PROCESS 0
   RECV FROM PROCESS 0
END IF
```

Figure 3: Pseudo-code that hangs for globally-blocking sends

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and therefore waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will therefore never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. Section 6.2 discusses this in detail. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. Therefore, if process 0 sends three messages (label them A, B, and C) to process 1, process 1 *must* receive A before it can receive B, and message C can be received only after both A and B. The main reason for this restriction is that it allows for the computation of message identifiers, which is discussed in section 6.1.

It should be noted, however, that messages from different processes are not ordered. Therefore, if processes $0, \ldots, 3$ send messages $A, \ldots, D$ to process 4, process 4 may receive these messages in any order that is convenient.

3.2 Syntax

The names of the communication routines follow the template `vXXYY2D`, where the letter in the `v` position indicates the data type being sent, `XX` is replaced to indicate the shape of the matrix, and the `YY` positions are used to indicate the type of communication to perform. This is shown in table 2.

The calling sequences for these routines are:

```
vGESD2D(             M, N, A, LDA, RDEST, CDEST )
vGERV2D(             M, N, A, LDA, RSRC,  CSRC  )
vTRSD2D(UPLO, DIAG, M, N, A, LDA, RDEST, CDEST )
vTRRV2D(UPLO, DIAG, M, N, A, LDA, RSRC,  CSRC  )
```

The function of the parameters depends largely on whether the routine sends data (`vXXSD2D`) or receives (`vXXRV2D`) data. Output parameters are underlined. All other parameters are input, and thus not modified by the call.

Parameters:

`UPLO`	Specifies if matrix is stored as Lower or Upper trapezoidal matrix. See section 2.1 for details on trapezoidal matrices.
`DIAG`	Specifies if the matrix is unit diagonal. See section 2.1 for details on trapezoidal matrices.
`M`	Row dimension of matrix.
`N`	Column dimension of matrix.
<u>`A`</u>	Two dimensional array of data to be sent/received into. `vXXSD2D`: Array of data to be sent. `vXXRV2D`: Array where data is to be received.

vXXYY2D

v	MEANING
`I`	Integer data is to be communicated.
`S`	Single precision real data is to be communicated.
`D`	Double precision real data is to be communicated.
`C`	Single precision complex data is to be communicated.
`Z`	Double precision complex data is to be communicated.

XX	MEANING
`GE`	The data to be communicated is stored in a general rectangular matrix.
`TR`	The data to be communicated is stored in a trapezoidal matrix.

YY	MEANING
SD	Send. One process sends to another.
RV	Receive. One process receives from another.
BS	Broadcast/send. A process begins the broadcast of data within a scope.
BR	Broadcast/recv. A process receives and participates in the broadcast of data within a scope.

Table 2: Values and meanings of the communication routines' name positions

`LDA` Leading dimension of the array `A`.

`RDEST` `vXXSD2D`: Row index of destination process.

`CDEST` `vXXSD2D`: Column index of destination process.

`RSRC` `vXXRV2D`: Row index of source process.

`CSRC` `vXXRV2D`: Column index of source process.

As a simple example, the pseudo code given in figure 3 is rewritten in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector X, which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)

IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
   CALL DGESD2D(5, 1, X, 5, 1, 0)
   CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
   CALL DGESD2D(5, 1, X, 5, 0, 0)
   CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

3.3 Timings

One of the main reasons we present times in this report is for comparison with the system communication routines. In order to make these comparisons, we confine ourselves to using only a subset of the BLACS. For instance, most platforms support communication of contiguous data (vectors), so the timings are presented using general rectangular matrices with only one column. Broadcast and global timings are also restricted, and this is covered in their respective sections.

We present timings for four of the machines on which the BLACS run. Because the correct names of these platforms are rather long, table 3 gives a list of the four platforms and the abbreviations that refer to them. Table 4 gives a list of the system calls and the message passing layers used in the timings of not only the point to point, but also combines and broadcasts (abbreviated *bcast* in the table). Also listed is the routine used to synchronize processors (abbreviated *sync* in the table). Finally, appendix D contains the code used to obtain the times given throughout this report.

All of the tests given in this paper are repeated inside a timing loop in order to insure that the time being measured is above clock resolution. The number of times a given test was repeated (X) is indicated in each figure by the notation *reps*=X.

There are two times associated with point to point communication. These times are T_c (time for a complete communication) and T_s (time to post a send). A third quantity, T_m, the time to perform a malloc and memory-to-memory copy, will be of interest on those platforms where buffering is done in order to perform locally-blocking sends.

We define T_c as the time lapse between a sender process initiating a send, and the waiting receiver returning from the blocking receive. To find this value, an "echo" test is performed. In this test, one processor is the sender. All other processors will only echo back what is sent. The sender sends a wake-up message to a waiting echo processor. This tells the echo processor to start its receive. The sending processor now starts its timer, and loops over a send followed by a receive. The echo processor loops over a receive followed by a send. When they have done this the required number of times, the sender stops his timer, and divides the time by the number of repititions to get the time for one T_c. This test was originally proposed in [13], and that paper provides further details.

It should be noted that this measurement favors platforms with globally-blocking sends. The strength of globally-blocking sends is that no buffering is required. Its weakness is that if the receiver posts the receive after the corresponding send is posted, the sending processor must wait. In the echo test, the receive will always be posted at about the same time as the send. This means that the delays inherent in using globally-blocking sends will not show up in this test. Locally-blocking sends will still have to pay the cost for their buffering, but the ability for the sender to return before the receive is posted is not utilized.

Therefore, on systems where the native send is globally-blocking, the BLACS times (which use the BLACS locally-blocking sends) will appear to be much worse than the system's times. On such systems, the BLACS use buffering to create locally-blocking sends. This requires the BLACS to allocate a buffer (if the BLACS don't have one of the correct size available), and copy the entire message on each send (see section 6.2.2 for details). Therefore, on the CM-5 and SP1 (the systems with globally-blocking sends), we measure T_m as well.

The final time of interest is the time it takes post a send, T_s, i.e., the time from when the send routine is called, until it returns. This value may be less than T_c, especially if non-blocking or locally-blocking sends are used. If $T_s < T_c$, this is of obvious interest to optimizing codes. T_s becomes even more important when we discuss the times in broadcasts and combine operations.

The times presented throughout this paper were taken while no other users were present on the machines. When this is not the case, communication times can vary, with the degree of variance determined by the platform, and what the other users' processes are doing.

On each platform our sample range is from $0 \ldots 50,000$ double precision elements ($0 \ldots 400,000$ bytes), with a data point taken every 1000 double precision elements. This is our primary range of interest, a range which we believe most communication falls within. For this primary range, we check the reproducibility of our codes by re-running every fifth data point ten additional times. The distribution of these repeated data points gives us an idea of how accurate/reproducible our timings are. A least squares fit is then done on all of these data points to arrive at α and β.

We now have repeated data points at 11 different values of N within our range. If we label the i^{th} ($i \leq 11$) set of repeated data points as $D_r(i)$, then the relative error is given by $\text{RE} = \max_{1 \leq i \leq 11}(\frac{\max(D_r(i)) - \min(D_r(i))}{\text{average}(D_r(i))}) * 100.0$. After all of the timings had been taken, it was discovered that we had failed to set the value of reps high enough to measure the $N = 0$ data point on many systems. Therefore the RE calculations do not include the $N = 0$ data point.

We will perform a least squares fit on all of these times, so that we express each as a linear model. Therefore, each time will be written as $T = \alpha + \beta N$, where N is the number of double precision elements being communicated. If these models were completely accurate across the entire range, α would be latency, or startup cost, and β would be the cost/element of the operation. However, the cost/element does not remain constant across the range. This is especially true of data copying, where pipelining occurs. Indeed, we find that doing a least squares fit on the copy times yields a negative α, due to this pipelining. This results in those locally-blocking sends which use the memory copy also having negative α's on this range.

Therefore, we find a second range of interest. Thus the range $N = 0, \ldots, 1000$ is also timed, but since it is not our primary range, the relative error calculations are not done for it.

When modelling T_c, we find that α is constant between two given processors, but may vary depending on the distance between the processors. The T_c timings shown here are all for nearest neighbor communication. Detailed study of each individual machine is beyond the scope of this paper, and so non-nearest neighbor times are not investigated.

For all platforms T_c and T_s are given for the machine's communication primitives and for the machine's BLACS.

Figures 4, 5, 6, 7, 8, 9, 10, and 11, show the timings for T_c and T_s for each platform. Viewing these graphs should give the user an intuitive handle on the relevance of the quantities RE, α and β, which we will also specify. In the interest of conserving space, only the numbers are given for T_m.

Tables 5, 6, and 7 show the *alpha* and *beta* for the least squares fit of the T_c, T_s and T_m data on our primary range of $N = 0, \ldots, 50,000$. These tables also give the relative error of the data (not of the least squares fit).

Table 6 summarizes the T_s timings. We see that little is lost on the i860, Paragon, or SP1 platforms. The CM-5 loses some performance.

On the CM-5, both T_c for locally-blocking sends and T_m show a negative α. This is because packing displays pipelining: the operation is much more efficient for long vectors than for short. Therefore, we see that our approximation is inaccurate at the beginning of our curve. For our purposes in this paper, the linear model of the large range is what we want: we wish to see values over this whole range, and are not as concerned about the first few data points. However, since the reader may be interested the smaller range, we also present some numbers obtained in timings over the range $0, \ldots, 1000$. Since this is not our target range, we did not re-run the measurements to get a relative error.

The most important quantity, T_c, is summarized in table 5. On the i860, only the latency is affected, which is as expected. On those machines natively possessing locally-blocking sends, the BLACS just add a few routine calls, a message ID computation, etc., to the cost of the send/receive. Since none of these things depend on N, only α should be affected on those platforms. Notice that on the Paragon, β also seems to be changed. It is unsure if this is true, or if this is just a timing artifact, as timings on the Paragon tend to be chaotic. Notice that the number of repetitions is set to 1000 for this platform. Even with such a high number of repetitions, anomalous behavior is sometimes observed. At fewer repetitions, the BLACS routinely seemed faster than the system (a patent impossibility). At any rate, the curves indicate that the times are quite comparable.

On the SP1 and CM-5, we see that the effective bandwidth changes, as does the latency. This is due to the data copying mentioned previously. We see that the CM-5 shows the effect far more than the SP1. The CM-5 has weak processors (spark-2's), and a fairly decent

SYSTEM	Abbr.
Intel iPSC/860	i860
Intel Paragon XP/S	Paragon
Thinking Machine's CM-5	CM-5
IBM Scalable POWERparallel System 1	SP1

Table 3: Names and abbreviations for timing platforms

	i860	PGON	CM-5	SP1
Mssg. Passing Layer	NX 3.3.2	OSF/1 v 1.2β	CMMD 3.1	MPL (AIX 3.2.4 PE software 1.1/PTF 1)
send	csend	csend	CMMD_send_block CMMD_send_noblock	mpc_bsend
receive	crecv	crecv	CMMD_receive_block	mpc_brecv
bcast/send	csend	csend	CMMD_bc_to_nodes	mpc_bcast
bcast/recv	crecv	crecv	CMMD_receive_bc_from_node	mpc_bcast
combine	gdsum	gdsum	CMMD_scan_v	mpc_combine
sync	gsync	gsync	CMMD_sync_with_nodes	mpc_sync
send(BLACS)	csend	csend	CMMD_send_async	mpc_send
recv(BLACS)	crecv	crecv	CMMD_receive_block	mpc_brecv

Table 4: System calls used for timings

SYSTEM	BLOCK	SYSTEM TIMES			BLACS TIMES		
		α	β	Rel Err	α	β	Rel Err
i860	Local	200	2.8576	0.06%	211	2.8576	5.93%
Paragon	Local	212	0.2049	10.76%	209	0.2098	6.76 (32.26)%
CM-5	Global	169	0.9118	1.90%	N/A	N/A	N/A
CM-5	Local	-159	1.4787	3.09%	-72	1.6246	280%
SP1	Global	772	1.1400	12.35%	712	1.2463	6.62(40.45)%

Table 5: Least squares fit for T_c times (in microseconds), $N = 0, \ldots, 50000$

SYSTEM	BLOCK	SYSTEM TIMES			BLACS TIMES		
		α	β	Rel Err	α	β	Rel Err
i860	Local	48	2.8573	6.73%	47	2.8573	5.29%
Paragon	Local	109	0.1957	4.88%	129	0.2001	1.50%
CM-5	Global	150	0.9164	7.01%	N/A	N/A	N/A
CM-5	Local	173	1.4787	17.37%	337	1.5997	4.48%
SP1	Global	524	1.1165	23.56%	42	1.2316	10.99%

Table 6: Least squares fit for T_s times (in microseconds), $N = 0, \ldots, 50000$

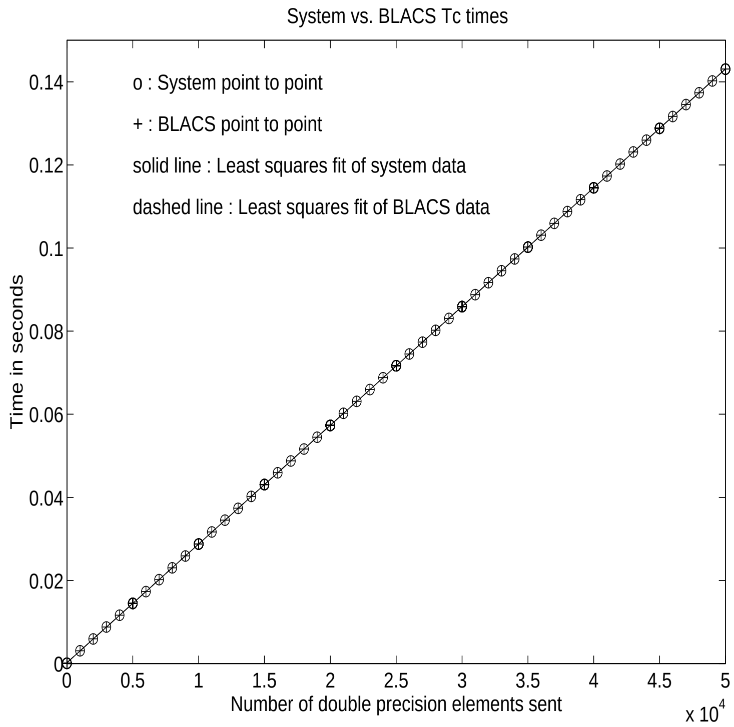

Figure 4: T_c for nearest neighbor communication on the Intel i860, reps=15

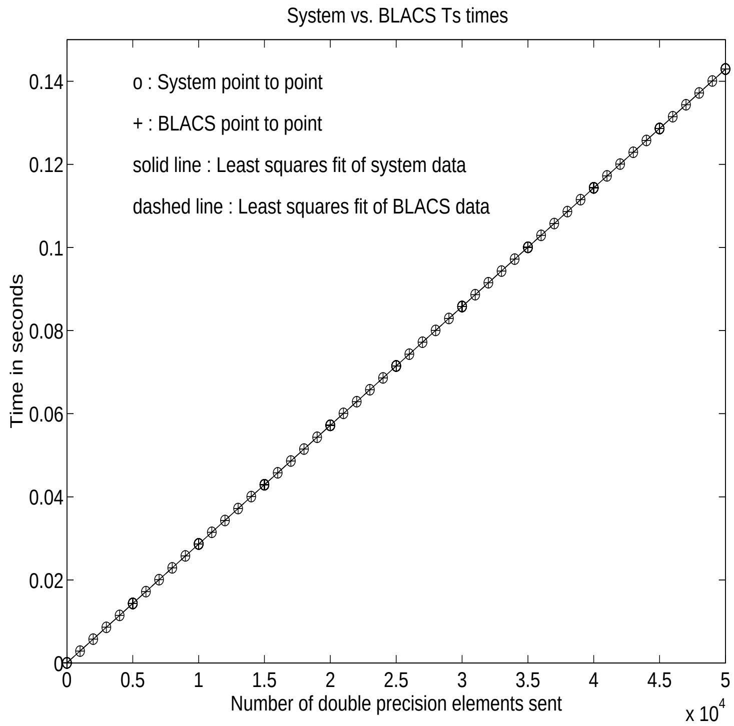

Figure 5: T_s on the Intel i860, reps=15

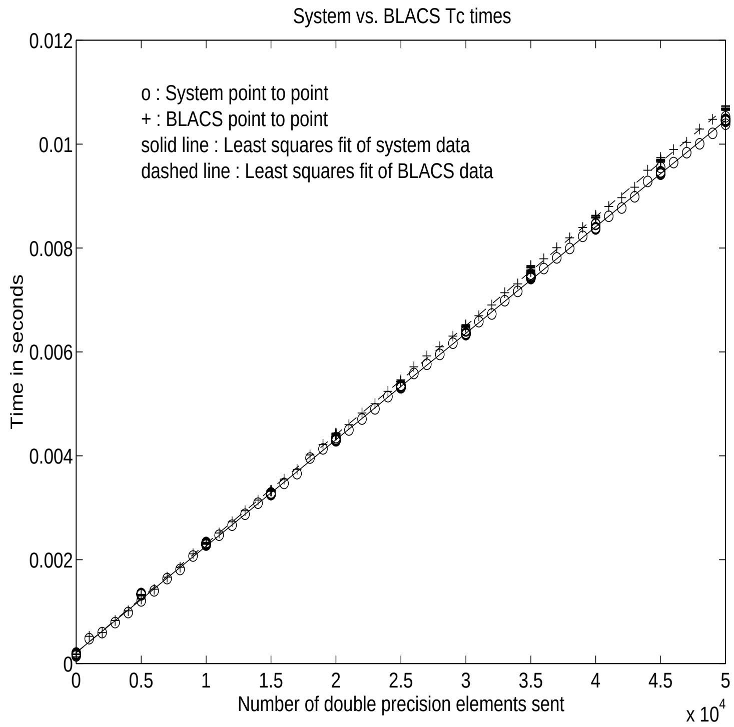

Figure 6: T_c for nearest neighbor communication on the Paragon, reps=1000

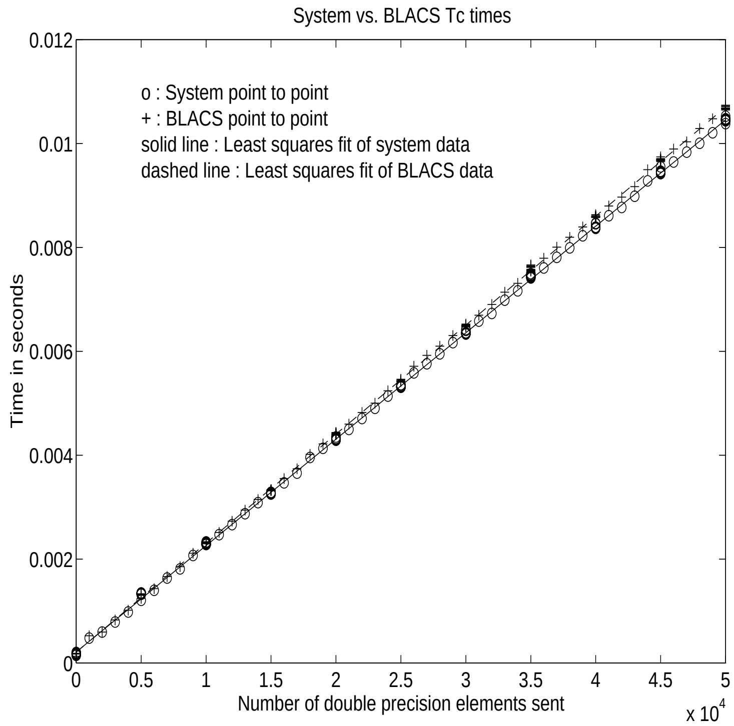

Figure 7: T_s on the Paragon, reps=1000

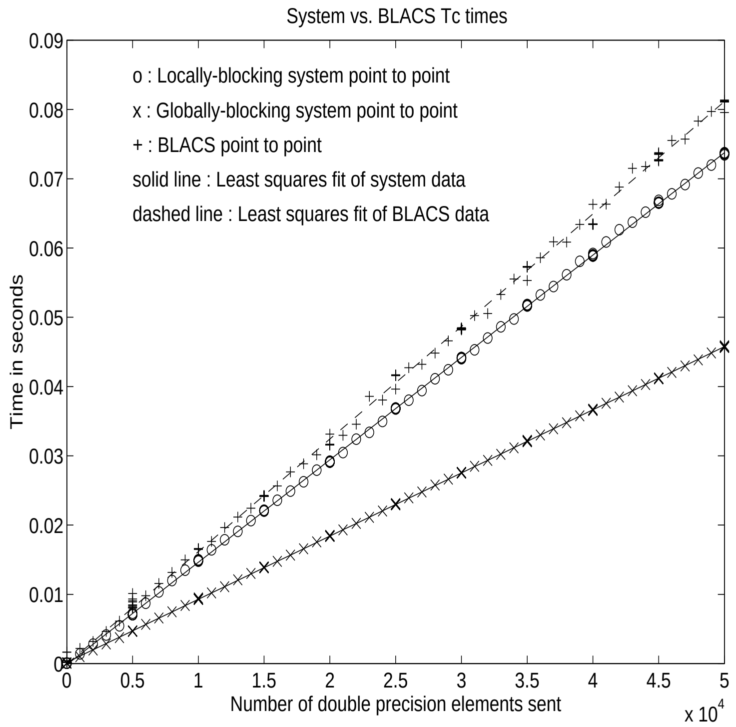

Figure 8: T_c for nearest neighbor communication on the CM-5, reps=30

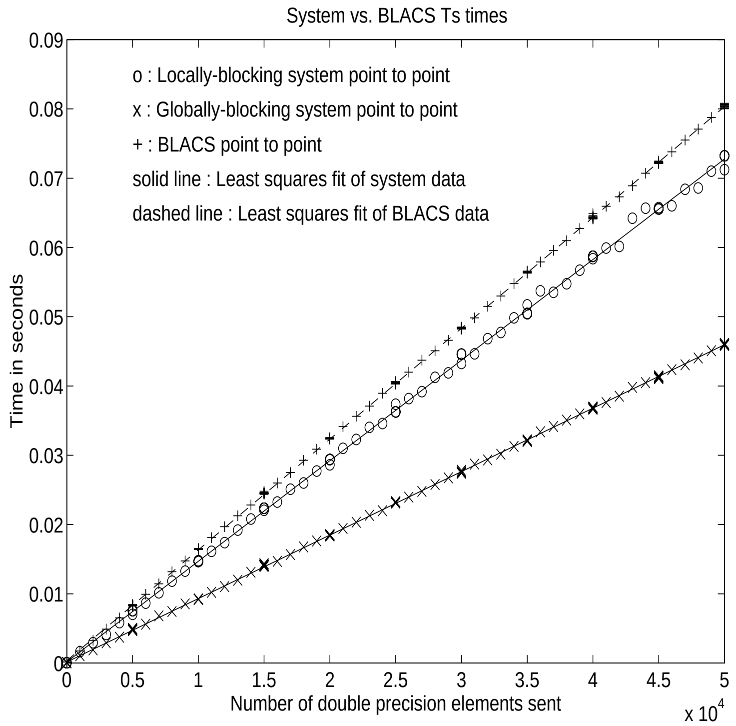

Figure 9: T_s on the CM-5, reps=30

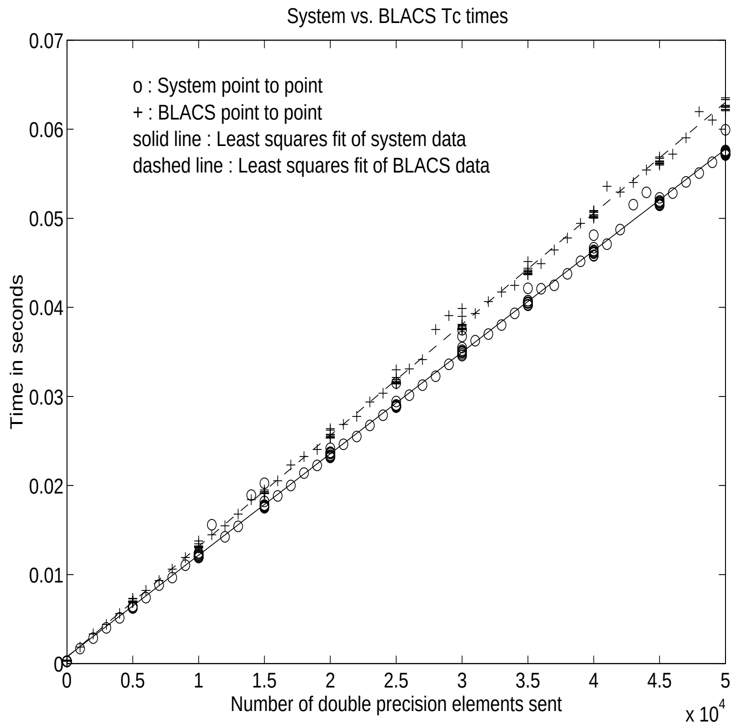

Figure 10: T_c for nearest neighbor communication on the SP1, reps=30

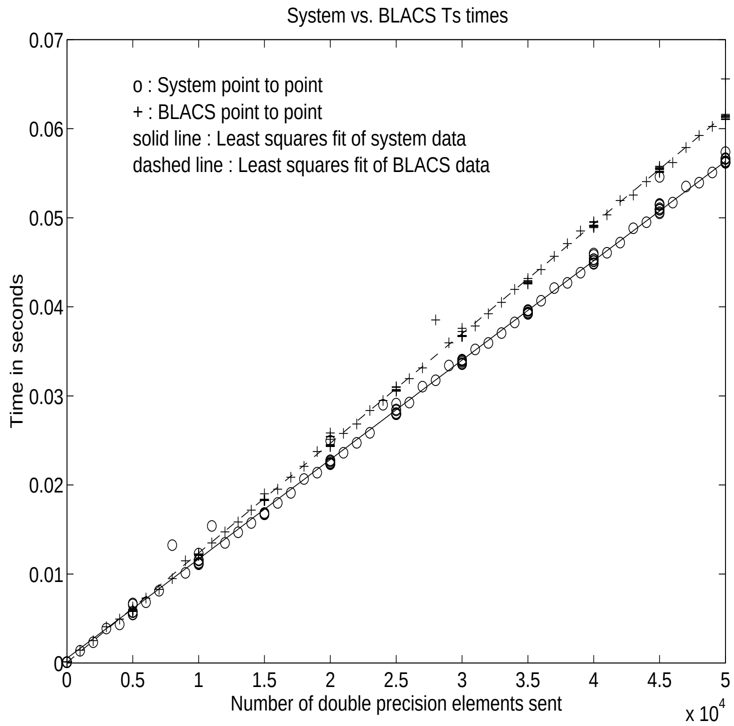

Figure 11: T_s on the SP1, reps=30

network. This means that the data copy costs us much more than on the SP1. The SP1 possesses RS6000's as its processors, and has a very high latency communication network. This means we lose less by doing the data copy. It should be noted that the CM-5, unlike the SP1, does offer a locally-blocking send. When the BLACS were first implemented on the platform our timings seemed to indicate that the locally-blocking system send was less efficient than the BLACS' locally-blocking send. Since that time, the BLACS have changed versions, and so has the CM-5's message passing library. We see that the CM-5's locally blocking send is now superior to that used in the BLACS, and so the next version of the BLACS will probably use the system's locally-blocking send.

These numbers are worst case for the BLACS. First, as previously mentioned, the nature of the T_c test favors globally-blocking sends. Secondly, if the user is sending a non-contiguous 2D array or a trapezoidal array, this extra buffering has to be done even for a globally blocking send, and the times are then unaffected.

A final note is in order here. We will later use these linear models to predict the speed of the various scoped operations presented later. We have seen that, especially on the CM-5, point to point messages have become slower due to the buffering required to support locally-blocking sends. However, we do not use these point to point times when we predict the speed of the BLACS scoped operations. The data copy is only done once, so we will use one T_m, and then use the system's T_c and T_s to predict scoped times. This corresponds to what is done in the code: the starting process does the buffering, and all other processes send/receive/operate on the already packed data.

Table 8 shows T_c and T_s for the $N = 0, \ldots, 1000$ range. Note that the SP1 and Paragon times should not be considered reliable. If the graph of these two machines on this range is viewed, it is apparent that the least squares fit is not very accurate.

All of these tables and graphs boil down to the result that the BLACS are extremely competitive on all platforms except where they must support locally-blocking sends. The SP1 does not see too large a slowdown, leaving the CM-5 as the only platform where true losses occur. We believe the added programmability of locally-blocking sends makes the loss worthwhile. Also, it appears we can do quite a bit better by simply using the CM-5's native locally-blocking send, so the next version of the BLACS should be more competitive.

4 Broadcasts

4.1 Semantics

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast/send*. All processes within the same scope must then issue the complementary *broadcast/receive*.

The BLACS define that both broadcast/send and broadcast/receive are globally-blocking. Broadcasts/receives cannot be locally-blocking since they must post a receive (remember that receives cannot be locally-blocking). When a given process can leave a broadcast/receive operation is topology dependent, so, in order to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be locally-blocking. Since no information is being received, as long as we use locally-blocking point to point sends, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, i.e., all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with `SCOPE = 'ALL'` must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

4.2 Syntax

The calling sequence for these routines is:

```
vGEBS2D( SCOPE, TOP,             M, N, A, LDA              )
vGEBR2D( SCOPE, TOP,             M, N, A, LDA, RSRC,  CSRC  )
vTRBS2D( SCOPE, TOP, UPLO, DIAG, M, N, A, LDA              )
vTRBR2D( SCOPE, TOP, UPLO, DIAG, M, N, A, LDA, RSRC,  CSRC  )
```

The function of the parameters depends largely on whether the routine sends data (`vXXBS2D`) or receives (`vXXBR2D`) data. Output parameters are underlined. All other parameters are input, and thus not changed inside the routines.

Parameters:

`SCOPE`	Scope of processes to participate in operation. Limited to `'ROW'`, `'COLUMN'`, or `'ALL'`. See section 2.2 for additional details.
`TOP`	Network topology to be emulated during communication. Topologies presently supported are discussed in section 4.3
`UPLO`	Specifies if matrix is stored as Lower or Upper trapezoidal matrix. See section 2.1 for details on trapezoidal matrices.
`DIAG`	Specifies if the matrix is unit diagonal. See section 2.1 for details on trapezoidal matrices.
`M`	Row dimension of matrix.
`N`	Column dimension of matrix.
<u>`A`</u>	Two dimensional array of data to be sent/received into. `vXXBS2D`: Array of data to be sent. `vXXBR2D`: Array where data is to be received.
`LDA`	Leading dimension of the array `A`.

RDEST vXXBS2D: Row index of destination process.

CDEST vXXBS2D: Column index of destination process.

RSRC vXXBR2D: Row index of source process.

CSRC vXXBR2D: Column index of source process.

As described above, the parameters M, N, and LDA dictate the shape of the array being communicated. All processes participating in a given send operation or its receive complement must have the same amount of array space available (i.e. M * N must be the same). However, it is not necessary that they all receive the data in the same way (this holds true for point to point communication, as well). An example should help illustrate this principle:

Process {0,2} has a double precision matrix B, with a total size of 500 x 200. All the other processes in its process column require five rows and seven columns of this matrix starting at the matrix position (9,4). Process {0,2} would make the following call to the BLACS:

```
DGEBS2D('COLUMN', 'HYPERCUBE', 5, 7, B(9,4), 500)
```

Since process {0,2} has initiated a broadcast, the processes $\{i, 2\}, i = 1, 2, \ldots, \mathcal{P}-1$ *must* call DGEBR2D. However, their receive calls need not be exactly the same. For instance, process {1,2} might want to receive the information into a work vector, WORK. It would make the following call:

```
DGEBR2D('COLUMN, 'HYPERCUBE', 5, 7, WORK, 5, 0, 2)
```

The other processes in the process column could receive the message into their copy of B with the following command:

```
DGEBR2D('COLUMN', 'HYPERCUBE', 5, 7, B(9,4), 500, 0, 2)
```

NOTE: all versions of the BLACS except PVM allow the user to vary M and N, as long as M * N is the same across all processes. However, in PVM the data must be unpacked in the same manner that it is packed. Therefore, the shape of the matrix being communicated should be changed only by varying LDA.

4.3 Topologies

The topology parameter determines how the messages involved in a distributed operation are sent. The use of the topology idea allows the user to exploit the following fact: even if the time to perform a distributed operation cannot be reduced, which processors bear the brunt of the cost of the operation *can* be varied.

Many factors go into deciding which topology is optimal. First, the user must decide if any processor is more important than others. For instance, if the source processor's time is more important than other processors', a ring topology is often optimal. On the other hand, if everyone needs the information quickly, some type of tree is often best.

Some topologies tie up the sending processor for large amounts of time, and different processors get the information at different times depending on topology. Also, some topologies are "noisy", i.e. many communications are issued simultaneously, while others are "quiet". Noisy algorithms will cause problems on systems where network conflicts are problematic. Quiet algorithms are likely to force some processors to wait much longer than they would if a "noisy" topology had been used, since less communication is going on in parallel.

Some topologies are "pipelining", i.e., the first such operation synchronizes the processors so that subsequent operations will be cheap.

In the discussion of the presently supported topologies is given below, we use the following symbols: N_p, the number of processors involved in the operation, T_s, the time to send a message, and T_c, the time for a complete communication (send and receive).

All figures displaying communication patterns are shown with $N_p = 8$, because this size is adequate to show off the features of the topologies, and is still small enough to fit into a reasonable amount of space. Further, the processors are numbered from $0, \ldots, (N_p - 1)$. We do not specify grid coordinates because these broadcasts can operate on rows or columns, or the entire grid. If we instantiate such a picture as a row broadcast, for instance, these values are column indexes. For ease of reference, we will still refer to a given index as "processor I", but this should be taken to mean the processor at the I'th position in a row, a column, or in the grid. Please note as well that the term processor has now replaced process. We present timing analysis in this section, and they will not be accurate if more than one process is spawned to a given processor.

To be consistent, processor 0 is always shown as the source/dest of the broadcast/combine. Finally, a label `S = I` to the left of a figure indicates that the algorithm is in the I'th step. For the time analysis discussed in the text, it is assumed the BLACS are operating in an environment where an arbitrary number of processors may be communicating simultaneously. This assumption will affect the accuracy of our prediction if the number of actual links is less than those assumed by the algorithm.

At the present time there are two classes of broadcast topology. The first class involves topologies based on rings. The second classification consists of topologies based on trees. Within these classes, there are several different algorithms, which differ slightly from each other. For ring topologies, the main differences involve which direction within the ring messages flow (increasing/decreasing), and the number of rings the scope is separated into (N_r). For tree topologies, the main variables involve the number of branches (N_b) at each node of the tree, and which branch is sent to first.

These classes are explained in detail below, and table 9 provides a quick summation of some of the more important properties. This table specifies the number of steps until the algorithm completes (STEPS), the number of messages sent during step i (SENDS, $S = i$), the number of processors who are finished with the routine after step i is complete (PROCS DONE, $S = i$), the time the source processor spends in the algorithm (SRC TIME), and finally the maximum time spent by any processor in the operation (MAX TIME). The analyses shown in table 9 have been simplified by assuming that N_r is an even multiple of N_p, and $N_b = 1$, with N_p an integer multiple of 2. The specific topology section should be examined for full details.

4.3.1 Broadcast Ring Topologies

The various ring topologies are discussed below. All of these topologies can experience pipelining of various degrees. Our timing models assume that processors are roughly synchronized when entering the broadcast. However, when a ring broadcast is performed, it forces an obvious ordering onto the processors; i.e, the first processor in the ring will leave the operation before the processor which follows it in the ring. This means that once the

SYSTEM	T_m		
	α	β	Rel Err
CM-5	-152	0.7248	1.54%
SP1	-58	0.1072	0.40%

Table 7: Least squares fit for T_m times (in microseconds), $N = 0, \ldots, 50000$

LIBRARY	BLOCK	T_c		T_s	
		β	α	β	α
Gamma System	Loc	2.92	162	2.845	81
Gamma BLACS	Loc	2.92	173	2.844	82
Paragon System	Loc	0.22	121	0.27	49
Paragon BLACS	Loc	0.22	140	0.28	52
CM-5 System	Glob	0.92	75	0.91	62
CM-5 System	Loc	1.32	87	1.32	123
CM-5 BLACS	Loc	1.49	150	1.45	183
SP1 System	Glob	1.26	448	0.97	449
SP1 BLACS	Loc	1.36	494	?1.12?	484

Table 8: Least squares fit for point to point times (in microseconds), $N = 0, \ldots, 1000$

	N_r–RING	1–TREE
Steps	N_p/N_r	$\log_2(N_p)$
SENDS, $S = i$	N_r	$(2)^i$
PROCS DONE, $S = i$	$1 + N_r * i$	0
SRC TIME	$N_r * T_s$	$2\log_2(N_p) * T_s$
MAX TIME	$((N_p-1)/N_r) * T_c + (N_r - 1) * T_s$	$\log_2(N_p) * T_c$
PIPELINING?	YES	NO

Table 9: Broadcast topology highlights

cost of the first broadcast is payed, the processors are optimally ordered to perform another ring broadcast. The time each processor pays for the second broadcast will be roughly T_c ($T_c + T_m$ if the BLACS are supporting locally-blocking sends via buffering), rather than that given in the text. Therefore, whenever a given processor is to issue several consecutive broadcasts, use of a ring topology should be considered. It will result in minimization of the sender's time as usual, but since the ordering cost is payed only once, it may result in faster overall transfer rates as well.

Pipelines can be maintained if the algorithm flows across processors in an orderly way. I.e., if the sender of row broadcasts starts out as the first process column, and then is the second, etc, an increasing ring pipeline will be maintained. If the flow is in the opposite direction, it may be possible to set up a decreasing ring pipeline. The affects of pipelining on broadcast times will be discussed in greater detail after all ring-based topologies have been explained.

Unidirectional Ring Unidirectional ring topologies require the source processor to issue one broadcast, and each processor then receives and forwards the message. The two unidirectional ring topologies are increasing ring (`TOP = 'I'`), and decreasing ring, (`TOP = 'D'`). These algorithms have the advantage that the originating processor must spend only T_s time in the broadcast. However, the last processor in the ring will spend $(N_p - 1) * T_c$ time in algorithm. Figures 12 and 13 respectively show increasing and decreasing ring broadcast. Unidirectional rings are the "quietest" algorithms possible: only one processor is sending at a time.

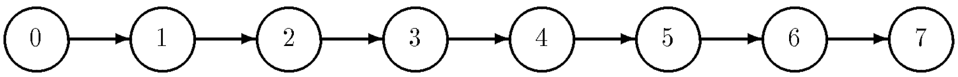

Figure 12: Increasing ring broadcast

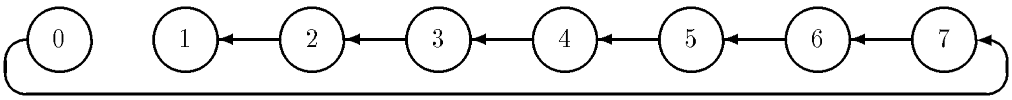

Figure 13: Decreasing ring broadcast

Split Ring The split ring attempts to alleviate the long waiting time inherent in unidirectional rings, without tying up the originating node. Examining figure 14 should convince the reader that the longest time spent in the algorithm is roughly $\lfloor P/2 \rfloor * T_c$, and that the source spends $(2 * T_s)$ time in broadcast. The split ring topology is called by `TOP = 'S'`. Although it is unlikely to be important in all but the most critical of optimizations, the user should know that the split ring sends in the increasing direction first. This is a relatively "quiet" algorithm as only two processors will be sending at any one time.

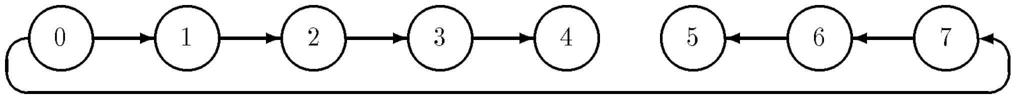

Figure 14: Split ring broadcast

Multiring The multiring algorithm (also referred to as multipath) provides a scalable ring algorithm. By definition, the graphs created by a multiring topology is not a ring at all, but is instead a special kind of tree. We call it a ring topology despite this, because it behaves like the true ring topologies: pipelining may occur, and maximum time in the algorithm scales linearly with the number of processors involved.

In this algorithm, the user provides the number of rings (N_r) the broadcast is to proceed on. The processors participating in the broadcast are then split up into N_r separate increasing rings. Figure 15 shows a multiring with $N_r = 3$. Note that the source sends to the closest ring first, and the farthest ring last. This may seem counter-productive, in the sense that if we would like to minimize link contention, sending the to far ring first makes more sense. However, ring topologies are most useful in pipelined codes, where, since the flow of the algorithm proceeds in one direction across the processors, the time spent by the nearer processors is more important than that of the far processors.

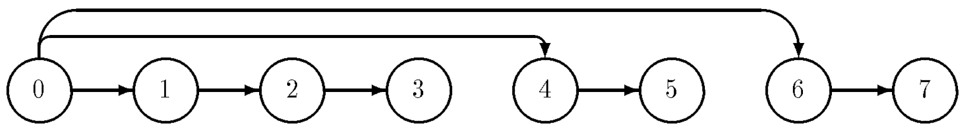

Figure 15: Multiring broadcast with $N_r = 3$

This algorithm requires $\lceil N_p/N_r \rceil$ steps, and at each step N_r sends will be initiated. The source processor is finished after the first step, and N_r processors finish each step thereafter. The source processor must send to all rings, and so its time in the algorithm should be $N_r * T_s$. If N_r does not evenly divide N_p, some rings must be longer than others. This topology specifies that if $N_{lr} = (N_p - 1) \bmod N_r$, then the first N_{lr} (number of long rings) rings will get an extra processor. With this in mind, it is easily seen that the processor who must wait the longest for this algorithm is either the ending processor in the last long ring, or, if all rings are of the same length, the last processor in the last ring. Therefore, if all rings are of equal length, the longest wait time will be $((N_p - 1)/N_r) * T_c + (N_r - 1) * T_s$. Otherwise, it will be $\lceil (N_p - 1)/N_r \rceil * T_c + ([(N_p - 1) \bmod N_r] - 1) * T_s$. Note: it is possible that if N_r is large, enough T_s's could build up to make it so that the last ring waits longer than the last long ring. This is a detail, and should make no real difference.

Most instantiations of the multiring topology will be relatively "quiet", since at worst N_r processors will be sending at the same time.

Calling the multiring algorithm is more complicated than the less general algorithms described above. Not only must a topology be selected, but a number of rings must be passed to the BLACS. Therefore, there is an support routine, `SETBRANCHES`, which allows the user to set an internal variable in the BLACS describing the number of rings to use. Multiring is called by setting `TOP = 'm'`. Here is an example of the recommended way to call the multiring topology:

```
call setbranches(3)
call dgebs2d('Row', 'm', m, n, A, lda)
⋮
call dgebs2d('Column', 'm', 3, 2, work, 5)
```

Notice that `SETBRANCHES` need only be called when changing N_r, therefore, in the example above, both the row and column broadcasts will split their processors into 3 increasing rings.

Pipelining All ring-based topologies can display pipelining. However, as the number of rings (N_r) increases, the pipeline advantage tends to decrease. After a ring broadcast, each separate ring is correctly pipelined with respect to the processors within its ring, but not with the source processor. As the source processor sends more and more messages, this lack of synchronization becomes worse. An example illustrates this principle. Assume we have just finished a N_r-ring broadcast. At this point the maximum cost payed is that given in the topology description above (call this time T^1). We then repeat this broadcast k times. If we have a 1-ring, all processors are synchronized so that the total cost is just $T^1 + k * T_c$. If $N_r > 1$, however, for each iteration beyond the first we pay the T_c cost, plus the cost of the other sends the source has had to issue before sending to our ring again. Thus, in general, the cost is $T^1 + k * (T_c + (N_r - 1)T_s)$ (or $T^1 + k * [T_m + T_c + (N_r - 1)T_s]$, if the BLACS are buffering to support locally-blocking sends).

4.3.2 Broadcast Tree Topologies

Hypercube The first tree-based topology is called *hypercube*. This algorithm is a specialized broadcast which matches the Intel i860's hypercube network. It uses bit level operations to achieve low overhead in computing source and destination of messages. It was originally coded by Robert van de Geijn[11, 12], and only slightly modified for inclusion in the BLACS. This topology requires that N_p be an integer power of 2. If it is not, the general tree algorithm described below is called instead. A final detail is that at each node in the tree, messages are sent to the nearest node first. This broadcast strategy is shown in figure 16.

Hypercube broadcasts are most useful when getting the information out to all processors is more important than saving origin node time. It requires the origin node to spend $T_s * log_2(N_p)$ time in the broadcast. However, the longest any node need wait is $T_c * \log_2(N_p)$. Hypercube broadcasts are relatively "noisy", since the number of processors sending at one time grows with N_p. In the last step of the broadcast, $N_p/2$ processors will be sending simultaneously.

General Tree The final topology that is supported is the general tree broadcast. It allows the user to choose the number of branches (N_b) at each step in the broadcast tree. Figures 17, 18 and 19 show general tree broadcasts with $N_b = 1, 2, 3$. Note that general tree with $N_b = 1$ is a hypercube broadcast where at each node in the tree, the node furthest from the present node is sent to first. This tends to minimize link contentions, if the assumption is made that processors far away from each other tend not to share the same link.

With this algorithm, N_p does *not* have to be an integer power of N_b. Since N_b varies, more terminology is required to discuss this algorithm. The height of the tree required to finish the broadcast will be $H_t = \lceil log_{N_b+1}(N_p) \rceil$. The number of initial sends (sends done by origin node in $S = 0$) is given by $N_{is} = \lceil N_p/(N_b + 1)^{(H_t-1)} \rceil - 1$. With these quantities defined, it can be shown that the time the origin node spends in the broadcast is $[N_b(H_t - 1) + N_{is}] * T_s$. The longest time any processor spends in the algorithm is: $[(H_t - 1)(N_b - 1) + (N_{is} - 1)] * T_s + H_t * T_c$. General tree broadcasts are obviously "noisy", and the greater N_b and N_p are, the more "noisy" the algorithm becomes. This topology may be called in several ways. If the user sets `TOP = 't'`, the routine `setbranches` should be used in the same way as discussed for multiring. An example should clarify this:

```
call setbranches(2)
call dgebs2d('Row', 't', m, n, A, lda)
```

This would call the general tree algorithm with $N_b = 2$. Table 10 summarizes the ways to call the general tree broadcast.

TOP	Explanation
'1'	tree with $N_b = 1$.
'2'	tree with $N_b = 2$.
'3'	tree with $N_b = 3$.
'4'	tree with $N_b = 4$.
'5'	tree with $N_b = 5$.
'6'	tree with $N_b = 6$.
'7'	tree with $N_b = 7$.
'8'	tree with $N_b = 8$.
'9'	tree with $N_b = 9$.
't'	tree with $N_b =$ `I`,where `I` is set by `CALL SETBRANCHES(I)`.
'f'	perform fully-connected broadcast: source sends to all participating processes

Table 10: General tree topology entry points

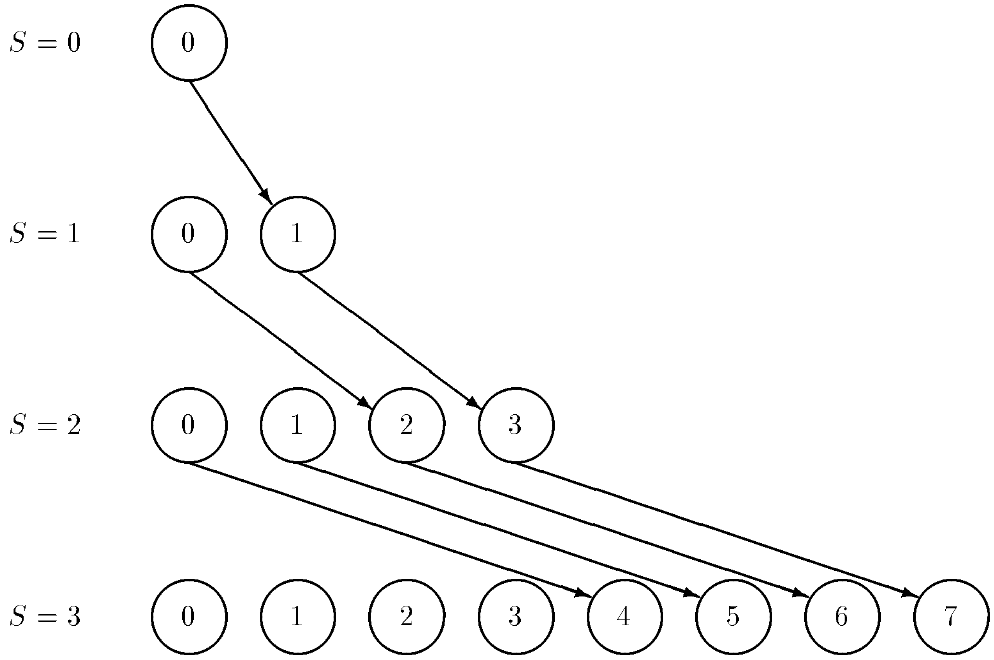

Figure 16: Hypercube broadcast, nearest node first.

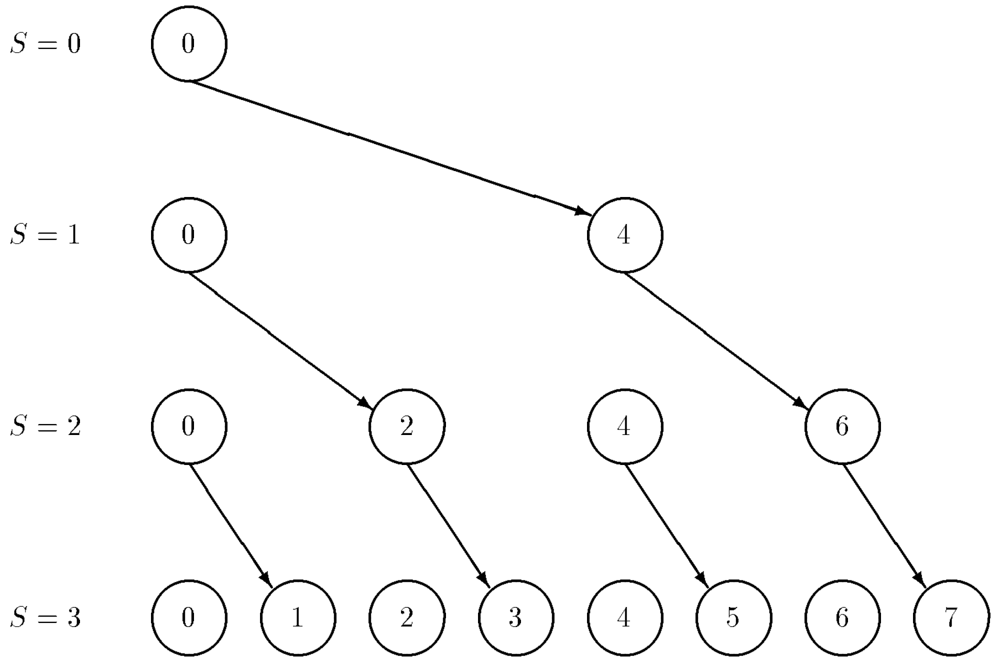

Figure 17: General tree broadcast with $N_b = 1$

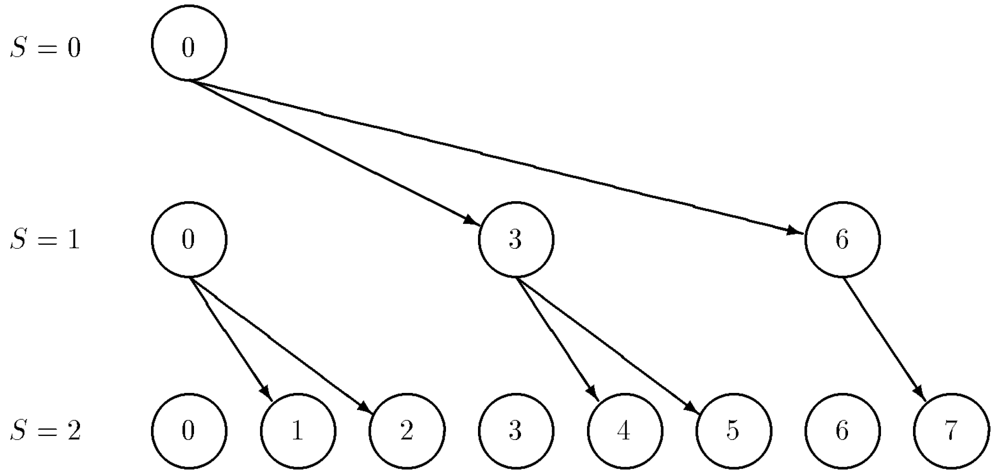

Figure 18: General tree broadcast with $N_b = 2$

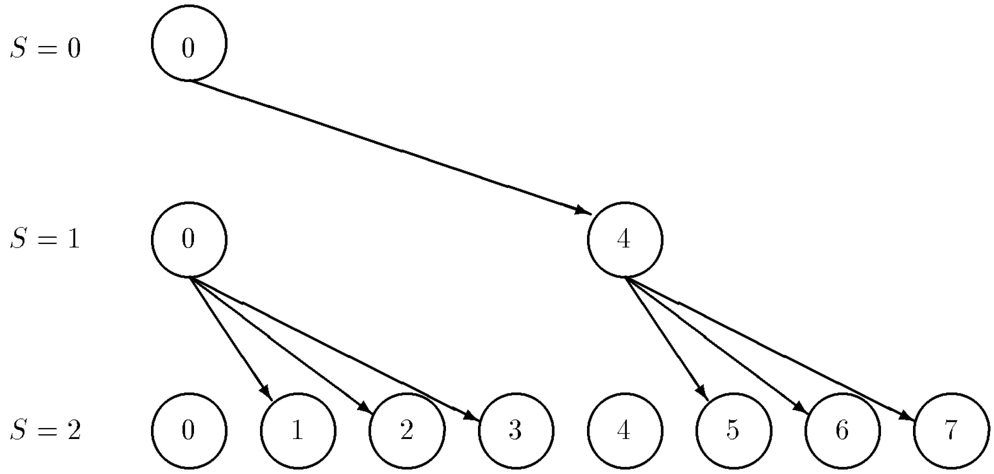

Figure 19: General tree broadcast with $N_b = 3$

4.4 Timings

Here we present the broadcast times for each machine. Two times are presented for each broadcast. The first is the time the source processor spends in the broadcast, and the second is the maximum time spent by any processor in the broadcast. As with the point to point timings, we concentrate on those timings that allow us to compare against system broadcasts. This means that we once again send 1-D arrays, and that we primarily concern ourselves with the topology that minimizes the maximal time in the broadcast.

As mentioned before, there are many ways to define the "best" broadcast topology. If pipelining can occur, rings are often best. If a particular processor's time is more important that others, again a ring may be the best choice. However, the most widely-used standard for "best" is the algorithm that, starting from synchronized processors, gets the answer to all participating processors in the least amount of time. Even here, it is impossible to say one topology is best, because it may depend on the size of the message being sent. We choose the topology that performs best over the entire curve, even if it is not as good in a specific region. In the following sections, we will concentrate our efforts on the topology that meets these qualifications.

In analyzing these times, it is important to understand how they were obtained. First, all processors are roughly synchronized by a call to the system's barrier or synchronization routine. Like the timings for T_c, the broadcast is repeated several times within a loop to insure the timings are above clock resolution. However, this can lead to results which are biased either for or against the topology. Ring topologies, for instance, will gain a positive bias, since the first broadcast will synchronize the processors so that any additional broadcasts will cost roughly T_c (i.e., pipelining occurs). On the other hand, it could be the case that messages sent in the last stage of a previous iteration interfere with the messages being sent in the initial stage of the present iteration, and the results are therefore negatively influenced. This issue is addressed in section 4.4.3.

The broadcast timings are split into three sections. Section 4.4.1 shows the results of a single run of several interesting topologies. The next section uses these timings to see how closely the theoretical models proposed in the topology section match with observed data. Finally we further analyze the BLACS "best" topology and the system broadcast to see how reproducible our timings are, and we give a least squares fit for both.

4.4.1 Survey of Topologies

The first test run on each platform is a survey of topologies with a range of $N = 0, \ldots, 50,000$ where a a data point is taken every 1000 double precision elements.

For each platform, figures 20, 21, 22, and 23, show four broadcast strategies that we find interesting. One curve is always the system broadcast, represented by a solid line. The curve represented by a dashed line is the "best" BLACS topology. By "best", we mean the topology that, starting from roughly synchronized processors, gets the message to all processors most rapidly. As was mentioned in the topology section, this does not imply the topology is best for all broadcasts. The other two curves show timings of other topologies of interest, such as rings, where the effects of pipelining make them interesting.

The only real surprise here is on the CM-5. On all other platforms, the BLACS are quite competitive, but, as expected, not quite as good as the system. On the CM-5, how-

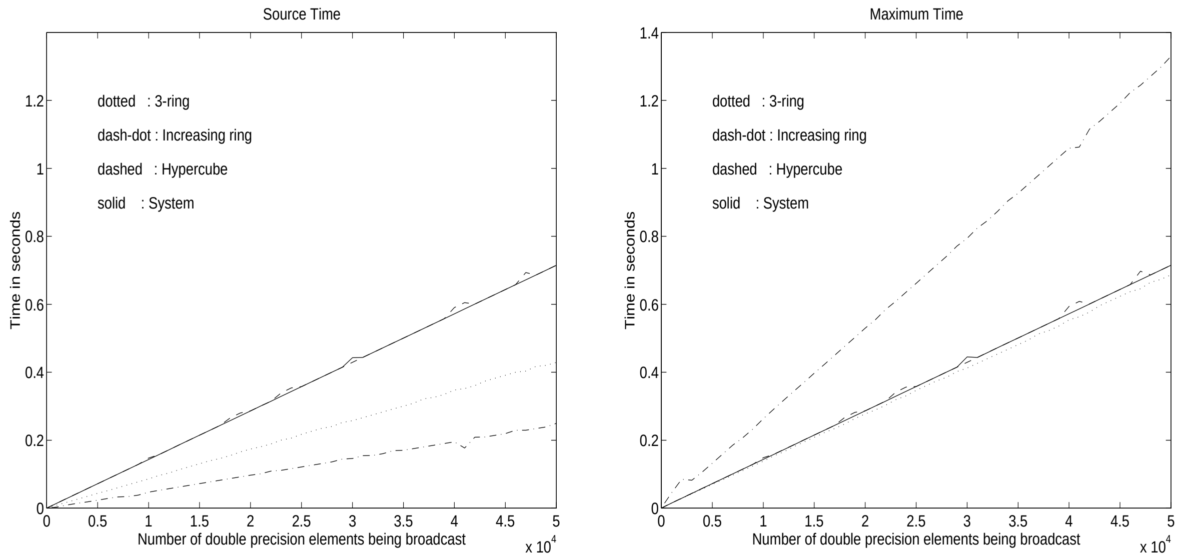

Figure 20: Survey of 32 processor i860 broadcast topologies, reps=5

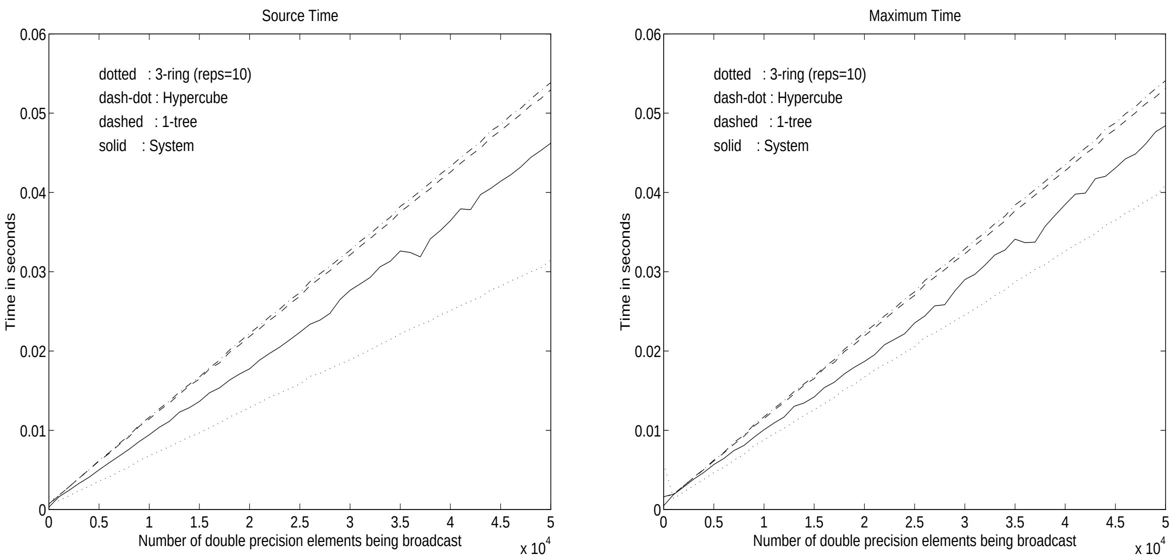

Figure 21: Survey of 32 processor Paragon broadcast topologies, reps=30

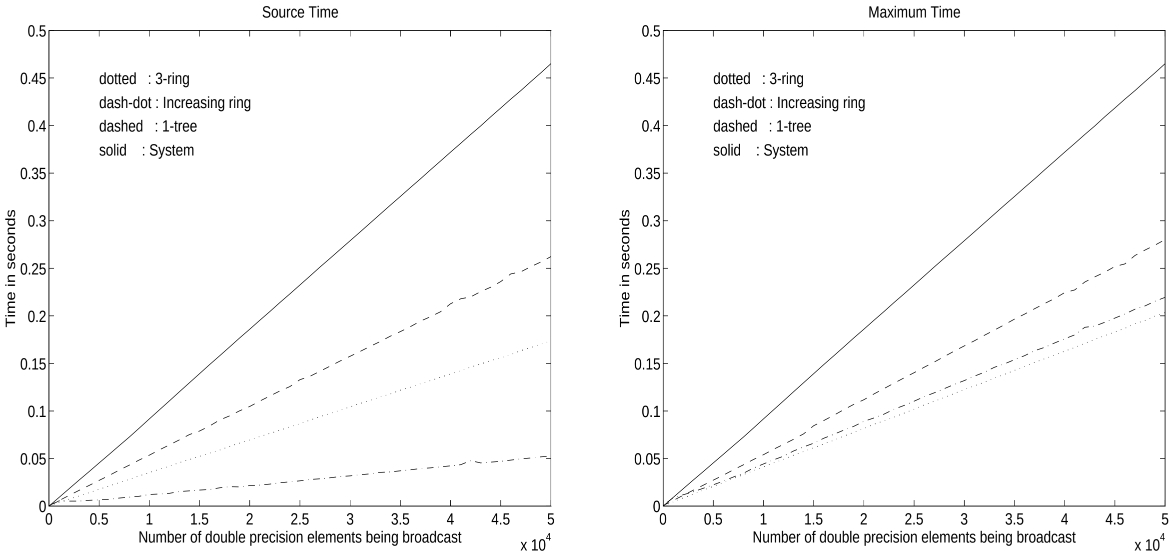

Figure 22: Survey of 32 processor CM-5 broadcast topologies, reps=15

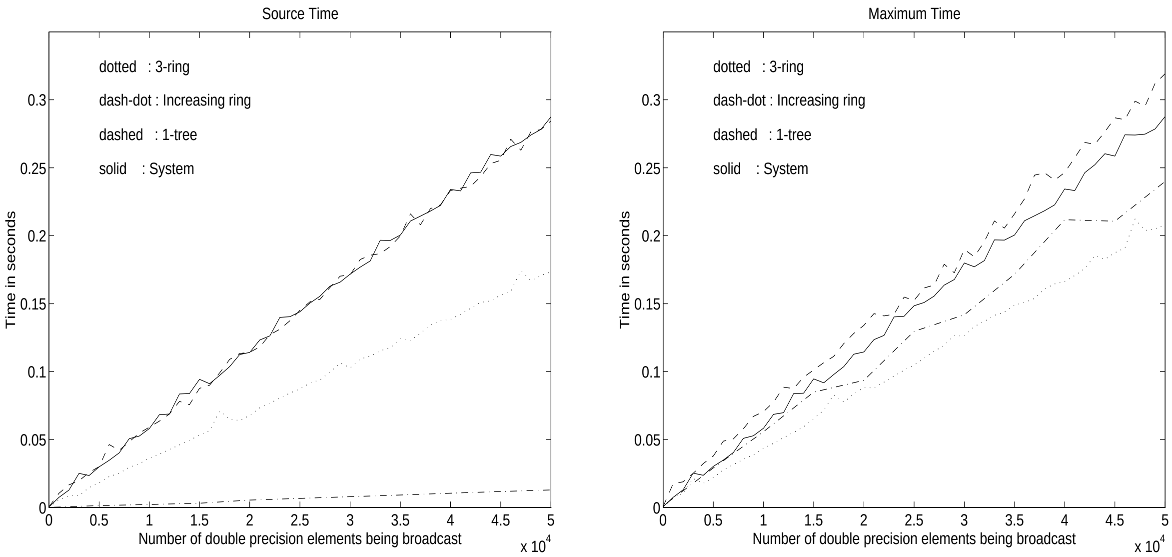

Figure 23: Survey of 32 processor SP1 broadcast topologies, reps=15

ever, the BLACS are considerably faster than the system. These system broadcast times were confirmed by Thinking Machines. The BLACS broadcast times are confirmed by the theoretical predictions, and are shown to be repeatable. Thinking Machines has stated that the next version of CMMD features faster broadcast and combine operations. Also, they noted that the system broadcast provides a better synchronization point than the tree-based topology used by the BLACS.

4.4.2 Accuracy of Theoretical Models

In this section we demonstrate how closely the topology models presented in section 4.3 match our experimental results. On the graphs 24 through 31, each observed data point is represented by 'o' (these points are obtained by repeating the broadcast within the timing loop), and the prediction given by the theoretical model is represented by a solid line.

For each platform, we choose to look at two topologies, one a ring, and one a tree. The first topology will be 3-ring, which as we see displays pipelining. In order to highlight the value of pipelining, two values are shown. Data points indicated by +'s were obtained by calling the broadcast only one time, so there is no pipelining. The predicted time for the non-pipelined broadcast is shown as a dotted line. We do not bother to plot the non-pipelined source time, since source time does not experience pipelining. The second topology shown is the BLACS "best" topology for each platform.

It should be noted that the single repetition times may not be individually meaningful, i.e., the length of time being measured is small enough that relatively large errors may occur. However, looking at all the data points provides a good indicator of whether the repeating of the test is noticeably effecting the times. Since these data points are not reliable, we do not calculate a relative error for the single repetition runs.

An example of how the predicted times are derived may be of interest. We will illustrate how the prediction for the CM-5's 3-ring broadcast was made. First we note that present version of the CM-5 BLACS are designed to have locally-blocking broadcast/sends (which will change for the next release), so we must figure the cost of the memory copy (T_m) in with the cost of the communication. The cost for one broadcast should be $N_r * T_s$ for the source processor, and taking into account that T_s is almost the same size as T_c, the maximal time in the algorithm is $((N_p - 1)/N_r) * T_c + (N_r - 1) * T_s$ (this comes from the analysis presented in the topology section). Now we instantiate these predictions using the following facts:

1. Number of processors, $N_p = 32$
2. Number of rings, $N_r = 3$
3. The T_m cost must be paid
4. After paying the T_m costs, the system T_s and T_c values will be used
5. These numbers predict single iteration runs: we have pipelining

Source time does not display pipelining, so predicted source time for all repetitions is $T_m + 3 * T_s$. The maximal time in the algorithm does display pipelining, so we have two

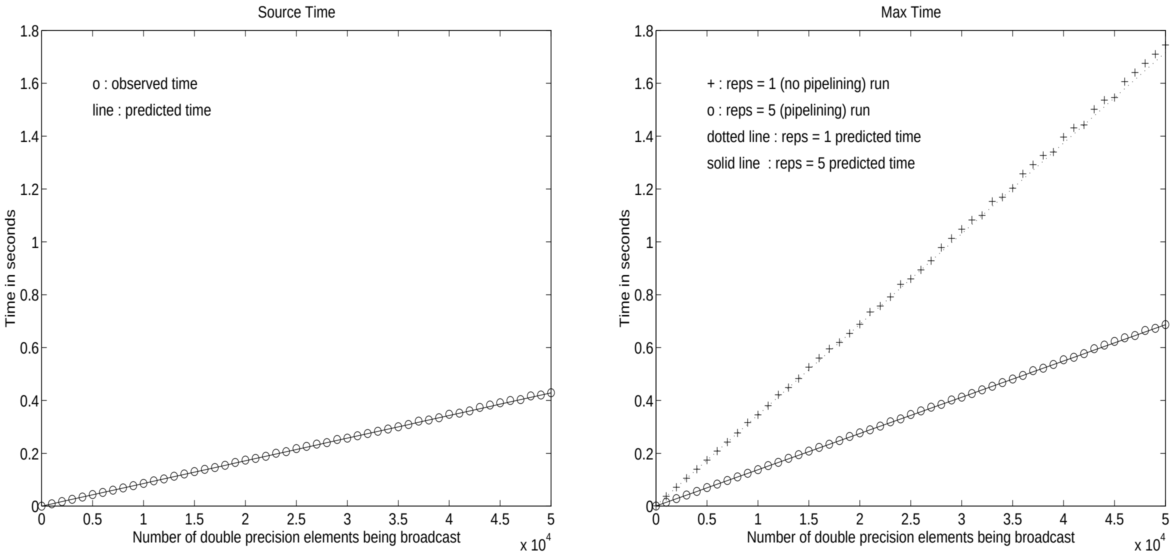

Figure 24: Predicted vs. measured time for 32 processor i860 BLACS 3-ring broadcast

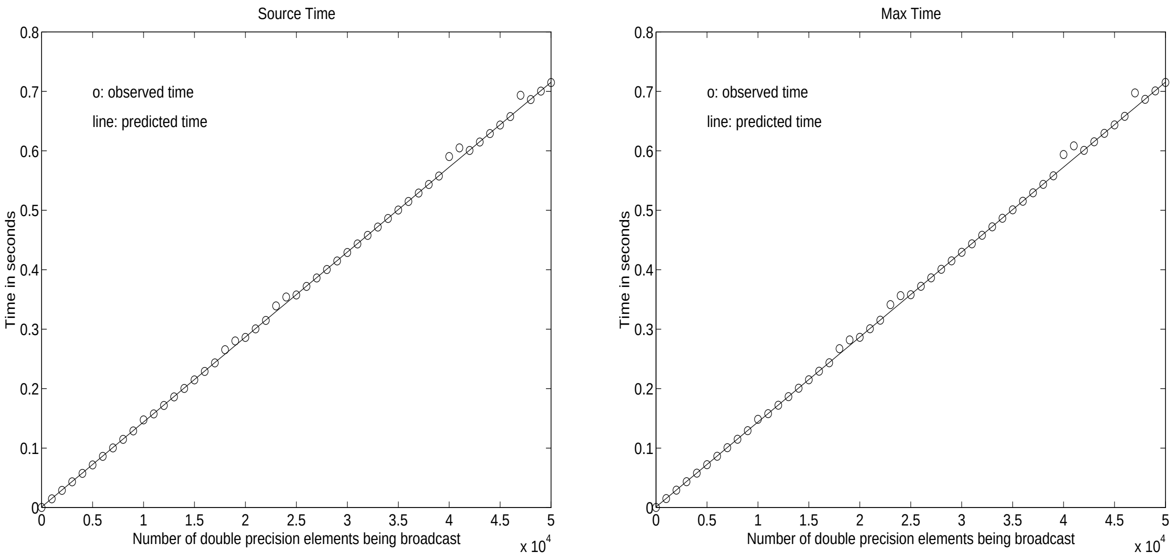

Figure 25: Predicted vs. measured time for 32 processor i860 BLACS hypercube broadcast

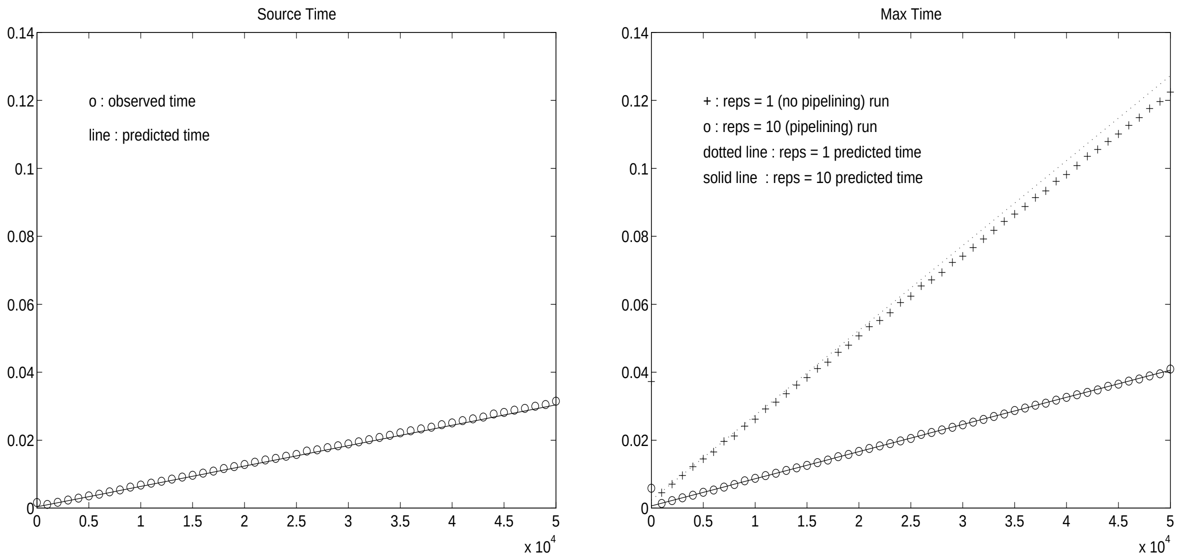

Figure 26: Predicted vs. measured time for 32 processor Paragon BLACS 3-ring broadcast

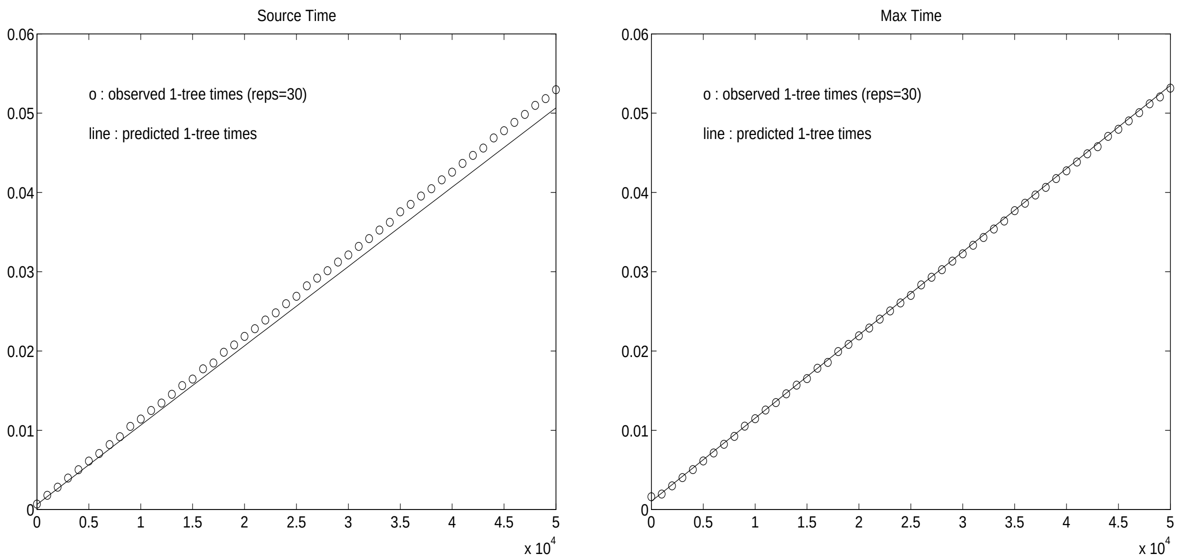

Figure 27: Predicted vs. measured time for 32 processor Paragon BLACS 1-tree broadcast

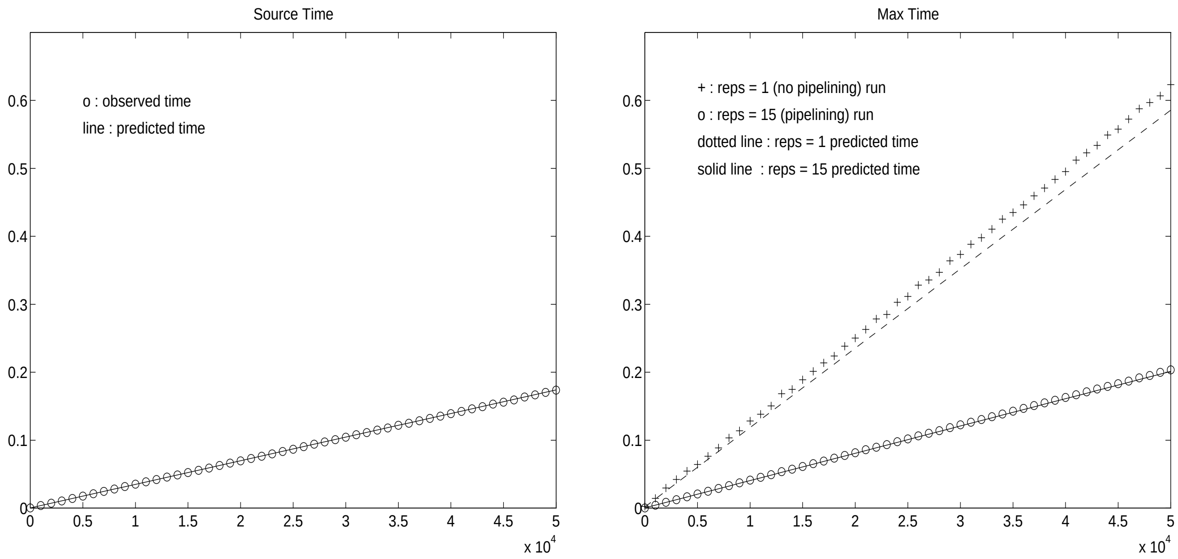

Figure 28: Predicted vs. measured time for 32 processor CM-5 BLACS 3-ring broadcast

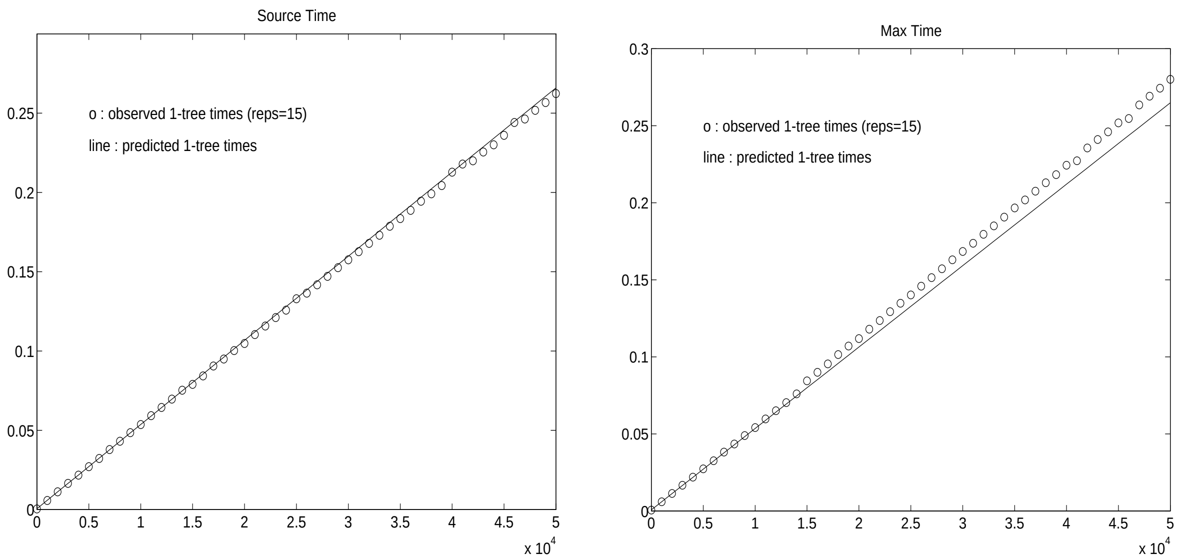

Figure 29: Predicted vs. measured time for 32 processor CM-5 BLACS 1-tree broadcast

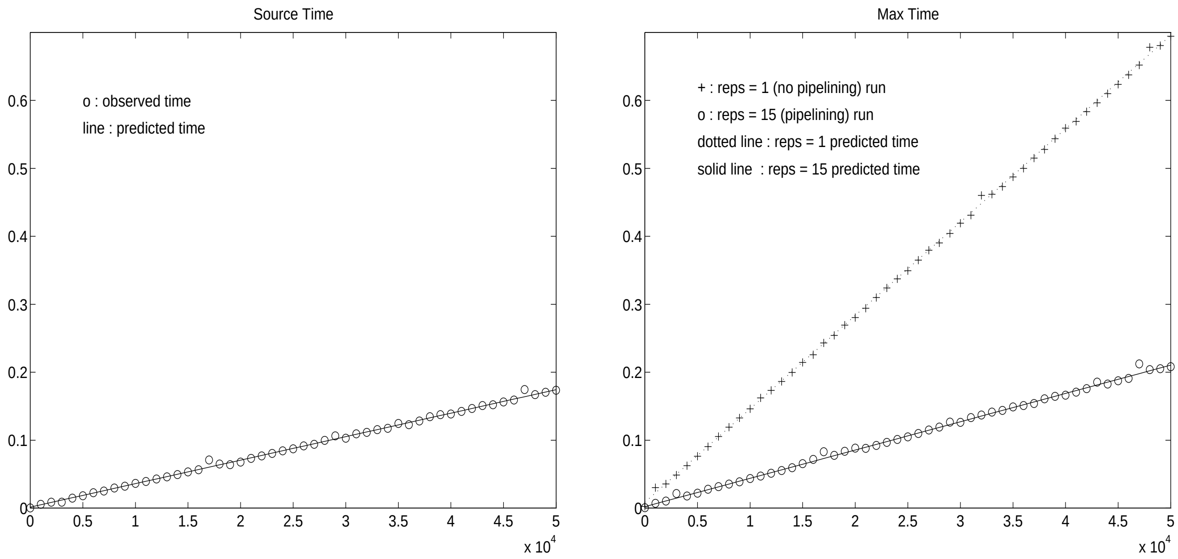

Figure 30: Predicted vs. measured time for 32 processor SP1 BLACS 3-ring broadcast

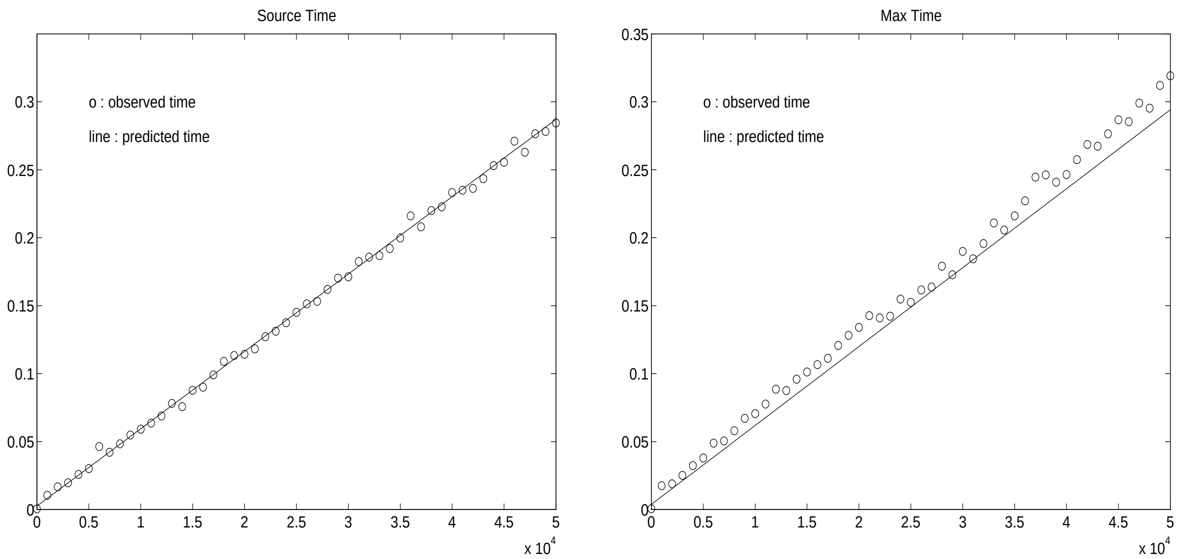

Figure 31: Predicted vs. measured time for 32 processor SP1 BLACS hypercube broadcast

predictions. For the reps=1 run, the time is $T_m + 10 * T_c + 2 * T_s$. The reps=15 broadcast will benefit from pipelining. The first broadcast's cost is the reps=1 time given above. Each additional broadcast, however, will only add $T_m + T_c + 2 * T_s$ to the cost of the algorithm. We now use our linear models of T_s, T_c, and T_m to arrive at our prediction.

Examining figures 24, 26, 28, and 30 should convince the reader that, when it can be employed, pipelining is a very effective optimizer of broadcasts. Further, it can be seen that the timing models are quite accurate on all platforms for the multiple repetition runs of the ring topologies.

Figures 25, 27, 29, and 31 show the predicted times for the tree broadcasts. We see that as messages grow larger, our prediction is not precise for the CM-5. Remember that we made the assumption that all our communication was nearest neighbor, and that we had at least the number of links required by the algorithm. Neither of these assumptions are true for tree broadcasts on the CM-5, and this causes our tree predictions to be inaccurate.

The SP1 shows a more consistent underestimation of maximal time in the tree broadcast. According to the system specifications, the network should behave as if it were fully-connected: all processors are the same distance away, and link conflicts do not occur. As we will see in the next section, it appears that repeating the broadcast multiple times does negatively affect communication time, however. This may be due to the way the BLACS are coded, and more experimentation will be required to isolate this problem. Since the theoretical models work for other platforms, it unlikely they are wrong. The fact that the combine models also underestimate the time required by the SP1 indicates that it is indeed a system-dependent error. In both cases, however, the error is not that much greater than the variance separate timing runs display.

Table 11 shows the relative errors of prediction versus actual times. If $T^o(i)$ is the observed time at the i^{th} data point, and $T^p(i)$ is the predicted time, then we define the relative error of the prediction as $\text{RE} = \max_{1 \le i \le 11}(|T^o(i) - T^p(i)|/|T^o(i)|)$. As with the point to point timings, we find that at the first data point ($N = 0$) the time is not large enough to give reliable results at the number of repetitions chosen, and thus it is ignored in our computation of RE. The SP1 has a large relative error, but this is at least partially due to inaccurate timings, as the error is high in only the first few data points.

SYSTEM	3-RING		"BEST"	
	Source	Maximum	Source	Maximum
i860	2.95 %	3.16 %	4.18 %	3.56 %
Paragon	10.40%	4.60%	10.00%	6.84%
CM-5	1.71 %	2.07%	2.45 %	5.61 %
SP1	26.55%	44.41%	26.76%	42.64%

Table 11: Relative errors for predicted times

4.4.3 Validity of Timings

Finally, we would like assurance that our times are meaningful, in the sense that they represent reproducible behavior for the machine. Also, we need to show that the tree topologies are relatively unaffected by the fact that we have repeated the broadcast multiple times (we account for this effect on ring-based topologies – for rings it is pipelining).

To accomplish this, we concentrate on our two most important times, the maximum time in the system broadcast, and the maximum time in the BLACS' "best" topology. For these quantities, every fifth data point is run an additional ten times. If we get a large spread of values, we know our timings are unreliable. These data points, together with those done in the survey, are shown as o's in figures 32, 33, 34, and 35. To show that repeating the broadcast within the timing loop has little effect on broadcast time, we run the larger data points with repetitions set to 1; these points are shown on the graph as +'s (Note that the Paragon graph has no reps=1 points: the times were not even vaguely repeatable). Finally, the least squares fit of all of these points is shown on the graph by the solid line.

On the i860, we see that the reps=1 times are well in line with the reps=5 runs, and we conclude that the timings done with multiple repetitions are accurate for reps=1 as well. All of the CM-5's reps=1 times, on the other hand, are noticeably (but not grossly) greater those of the multiple repetitions runs. This may be a timing artifact, but since all of the reps=1 runs are above the reps=15 runs, this seems unlikely. We recall that T_m is relatively expensive on the CM-5. Since we are using non-blocking sends in the BLACS, we may be able to hide some of this T_m time. The way this would work is that a process would perform the first pack (T_m), and start the non-blocking send. It returns before the message is actually sent, and does at least part of the next iteration's pack before being interrupted to actually send the message. If this is occurring, it could easily explain the difference we see between the multiple repetition and single repetition times.

Finally, we see that the SP1's single repetition runs are slightly below those of the multiple repetition runs. Again, this difference could easily be a timing artifact, but the fact that so many of the reps=1 data points agree argue against it. More to the point, the reps=1 data points fall more in line with the predicted time for the algorithm. As mentioned in the previous section, it will take a more detailed analysis to determine what, if anything, is occurring here.

Table 12 gives the relative error of the system and BLACS broadcasts. This table also gives the least squares fit of the multiple repetition data points, so that the reader can get a numerical value to associate with the graphs. The BLACS are quite competitive across all platforms. It should be noted that we have not chosen the topology to optimize the small size broadcasts. Using a different topology could cut the latency, at the expense of bandwidth.

SYSTEM	SYSTEM TIMES			BLACS TIMES		
	α	β	Rel Err	α	β	Rel Err
i860	557	14.2976	3.69%	1086	14.3598	3.78%
Paragon	403	0.9444	4.82%	1028	1.0437	1.69%
CM-5	-755	9.3167	0.78%	-523	5.6090	4.96%
SP1	2764	5.7323	25.88%	9150	6.0549	24.78%

Table 12: Least squares fit (in microseconds) and relative error of broadcast times for $N = 0, \ldots, 50000$

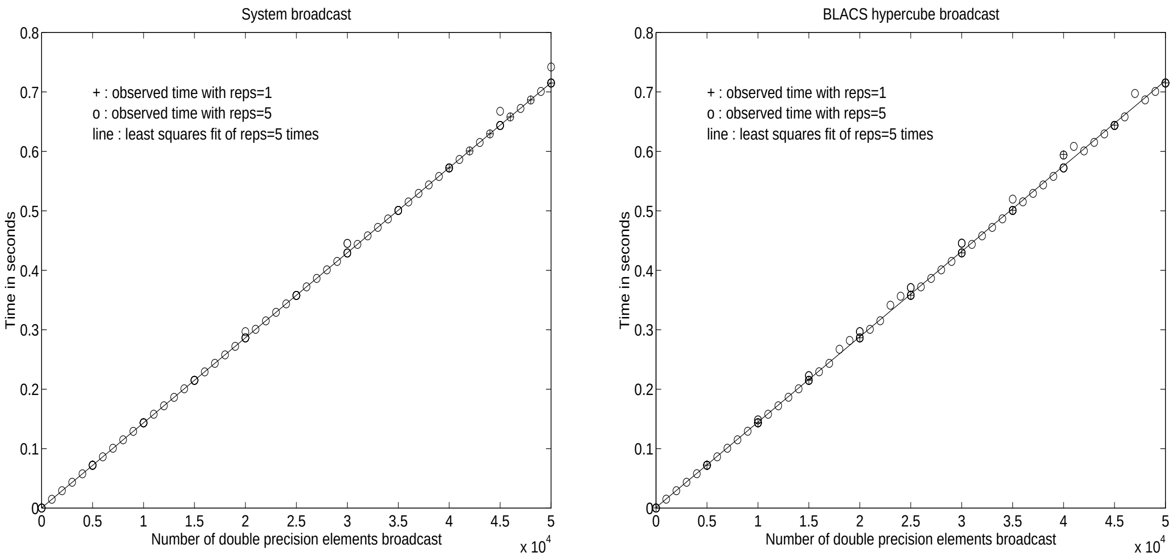

Figure 32: Variance between runs on 32 processor i860 broadcasts

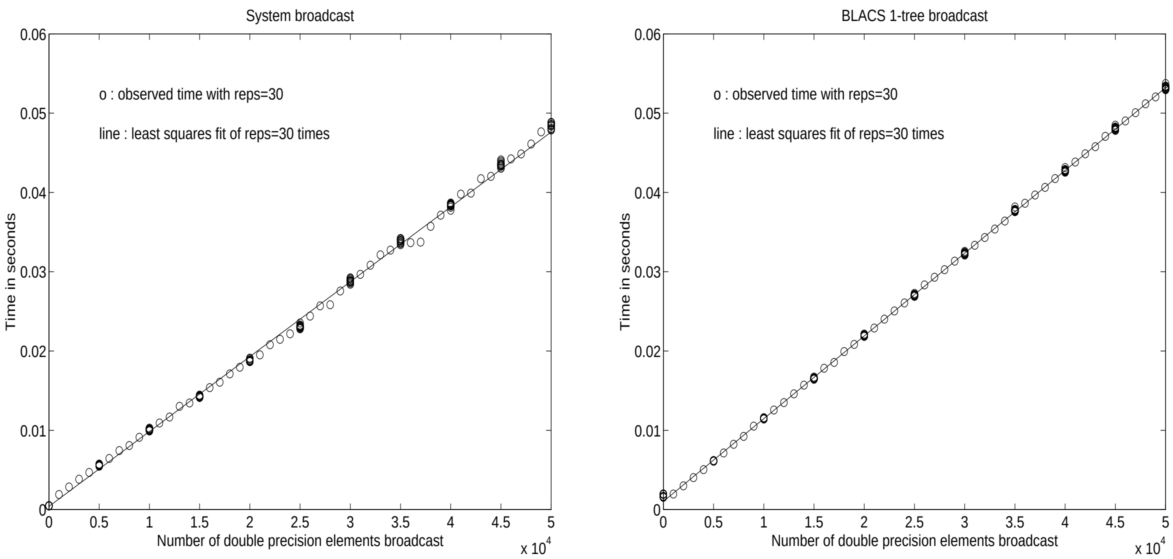

Figure 33: Variance between runs on 32 processor Paragon broadcasts

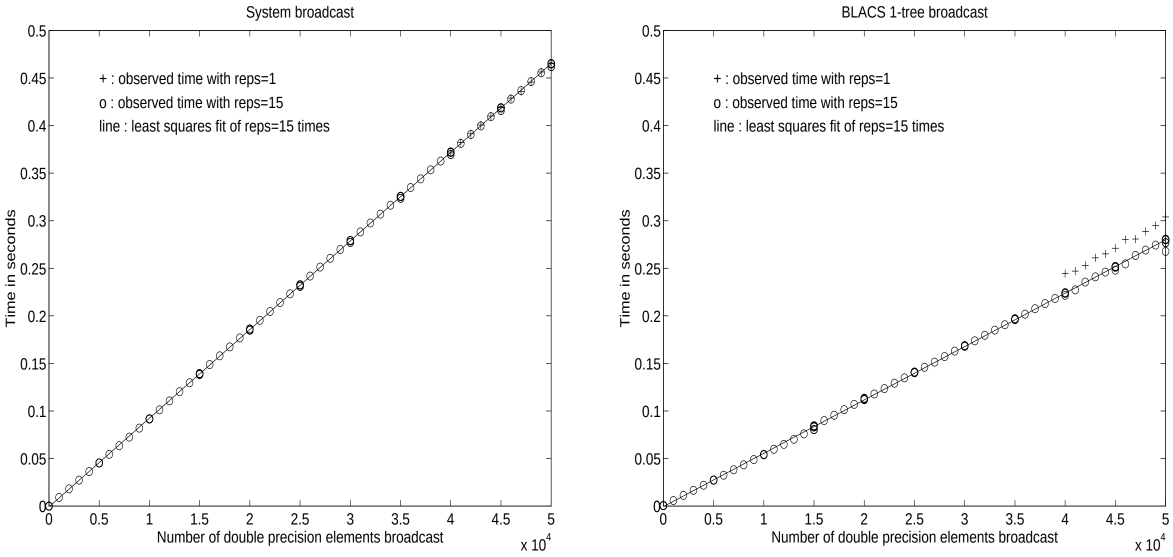

Figure 34: Variance between runs on 32 processor CM-5 broadcasts

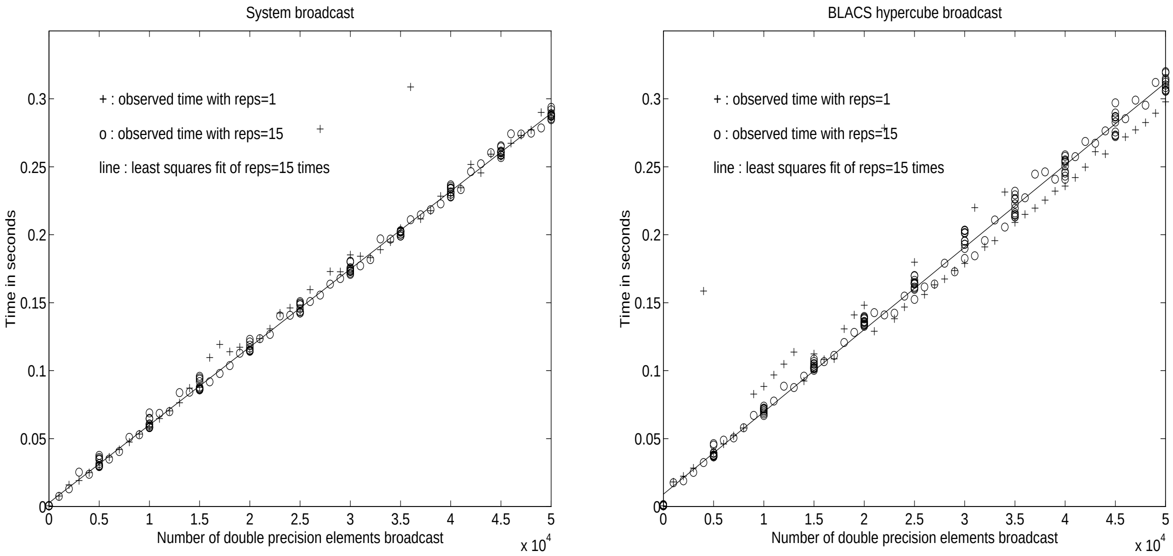

Figure 35: Variance between runs on 32 processor SP1 broadcasts

5 Combines

5.1 Semantics

In a combine operation, each participating process contributes data which is combined with all other processes' data to produce a result. This result can be left on a particular process (called the *destination* process), or on all participating processes. If the result is left on only one process, we refer to the operation as a *leave-on-one* combine, and if the result is given to all participating processes we reference it as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are element-wise summation, maximization, and minimization of general rectangular arrays. Note that a combine operation combines data between processors. By definition, then, a combine done across a scope of only one processor does not change the input data. This is why we specify that the operations (max/min/sum) are *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4×2 array of inputs produces a 4×2 answer array. The example given in the next section should help if further clarification is required.

If a processor is to receive the result, then obviously it cannot leave the routine until all processes have contributed their data to the combine operation. Therefore, the combine is globally-blocking (i.e., no process returns from the routine until all participating processes call it) for at least one process. If the answer is left on all processes, then the combine is globally-blocking for all processes. Finally, it may be possible to perform superior optimizations on certain platforms it it is allowed to create even a leave-on-one combine which is globally-blocking. The BLACS standard therefore states that while it may not always be the case, all combines should be programmed as if they were globally-blocking for all participating processes.

5.2 Syntax

The general form of the names for combines is `vGZZZ2D`, where `v` is the same as shown in table 2. The position `ZZZ` indicates what type of operation should be performed when sending the data. The operations presently supported are shown on table 13.

vGZZZ2D

ZZZ	MEANING
MAX	Entries of result matrix will have the value of the greatest absolute value found in that position.
MIN	Entries of result matrix will have the value of the smallest absolute value found in that position.
SUM	Entries of result matrix will have the summation of that position.

Table 13: Values and meanings of combine routines' name positions

The calling sequences for these routines are:

```
vGSUM2D( SCOPE, TOP, M, N, A, LDA,             RDEST, CDEST )
```

```
vGMAX2D( SCOPE, TOP, M, N, A, LDA, RA, CA, LDIA, RDEST , CDEST )
vGMIN2D( SCOPE, TOP, M, N, A, LDA, RA, CA, LDIA, RDEST , CDEST )
```

Parameters:

As before, output parameters are underlined.

`SCOPE`	Scope of processes to participate in operation. Limited to `'ROW'`, `'COLUMN'`, or `'ALL'`. See section 2.2 for details.
`TOP`	Network topology to be emulated during communication. Topologies presently supported are discussed in section 5.4.
`M`	Row dimension of matrix being compared/summed.
`N`	Column dimension of matrix being compared/summed.
<u>`A`</u>	Input: Two dimensional array of values being compared/summed (element-wise). Output: 2D array of results. If a process calls the routine but is not indicated to receive the final result, his array may be overwritten with intermediate results.
`LDA`	Leading dimension of the 2D array A.
<u>`RA`</u>	Integer array (of size at least `M x N`) indicating the row index of the process that provided the maximum/minimum. If a process calls the routine but is not indicated to receive the final result, his array may be overwritten with intermediate results.
<u>`CA`</u>	Integer array (of size at least `M x N`) indicating the column index of the process that provided the maximum/minimum. If a process calls the routine but is not indicated to receive the final result, his array may be overwritten with intermediate results.
`LDIA`	Leading dimension of the integer arrays RA and CA.
`RDEST`	Row index of process on which result is to be accumulated. On all other processes `A`, `RA`, and `CA` may be overwritten with intermediate results. `RDEST = -1` indicates the result is to be left on all participating processes.
`CDEST`	Column index of process on which result is to be accumulated. On all other processes `A`, `RA`, and `CA` may be overwritten with intermediate results. NOTE: if `RDEST = -1`, `CDEST` is not referenced.

For an operation to proceed, all processes indicated by the SCOPE command must call the given routine. Once again, an example should demonstrate how these routines are used. Assume we have a 2 x 4 process mesh (as shown in figure 1). Process {1,3} needs the maximum of the matrix `B` (of size 4 x 4) over all processes. All processes would make the following call:

```
DGMAX2D('ALL', '1-TREE', 4, 4, B, 4, RA, CA, 4, 1, 3)
```

Upon completion, process {1,3} would have three matrices that contain information on the maximize function. The matrix `B` is still of size 4 x 4. Element (1,2) of `B` would contain the element with the largest absolute value found on any process at matrix location (1,2). RA(1,2) would indicate what process row that maximum was found on, while CA(1,2) would tell which process column it was found on.

5.3 Related Topics

5.3.1 Additional Buffering Demands

In a broadcast or a send, only one buffer is required (assuming the send is not being used to simulate a send with a different level of blocking – see section 6.2 for details). Data can be sent from, received into, or in a broadcast/receive operation, both sent and received into this one buffer. Combine operations require at least two buffer spaces. One space is needed to receive other processes' data, and the second buffer stores the answer (so far) that the operation has produced. A simple example would be in the `DGSUM2D` operation. One buffer is received into, and then its' contents are added into another buffer.

This leads to two types of buffers. The first is a receive buffer, which is used to receive other process's contribution to the combine. The second buffer is the result buffer, which stores the result of the operation. The receive buffer must be contiguous, since it is receiving contiguous messages. It is highly desirable that the result buffer be contiguous as well.

The Intel BLACS were originally written on the Intel i860, and our machine had only 8MB of memory per node. The Touchstone Delta was a little better, with 16MB. However, on both machines, insufficient memory was a constant problem. Therefore, in order to conserve memory space, the Intel BLACS never get more than one buffer. For combines, this BLACS' allocated buffer is used as the receive buffer. The user's matrix A is then used as the result buffer. This saves memory, but it has several drawbacks. First, the user's buffer may not be contiguous in memory. This means that it requires two loops to perform the operation, and if the columns of A are short, there may be numerous cache misses. Further, messages cannot be directly sent from the result buffer if it is not contiguous in memory. If a message is to be sent from the result buffer, it must be packed into the receive buffer, and sent from there. This will require data copying.

Unfortunately, it is almost always the case that we wish to send from the result buffer. The usual way an operation proceeds is: receive into receive buffer, combine this information with that already stored in the result buffer, and then send this result on to another process. With this in mind, we see that a contiguous result buffer will save many unneeded data copies, since we could send directly from the result buffer.

On the SP1 and CM-5, the BLACS get two buffers, and therefore this situation is handled more efficiently (these platforms require at least two buffers anyway: one to perform non-blocking sends out of, and one to receive into – see section 6.2 for full details).

As mentioned earlier, PVM does its own buffering. Since there is already the cost of packing and unpacking into the PVM buffers, it does not seem worthwhile for the BLACS to add yet another layer of buffering. Therefore, the user's matrix is used as the operation buffer, and the PVM buffer serves, mostly, as the communication buffer. PVM allows only pack/unpack access on its buffers, so we cannot perform the operation directly using the PVM buffers. The BLACS therefore allocate a buffer equal to the column length, which is M (see section 2.1 for details). A column of data is then unpacked, operated on (leaving the result in A), and then the M-length buffer is overwritten with the next column. For `DGMAX2D` and `DGMIN2D` these buffers are slightly longer. For the sum routines, the length is $M * \text{sizeof(type)}$, where type is the data type being operated on. For max/min operations, the buffer is of size $M * (\text{sizeof(type)} + \text{sizeof(short)})$. The extra $M * \text{sizeof(short)}$ space is required to hold values indicating which process the max/min came from.

5.3.2 Communication and Its Effect on Fan-in

Figure 36 shows a simple 5 to 1 fan-in. An understanding of the fan-in operation is necessary for an informed analysis of the combine topologies. The behavior of fan-in is system dependent.

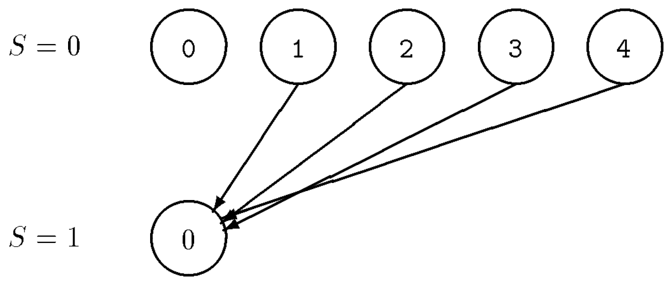

Figure 36: Simple 5 to 1 Fan-in

If the sending processes are not synchronized, it is very important that no ordering be imposed by the receiver. The BLACS avoid this ordering by using the ID generating algorithm discussed in section 6.1.

To analyze the cost of this operation, it is necessary to understand how the underlying message passing platform works. An example should highlight the effect of different schemes on performance.

Assume that process 0 above is the last to enter the operation. On the Intel machines, this means that when 0 does begin the code, no T_c costs will be levied. This is because the Intel system buffers messages on the receiving process. Therefore, when 0 enters the routine, the messages from all four processors are waiting to be accessed, and the cost, instead of T_c, will be that of a memory to memory copy (T_m). The cost of this operation is then $4 * (T_m + T_o)$. An additional note is that processors $1, \ldots 4$ can leave as soon as their send is completed.

On the other hand, when using a blocking send such as available on the CM-5 or SP1, process $1, \ldots, 4$ will enter the routine, and then wait. When 0 arrives, the receives are posted, the operations done, and the cost is $4 * (T_c + T_o)$. Further, process $1, \ldots 4$ must either wait until process 0 arrives, or issue non-blocking sends so that they may leave immediately.

This sequence of execution is not necessary to obtain speedup on the Intel systems. Assume 0 enters the operation first. It then waits until the first processor arrives and sends its message. At that point, it pays the $T_c + T_o$ cost of receiving and operating on the data. After that period of time, however, it is likely that at least one of the other processes will have entered the code, and already have begun its send. If the processors are poorly synchronized, this kind of savings can become appreciable.

The SP1 or CM-5 will have to pay the full T_c cost regardless of order. Since there is no communication/computation overlap on these machines, not even the intelligent use of non-blocking message passing can prevent this.

5.4 Topologies

At the present time, only two topologies are supported for combines. All of the notation used in the discussion of broadcast topologies is required in this discussion. In addition, the time T_o, defined to be the time required to perform the given operation (max, min, or sum) and T_D, the time the destination processor spends in the algorithm, are also needed. To rigorously define the T_o as used in the discussion below, T_o should include all time during the operation where the process is not communicating, and as such, its true value would be $T_o + \epsilon$. Then, ϵ would indicate the time to set up the loops, perform the function calls required by the algorithm, etc. Note that this ϵ was ignored in our discussion of broadcast topology, and we do the same here - it should be small enough to be overwhelmed by T_o.

5.4.1 General Tree Gather

The first combine topology is the general tree gather, or fan-in, which is basically the same algorithm as the general tree broadcast (or fan-out) described in the 4.3.2, except that communication flows in the opposite direction. Figures 37 and 38 show the communication patterns of this algorithm with $N_b = 1$ and $N_b = 4$ (as before, N_b refers to the number of branches at each node of the tree).

If all processors in the scope of the operation need the information, it is rebroadcast using broadcast's general tree algorithm. This topology can be called in the exact same way as broadcast's general tree algorithm, i.e. through the use of `SETBRANCHES` and setting `TOP = 't'`, or by setting `TOP ='1' ...'9'`.

Assuming that only one processor needs the answer (the case when all processors require the answer will be dealt with later) this topology has many desirable features. First, at each step of the algorithm only $\frac{1}{N_b}$ of the processors left in the operation go on to the next step.

It is difficult to write a general tree timing analysis because much of the behavior of these trees depends on how the machine handles communication (i.e., is there computation/communication overlap, where are messages buffered, etc.). However, careful analysis can demonstrate that $N_b = 1$ will usually be the best choice to minimize T_D on the systems presently supported. It will be shown below that $N_b \geq 2$ will, in general, be slower than $N_b = 1$, assuming a processor can receive only one message at a time (this is the true for all present platforms).

We may break the analysis of this algorithm into two distinct cases. The first is when T_o (time to perform sum/max/min) is greater than T_c (time to communicate). As we increase N_b, the height of the tree will decrease, with a corresponding increase in the number of T_o's that a receiving processor must perform at each step.

Since $T_o \geq T_c$, only the first send in each step is costly. The rest of the T_c's can be accomplished in the time where the previous T_o is operating (assuming computation/communication overlap). However, for the case where $T_o \geq T_c$, it is obviously not useful to subtract a T_c at the price of adding a T_o.

The other case, where $T_o < T_c$ might then seem like an opportunity for high degree trees to shine. A close analysis reveals this is not true. Again, as N_b increases, each re-

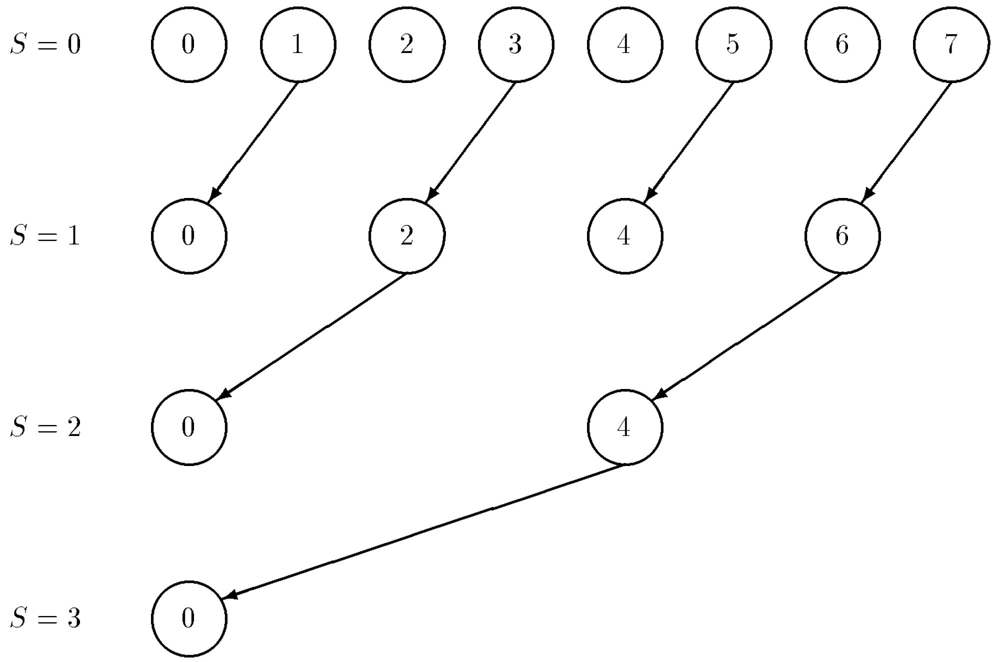

Figure 37: General tree gather with $N_b = 1$

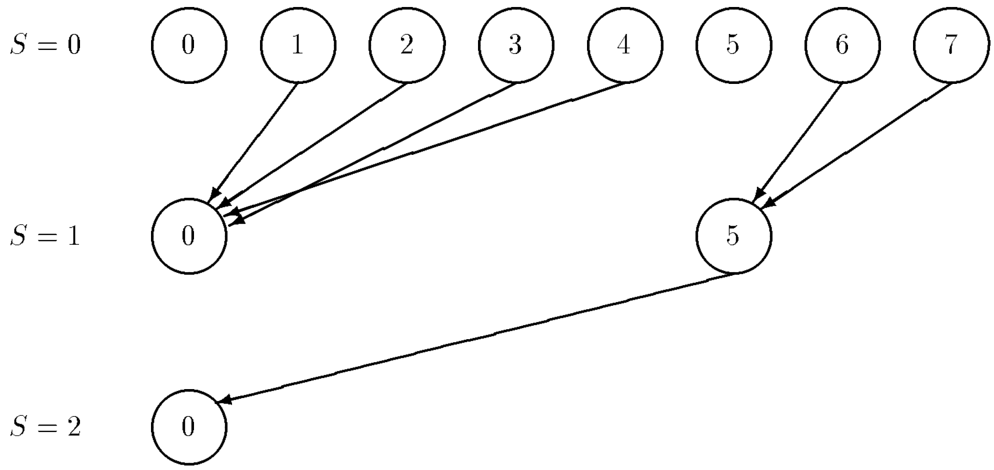

Figure 38: General tree gather with $N_b = 4$

ceiver performs the operation more and more times. However, since $T_o < T_c$, even assuming computation/communication overlap, the cost of the communication cannot be hidden entirely by T_o. The time each receiver spends will then be (again assuming a processor can receive only one message at a time) $T_c + N_b * T_o + (N_b - 1)(T_c - T_o) = N_b * T_c + T_o$, thus $T_D \approx \lceil \log_{N_b+1}(N_p) \rceil (N_b * T_c + T_o)$ (the $\approx$ is here because the last step of the algorithm may not have the full N_b senders).

If we differentiate T_D with respect to N_b, we get $T'_D = \frac{ln(N_p)[(N_b+1)ln(N_b+1)-N_b]*T_c-T_o}{(N_b+1)\ln^2(N_b+1)}$. If $T_o < T_c$, this function is strictly positive for all $N_b \geq 2$. This implies that T_D is strictly increasing, which tells us that the destination processor's time increases with N_b. Therefore, once again, $N_b = 1$ is the best choice.

Therefore, $N_b > 1$ will only be of use in special situations. If a processor can overlap receives, $N_b > 1$ would be useful. If processors will be poorly synchronized when entering the operation, larger N_b's may be useful. In this case, the large N_b would increase the likelihood of a receiving processor having something to do while it waits for the next processor to enter the operation. $N_b > 1$ also allows more processors to leave at each step. On a platform where unreceived sends are buffered on the receiver (at the moment, only the Intel machines), if sending processors arrives before receivers, all communication is essentially free, leaving only T_o costs. Note that the number of receivers decreases as N_b increases, so this can be useful.

With these caveats, we can say that $N_b = 1$ is the interesting choice, and then, $T_D = \lceil log_2(N_p) \rceil (T_c + T_o)$. If all processors require the answer, it is found as above, and then broadcast to all processors via the general tree algorithm described in section 4.3.2. The longest time any processor would then spend in the algorithm would be $\lceil log_2(N_p) \rceil (2*T_c+T_o)$

5.4.2 Bidirectional Exchange

This topology is specialized for the case where all processors require the information (i.e. `RDEST = -1`, as described in section 5), and if `RDEST` does not equal `-1`, the general tree algorithm with $N_b = 1$ is called instead. It is based on an algorithm presented in [11]. This topology involves having pairs of processors exchange information, and thus it performs best when N_p is an integer power of 2. The communication pattern inherent in this algorithm is shown in figure 39. As the user can see, this an extremely "noisy" algorithm: every processor is sending and receiving at every step in the algorithm. It is called by setting `TOP = 'h'`.

Unless the platform supports the overlap of sends and receives, this topology is inferior to fan-in/fan-out. If sends and receives cannot occur simultaneously, the best speed this algorithm can achieve is $T_D = log_2(N_p) * (2 * T_c + T_o)$. This T_D is for all processors. Fan-in/fan-out with $N_b = 1$ has the same maximal cost, but half of the processors finish earlier. Therefore, this topology should only be used on platforms where sends and receives can be overlapped.

Assuming simultaneous send and receive, we have two interesting cases. If N_p is an integer power of two, all processors will spend roughly $T_D = log_2(N_p) * (T_c + T_o)$ in the algorithm. If N_p is not an integer power of two, the first step of the algorithm requires processors beyond the power of two to send their values to processors within an integer power of two, the normal bidirectional exchange takes place, and then the answers are sent

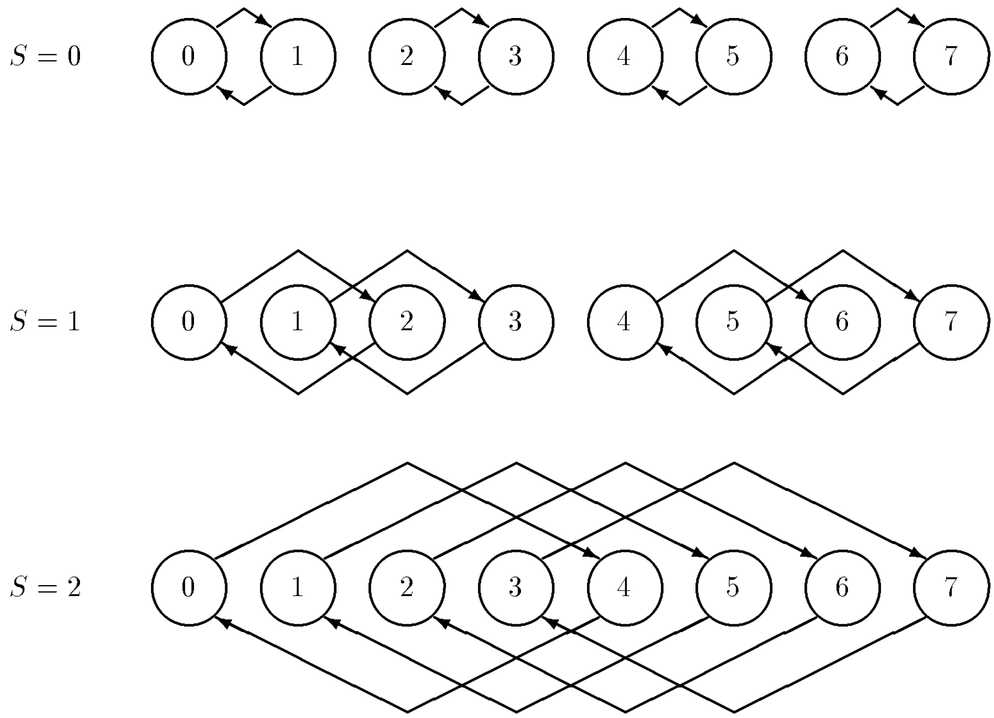

Figure 39: Bidirectional exchange

back out to the non-power of two processors.

This means that processors within an integer power of two will spend $T_D = T_c + T_o + T_s + \lfloor log_2(N_p) \rfloor * (T_c + T_o)$ in the algorithm, and those processors beyond the integer power of two will spend $T_D = 2 * T_c + T_o + \lfloor log_2(N_p) \rfloor * (T_c + T_o)$.

In the best case, this algorithm will give all processors the answer in the same amount of time that it takes to get the answer to one processor using the fan-in algorithm. However, it will rarely be the case that this speed is realized. Not only must simultaneous send/receive be allowed, but a twice the bandwidth is required, and a network of at least the richness of a hypercube is required to avoid link conflicts. Therefore, fan-in/fan-out should be used in the general case, and this topology should be utilized only when timings show that it is superior.

5.5 Timings

Here we present timings for the sum combine operation, where the result is left on all participating processors. All platforms except the CM-5 possess only leave-on-all combines. The CM-5 possesses only leave-on-one combines. The BLACS possess both. Since the majority of the platforms have only leave-on-all combines, our comparisons are made with leave-on-all. On the CM-5, a leave-on-all combine is constructed by performing a leave-on-one, followed by a broadcast (analogous to the leave-on-all 1-tree of the BLACS).

None of the machines natively possessed a maximization/minimization combine operation which returned the processor which supplied the max/min. Therefore, max/min operations are not suitable for a system/BLACS comparison. Further, max/min are only rarely used on anything but scalars, so their performance throughout our $0, \ldots, 50,000$ range is relatively uninteresting. Therefore, timings for maximization and minimization are not presented. The user should be aware, however, that for large problems, the BLACS will be considerably slower on max/min operations, since the extra information inherent in supplying the source of the max/min must be communicated along with the max/min data.

5.5.1 Survey of Topologies

Our first action for global timings was to run a range of topologies and see which ones are best. It was discovered that the 2 and higher level trees were quite bad for large N, just as our analysis had predicted. For extremely small N (i.e. in the range of $N < 30$), higher level trees were faster than 1-tree's. This was because synchronization became an issue for such small messages (as mentioned in 5.4). For larger sizes, these trees inevitably did poorly. Therefore, the survey graphs (figures 40, 41, 42, and 43) feature 3 topologies, 1-tree (dashed line), bidirectional exchange (dotted line), and the system's sum (solid line).

Since the CM-5 did not possess a leave-on-all combine, it was necessary to use a broadcast to create a leave-on-all. However, since the BLACS have already been shown to possess a superior broadcast, this may seem prejudicial. Therefore, our CM-5 survey includes a system and BLACS leave-on-one combine.

One look at the Paragon timings (figure 41) should be enough to alert the reader that something is wrong. When these timings were taken, a new (β release) version of the operating system had just been installed on the Paragon, and it seems that there is an error in the system's sum combine. It has been reported to Intel, and they are looking into it. The BLACS are also noticeably superior to the CM-5's system combine. As with the broadcast times, the next version of CMMD is supposed to feature faster combines.

5.5.2 Accuracy of Theoretical Models

The time T_o is required to apply the theoretical models. T_o was obtained by looping over calls to the BLACS summation routine. We then did a least squares fit over the $0, \ldots, 50,000$ data range, as with other times. No particular insight is gained by the graphs, and so for space purposes we just report the results. Table 14 gives the least squares fit of the times for the BLACS summation operation on the various platforms. Notice that the Paragon is slower than the i860 for T_o. This is rather surprising since the Paragon possesses a faster version of the same chip (a 50MHz i860, as opposed to a 40MHz). However, the Paragon is running a full-blown unix operating system, including virtual memory, etc. It is assumed that the many background processes present in a full unix implementation cause this slowdown.

Figures 44, 45, 46, and 47 show the predicted time (solid line) and the observed time for bidirectional exchange (o's) and 1-tree (+'s).

For the i860, we see that the prediction of 1-tree is accurate, but that for large N, the bidirectional exchange prediction is considerably low. The Intel BLACS do not make optimally use the Intel's primitives in bidirectional exchange, with the result that some link conflicts occur between iterations of bidirectional exchange, and thus we see this gap.

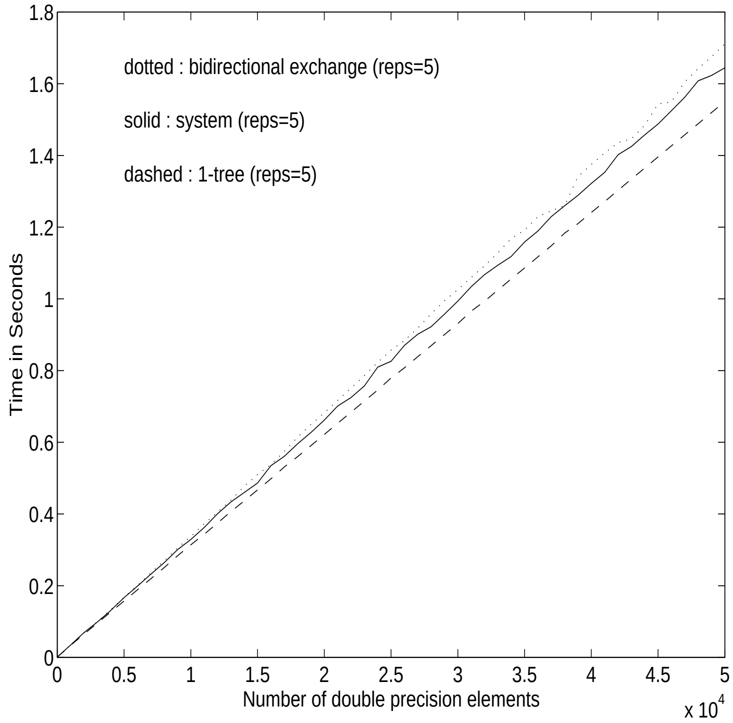

Figure 40: Survey of 32-processor i860 combines

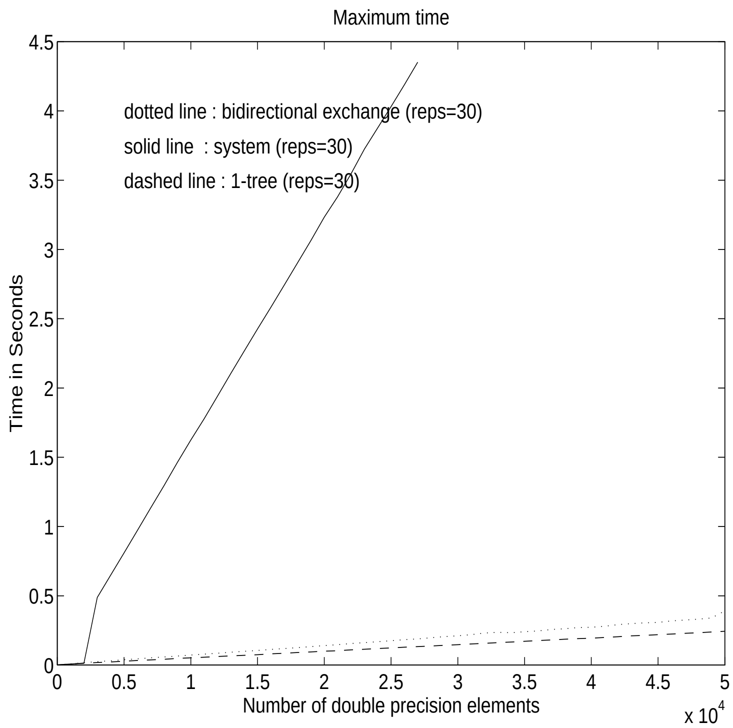

Figure 41: Survey of 32-processor Paragon combines

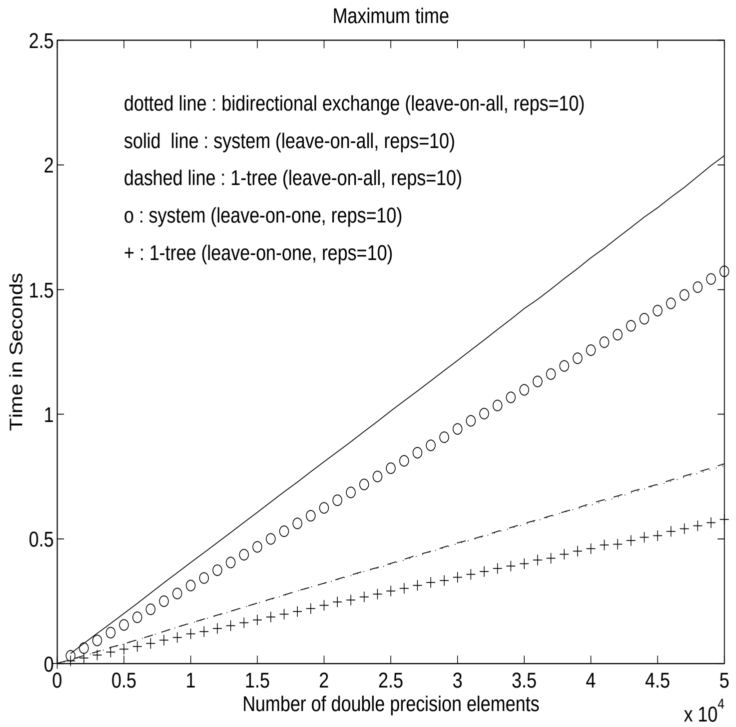

Figure 42: Survey of 32-processor CM-5 combines

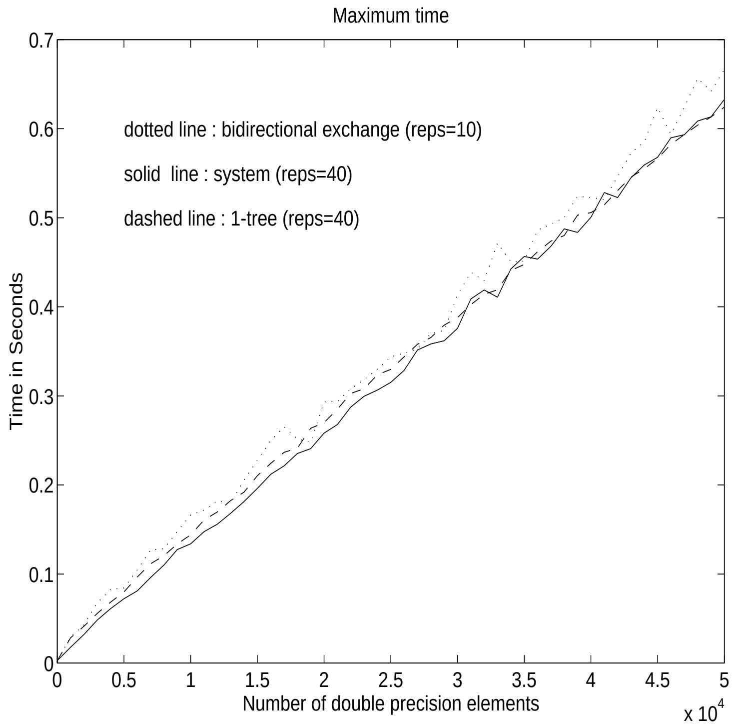

Figure 43: Survey of 32-processor SP1 combines

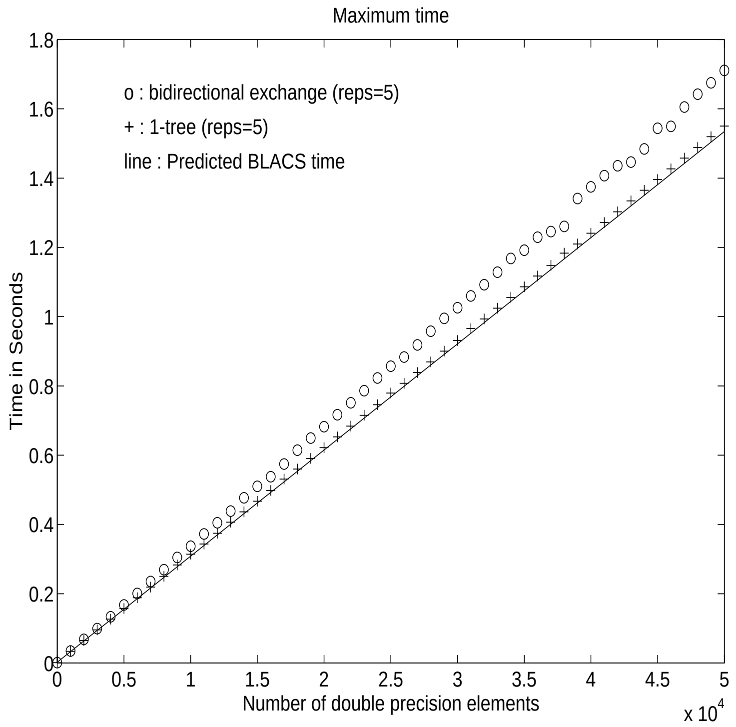

Figure 44: Predicted vs. measured maximum time for i860 BLACS combine (sum)

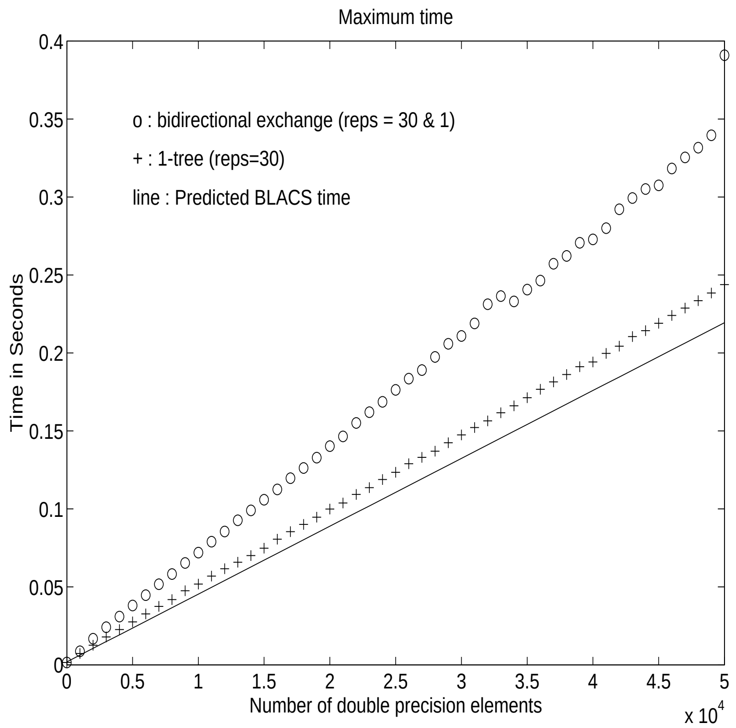

Figure 45: Predicted vs. measured maximum time for Paragon BLACS combine (sum)

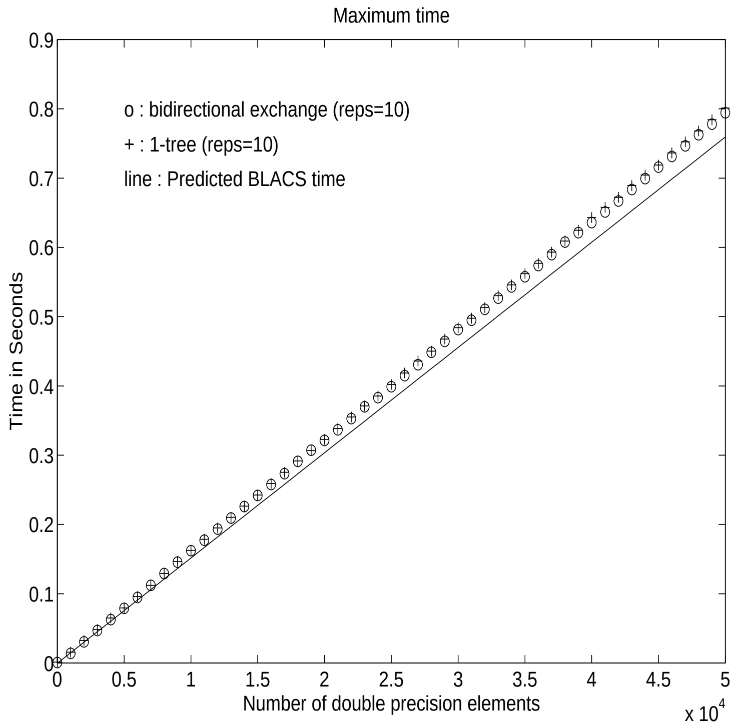

Figure 46: Predicted vs. measured maximum time for CM-5 BLACS combine (sum)

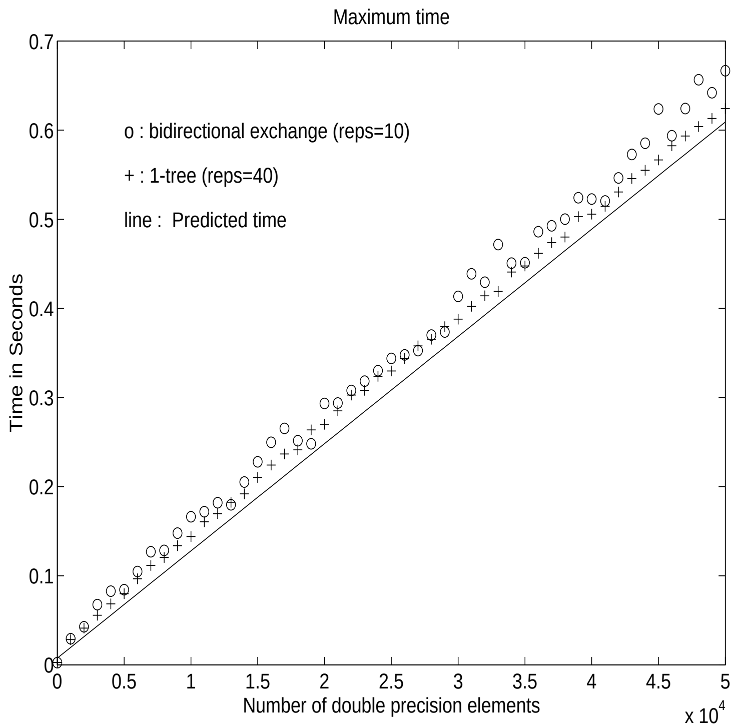

Figure 47: Predicted vs. measured maximum time for SP1 BLACS combine (sum)

SYSTEM	T_o	
	α	β
i860	-14	0.4145
Paragon	-45	0.4506
CM-5	-392	1.0706
SP1	-24	0.1052

Table 14: Least squares fit of T_o (in microseconds) for various platforms

The Paragon prediction is low for both bidirectional exchange and 1-tree. Partly this is because neither of the topologies will result in nearest neighbor communication on the Paragon. The main cause of this gap, however, is link contention. Both strategies call for more links than the Paragon, which is a 2D grid, possesses. We see that these factors, plus the non-optimal coding of bidirectional exchange mentioned above, cause the prediction to be extremely poor for bidirectional exchange. The reader should notice that the bidirectional exchange algorithm curve contains reps = 30 and reps = 1 runs. This is because for large problem sizes the extreme link contention involved in performing multiple repetitions of bidirectional exchange would effectively cause a hang on the Paragon. We therefore ran some of the larger sizes by doing only a single repetition. We may therefore conclude that bidirectional exchange (at least in its present form) should never be used on the Paragon.

On the CM-5, we see that our prediction is slightly low for both algorithms. This small error in the prediction is probably due to the fact that the CM-5 does not possess enough links to stop link contention, and that this communication pattern will not be nearest neighbor.

The SP1's combine times, like those of it's broadcast, are slower than predicted. Again, further investigation will be required to determine the cause. At any rate, while the underestimation is noticeable, it is not gross.

Table 15 shows the relative errors of prediction versus actual times. If $T^o(i)$ is the observed time at the i^{th} data point, and $T^p(i)$ is the predicted time, then we define the relative error of the prediction as $\text{RE} = \max_{1 \leq i \leq 11}(|T^o(i) - T^p(i)|/|T^p(i)|)$. Note that the predicted time is in the denominator, unlike the relative error used for broadcasts. We choose this relative error because, for the systems supported here, bidirectional exchange and 1-tree have the same predicted time. Therefore, by using the predicted time in the denominator, the relative error will tell us which algorithm is closest to the predicted time.

SYSTEM	Bidirectional Exchange	1-tree
i860	10.72%	3.05%
Paragon	78.16%	19.99%
CM-5	7.04%	7.98%
SP1	55.08%	45.13%

Table 15: Relative errors for predicted combine times

As with the point to point timings, we find that at the first data point ($N = 0$) the time is not large enough to give reliable results at the number of repetitions chosen, and thus it is ignored in our computation of RE. The SP1 once again has a large relative error. As with the broadcast predictions, only in the first few points have this large error.

5.5.3 Validity of Timings

As in the broadcast section, the system and the best BLACS topology are further analyzed to determine validity. Every fifth point is ran 10 additional times to observe the variance. Normal data points are indicated by o's, and those data points with reps=1 are indicated by +'s (due to time constraints, we were unable to obtain any reps=1 times on the SP1). The least squares fit is the solid line.

Examining figures 48, 49, 50, and 51 should convince the reader that the use of multiple repetitions has no real effect on our timings on the i860, Paragon, or CM-5. We cannot make this assertion for the SP1, since we have no reps=1 times.

Table 16 gives the relative errors and least squares fit of the data points. These least squares fits should demonstrate to the reader that not only are the BLACS quite competitive across all platforms, but indeed are slightly better on some. Again, the CM-5 is the only platform where the difference is large enough to be alarming, and we are told that the next version of CMMD provides for faster combines. As far as validity of the data is concerned, we see that, as usual, the Paragon and SP1 have the most suspicious data. However, with a maximal RE of around 10%, even these times are fairly reliable. Notice that the we do not present the Paragon's system data, since there appears to be an error in the present version.

SYSTEM	SYSTEM TIMES			BLACS TIMES		
	α	β	Rel Err	α	β	Rel Err
i860	-214	33.1828	3.60%	2828	30.9618	0.66%
Paragon	N/A	N/A	N/A	3502	4.8024	7.47%
CM-5	-3529	40.7289	0.29%	1883	15.8834	1.12%
SP1	8968	12.4721	9.88%	23675	12.1638	10.07%

Table 16: Least squares fit (microseconds) and relative error of combine times for $N = 0, \ldots, 50000$

6 Implementation and Portability Issues

6.1 Message Identifiers

Because it may be of interest to other researchers, a sketch of the BLACS' ID generation algorithms follows. There are two different algorithms, one for point to point ID generation, and one for when the operation involves a scope.

After the user has accepted the default, or has specified a legal range of msgid values for the BLACS to use (via the support routine `SHIFT_RANGE`), this range is split in two. The first range is reserved for point to point IDs, and the second is reserved for the scoped routines.

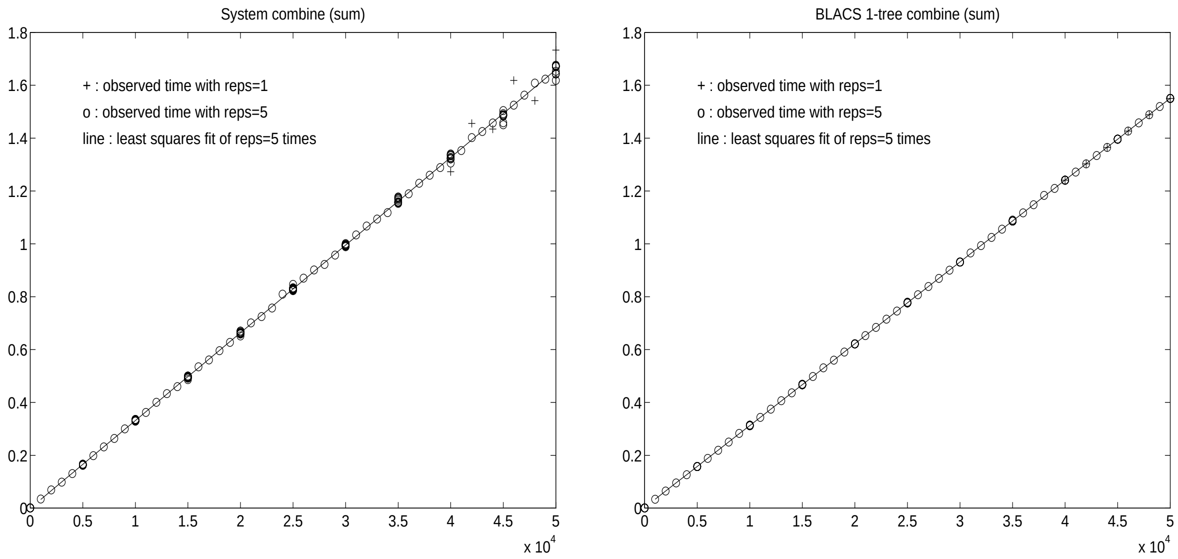

Figure 48: Variance between 32-processor i860 combine runs

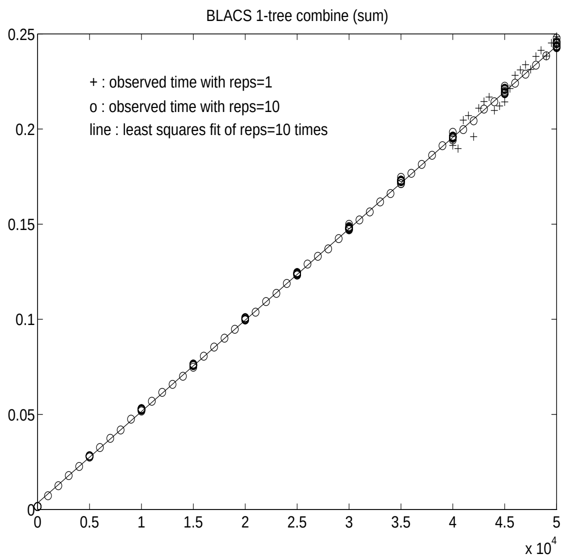

Figure 49: Variance between 32-processor Paragon combine runs

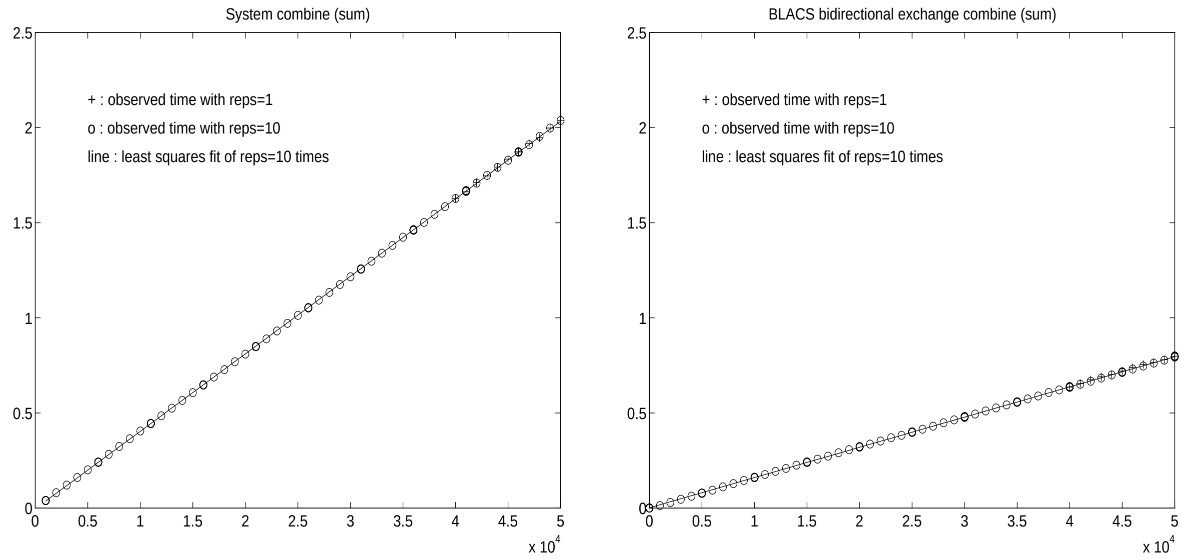

Figure 50: Variance between 32-processor CM-5 combine runs

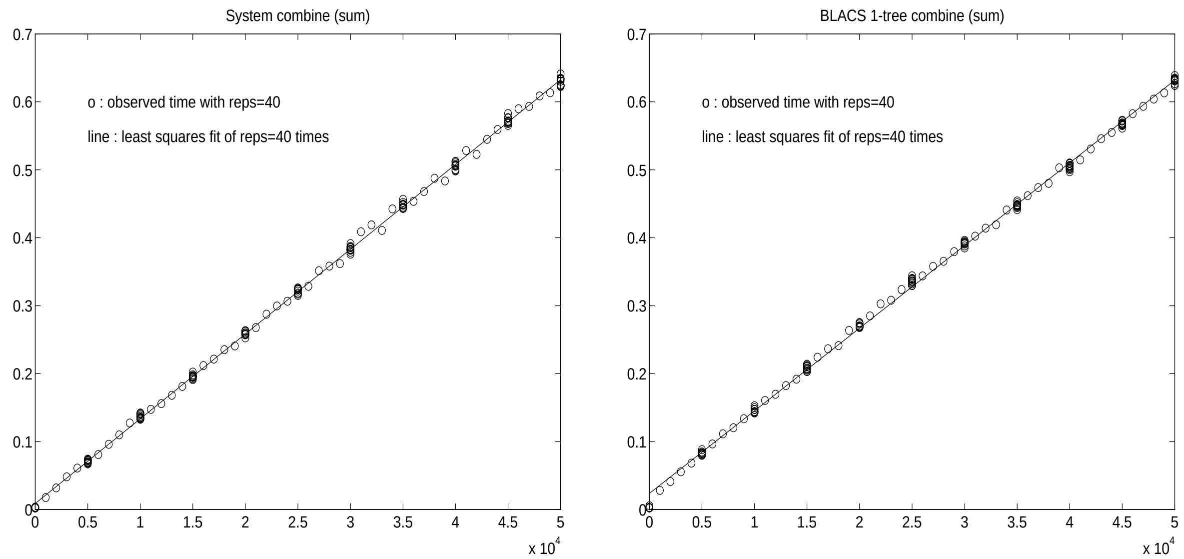

Figure 51: Variance between 32-processor SP1 combine runs

Before the details of the algorithm are described, the cost of this algorithm should be mentioned. In terms of memory, each process requires: 3 integer vectors of size N_g, 1 integer vector of size $\mathcal{P}$, and 1 integer vector of size $\mathcal{Q}$. The need for these vectors is explained below. The computation of a msgid requires roughly five integer operations.

6.1.1 Point To Point Message ID Generation

Let $N_g = \mathcal{P} * \mathcal{Q}$, the number of processes in the grid. Each process keeps two vectors (`sndcount` and `rcvcount`), each of length N_g, which contain a history of its point to point communication within the grid. These vectors are indexed by a *virtual* process ID (V_{pid}). The actual process ID cannot be used since, if the grid was set up by calling `GRIDMAP`, process IDs do not have any fixed relation to grid coordinates.

To obtain the virtual process ID of process {p, q}, the formula is $V_{pid} = p * \mathcal{Q} + q$. Therefore, `rcvcount[i]` contains the number of times the process has received from the process with $V_{pid} =$ `i`. Similarly, `sndcount[i]` indicates how many times the process has sent to $V_{pid} =$ `i`.

We split the point to point range of IDs into N_g subranges. Subrange `i` will be reserved for messages who's source has a V_{pid} of `i`. Messages from source `i` will have a msgid that starts in its subrange, plus the count of the number of sends it has made to the receiving process.

The receiver calculates the ID similarly. It knows the source's V_{pid} from the input parameters `RSRC` and `CSRC`. It then uses the same calculation as the source process did, except where the source uses the number of times it has sent to the receiver, the receiver uses its count of the number of times it has received from the sender.

6.1.2 Scoped Message ID Generation

At present, all of our scoped operations are implemented as a series of point to point communications. We could therefore use point to point msgids for scoped operations. There are several reasons this is unsatisfactory.

The most important reason is that if we are fanning information into a process, such as occurs in the combine operations (see section 5.4), one process may be receiving information from several other processes. If we use the above scheme of message generation, we must know who is sending the message before we can receive. In a fan-in operation, however, all the messages may be identical in the sense that the order in which they are received is unimportant. Then, for optimization reasons, it is not a good idea to force an order onto the receives (in the worst-case scenario, forcing an artificial ordering can almost double the time for the fan-in).

A second reason to avoid using the point to point ID generating routines is that a scoped operation may involve several steps, and at each one a new ID would have to be generated. Furthermore, the only information we have is the process that is the source (for broadcast) or destination (for combines) – for point to point msgids, we need to know who is doing the actual sending. The process presently sending is topology dependent, and the special case code could wind up being too costly.

We therefore implement a slightly different ID generator for scoped operations. The scoped ID subrange is subdivided in three, one for each scope. For each scope, two counters are kept. One is a scalar that counts the number of broadcast/sends the process has initiated. The other is a vector, which keeps track of the number of times the process has participated in a broadcast/receive initiated from other processes in the scope, and is therefore of the same length as the number processes in the scope. This means that the count for scope = 'ROW' is of size $\mathcal{Q}$, for scope = 'COLUMN' it's of size $\mathcal{P}$, and for scope = 'ALL', it's of size N_g.

The algorithm is now the same as for point to point, each subrange is further divided to provide a range for each sender, etc. Note that we now need only a scaler to count the number of sends, because the destination for each scope is always the same: the entire scope.

6.2 Buffering

At the moment, buffering is done differently on all platforms except the CM-5 and the SP1. On PVM, the PVM system does the buffering of messages, and the only buffer space the BLACS ever need is for combine operations, where a workspace corresponding to the parameter `M` is required (see section 5.4 for details).

6.2.1 Buffering On the Intel Machines

The Intel platforms feature locally-blocking sends, so if the user's data is already contiguous, the message is sent/received directly from/into the user's space. A BLACS buffer is required only when the user communicates non-contiguous data. There are two ways for the BLACS to be given non-contiguous data to send/receive. The first is to use one of the general rectangular routines, and specify the parameters `N > 1` and `M` $\neq$ `LDA` (see section 2.1 for explanation of matrices and these parameters). The other option leading to non-contiguous data is the sending/receiving of trapezoidal matrices. Since only a section of the whole matrix is to be sent/received, it must be packed into contiguous storage for the communication.

The buffering strategy on the Intel systems is therefore fairly straightforward. The system starts out with no buffers allocated. If the user issues a request to communicate non-contiguous data, a buffer of the correct size (`M * N`) is allocated. Allocating memory is not free, so we do not release the buffer once the send/receive is complete. If we now receive further calls which require a buffer space less than or equal to what we already have, we don't have to pay the cost of another memory allocation.

The BLACS buffer will be released on only three occasions. The first is at a call to the support routine `BLACSEXIT`, which indicates all use of the BLACS is over. The support routine `FREEBBUFF` exists so that if the user needs the space for his own code, he can explicitly free the BLACS buffer. Finally, if a non-contiguous message requires more buffer space than is currently available, the present buffer will be released and a new buffer of the correct size will be allocated.

6.2.2 Buffering for the CM-5 and SP1 Platforms

Since the CM-5 and SP1's native sends are globally-blocking, the BLACS must simulate a locally-blocking send using non-blocking sends coupled with buffering.

Buffering the data is a straightforward way to simulate locally-blocking sends. Each send request results in the BLACS copying the data to an internal buffer, starting an non-blocking send, and then returning control to the user. The BLACS must not touch the buffer until the non-blocking operation is complete, but the user does not need to know about that – it is handled behind the scenes.

To fully simulate locally-blocking sends requires dynamic buffering, which allows as many unreceived sends to be issued as available memory will support. However, the overhead in managing dynamic buffers can be large, and the sender's time must be optimized. This is due to the fact that when performing a send, the time required to initiate the send is added not only to the sender's time, but to the time of all processors waiting to receive from the sender as well. For instance, if a processor is sending to four other processors, any delay experienced by the sender also delays the receivers, assuming they have called the receive routine.

It is obvious that buffering causes a slight delay in each send, but we feel it is worth the cost of a memory copy to avoid the kinds of hanging problems previously discussed. If data being communicated is contiguous, we can still receive directly into the user's space.

In the first release of the CM-5 BLACS (the one presently available on netlib), due to its high overhead, dynamic buffering was not supported. Instead, two buffers at most were allocated. When those buffers filled up, if further unreceived send requests were posted, a hang would occur. This solution would make it so the code fragment given in section 3 would run, but if that code were changed so that each process sent *twice* to the other before posting the corresponding receives, a hang would occur.

This was a workable temporary solution, simply because it is rare to find codes that require more to prevent a hang. This state of affairs continued until a more general method was discovered.

The key concept for performing less expensive dynamic buffering is that all of the overhead associated with dynamic buffering should be performed *after* the non-blocking send is initiated, but before the function returns. The present buffering strategy is explained below.

There are three states a buffer can be in. The first state is *active*. Active buffers are buffers from which non-blocking operations are being performed. Usually there is only one operation per buffer, but during broadcasts, the BLACS may send up to $N_g - 1$ messages from the same buffer.

The second state is *ready*. A ready buffer is a buffer that is available for use. Only one ready buffer is kept by the BLACS.

The last state is transitory. When the non-blocking operations of an active buffer are finished, the buffer becomes *inactive*. When the BLACS poll and discover the buffer is inactive, the inactive buffer is compared with the present ready buffer (if a ready buffer exists). If the inactive buffer is larger than the ready buffer, the ready buffer is released, and the inactive buffer becomes the ready buffer. If the ready buffer is bigger than the inactive buffer, the inactive buffer is released.

In general then, the way this system works is that when a sending routine is called, it immediately uses the ready buffer for its packing. If the ready buffer does not exist, one of the needed size is allocated. If the ready buffer exists, but is too small, the present ready buffer is released, and a new buffer of the correct size is allocated. After the send is begun, the ready buffer is moved onto the active buffer queue. Then, before control is returned to the user, the active queue is checked for buffers that have become inactive. Inactive buffers become the ready buffer so that the next send operation won't have to allocate its own.

Whenever any of the main BLACS routines are called, the status of the buffers is checked. If there are any active buffers, they are polled to determine if they have become inactive. If they have, they are treated as described above.

This algorithm results in a system where at most one unused buffer is in memory. It may occur, however, that enough outstanding sends are issued that the BLACS are unable to allocate further memory. If this occurs, the BLACS call an emergency routine which waits for a user definable amount of time for active buffers to become inactive. If time expires before a buffer becomes available, it is assumed a hang has occurred (i.e. the user has issued many sends, but has not issued the corresponding receives), and the BLACS exit with an error message.

The most important thing to note about this algorithm is that the handling of the queues, the polling for non-blocking operation completion, etc., is done *after* the send is begun. This means that the sender will have to pay these costs, but those processors waiting to receive from him will not.

7 Future Directions

This section presents some proposed future directions for the BLACS. Some are more likely to be pursued than others. These directions are roughly separated into two categories. The first category involves extensions of the BLACS standards; optimizations to the present code make up the second.

7.1 Possible Extensions to the BLACS

7.1.1 Arbitrary Scopes

The 2D process grid allows for three natural scopes, as has been previously discussed. This may seem unnecessarily restrictive. It is a relatively simple matter to modify the scoped routines to allow for arbitrary scopes, where the scope is defined, for instance, by a linear array of processes passed in as a vector. However, there are certain drawbacks to this. First, all message IDs will have to be generated using the point to point algorithm presented in section 6.1, since the scoped ID generation algorithm depends on having static scopes. There is also an increased opportunity for user error, since it would be up to the user to ensure that everyone calling the operation had the correct scope vector.

Using arbitrary scopes would obviate the idea of a 2D process grid, and the entire interface would have to be re-designed. Therefore, if arbitrary scopes are later seen to be required for certain types of algorithms, a new interface using the same BLACS core routines will have to be written, and this version will not be the 2D BLACS, since the grid would no longer possess any special relevance.

7.1.2 Wildcard Receive

At present, the user must specify which process is the source of the incoming message in order to post a receive. This can result in operations that are inefficient because of unneeded ordering of receives. The most obvious example of this is the simple fan-in with functionally equivalent messages (i.e., a group of processors send data to a receiving processor, which does not care about the order in which it receives the messages). This kind of fan-in occurs regularly in combine operations, but by using the ID generating scheme presented in section 6.1, the BLACS avoid the unneeded ordering. If the fan-in is written using the BLACS' point to point communication, however, the user cannot avoid the ordering.

It is therefore proposed to extend the point to point receive by allowing the user to pass in `RSRC = -1`, which indicates that any message will be received. The address of the process that actually sent the message will be returned in `RSRC` and `CSRC`. This is potentially dangerous: the BLACS cannot compute a message ID until the sender is known, and therefore any outstanding message directed to the receiving process will be accepted. It will be the user's responsibility to ensure that no unwanted messages are outstanding when this wildcard receive is issued.

7.1.3 Additional Combine Operations

The BLACS presently provide three combine operations, which allow maximization, minimization, or summation on rectangular matrices. Several algorithms require trapezoidal summation, and there have been requests for exclusive OR, sum of squares, and other combines. It is clearly impractical for one package to support all conceivable operations. A promising idea, however, is to provide a combine called, for example, `vTUOP2D`, which is a trapezoidal user-defined operation. If `UPLO = 'G'`, the matrix will be assumed to have a general rectangular shape, rather than a trapezoidal.

In addition to the usual parameters, the user will pass in three function pointers. The first will be an initialization routine, the second points to the function which performs the user's operation (exclusive OR, for instance), and the final will be a routine which performs any needed post-operation computation. In this way, the BLACS combines may support almost any associative and commutative operation.

7.1.4 Built-in Debug and Timing Levels

To avoid unnecessary overhead, the BLACS presently perform almost no error checking and take no timings at all. However, when an error occurs in a code, it is often difficult to track the error down. Therefore, we propose to add the pre-processor variable `BlacsDebugLvl`. So that certain statistics useful in code optimization can be determined, the pre-processor `BlacsTimingLvl` will also be added. These variables will be used to selectively compile different portions of code. If both are set to 0, the code will be the same as today: no timing or debug information available.

At the present, roughly three debug levels seem practical. Level 1 would perform mainly parameter checking, and its effect on speed should be negligible.

Level two would be more active, and would involve using non-blocking receives coupled with polling to detect hangs. When such a hang occurred, messages explaining what operation was being performed, etc., should be helpful in finding the cause. This level of debugging would include everything short of off-process access of data in its attempt to discover errors. Its heavy use of polling would cause it to be noticeably slower than level 0 or 1.

Level three would be extremely intrusive. It could involve anything, including sending/receiving extra messages to ensure things are working correctly.

Since these levels are determined at compile time, no speed is lost if the user needs no debugging help. If required, however, the BLACS could be compiled with a higher level of debugging, and the user could link to the debug version until the code was fully developed. Then, for production runs, the code would be linked to the optimized version (debug level 0).

The timing level would work similarly, with several levels of increasingly intrusive timings. A survey of users is required before these levels may be finalized, but a few statistics are of obvious use. These include time spent in each BLACS routine, time spent waiting for/sending messages, number of messages sent, maximum, minimum, and average message length, etc. These kinds of data will allow the developer to get an idea of where the majority of time is spent, and thus where optimization is required.

7.2 Optimizations

There are many optimizations that are available for exploration. In this section we discuss some of the more interesting ideas.

7.2.1 Intel BLACS

The first implementation of the BLACS was on the Intel machines, and since we learn from experience (we hope), it is not surprising that this platform contains the most easily seen areas for improvement. The greatest opportunity for optimization should come from the use of non-blocking messages. At present, the Intel BLACS use no non-blocking messages at all. Those times when the BLACS copy data to a buffer anyway (when the message is not contiguous in memory), it certainly makes sense to exploit non-blocking sends. Even when the message is sent from the user's buffer, non-blocking sends may help speed up broadcasts, where the sender sends the same message to more than one processor.

A second area for improvement is in the combine's buffering. When enough memory is available, two buffers can be allocated, so that cache usage is maximized, and the result buffer can be used to send data directly.

The use of *forced type messages* needs to be investigated. Forced type messages are locally-blocking sends with no buffering. As previously mentioned, regular messages are buffered on the receiving process. To accomplish this, an Intel send first sends a request for buffer space to the receiver, and when an acknowledgment is returned, the actual message is sent. A forced type message avoids this buffer request, resulting in faster communication. Forced type messages are also the only practical way to allow a processor to simultaneously send and receive messages. Because this implementation does not use forced types for the bidirectional exchange algorithm, we have seen that it does poorly on the Intel. Intelligent use of forced type messages should noticeably decrease the time required for bidirectional exchange. Other uses of forced types should also be explored.

Finally, with the introduction of the Paragon, Intel added new routines to their communication library. Use of these new routines needs to be investigated, to see if it is worth having a Paragon-specific version of the BLACS.

7.2.2 CM-5 BLACS

It should first be noted that the primary problem on the CM-5 at this time is an inability to access the vector units of the machine. Because of the way the machine is set up, fortran 77 or C message passing codes basically see the machine as a collection of Spark-2 processors. In this configuration, codes are far more likely to be computation bound than communication bound. Therefore, unless faster processor speeds are achieved, optimizations to the BLACS are unlikely to have a real effect on the speed of most codes.

With this in mind, we briefly survey a few optimizations for the CM-5. We have seen the the CM-5's locally-blocking send is better, at least in regards to the echo test, than that presently used in the BLACS. We need to investigate where in the BLACS this primitive can be efficiently utilized. It seems likely that we can use the CM-5's locally-blocking send to support the BLACS point to point communication, and thus improve the BLACS point to point performance.

When the CM-5 BLACS were written, it had not been defined that the broadcast and combine operations were globally-blocking. We therefore did a data copy to make them (when possible) locally-blocking. Since they are now defined to be globally-blocking, we can save the cost of the data copy. This will result in even more efficient scoped operations, which are already faster than those provided by the system.

The CM-5 possesses a message passing layer beneath the CMMD layer presently used in the BLACS. This layer (called CMAML) can be used to build a very rich message passing system. For instance, a routine allowing a processor to execute a remote procedure call, i.e., call a routine on another processor, is available. This capability could be exploited in order to cause the receiver to do buffering, for instance.

There are numerous other optimizations that CMAML would allow. The present CM-5 BLACS implementation does not use CMAML because it is not guaranteed to remain constant as software is updated. Still, if the speed win is great enough, it may be worth having to update the code when CMAML is changed.

7.2.3 SP1 BLACS

As with the CM-5, now that the broadcast and combine operations are defined as globally blocking, we can speed up these operations by avoiding the memory copy.

We have seen that the SP1's combine and broadcast are faster than the BLACS. If this is still the case after optimization for the platform is finished, we can add the SP1's broadcast and combine as topologies for the BLACS. Unlike the other platforms, the SP1's primitives allow groupings which can be used to support the BLACS scoped operations.

7.2.4 PVM BLACS

By its very nature, little can be said about the underlying architecture of a PVM machine. One area where PVM often differs from other supported platforms, however, is that PVM is often implemented on systems where there is only one communication link for the entire system (ethernet, token ring, etc). In this case, most topologies are useless. Since only one process may be sending at a time in such a system, using trees or multiple rings for the broadcast changes nothing.

However, there is a topology that can be added that, at least theoretically, should provide speedup for combines. Broadcast topologies cannot be improved, because the entire time in the algorithm consists of message passing. Combines, as discussed earlier, have two time components. While one section of processors are communicating, another section can be performing the local operation (max/min/sum).

It is therefore proposed to add a multiring combine topology. The amount of speedup that can be obtained will depend on the ratio of T_o to T_c. If $T_o \geq T_c$, then we may use T_o/T_c rings to achieve a speedup of of roughly T_o/T_c. True speedup will be less than this, due to link contention and improper synchronization.

If $T_o \ll T_c$, the speedup obtained by using multiple rings will probably not be worth the added link contention. Unfortunately, for most systems using standard ethernet, this may be the case. However, we are increasingly seeing systems that, while they have only one link, that link is very fast. In this case, we should see some speedup. Therefore, a multiring scoped operator should be interesting. Codes that are strongly pipelined might conceivably benefit from this topology as well.

8 Conclusion

While there are many avenues of investigation still to be pursued, we believe we have met the basic goals of the project. Codes written using the BLACS can run unchanged on the iPSC2, i860, Touchstone Delta, Paragon, CM-5, SP1, and PVM. Support for scoped operations, message ID computation, sending of matrices, and locally-blocking sends greatly enhance the programmability and ease of use of the library.

We have shown that the only real loss of performance comes in point to point communication, where supporting locally-blocking sends causes the CM-5 BLACS, and to a lesser extent the SP1 BLACS, to compare unfavorably to the system code. We feel the added programmability of locally-blocking sends makes this sacrifice worthwhile.

In broadcasts and combines, the BLACS are quite competitive, and sometimes even beat the system primitives in minimizing maximal time in the algorithm. However, by varying the topology parameter, the BLACS allow for a much greater variety of broadcast/combine behavior, resulting in code that is more easily made to fit the user's specific needs.

REFERENCES

References

[1] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam., *PVM 3 User's Guide and Reference Manual*, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May, 1993

[2] MPI Forum, *MPI: A Message Passing Interface*, Proceedings of Supercomputing '93, pgs 878-885 IEEE Computer Society Press, 1993.

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[5] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.

[6] Jack J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley *Two Dimensional Basic Linear Algebra Communication Subprograms.* Environments and Tools for Parallel Scientific Computing, Elsevier Science Publishers B.V., 1993.

[7] E. Anderson, Z. Bai, C. Bischof, J.W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, 1990. (LAPACK Working Note 20).

[8] J. Dongarra, R. van de Geijn and D. Walker, *A Look at Scalable Dense Linear Algebra Libraries*, LAPACK Working Note 43, technical report, University of Tennessee, 1992.

[9] J. Dongarra and R. van de Geijn, *Two Dimensional Basic Linear Algebra Communication Subprograms*, LAPACK Working Note 37, technical report, University of Tennessee, 1991.

[10] J. Choi, J. Dongarra R.Pozo and D. Walker, *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, LAPACK Working Note 55, technical report, University of Tennessee, 1992.

[11] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn, "Global Combine on Mesh Architectures with Wormhole Routing, to appear in the proceedings of the *7th International Parallel Processing Symposium*, Newport Beach, CA, April 13-16, 1993

[12] Ching-Tien Ho and S. Lennart Johnsson, *Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes*, Proceedings of the 1986 International Conference on Parallel Processing, IEEE, 1986.

[13] T. H. DUNIGAN, *Performance of the INTEL iPSC/860 Hypercube* Technical Report ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May, 1990.

[14] G. A. GEIST AND M. T. HEATH AND B. W. PEYTON AND P. H. WORLEY, *A users' guide to PICL: a portable instrumented communication library*, Oak Ridge National Laboratory, September 1990.

APPENDICES

A Example Code: Matrix Vector Multiply

The following program performs a distributed matrix-vector multiply, and figures infinity norms of distributed vectors and matrices using the BLACS. This example uses point-to-point communication, global operations, and broadcasts. Notice in the code that we may make BLACS calls with only the first letter of scope and topology as parameters, or we may write out the full words. Figure 52 shows how the data would be distributed on a 2 x 2 processor grid. LM and LN refer to **L**ocal matrix rows and **L**ocal matrix columns, respectively.

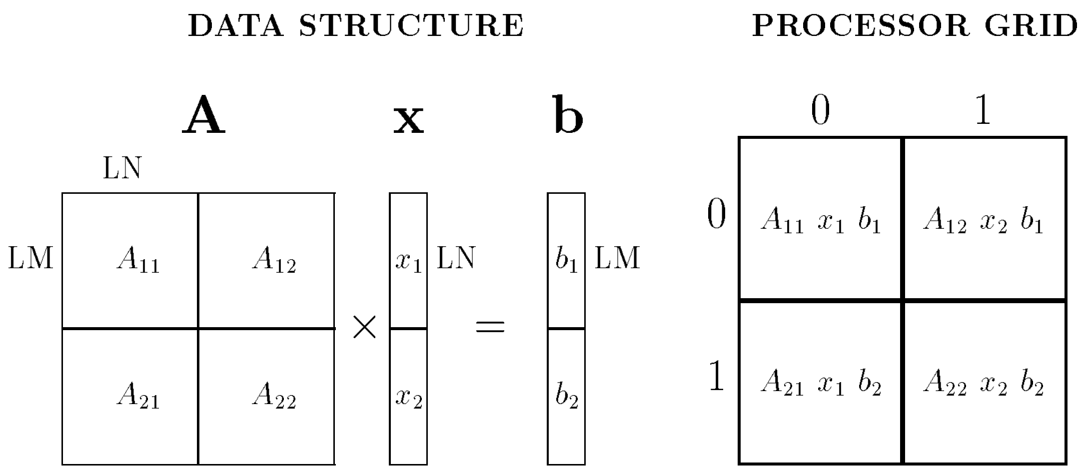

Figure 52: Matrix-vector multiply on 2 x 2 processor grid.

```
      PROGRAM MVMULT
*
*     .. External Functions ..
      INTEGER IDAMAX
      DOUBLE PRECISION DRAND, DINFNRM
      EXTERNAL IDAMAX, DRAND, DINFNRM
*
*     .. External Subroutines ..
      EXTERNAL AUXSETUP, BLACSINIT, GRIDINFO, DGMAX2D, DGSUM2D, DGEMV
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC INT, REAL, SQRT
*
*     .. Scalars  ..
      INTEGER LDA, LM, LN
      PARAMETER (LDA = 50)
      PARAMETER (LM = LDA)
      PARAMETER (LN = LDA)
      INTEGER IAM, NNODES, NPROW, NPCOL, MYROW, MYCOL
      INTEGER I, J, ITMP1, ITMP2
      DOUBLE PRECISION NORMA, NORMB, NORMX, NORMTST
*
*     .. Arrays ..
      DOUBLE PRECISION A(LM,LN), X(LN), B(LM)

*
*     Find out how many processors have been allocated for use
*
      CALL INITPINFO(IAM, NNODES)
*
*     For PVM only: if virtual machine not set up, allocate
*     it with 8 processes
*
      IF (NNODES .LT. 1) THEN
         NNODES = 8
         CALL AUXSETUP(IAM, NNODES)
      END IF
*
*     Figure processor grid I want to use
*
      NPROW = INT(SQRT(REAL(NNODES)))
      NPCOL = NNODES / NPROW
*
*     Define the processor grid, and get grid information
*
```

```
      CALL BLACSINIT(NPROW, NPCOL)
      CALL GRIDINFO(NPROW, NPCOL, MYROW, MYCOL)
*
*     If I'm not in new processor grid, goto end of program
*
      IF (MYCOL .GE. NPCOL .OR. MYROW .GE. NPROW) GOTO 100
*
*     Generate distributed matrix A
*
      A(1,1) = DRAND(MYROW*NPCOL+MYCOL)
      DO 10 J = 1, LN
         DO 10 I = 1, LM
            A(I,J) = DRAND(0)
10    CONTINUE

*
*     Figure the infinity norm of A
*
      NORMA = DINFNRM(LM, LN, A, LDA, B)
*
      IF (MYROW .EQ. 0) THEN
*
*        Generate vector X, distributed over processor row 0.
*
         X(1) = DRAND(MYCOL)
         DO 20 J = 1, LN
            X(J) = DRAND(0)
20       CONTINUE
*
*     All processor rows need a copy of X, so broadcast within columns
*
         CALL DGEBS2D('COLUMN', 'HYPERCUBE', LN, 1, X, LN)
*
      ELSE
*
*        Receive my piece of X from processor row 0.
*
         CALL DGEBR2D('COLUMN', 'HYPERCUBE', LN, 1, X, LN, 0, MYCOL)
*
      END IF
*
*     Figure the infinity norm of X
*
      NORMX = X(IDAMAX(LN, X, 1))
      CALL DGMAX2D('R', 'H', 1, 1, NORMX, 1, ITMP1, ITMP2, 1, -1, 0)
```

```
*
*     Do b = A*x, where b is distributed over a processor column
*     NOTE: all processor columns have a copy of b
*
*     perform local A*x
*
      CALL DGEMV('N', LM, LN, 1.0D0, A, LDA, X, 1, 0.0D0, B, 1)
*
*     Add in pieces of A*x done on other processors
*
      CALL DGSUM2D('ROW', 'HYPERCUBE', LM, 1, B, LM, -1, 0)
*
*     Figure the infinity norm of B
*
      NORMB = B(IDAMAX(LM, B, 1))
      CALL DGMAX2D('C', 'H', 1, 1, NORMB, 1, ITMP1, ITMP2, 1, -1, 0)
*
*     Print out norms
*
      IF (MYROW.EQ.0 .AND. MYCOL.EQ.0) THEN
         WRITE (*, 1000), 'A', NORMA
         WRITE (*, 1000), 'x', NORMX
         WRITE (*, 1000), 'b', NORMB
      END IF
*
*     Mainly in order to show an example of using point-to-point
*     communication, the following test is done: is processor {0,0}'s
*     NORMB the same as processor {nprow-1, npcol-1}'s.
*
      IF (MYROW .EQ. NPROW-1 .AND. MYCOL .EQ. NPCOL-1) THEN
         CALL DGESD2D(1, 1, NORMB, 1, 0, 0)
      ELSE IF (MYROW .EQ. 0 .AND. MYCOL .EQ. 0) THEN
         CALL DGERV2D(1, 1, NORMTST, 1, NPROW-1, NPCOL-1)
         IF (NORMTST .NE. NORMB) WRITE(*, 2000) NORMB, NORMTST
      END IF
*
100   STOP
*
1000  FORMAT ('||',A,'|| = ',G20.15)
2000  FORMAT ('ERROR: ||B||''s do not match. Values are: ',
     $        G20.15,G20.15)
*
*     End program MVMULT
*
      END
```

```
      DOUBLE PRECISION FUNCTION DINFNRM(LM, LN, A, LDA, WORK)
*
*  -- BLACS example routine --
*
*     .. Scalar Arguments ..
      INTEGER LM, LN, LDA
*
*     .. Array Arguments ..
      DOUBLE PRECISION A(LDA, *), WORK(*)
*
*  PURPOSE:
*  ========
*
*  Compute the infinity norm of a distributed matrix, where
*  the matrix is spread across a 2D processor grid.
*
*  Arguments
*  =========
*
*  LM        (input) INTEGER
*            Number of rows of the global matrix owned by this processor.
*
*  LN        (input) INTEGER
*            Number of columns of the global matrix owned by this processor.
*
*  A         (input) DOUBLE PRECISION, dimension (LDA,N)
*            The matrix who's norm you wish to compute.
*
*  LDA       (input) INTEGER
*            Leading Dimension of A.
*
*  WORK      (temporary) DOUBLE PRECISION array, dimension (LM)
*            Temporary work space used for summing rows.
*
*     .. External Subroutines ..
      EXTERNAL DGSUM2D, DGMAX2D, GRIDINFO
*
*     .. External Functions ..
      INTEGER IDAMAX
      DOUBLE PRECISION DASUM
*
*     .. Local Scalars ..
      INTEGER NPROW, NPCOL, MYROW, MYCOL,  I, J
      DOUBLE PRECISION MAX
*
```

```
*     .. Executable Statements ..
*
*
*     Get processor grid information
*
      CALL GRIDINFO(NPROW, NPCOL, MYROW, MYCOL)
*
*     Add all local rows together
*
      DO 20 I = 1, LM
         WORK(I) = DASUM(LN, A(I,1), LDA)
20    CONTINUE
*
*     Find sum of global matrix rows and store on column 0 of processor grid
*
      CALL DGSUM2D('r', '1', LM, 1, WORK, LM, MYROW, 0)
*
*     Find maximum sum of rows for supnorm
*
      IF (MYCOL .EQ. 0) THEN
         MAX = WORK(IDAMAX(LM,WORK,1))
         IF (LM .LT. 1) MAX = 0.0D0
         CALL DGMAX2D('c', 'h', 1, 1, MAX, 1, I, J, 1, -1, 0)
      END IF
*
*     Processor column 0 has answer: send answer to all nodes
*
      IF (MYCOL.EQ.0) THEN
         CALL DGEBS2D('r', 'h', 1, 1, MAX, 1)
      ELSE
         CALL DGEBR2D('r', 'h', 1, 1, MAX, 1, 0, 0)
      END IF
*
      DINFNRM = MAX
*
      RETURN
*
*     End of DINFNRM
*
      END
```

B Obtaining the BLACS from Netlib

Netlib is an automated system which allows users to obtain software, papers, etc., via e-mail. The BLACS are in the ScaLAPACK directory on netlib. To obtain a list of of the presently available ScaLAPACK codes, mail to `netlib@ornl.gov` and in the body of the message type `send index from scalapack`. This index will also explain other options for downloading software, such as Xnetlib or anonymous rcp.

To obtain the BLACS User's Guide from netlib, mail to `netlib@ornl.gov` and in the body of the message type `send blacs_ug.ps from scalapack`. the various versions of the BLACS can be obtained by mailing to `netlib@ornl.gov`, and typing the message indicated in table 17 in the body of the message.

VERSION	MESSAGE
Intel	`send intelblacs.uue from scalapack`
CM-5	`send cm5blacs.uue from scalapack`
PVM	`send pvmblacs.uue from scalapack`

Table 17: How to obtain various BLACS versions from netlib

C Quick Reference To The BLACS

Fortran Interface

Initialization

```
INITPINFO( MYPNUM, NPROCS )
AUXSETUP( MYPNUM, NPROCS )
SETPVMTIDS( NTASKS, TIDS )
BLACSINIT( NPROW, NPCOL )
GRIDMAP( USERMAP, LDUMAP, NPROW, NPCOL )
IDRANGES( MINSND, MAXSND, MINBS, MAXBS )
SHIFT_RANGE( MINSND, MAXSND, MINBS, MAXBS )
```

Sending

```
□GESD2D(                          M, N, A, LDA, RDEST, CDEST )

□GEBS2D( SCOPE, TOP,              M, N, A, LDA               )

□TRSD2D(             UPLO, DIAG, M, N, A, LDA, RDEST, CDEST )

□TRBS2D( SCOPE, TOP, UPLO, DIAG, M, N, A, LDA               )
```

Receiving

```
□GERV2D(                          M, N, A, LDA, RSRC, CSRC )

□GEBR2D( SCOPE, TOP,              M, N, A, LDA, RSRC, CSRC )

□TRRV2D(             UPLO, DIAG, M, N, A, LDA, RSRC, CSRC )

□TRBR2D( SCOPE, TOP, UPLO, DIAG, M, N, A, LDA, RSRC, CSRC )
```

Global Maximum, Minimum, Sum

```
□GMAX2D( SCOPE, TOP, M, N, A, LDA, RA, CA, LDIA, RDEST, CDEST )

□GMIN2D( SCOPE, TOP, M, N, A, LDA, RA, CA, LDIA, RDEST, CDEST )

□GSUM2D( SCOPE, TOP, M, N, A, LDA,              RDEST, CDEST )
```

Auxiliary Routines

```
BARRIER( SCOPE )
GRIDINFO(  NPROW, NPCOL, PROW, PCOL )
IDRANGES( MINSND, MAXSND, MINBS, MAXBS )
KBRID( SCOPE, RSRC, CSRC )
KBSID( SCOPE )
KPNUM( PROW, PCOL )
KRECVID( RSRC, CSRC )
KSENDID( RSRC, CSRC, RDEST, CDEST )
PCOORD( PNUM, PROW, PCOL )
SETBRANCHES( NBRANCHES )
FREEBUFF
BLACSEXIT( CONTINUE )
```

Note that all routines preceded by a □ have the following prefixes: S, D, C, Z, I.

Declarations

```
INTEGER         CDEST, CONTINUE, CSRC, LDA, LDIA
INTEGER         M, MAXBS, MAXSND, MINBS, MINSND
INTEGER         N, NBRANCHES, NPCOL, NPROW
INTEGER         PCOL, PNUM, PROW, RDEST, RSRC
INTEGER         CA( LDIA, * ), RA( LDIA, * )
CHARACTER       DIAG, SCOPE, TOP, UPLO
REAL/DOUBLE     A( LDA, * )
 or
COMPLEX/COMPLEX*16 A( LDA, * )
 or
INTEGER         A( LDA, * )
```

Meaning of prefixes

```
S - REAL
D - DOUBLE PRECISION
C - COMPLEX
Z - COMPLEX*16
I - INTEGER

K - INTEGER Function

GE - GENERAL
TR - TRAPEZOIDAL

SD - SEND
RV - RECEIVE

BS - BROADCAST SEND
BR - BROADCAST RECEIVE

GMAX - GLOBAL element-wise MAXIMUM
GMIN - GLOBAL element-wise MINIMUM
GSUM - GLOBAL element-wise SUMMATION
```

Options

```
UPLO  = 'Upper triangular', 'Lower triangular';
DIAG  = 'Non-unit triangular', 'Unit triangular';
SCOPE = 'All', 'row', 'column';
TOP   = (SEE DESCRIPTION BELOW).
```

Broadcast Topologies

```
TOP   = 'I' increasing ring;
      = 'D' decreasing ring;
      = 'H' hypercube (minimum spanning tree);
      = 'S' split-ring;
      = 'F' fully connected;
        (calls multipath with NPATHS = NNODES-1)
      = 'M' : nodes divided into I increasing
              rings, where I is set with call
              to SETBRANCHES
      = '1' : tree broadcast with NBRANCHES = 1
      = '2' : tree broadcast with NBRANCHES = 2
      = '3' : tree broadcast with NBRANCHES = 3
      = '4' : tree broadcast with NBRANCHES = 4
      = '5' : tree broadcast with NBRANCHES = 5
      = '6' : tree broadcast with NBRANCHES = 6
      = '7' : tree broadcast with NBRANCHES = 7
      = '8' : tree broadcast with NBRANCHES = 8
      = '9' : tree broadcast with NBRANCHES = 9
      = 'T' : tree broadcast with NBRANCHES = I,
              where I is set with call to
              SETBRANCHES
```

Global Topologies

```
TOP   = '1' : tree gather with NBRANCHES = 1
      = '2' : tree gather with NBRANCHES = 2
      = '3' : tree gather with NBRANCHES = 3
      = '4' : tree gather with NBRANCHES = 4
      = '5' : tree gather with NBRANCHES = 5
      = '6' : tree gather with NBRANCHES = 6
      = '7' : tree gather with NBRANCHES = 7
      = '8' : tree gather with NBRANCHES = 8
      = '9' : tree gather with NBRANCHES = 9
      = 'T' : tree gather with NBRANCHES = I,
              where I is set with call to
              SETBRANCHES
      = 'F' : Fully connected --  calls tree gather
              with NBRANCHES = NNODES - 1
      = 'H' : if IRDEST = -1, a specialized
              "leave on all" hypercube topology
              called bidirectional exchange is used.
              Otherwise, TOP = '1' is substituted.
```

Initialization

An initial call to BLACSINIT or GRIDMAP must occur at the beginning of your program before calling any non-initialization BLACS routine. BLACSINIT or GRIDMAP may be called repetitively in order to change the processor grid.

A call to BLACSINIT or GRIDMAP must be made after SHIFT_RANGE in order to have the range shift take affect.

Notation

Any subroutine parameter argument that is underlined is an output argument. If a subroutine is underlined it is a function that returns a value. Specifically, if it is prefixed by K-, it returns an integer value. The prefix P- stands for processor. For example, integer function KPNUM returns the processor number for the specified grid coordinates.

Topology Hints

Topologies allow the user to optimize communication patterns for a particular operation. For the Intel BLACS, good defaults are broadcast and global TOP = 'HYPERCUBE'. For the PVM BLACS, good default values are broadcast TOP = 'I' and global TOP = '1'. For more details read the paper referenced below.

References

J. J. Dongarra, LAPACK Working Note 34, *Workshop on the BLACS*, Computer Science Dept. Technical Report CS-91-134, University of Tennessee, Knoxville, May, 1991. To receive a postscript copy, send email to netlib@ornl.gov and in the mail message type: `send lawn34.ps from lapack/lawns.` For a more up-to-date report, send your request for the BLACS users' guide to `blacs@cs.utk.edu`.

Obtaining the BLACS

In order to get instructions for downloading the BLACS, send email to netlib@ornl.gov and in the body of the message type `send index from scalapack`.

Send comments and questions to `blacs@cs.utk.edu`.

Basic

Linear

Algebra

Communication

Subprograms

Quick Reference Guide

Release 1.0BETA

April 22, 1994

University of Tennessee

D Timing Codes

This appendix contains the actual code used to perform the timings given in this paper. It is impossible to construct a 100% correct test for some of the quantities measured here. We therefore present the timers that were used, so the biases, if any, can be examined by the reader.

D.1 T_c Timing Code

```
#include <stdio.h>

#define BLACS 1
#define INTEL 2
#define SP1 3
#define CM5 4

#define BLOCK 0
#define INTR 1 /* are interrupts off (0) or on? */

#define dwalltime00_ dcputime00_
#define SYS BLACS

#if (SYS == BLACS)

#define PLAT "Intel BLACS"
#define Tsend(msgid, dest, length, buff)\
        sgesd2d_(&(length), &one, (buff), &(length), &zero, &(dest));
#define Trecv(msgid, src, length, buff)\
        sgerv2d_(&(length), &one, (buff), &(length), &zero, &(src));

#elif (SYS == INTEL)

#define PLAT "Intel"
#define dwalltime00_ dclock
#define Tsend(msgid, dest, length, buff)\
        csend((msgid), (buff), (length), (dest), 0);
#define Trecv(msgid, src, length, buff)\
        crecv((msgid), (buff), (length));

#elif (SYS == CM5)

#include </usr/include/cm/cmmd.h>
#define PLAT "CM5"
#define dwalltime00_ dcputime00_
#if (BLOCK)
#define Tsend(msgid, dest, length, buff)\
        CMMD_send_block((dest), (msgid), (buff), (length));
#else
#define Tsend(msgid, dest, length, buff)\
        CMMD_send_noblock((dest), (msgid), (buff), (length));
#endif
#define Trecv(msgid, src, length, buff)\
        CMMD_receive_block((src), (msgid), (buff), (length));

#elif (SYS == SP1)

#define PLAT "SP1"
#define Tsend(msgid, dest, length, buff)\
```

```
        mpc_bsend((buff), (length), (dest), (msgid));
#define Trecv(msgid, src, length, buff)\
        mpc_brecv((buff), (length), &(src), &(msgid), &(itmp));

#endif

main(nargs, args)
int  nargs;
char  *args[];
/*
 *  Measures time for point to point communication.  Usage:
 *  Tc <len1> <lenN> <inclen> <repititions> <sending node> <1st recving node>
 *        <last recving node> <recv increment> [<outfile>]
 *  if <recv increment> greater than <last recv node>, then increment is by
 *  power of 2.
 */
{
   double dwalltime00_();
   char *outfile;
   FILE *fp=stdout;
   int len, len1, lenN, inclen, reps, sender, recv1, recvN, recvinc;
   int iam, nnodes, msgid, dest, length, i, j, itmp;
   int *ibuff=NULL;
   int one=1, zero=0;
   double time1, *time;

/*
 * Get my node number and the number of nodes in system
 */
#if (SYS == BLACS)
   initpinfo_(&iam, &nnodes);
#elif (SYS == INTEL)
   iam = mynode();
   nnodes = numnodes();
#elif (SYS == CM5)
   iam = CMMD_self_address();
   nnodes = CMMD_partition_size();
   CMMD_node_timer_clear(63);
   CMMD_node_timer_start(63);
   CMMD_fset_io_mode(stdout, CMMD_independent);
   CMMD_fset_io_mode(stderr, CMMD_independent);
#if (INTR)
   CMMD_enable_interrupts();
#else
   CMMD_disable_interrupts();
#endif

#elif (SYS == SP1)
   mpc_environ(&nnodes, &iam);
#endif
/*
```

```
 * Read in params from command line, or use defaults
 */
   if (nargs >= 9)
   {
      len1 = atoi(args[1]);
      lenN = atoi(args[2]);
      inclen = atoi(args[3]);
      reps = atoi(args[4]);
      sender = atoi(args[5]);
      recv1 = atoi(args[6]);
      recvN = atoi(args[7]);
      nnodes = recvN + 1;
      recvinc = atoi(args[8]);
      if (nargs > 9) fp = fopen(args[9], "w");
   }
   else
   {
      if (iam == sender)
         fprintf(stderr,
         "Incorrect number of parameters.  Should be 7.  Using defaults.\n");
      len = 1024;
      reps = 10;
      sender = 0;
      recv1 = 1;
      if (nnodes > 0) recvN = nnodes - 1;
      else recvN = nnodes = 4;
      recvinc = 1;
   }

#if (SYS == BLACS)
   auxsetup_(&iam, &nnodes);
   blacsinit_(&one, &nnodes);
#endif

   time = (double *) malloc(nnodes * sizeof(*time));

   for(dest=recv1, i=0; dest <= recvN; i++)
   {
      if (recvinc < 0) dest = (dest << 1) + 1;
      else dest += recvinc;
   }
   for (len=len1, j=0; len <= lenN; j++)
   {
      if (inclen < lenN) len += inclen;
      else len *= len;
   }
   if (iam == sender)
      fprintf(fp,
      "Start of %d node '%s' run: sender=%d, nlengths=%d, nrecvs=%d, reps=%d.\n",
      nnodes, (PLAT), sender, j, i, reps);
```

```
   len = len1;
   while (len <= lenN)
   {
#if (SYS == BLACS)
      length = len/4;
#else
      length = len;
#endif
      if (ibuff) free(ibuff);
      ibuff = (int *) malloc(len);
      if (iam == sender)
         fprintf(fp,
" Message Len=%d, reps=%d, sender=%d, recv1=%d, recvN=%d, recvinc=%d.\n",
           len, reps, sender, recv1, recvN, recvinc);

      for(dest=recv1, i=0; dest <= recvN; i++)
      {
         for (j=0; j < len/4; ibuff[j++]=0);  /* page in ibuff if needed */
         msgid = dest;

         if (iam == sender)
         {
/*
 *          Send out wake-up message, get back ready acknowledgement
 */
            Tsend(msgid, dest, zero, ibuff);
            Trecv(msgid, dest, zero, ibuff);
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
               Tsend(msgid, dest, length, ibuff);
               Trecv(msgid, dest, length, ibuff);
            }
            time[i] = dwalltime00_() - time1;
         }
         else if (iam == dest)
         {
/*
 *          Wait for wake-up call, then tell sender I'm ready
 */
            Trecv(msgid, sender, zero, ibuff);
            Tsend(msgid, sender, zero, ibuff);
            for (i=0; i < reps; i++)
            {
               Trecv(msgid, sender, length, ibuff);
               Tsend(msgid, sender, length, ibuff);
            }
         }
         if (recvinc < 0) dest = (dest << 1) + 1;
         else dest += recvinc;
      }
```

```
      if (iam == sender)
      {
         for(dest=recv1, i=0; dest <= recvN; i++)
         {
            fprintf(fp,
"  Node %d send to %d: length=%d, Tc = %8.2lf (us), ThruPut = %lf (KB/s).\n",
               sender, dest, len, (0.5e6*time[i])/reps,
               (len*reps)/(1000.0*time[i]));
            if (recvinc < 0) dest = (dest << 1) + 1;
            else dest += recvinc;
         }
         fprintf(fp, " End of tests, length=%d.\n", len);
      }
      if (inclen < lenN) len += inclen;
      else len *= len;
   }
   if (iam == sender) fprintf(fp, "End of Run.\n");
   if (fp != stdout) fclose(fp);
}
```

D.2 T_s Timing Code

```
#include <stdio.h>

#define BLACS 1
#define INTEL 2
#define SP1 3
#define CM5 4

#define INTR 1 /* are interrupts off (0) or on? */
#define BLOCK 1  /* use blocking send? */

#define SYS BLACS
#define dwalltime00_ dcputime00_

#if (SYS == BLACS)

#define PLAT "CM5 BLACS"
#define Tsend(msgid, dest, length, buff)\
        sgesd2d_(&(length), &one, (buff), &(length), &zero, &(dest));
#define Trecv(msgid, src, length, buff)\
        sgerv2d_(&(length), &one, (buff), &(length), &zero, &(src));

#elif (SYS == INTEL)

#define PLAT "Intel"
#define dwalltime00_ dclock
#define Tsend(msgid, dest, length, buff)\
        csend((msgid), (buff), (length), (dest), 0);
#define Trecv(msgid, src, length, buff)\
        crecv((msgid), (buff), (length));

#elif (SYS == CM5)

#include </usr/include/cm/cmmd.h>
#define PLAT "CM5"
#define dwalltime00_ dcputime00_
#if (BLOCK)
#define Tsend(msgid, dest, length, buff)\
        CMMD_send_block((dest), (msgid), (buff), (length));
#else
#define Tsend(msgid, dest, length, buff)\
        CMMD_send_noblock((dest), (msgid), (buff), (length));
#endif
#define Trecv(msgid, src, length, buff)\
        CMMD_receive_block((src), (msgid), (buff), (length));

#elif (SYS == SP1)

#define PLAT "SP1"
#define Tsend(msgid, dest, length, buff)\
```

```
        mpc_bsend((buff), (length), (dest), (msgid));
#define Trecv(msgid, src, length, buff)\
        mpc_brecv((buff), (length), &(src), &(msgid), &(itmp));

#endif

main(nargs, args)
int  nargs;
char  *args[];
/*
 *  Measures time for point to point communication.  Usage:
 *  Ts <outfile> <# reps> <len1> <lenN> <inclen> <sending node>
 *     <recv1> .. <recvN>
 */
{
   double dwalltime00_();
   char *outfile;
   FILE *fp=stdout;
   int len, len1, lenN, inclen, reps, sender, nrecvers;
   int iam, nnodes, msgid, dest, length, i, j, itmp;
   int *ibuff=NULL, *recvers=NULL;
   int one=1, zero=0;
   double time1, time;

/*
 * Get my node number and the number of nodes in system
 */
#if (SYS == BLACS)
   initpinfo_(&iam, &nnodes);
#elif (SYS == INTEL)
   iam = mynode();
   nnodes = numnodes();
#elif (SYS == CM5)
   iam = CMMD_self_address();
   nnodes = CMMD_partition_size();
   CMMD_node_timer_clear(63);
   CMMD_node_timer_start(63);
   CMMD_fset_io_mode(stdout, CMMD_independent);
   CMMD_fset_io_mode(stderr, CMMD_independent);

#if (INTR)
   CMMD_enable_interrupts();
#else
   CMMD_disable_interrupts();
#endif

#elif (SYS == SP1)
   mpc_environ(&nnodes, &iam);
#endif
/*
 * Read in params from command line, or use defaults
```

```
 */
   if (nargs >= 8)
   {
      if ( !strcmp(args[1], "stdout") ) fp = stdout;
      else if ( !strcmp(args[1], "stderr") ) fp = stderr;
      else fp = fopen(args[1], "w");
      reps = atoi(args[2]);
      len1 = atoi(args[3]);
      lenN = atoi(args[4]);
      inclen = atoi(args[5]);
      sender = atoi(args[6]);
      nrecvers = nargs - 7;
      recvers = (int *) malloc( nrecvers * sizeof(*recvers) );
      nnodes = 1;
      for (i=0; i < nrecvers; i++)
      {
         recvers[i] = atoi(args[7+i]);
         if (recvers[i] >= nnodes) nnodes = recvers[i] + 1;
      }
   }
   else
   {
      if (iam == 0) fprintf(stderr,
         "Incorrect number of parameters.  Should be >= 8.  Using defaults.\n");
      fp = stdout;
      reps = 1;
      len1 = 0; lenN = 100; inclen = 10;
      sender = 0;
      nrecvers = 1;

      recvers = (int *) malloc( nrecvers * sizeof(*recvers) );
      for (i=0; i < nrecvers; i++) recvers[i] = i+1;
      nnodes = nrecvers + 1;
   }

#if (SYS == BLACS)
   auxsetup_(&iam, &nnodes);
   blacsinit_(&one, &nnodes);

#endif

   for (len=len1, j=0; len <= lenN; j++)
   {
      if (inclen < lenN) len += inclen;
      else len *= 2;
   }
   if (iam == sender)
   {
      fprintf(fp,
      "Start of %d node '%s' run: sender=%d, nlengths=%d, nrecvs=%d, reps=%d.\n",
      nnodes, (PLAT), sender, j, nrecvers, reps);
```

```
      fprintf(fp, "Receivers:");
      for (i=0; i < nrecvers; i++) fprintf(fp, " %d",recvers[i]);
      fprintf(fp, "\n");
   }
/*
 * Quit if I am not sender or recver
 */
   else
   {
      for(i=0; i < nrecvers; i++) if (recvers[i] == iam) break;
      if (i == nrecvers) exit(0);
   }

   if (ibuff) free(ibuff);
   ibuff = (int *) malloc(lenN);
   len = len1;
   while (len <= lenN)
   {
#if (SYS == BLACS)
      length = len/4;
#else
      length = len;
#endif
      if (iam == sender)
         fprintf(fp, " Message Len=%d.\n", len);

      for (j=0; j < len/4; ibuff[j++]=0);  /* page in ibuff if needed */
      msgid = 0;

      if (iam == sender)
      {
/*
 *       Send out wake-up message, get back ready acknowledgement
 */
         for(i=0; i < nrecvers; i++)
         {
            j = zero;
            Tsend(msgid, recvers[i], j, ibuff);
            Trecv(msgid, recvers[i], j, ibuff);
         }

         time1 = dwalltime00_();
         for (j=0; j < reps; j++)
            Tsend(msgid, recvers[j%nrecvers], length, ibuff);
         time = dwalltime00_() - time1;
      }
      else
      {
/*
 *       Wait for wake-up call, then tell sender I'm ready
 */
```

```
         j = zero;
         Trecv(msgid, sender, j, ibuff);
         Tsend(msgid, sender, j, ibuff);

         for (i=0; i < reps; i++)
            if (iam == recvers[i%nrecvers]) Trecv(msgid, sender, length, ibuff);
      }
      if (iam == sender)
      {
         fprintf(fp, "  %d: length=%d, Ts = %8.2f (us).\n",
                 sender, len, (1.0e6*time)/reps);
         fprintf(fp, " End of tests, length=%d.\n", len);
      }
      if (inclen < lenN) len += inclen;
      else len *= 2;
   }
   if (iam == sender) fprintf(fp, "End of Run.\n");
   if (fp != stdout && fp != stderr) fclose(fp);
}
```

D.3 Broadcast Timing Code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BLACS 1
#define INTEL 2
#define SP1 3
#define CM5 4

#define SYS CM5

#if (SYS == INTEL)

#define PLAT "Intel"
#define dwalltime00_ dclock
#define Tsync() gsync()

#elif (SYS == CM5)

#define PLAT "CM5"
#define dwalltime00_ dcputime00_
#define Tsync() CMMD_sync_with_nodes()

#elif (SYS == SP1)

#define PLAT "SP1"
#define Tsync() mpc_sync(allgrp)

#endif

main(int nargs, char *args[])
/*
 *  Measures time for point to point communication.  Usage:
 *  Tbs <top> <len1> <lenN> <inclen> <repititions> <src node> [<outfile>]
 *  if <recv increment> less than 1, then increment is by power of 2.
 */
{
   double dwalltime00_();

   FILE *fp=stdout;
   char *outfile, *systyp;
   char top;
   int len, len1, lenN, inclen, nlens, reps, src, rsrc;
   int iam, nnodes, msgid, length, i, j, itmp, allgrp;
   int one=1, zero=0;
   int *rA=NULL;
   double *dbuff=NULL;
   double time1, time, srcT, maxT, minT, avgT;
```

```
/*
 * Get my node number and the number of nodes in system
 */
   initpinfo_(&iam, &nnodes);

/*
 * Read in params from command line
 */
   if (nargs >= 7)
   {
      top = *args[1];
      len1 = atoi(args[2]);
      lenN = atoi(args[3]);
      inclen = atoi(args[4]);
      reps = atoi(args[5]);
      src = atoi(args[6]);
   }
   else
   {
       fprintf(stderr, "\nIncorrect usage\n");
       exit(1);
   }

   auxsetup_(&iam, &nnodes);
   blacsinit_(&one, &nnodes);

   if ( (iam == 0) && (nargs > 7) ) fp = fopen(args[7], "w");

#if (SYS == SP1)
   rA = (int *) malloc(sizeof(int)*4);
   mpc_task_query(rA, 4, 3);
   allgrp = rA[3];
   free(rA);
#endif

   dbuff = (double *) malloc(lenN*sizeof(double));
   assert (dbuff != NULL);

/*
 *  If we're on a system that requires buffering even for contiguous messages,
 *  get the buffer of largest size by performing broadcast
 */
#if ( (SYS == SP1) || (SYS == CM5) )
   if (top != 'p')
   {
      if (iam == 0) dgebs2d_("a", &top, &lenN, &one, dbuff, &lenN);
      else dgebr2d_("a", &top, &lenN, &one, dbuff, &lenN, &zero, &zero);
   }
#endif

   for (nlens=0, len=len1; len < lenN; nlens++)
```

```
   {
      if (inclen > 0) len += inclen;
      else len *= len;
   }

   if (top == 'p') systyp = "primitive";
   else systyp = "BLACS";

   if (iam == 0)
   {
      fprintf(fp,
"\n======================================================================\n");
      fprintf(fp,
         "Time in milliseconds for double precision '%s %s' bcast, top='%c',\n",
              PLAT, systyp, top);
      fprintf(fp, "nnodes=%d, reps=%d, src=%d, ntests=%d.\n",
              nnodes, reps, src, nlens);
      fprintf(fp,
"======================================================================\n\n");
      fprintf(fp,
"   ELEMENTS           SRC TIME      MIN TIME      MAX TIME      AVG TIME\n");
      fprintf(fp,
"   --------          ----------    ----------    ----------    ----------\n\n");
   }

   len = len1;
   while (len <= lenN)
   {
      for (j=0; j < len; j++) dbuff[j] = 1.0;  /* init buffer */

      if (iam == src)
      {
         if (top == 'p')  /* use system primitive */
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
#if (SYS == INTEL)
               msgid = (reps)*nlens + j;
               csend(msgid, dbuff, len*sizeof(double), -1, 0);
#elif (SYS == SP1)
               mpc_bcast(dbuff, len*sizeof(double), iam, allgrp);
#elif (SYS == CM5)
               CMMD_bc_to_nodes(dbuff, len*sizeof(double));
#endif
            }
            time = dwalltime00_() - time1;
         }
         else          /* call the BLACS */
         {
```

```
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
               dgebs2d_("a", &top, &len, &one, dbuff, &len);
            time = dwalltime00_() - time1;
         }
         srcT = time;
         if (src != 0)  /* send source time to node 0 */
            dgesd2d_(&one, &one, &srcT, &one, &zero, &zero);
      }
/*
 *    If I am receiving and participating in broadcast
 */
      else
      {
         if (top == 'p')
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
#if (SYS == INTEL)
               msgid = (reps)*nlens + j;
               crecv(msgid, dbuff, len*sizeof(double));
#elif (SYS == SP1)
               mpc_bcast(dbuff, len*sizeof(double), src, allgrp);
#elif (SYS == CM5)
               CMMD_receive_bc_from_node(dbuff, len*sizeof(double));
#endif
            }
            time = dwalltime00_() - time1;
         }
         else
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
               dgebr2d_("a", &top, &len, &one, dbuff, &len, &zero, &src);
            time = dwalltime00_() - time1;
         }
         if (iam == 0) dgerv2d_(&one, &one, &srcT, &one, &zero, &src);
      }

      time = (time * 1000.0) / reps;  /* get time in msecs */
      srcT = (srcT * 1000.0) / reps;  /* get time in msecs */
      minT = maxT = avgT = time;
      dgmin2d_("a", "1", &one, &one, &minT, &one, dbuff, &dbuff[1],
              &one, &zero, &zero);
      dgmax2d_("a", "1", &one, &one, &maxT, &one, dbuff, &dbuff[1],
              &one, &zero, &zero);
      dgsum2d_("a", "1", &one, &one, &avgT, &one, &zero, &zero);
```

```
      avgT /= nnodes;

      if (iam == 0)
         fprintf(fp, "%11d       %13.3lf %13.3lf %13.3lf %13.3lf\n",
                 len, srcT, minT, maxT, avgT);
      if (inclen > 0) len += inclen;
      else len *= len;
   }
   if (iam == 0)
   {
      fprintf(fp,
"\n=========================================================================\n");
      fprintf(fp, "End of Run.\n");
      if (fp != stdout) fclose(fp);
   }
}
```

D.4 Combine Timing Code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BLACS 1
#define INTEL 2
#define SP1 3
#define CM5 4

#define SYS CM5

#if (SYS == INTEL)

#define PLAT "Intel"
#define dwalltime00_ dclock
#define Tsync() gsync()

#elif (SYS == CM5)

#define PLAT "CM5"
#define dwalltime00_ dcputime00_
#define Tsync() CMMD_sync_with_nodes()
#include </usr/include/cm/cmmd.h>

#elif (SYS == SP1)

#define PLAT "SP1"
#define Tsync() mpc_sync(allgrp)

#endif

main(int nargs, char *args[])
/*
 *  Measures time for point to point communication.  Usage:
 *  Tg <op> <top> <len1> <lenN> <inclen> <repititions> <dest node> [<outfile>]
 *  if <recv increment> greater than <last recv node>, then increment is by
 *  power of 2.
 */
{
   double dwalltime00_();
#if (SYS == SP1)
   extern void d_vadd();
#endif

   FILE *fp=stdout;
   char *outfile, *systyp;
   char op, top;
   int len, len1, lenN, inclen, reps, dest, rdest;
   int iam, nnodes, msgid, length, i, j, itmp, allgrp;
```

```
   int one=1, zero=0;
   int *rA=NULL, *cA=NULL;
   double *dbuff=NULL, *dbuff2=NULL;
   double time1, time, maxT, minT, avgT;

/*
 * Get my node number and the number of nodes in system
 */
   initpinfo_(&iam, &nnodes);

/*
 * Read in params from command line
 */
   if (nargs >= 8)
   {
      op = *args[1];
      top = *args[2];
      len1 = atoi(args[3]);
      lenN = atoi(args[4]);
      inclen = atoi(args[5]);
      reps = atoi(args[6]);
      dest = atoi(args[7]);
      if (dest < 0) rdest = -1;
      else rdest = 0;
   }
   else
   {
       fprintf(stderr, "\n Incorrect usage.\n");
       exit(1);
   }

   auxsetup_(&iam, &nnodes);
   blacsinit_(&one, &nnodes);

   if ( (iam == 0) && (nargs > 8) ) fp = fopen(args[8], "w");

#if (SYS == SP1)
   rA = (int *) malloc(sizeof(int)*4);
   mpc_task_query(rA, 4, 3);
   allgrp = rA[3];
   free(rA);
#endif

   dbuff = (double *) malloc(lenN*sizeof(double));
   assert (dbuff != NULL);
/*
 * If using platform dependant routine, get work buffer
 */
   if (top == 'p')  /* platform dependant routine */
   {
      dbuff2 = (double *) malloc(lenN*sizeof(double));
```

```
      assert (dbuff2 != NULL);
      systyp = "primitive";
   }
/*
 * If using BLACS, get buffer by doing an operation
 */
   else
   {
      systyp = "BLACS";
      if (op == '+')
      {
         for (j=0; j < lenN; j++) dbuff[j] = 0.0;  /* init buffer */
         dgsum2d_("a", &top, &lenN, &one, dbuff, &lenN, &rdest, &dest);
      }
      else
      {
         rA = (int *) malloc(lenN*sizeof(int));
         cA = (int *) malloc(lenN*sizeof(int));
         assert (rA != NULL);
         assert (cA != NULL);
         for (j=0; j < lenN; j++) dbuff[j]=0.0;  /* init buffer */
         dgmax2d_("a", &top, &lenN, &one, dbuff, &lenN, rA, cA, &len,
                  &rdest, &dest);
      }
   }

   if (iam == 0)
   {
      for (i=0, len=len1; len <= lenN; i++)
      {
         if (inclen > 0) len += inclen;
         else len *= 2;
      }

      fprintf(fp,
"\n=======================================================================\n");
      fprintf(fp,
         "Time in milliseconds for double precision '%s %s' op='%c', top='%c',\n",
              PLAT, systyp, op, top);
      fprintf(fp, "nnodes=%d, reps=%d, dest=%d, ntests=%d.\n",
              nnodes, reps, dest, i);
      fprintf(fp,
"=======================================================================\n\n");
      fprintf(fp,
         "   ELEMENTS              MIN TIME      MAX TIME      AVG TIME\n");
      fprintf(fp,
         "   --------             ----------    ----------    ----------\n\n");
   }
   len = len1;
   while (len <= lenN)
   {
```

```
      srand(len);
      for (j=0; j < len; j++) dbuff[j] = rand();  /* init buffer */

      if (op == '+')
      {
         if (top == 'p')
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
#if (SYS == INTEL)
               gdsum(dbuff, len, dbuff2);
#elif (SYS == SP1)
               mpc_combine(dbuff, dbuff2, len*sizeof(double), d_vadd, allgrp);
#elif (SYS == CM5)
               CMMD_scan_v(dbuff2, dbuff, CMMD_combiner_dadd, CMMD_downward,
                           CMMD_none, 0, CMMD_inclusive, sizeof(double), len);
               if (rdest == -1)
               {
                  if (iam == 0) CMMD_bc_to_nodes(dbuff2, len*sizeof(double));
                  else CMMD_receive_bc_from_node(dbuff2, len*sizeof(double));
               }
#endif
            }
            time = dwalltime00_() - time1;
         }
         else
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
               dgsum2d_("a", &top, &len, &one, dbuff, &len, &rdest, &dest);
            }
            time = dwalltime00_() - time1;
         }
      }
      else if (op == '>')
      {
         if (top == 'p')
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
#if (SYS == INTEL)
               gdhigh(dbuff, len, dbuff2);
#elif (SYS == SP1)
               mpc_combine(dbuff, dbuff2, len*sizeof(double), d_vmax, allgrp);
#elif (SYS == CM5)
```

```
               CMMD_scan_v(dbuff2, dbuff, CMMD_combiner_dmax, CMMD_downward,
                           CMMD_none, 0, CMMD_inclusive, sizeof(double), len);
               if (rdest == -1)
               {
                  if (iam == 0) CMMD_bc_to_nodes(dbuff2, len*sizeof(double));
                  else CMMD_receive_bc_from_node(dbuff2, len*sizeof(double));
               }
#endif
            }
            time = dwalltime00_() - time1;
         }
         else
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
               dgmax2d_("a", &top, &len, &one, dbuff, &len,
                        rA, cA, &len, &rdest, &dest);
            }
            time = dwalltime00_() - time1;
         }
      }
      else if (op == '<')
      {
         if (top == 'p')
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
#if (SYS == INTEL)
               gdlow(dbuff, len, dbuff2);
#elif (SYS == SP1)
               mpc_combine(dbuff, dbuff2, len, d_vmin, allgrp);
#elif (SYS == CM5)
               CMMD_scan_v(dbuff2, dbuff, CMMD_combiner_min, CMMD_upward,
                           CMMD_none, 0, CMMD_inclusive, sizeof(double), len);
#endif
            }
            time = dwalltime00_() - time1;
         }
         else
         {
            Tsync();
            time1 = dwalltime00_();
            for (j=0; j < reps; j++)
            {
               dgmin2d_("a", &top, &len, &one, dbuff, &len,
                        rA, cA, &len, &rdest, &dest);
            }
```

```
            time = dwalltime00_() - time1;
         }
      }

      time = (time * 1000.0) / reps;  /* get time in msecs */
      minT = maxT = avgT = time;
      dgmin2d_("a", "1", &one, &one, &minT, &one, dbuff, &dbuff[1],
              &one, &zero, &zero);
      dgmax2d_("a", "1", &one, &one, &maxT, &one, dbuff, &dbuff[1],
              &one, &zero, &zero);
      dgsum2d_("a", "1", &one, &one, &avgT, &one, &zero, &zero);
      avgT /= nnodes;

      if (iam == 0)
         fprintf(fp, "%11d          %13.3lf %13.3lf %13.3lf\n",
                 len, minT, maxT, avgT);
      if (inclen < lenN) len += inclen;
      else len *= 2;
   }
   if (iam == 0)
   {
      fprintf(fp,
"\n======================================================================\n");
      fprintf(fp, "End of Run.\n");
      if (fp != stdout) fclose(fp);
   }
}
```