LAPACK Working Note 94
A User's Guide to the BLACS v1.1 *

Jack J. Dongarra, † R. Clint Whaley ‡

May 5, 1997

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 report describes the library which has arisen from this project.

*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, and Mathematical Sciences Section, ORNL, Oak Ridge, TN 37831, dongarra@cs.utk.edu

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

Contents

1 Introduction **1**

2 Availability of the BLACS Software **2**

3 What's new for this release **3**

4 BLACS Concepts and Features **3**
- 4.1 Array-based Communication . . . 3
- 4.2 Process Grid and Scoped Operations . . . 4
- 4.3 Contexts . . . 5
- 4.4 ID-less Communication . . . 6
- 4.5 Blocking Levels . . . 7

5 Naming Conventions **8**

6 Point To Point Communication **8**
- 6.1 Semantics . . . 8
- 6.2 Syntax . . . 11
  - 6.2.1 Point to Point Sends . . . 11
  - 6.2.2 Point to Point Receives . . . 12
- 6.3 Example . . . 13

7 Broadcasts **13**
- 7.1 Semantics . . . 13
- 7.2 Syntax . . . 13
  - 7.2.1 Broadcast/send . . . 14
  - 7.2.2 Broadcast/receive . . . 15
- 7.3 Example . . . 16
- 7.4 Topologies . . . 16

8 Combines **17**
- 8.1 Semantics . . . 17
- 8.2 Syntax . . . 18
- 8.3 Example . . . 19
- 8.4 Topologies . . . 20

9 Support Routines **20**
- 9.1 Initialization . . . 20
  - 9.1.1 BLACS_PINFO . . . 20
  - 9.1.2 BLACS_SETUP . . . 21
  - 9.1.3 BLACS_GRIDINIT . . . 21
  - 9.1.4 BLACS_GRIDMAP . . . 22
- 9.2 Destruction . . . 24
  - 9.2.1 BLACS_FREEBUFF . . . 24
  - 9.2.2 BLACS_GRIDEXIT . . . 24

9.2.3 BLACS_ABORT . . . 25
9.2.4 BLACS_EXIT . . . 25
9.3 Informational and Miscellaneous . . . 25
9.3.1 BLACS_GRIDINFO . . . 25
9.3.2 BLACS_PNUM . . . 26
9.3.3 BLACS_PCOORD . . . 26
9.3.4 BLACS_BARRIER . . . 27
9.4 General purpose . . . 27
9.4.1 BLACS_GET . . . 27
9.4.2 BLACS_SET . . . 28
9.5 Unofficial routines . . . 30
9.5.1 SETPVMTIDS . . . 31
9.5.2 DCPUTIME . . . 31
9.5.3 DWALLTIME . . . 31
9.5.4 KSENDID . . . 32
9.5.5 KRECVID . . . 32
9.5.6 KBSID . . . 32
9.5.7 KBRID . . . 32

REFERENCES **34**

A C Interface to the BLACS **36**
A.1 Support Routines . . . 36
A.1.1 Initialization . . . 36
A.1.2 Destruction . . . 36
A.1.3 Informational and Miscellaneous . . . 36
A.1.4 Unofficial . . . 37
A.2 Point to Point . . . 37
A.3 Broadcasts . . . 37
A.4 Combines . . . 37

B Degrees of Blocking **38**
B.1 Non-blocking communication . . . 38
B.2 Locally-blocking . . . 39
B.3 Globally-blocking . . . 40

C BLACS Error Handling **41**
C.1 BLACS Warning and Error Messages . . . 42
C.1.1 Examples . . . 42
C.2 System Error Messages . . . 43
C.2.1 Examples . . . 43

D Repeatability and coherence **44**
D.1 Repeatability . . . 44
D.2 Coherence . . . 45
D.2.1 Example of Incoherence . . . 45

D.2.2 Example of Homogeneous Coherence 46
D.2.3 Example of Heterogeneous Coherence 46
D.3 Summing it up 46

E Broadcast Topologies **48**
E.1 Broadcast Ring Topologies 49
E.2 Broadcast Tree Topologies 52

F Combine Topologies **56**
F.1 General Tree Gather 56
F.2 Bidirectional Exchange 56

G Multiring Combine **58**

H Example Program **60**

List of Tables

1 Presently supported message passing layers 2
2 Scopes provided by a 2D process grid 5
3 Values and meanings of the communication routines' name positions 9
4 Values and meanings of combine routines' name positions 9
5 Prefix to type declaration mapping 11
6 Prefix to C type declaration mapping 36
7 Broadcast topology highlights 49

List of Figures

1 8 processes mapped to a 2 x 4 process grid. 4
2 After first step of LU factorization 5
3 Increasing ring broadcast 50
4 Decreasing ring broadcast 50
5 Split ring broadcast 50
6 Multiring broadcast with $N_r = 3$ 51
7 Hypercube broadcast, nearest node first. 53
8 General tree broadcast with $N_b = 1$ 54
9 General tree broadcast with $N_b = 2$ 54
10 General tree broadcast with $N_b = 3$ 55
11 General tree gather with $N_b = 1$ 57
12 General tree gather with $N_b = 4$ 57
13 Bidirectional exchange 58

1 Introduction

The BLACS (Basic Linear Algebra Communication Subprograms) [8, 10, 16] is a package that attempts to provide the same ease of use and portability for distributed memory linear algebra communication that the BLAS [5, 6, 15] provide for linear algebra computation.

The concept of concentrating the most used computation into a kernel of highly optimized routines, such as the BLAS, has proven itself in work on LAPACK [2, 1]. LAPACK (Linear Algebra PACKage) provides linear algebra routines for sequential and shared memory machines.

When the ScaLAPACK [9, 4] project (which involves porting LAPACK to distributed memory parallel machines) was begun, it rapidly became evident that a similar kernel for communication would be required. Out of this need the BLACS arose.

With these two kernels in place, software for dense linear algebra on MIMD platforms can consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for that particular platform, good performance should be achieved with relatively little effort. Also, since both packages will be available on a wide variety of machines, code modifications required to change platforms should be minimal.

There are various packages designed to provide a message passing interface that remains unchanged across multiple platforms, including PICL [13], PVM [12] and more recently, MPI [11]. These packages are general libraries, however, and thus their interfaces are not as easily usable for linear algebra applications as we would like.

In contrast, since the audience of the BLACS is known, the interface and methods of using the routines can be specialized (and thus simplified). For example, the BLACS are written at a level where the manipulation of the matrices involved in linear algebra computations is both natural and convenient.

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 a level as to be easily usable by linear algebra programmers.

- *Portability* The BLACS must supply an interface which can be supported across a wide range of parallel computers, including parallel machines built from heterogeneous processors.

The first section of this report discusses downloading and availability issues. The following section familiarizes the reader with some of the more important concepts and features of the BLACS. We then discuss the four main categories of BLACS routines. 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. Then, 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 summation or absolute value maximization or minimization). Finally, we discuss the support routines, which perform many diverse

functions, often not directly related to communication (for example, returning a process ID).

The appendices discuss the C interface to the BLACS, error handling, the issues of repeatability and coherence, broadcast and combine topologies, as well as providing further description of the blocking levels outlined in Section 4.5.

This user's guide is supplemented by the BLACS web page. The URL is `http://www.netlib.org/blacs/Blacs.html`. This on-line document gives detailed examples, as well as providing reference, downloading options, installation instructions, and troubleshooting. If problems still remain after reading this guide and consulting the mosaic page, questions should be mailed to `blacs@cs.utk.edu`.

2 Availability of the BLACS Software

The BLACS source code and documentation is available through *netlib*. Netlib is a software distribution service set up on the Internet that contains a wide range of computer software. Software can be retrieved from netlib by http, ftp, or email.

At present, four different BLACS implementations are available from netlib. Each of these four BLACS implementations is based on a different message passing layer. These message passing layers and the machines they are normally supported on are shown in Table 1.

In addition to the BLACS versions available on netlib [7], several vendors (e.g. Cray, IBM and Meiko) are presently producing optimized versions for their machines.

MESSAGE PASSING LAYER	MACHINES
MPI	Most systems.
MPL	IBM's SP series (SP1 and SP2)
NX	Intel's supercomputer series (iPSC2, iPSC/860, DELTA and PARAGON).
PVM	Most systems.

Table 1: Presently supported message passing layers

NOTE: In the past we have also supported a CMMD version of the BLACS, designed for Thinking Machine's CM-5. It appears that there are no longer users for this package. If you need the CMMDBLACS, send mail to `blacs@cs.utk.edu`, and we will examine making this version available again.

The BLACS homepage can be accessed at URL `http://www.netlib.org/blacs/Blacs.html`. The BLACS homepage contains reference to the routines, examples, installation instructions, troubleshooting, and downloading options.

The BLACS files may be obtained via anonymous ftp to `netlib.org`. Look in the directory `blacs`. The file `index` describes the files available in this directory.

Finally, the BLACS may be obtained by email. To receive downloading instructions and a list of available files, send email to `netlib@netlib.org` with the message `send index from blacs`.

3 What's new for this release

This release is really just a minor update from the 1.0 release. The most important change is that there are now flags which control whether the combine operations are *coherent* and *repeatable*. See section 9.4 and appendix D for details.

Also worth noting is that the setting of the number of branches (rings) for tree (multiring) broadcast has been seperated from the number of branches (rings) for tree (multiring) combine. See section 9.4 for further details.

4 BLACS Concepts and Features

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.

In the BLACS, 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 effect 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.

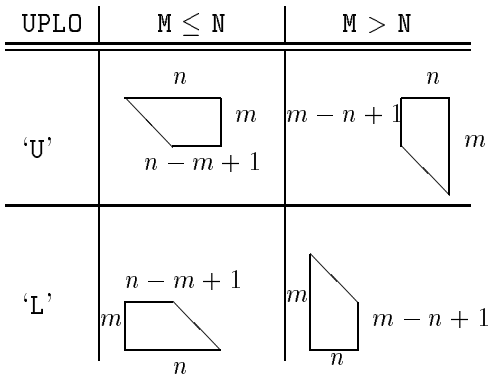
4.1 Array-based Communication

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 two dimensional 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 consists 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 two successive elements of a matrix row in memory (the BLACS assume column-major storage of arrays).

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 subclass of trapezoidal, so these matrices are also handled by the BLACS.

The shape of the trapezoid to be sent is determined by `M`, `N`, and `UPLO`:



The packing of arrays (if required) so that they may be sent efficiently is handled internally by the BLACS, allowing the user to concentrate on the logical matrix, rather than how the data is organized in the machine's memory.

4.2 Process Grid and Scoped Operations

The N_p processes involved in a parallel task or group 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 $\mathcal{P} * \mathcal{Q} = N_g \leq N_p$. A process can now be referenced by its coordinates within the grid (indicated by the notation $\{p, q\}$, where $0 \leq p < \mathcal{P}$, and $0 \leq q < \mathcal{Q}$), 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 2.

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

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 2: Scopes provided by a 2D process grid

these rows and columns actually are will depend upon the underlying hardware. 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, and therefore a 1D process grid usually yields the best performance. Fortunately, most modern parallel interconnection networks are at least as rich as a 2D grid, so that the additional levels of parallelism inherent in a 2D process grid can be successfully 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 matrix's elements. 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 $\tilde{A}$.

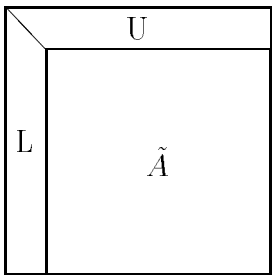


Figure 2: After first step of LU factorization

This very sketchy description of LU is analyzed much more completely in [9], which includes an examination of scalability and the advantages of using 2D process grids.

A more detailed understanding of the logical process grid will be obtained as we discuss the various BLACS routines later in the paper.

4.3 Contexts

In the BLACS, each logical process grid (hereafter referred to simply as the grid) is enclosed in a context. A context may be thought of as a message passing universe. This means that a grid can safely communicate even if other (possibly overlapping) grids are also communicating.

In most respects, we can use the terms 'grid' and 'context' interchangeably. For example,

we may say "perform operation in context X" or "in grid X". The slight difference here is that the user may define two exactly identical grids (say, two 1x3 process grids, both of which use processes 0, 1, and 2), but each will be wrapped in its own context, so that they are distinct in operation, even though they are indistinguishable from a process grid standpoint.

Contexts are used so that individual routines using the BLACS can, when required, safely operate without worrying if the user is running other distributed codes on the same machine.

Another example of the use of context might be to define a normal 2D process grid within which most computation takes place. However, in certain portions of the code it may be more convenient to access the processes as a 1D grid, and at certain other times we may wish, for instance, to share information among nearest neighbors. We will therefore want each process to have access to three contexts: the 2D grid, the 1D grid, and a small grid which contains the process and its nearest neighbors.

Therefore, we see that context allows us to:

- Create arbitrary groups of processes,
- Create an indeterminate number of overlapping and/or disjoint grids,
- Isolate each process grid so that grids do not interfere with each other.

In the BLACS, there are two grid creation routines (`BLACS_GRIDINIT` and `BLACS_GRIDMAP`) which create a process grid and its enclosing context. These routines return context handles, which are simple integers, assigned by the BLACS to identify the context. Subsequent BLACS routines will be passed these handles, which allow the BLACS to determine from which context/grid a routine is being called. The user should never actually manipulate these handles; they are opaque data objects which are only meaningful for the BLACS routines.

A defined context consumes resources. It is therefore advisable to release contexts when they are no longer needed. This is done via the routine `BLACS_GRIDEXIT`. When the entire BLACS system is shut down (via a call to `BLACS_EXIT`), all outstanding contexts are automatically freed.

4.4 ID-less Communication

One of the features 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 or tag) 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 internally 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 the BLACS to be used in conjunction with other communication platforms. An example that occurs regularly is linking a BLACS package (for example, ScaLAPACK) with a machine specific package.

These goals were achieved by placing two restrictions on communication, and allowing the user to optionally specify the BLACS msgid range. The first restriction on communication is that a receiving process must know the process grid coordinates of the sending process. 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.

Finally, in order for the BLACS to coexist with other communication packages, the BLACS allow the user the option of specifying what range of msgids the BLACS can use. In this case, it is the user's responsibility to ensure that the BLACS msgid range is not used in the code which utilizes the other communication package. If the user wishes to set the BLACS msgid range, he may do so by a call to the support routine `BLACS_SET`. Note that if the BLACS in use are written on top of a message passing system which natively supports the context concept (for instance, MPI), passing a unique system context for the BLACS to use will ensure BLACS communication will not interfere with other communication packages. In this case, setting msgid range will not be required.

4.5 Blocking Levels

An understanding of the level of *blocking* is required in order to safely use any communication package. A communication operation has various resources tied to it, the main such resource being the user's buffer. Since the buffer is the main resource BLACS users will be concerned with, in the following discussion we will refer only to the user's buffer, instead of using the more general term "resources".

The blocking level of a routine tells the user what correspondence, if any, there is between the return from a routine, and the availability of the buffer. For example, if the user posts a receive, he needs to know when the data he is receiving has actually been stored in the buffer.

In this paper, we define three levels of blocking: non-blocking, locally-blocking, and globally-blocking. These levels are briefly previewed below. Appendix B provides a short section explaining each of these levels in greater detail.

- *Non-blocking* communication: the return from the communication routine implies only that the message request has been posted. It is then the user's responsibility to probe and thus determine when the operation has completed.

- *Locally-blocking* communication: May be applied only to send operations, not receives. The return from the send implies that the buffer is available for re-use. It is further specified that the send will complete regardless of whether the corresponding receive is posted.

- *Globally-blocking* communication: The return from the operation implies that the buffer is available for re-use. The operation may not complete unless the complement of the operation is called (e.g., a send may not complete if the corresponding receive is not posted).

The BLACS provide globally-blocking point to point receive, broadcasts, and combines. The BLACS point to point send is locally-blocking. Appendix B provides the reasoning behind these choices for the BLACS blocking levels.

5 Naming Conventions

This section gives the naming conventions for each of the four BLACS routine classifications (point to point communication, broadcast, combine and support). Point to point, broadcast and combine are all typed routines, i.e., there is a separate routine for each data type.

Point to Point and Broadcast Routines 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 3.

Combines The general form of the names for combines is `vGZZZ2D`, where `v` is the same as shown in Table 3. The position `ZZZ` indicates what type of operation should be performed when sending the data. The operations presently supported are shown on Table 4.

Support Routines The support routines serve many diverse functions, and thus they do not have a great degree of standardization. All official BLACS support routines (i.e., those that are guaranteed by the standard to exist) have the form `BLACS_<name>`.

6 Point To Point Communication

6.1 Semantics

Point to point communication requires two complementary operations. The *send* operation produces a message, which is then consumed by the *receive* operation. The BLACS send is defined to be locally-blocking, and the receive is globally-blocking (see appendix B for details on blocking).

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

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 3: Values and meanings of the communication routines' name positions

vGZZZ2D

ZZZ	MEANING
AMX	Entries of result matrix will have the value of the greatest absolute value found in that position.
AMN	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 4: Values and meanings of combine routines' name positions

received only after both A and B. The main reason for this restriction is that it allows for the computation of message identifiers, as is discussed in Section 4.4.

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

6.2 Syntax

As mentioned in Section 5, these routines are type dependent, indicated here by the prefix `v`. The matrix type operated on by these routines will therefore vary with `v`, as shown in Table 5.

v	Data operated on is	TYPE declaration
I	integer	INTEGER
S	single precision real	REAL
D	double precision real	DOUBLE PRECISION
C	single precision complex	COMPLEX
Z	double precision complex	DOUBLE COMPLEX

Table 5: Prefix to type declaration mapping

With this in mind, the calling sequences and parameter declarations for these routines are given in the following sections. Note that output parameters are underlined. All other parameters will be input, and thus unchanged on exit from the routine.

6.2.1 Point to Point Sends

vGESD2D(ICONTXT, M, N, A, LDA, RDEST, CDEST)
vTRSD2D(ICONTXT, UPLO, DIAG, M, N, A, LDA, RDEST, CDEST)

Parameters:

ICONTXT (input) INTEGER
The BLACS context handle.

UPLO (input) CHARACTER*1
Indicates whether the matrix is upper (`UPLO = 'U'`) or lower (`UPLO = 'L'`) trapezoidal.

DIAG (input) CHARACTER*1
Indicates whether the diagonal of the matrix is unit diagonal (`DIAG = 'U'`), and thus need not be communicated, or otherwise (`DIAG = 'N'`).

M (input) INTEGER
The number of matrix rows to be sent.

N (input) INTEGER
The number of matrix columns to be sent.

A (input) TYPE array of dimension (LDA, N)
A pointer to the beginning of the (sub)array to be sent.

LDA (input) INTEGER
The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row.

RDEST (input) INTEGER
Process row coordinate of the destination process.

CDEST (input) INTEGER
Process column coordinate of the destination process.

6.2.2 Point to Point Receives

vGERV2D(ICONTXT, M, N, A, LDA, RSRC, CSRC)
vTRRV2D(ICONTXT, UPLO, DIAG, M, N, A, LDA, RSRC, CSRC)

Parameters:

ICONTXT (input) INTEGER
The BLACS context handle.

UPLO (input) CHARACTER*1
Indicates whether the matrix is upper (`UPLO = 'U'`) or lower (`UPLO = 'L'`) trapezoidal.

DIAG (input) CHARACTER*1
Indicates whether the diagonal of the matrix is unit diagonal (`DIAG = 'U'`), and thus need not be communicated, or otherwise (`DIAG = 'N'`).

M (input) INTEGER
The number of matrix rows to be received.

N (input) INTEGER
The number of matrix columns to be received.

A (output) TYPE array (LDA, N)
A pointer to the beginning of the (sub)array to be received.

LDA (input) INTEGER
The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row.

RSRC (input) INTEGER
Process row coordinate of the source of the message.

CSRC (input) INTEGER
Process column coordinate of the source of the message.

6.3 Example

For a simple example of using BLACS point to point message passing, we show code which has two processes swap their copies of a 5 element double precision vector `X`.

```
CALL BLACS_GRIDINFO(ICONTXT, NPROW, NPCOL, MYPROW, MYPCOL)

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

7 Broadcasts

7.1 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 (see appendix B for details). This 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.

7.2 Syntax

As with point to point communication, these routines vary with the data type, and Table 5 shows the mapping between the type prefix (indicated below by `v`) and the data type declaration. As before, output parameters are underlined. All other parameters will be input, and thus unchanged on exit from the routine. With these points in mind, the calling sequences and parameter declarations for these routines are given in the following sections.

7.2.1 Broadcast/send

vGEBS2D(ICONTXT, SCOPE, TOP, M, N, A, LDA)
vTRBS2D(ICONTXT, SCOPE, TOP, UPLO, DIAG, M, N, A, LDA)

Parameters:

ICONTXT (input) INTEGER
The BLACS context handle.

SCOPE (input) CHARACTER*1
Scope of processes to participate in operation. Limited to `'ROW'`, `'COLUMN'`, or `'ALL'`. See Section 4.2 for additional details.

TOP (input) CHARACTER*1
Network topology to be emulated during communication. Topologies presently supported are discussed in Section 7.4.

UPLO (input) CHARACTER*1
Indicates whether the matrix is upper (`UPLO = 'U'`) or lower (`UPLO = 'L'`) trapezoidal.

DIAG (input) CHARACTER*1
Indicates whether the diagonal of the matrix is unit diagonal (`DIAG = 'U'`), and thus need not be communicated, or otherwise (`DIAG = 'N'`).

M (input) INTEGER
The number of matrix rows to be broadcast.

N (input) INTEGER
The number of matrix columns to be broadcast.

A (input) TYPE array (LDA, N)
A pointer to the beginning of the (sub)array to be broadcast.

LDA (input) INTEGER
The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row.

7.2.2 Broadcast/receive

vGEBR2D(ICONTXT, SCOPE, TOP, M, N, A, LDA, RSRC, CSRC)
vTRBR2D(ICONTXT, SCOPE, TOP, UPLO, DIAG, M, N, A, LDA, RSRC, CSRC)

Parameters:

ICONTXT (input) INTEGER
The BLACS context handle.

SCOPE (input) CHARACTER*1
Scope of processes to participate in operation. Limited to `'ROW'`, `'COLUMN'`, or `'ALL'`. See Section 4.2 for additional details.

TOP (input) CHARACTER*1
Network topology to be emulated during communication. Topologies presently supported are discussed in Section 7.4.

UPLO (input) CHARACTER*1
Indicates whether the matrix is upper (`UPLO = 'U'`) or lower (`UPLO = 'L'`) trapezoidal.

DIAG (input) CHARACTER*1
Indicates whether the diagonal of the matrix is unit diagonal (`DIAG = 'U'`), and thus need not be communicated, or otherwise (`DIAG = 'N'`).

M (input) INTEGER
The number of matrix rows to be broadcast.

N (input) INTEGER
The number of matrix columns to be broadcast.

A (output) TYPE array (LDA, N)
A pointer to the beginning of the (sub)array to be received/broadcast.

LDA (input) INTEGER
The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row.

RSRC (input) INTEGER
Process row coordinate of the source of the broadcast.

CSRC (input) INTEGER
Process column coordinate of the source of the broadcast.

7.3 Example

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). It is not necessary for all participating processes to receive the matrix in the same way. For instance, process {1,2} might want to receive the information into a work vector, `WORK`, while the other processes in the process column receive the the broadcast into their copy of `B`. This could be accomplished as follows:

```
      CALL BLACS_GRIDINIT(ICONTXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*     If I participate in the broadcast
*
      IF (MYPCOL .EQ. 2) THEN
*
*        If I'm the source of the broadcast
*
         IF (MYPROW .EQ. 0) THEN
            CALL DGEBS2D(ICONTXT, 'COLUMN', ' ', 5, 7, B(9,4), 500)
*
*        If I want to receive into work
*
         ELSE IF (MYPROW .EQ. 1) THEN
            CALL DGEBR2D(ICONTXT, 'COLUMN, ' ', 5, 7, WORK, 5, 0, 2)
*
*        If I want to receive into B
*
         ELSE
            CALL DGEBR2D(ICONTXT, 'COLUMN', ' ', 5, 7, B(9,4), 500, 0, 2)
         END IF
```

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`.

7.4 Topologies

The topology parameter determines how the messages involved in a distributed operation are sent. The use of the topology concept 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. Topology also allows for the building of communication pipelines, as discussed below.

There are two main classes of topologies within the BLACS:

- Pipelining topologies (ring-based)
- Non-pipelining topologies (tree-based)

In a pipelining topology, the first operation synchronizes the processors so that subsequent operations will be cheap. Therefore, if the user is aware that several broadcasts will be performed with no interleaved synchronization, pipelining topologies should be considered. Further, the BLACS pipelining topologies are all based on rings, which means that pipelines can be maintained if the algorithm flows across processors in an orderly way. For example, 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 program flow is in the opposite direction, it may be possible to set up a decreasing ring pipeline. A pipeline for increasing direction can be obtained by setting `TOP = 'INCREASING RING'`; a pipeline for codes flowing across the processors in the opposite way can be obtained by setting `TOP = 'DECREASING RING'`.

The BLACS' pipelining topologies are usually much slower than non-pipelining topologies if only one operation is performed. Pipelining topologies are used to minimize the cost of several related operations. Therefore, if the broadcast does not pipeline, the user will probably wish to utilize the topology which minimizes the time spent in only one broadcast. The BLACS provide a default topology which attempts to do this, which is invoked by setting `TOP = ' '`.

One of the three topologies above will probably satisfy most users. However, there are many other BLACS topologies within the two main classes. In the pipelining category, the user may use a split ring (TOP = 'SPLIT RING') or a multiring (TOP = 'Multiring'), for instance. There are also several types of tree-based topologies. Appendix E provides full details on the available topologies.

8 Combines

8.1 Semantics

In a combine operation, each participating process contributes data which is combined with 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, element-wise absolute value maximization, and element-wise absolute value minimization of general rectangular arrays. Note that a combine operation combines data between processes. By definition, then, a combine performed across a scope of only one process does not change the input data. This is why we specify that the operations 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. If the element-wise operation concept is still unclear, the examples section should provide further clarification.

The maximization and minimization operations may require further explanation. When the max/min comparison is being performed, absolute value is used. Therefore, -5 and 5 are equivalent. However, the returned value is unchanged; i.e. it is not the absolute value, but instead is the signed value. Therefore, if we performed a BLACS absolute value maximum combine on the numbers $-5, 3, 1, -8$, the result would be -8.

The BLACS combines are globally-blocking (see appendix B for details).

8.2 Syntax

As with point to point communication, these routines vary with the data type, and Table 5 shows the mapping between the type prefix (indicated below by `v`) and the data type declaration. As before, output parameters are underlined. All other parameters will be input, and thus unchanged on exit from the routine. With these points in mind, the calling sequences and parameter declarations for these routines are given in the following sections.

vGSUM2D(ICONTXT, SCOPE, TOP, M, N, A, LDA, RDEST, CDEST)
vGAMX2D(ICONTXT, SCOPE, TOP, M, N, A, LDA, RA, CA,
RCFLAG, RDEST, CDEST)
vGAMN2D(ICONTXT, SCOPE, TOP, M, N, A, LDA, RA, CA,
RCFLAG, RDEST, CDEST)

Parameters:

ICONTXT (input) INTEGER
The BLACS context handle.

SCOPE (input) CHARACTER*1
Scope of processes to participate in operation. Limited to `'ROW'`, `'COLUMN'`, or `'ALL'`. See Section 4.2 for additional details.

TOP (input) CHARACTER*1
Network topology to be emulated during communication. Topologies presently supported are discussed in Section 8.4.

M (input) INTEGER
The number of matrix rows to be combined.

N (input) INTEGER
The number of matrix columns to be combined.

A (input/output) TYPE array (LDA, N)
A pointer to the beginning of the (sub)array to be combined.

LDA (input) INTEGER
The leading dimension of the matrix A, i.e., the distance between two successive elements in a matrix row.

RA (output) INTEGER array (RCFLAG, N)
If RCFLAG = -1, this array will not be referenced, and need not exist. Otherwise it is an integer array (of size at least RCFLAG x N) indicating the row index of the process that provided the maximum/minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

CA (output) INTEGER array (RCFLAG, N)
If RCFLAG = -1, this array will not be referenced, and need not exist. Otherwise it is an integer array (of size at least RCFLAG x N) indicating the column index of the process that provided the maximum/minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

RCFLAG (input) INTEGER
If RCFLAG = -1, then the arrays RA and CA are not referenced and need not exist. Otherwise, RCFLAG indicates the leading dimension of these arrays, and so must be $\geq$ M.

RDEST (input) INTEGER
The process row coordinate of the process who should receive the result. If RDEST or CDEST = -1, all processes within the indicated scope receive the answer.

CDEST (input) INTEGER
The process column coordinate of the process who should receive the result. If RDEST or CDEST = -1, all processes within the indicated scope receive the answer.

8.3 Example

An example should demonstrate how these routines are used. Assume we have a 2 x 4 process grid (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:

```
CALL DGMAX2D(ICONTXT, 'ALL', ' ', 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.

As another example, assume that process row 1 requires the minimum of the double precision scalar `DMIN`, and there is no need to know what process possessed the min. The code would then be:

```
IF (MYPROW .EQ. 1) THEN
```

```
      CALL DGAMN2D(ICONTXT, 'ROW', ' ', 1, 1, DMIN, 1, I, I, -1, -1, -1)
   END IF
```

8.4 Topologies

In broadcasts, the BLACS provide both pipelining and non-pipelining topologies. At the moment, the BLACS provide pipelining combine topologies only in the MPI version. Therefore, the user is encouraged to use the default topology (`TOP = ' '`) when calling a combine operation. The default `TOP` option will attempt to use the topology which will minimize the cost of one call to a combine operation.

Appendix F provides detailed descriptions of presently supported topologies.

9 Support Routines

There are a number of routines which do not deal directly with communication that are nonetheless required for programming in a parallel environment. The BLACS label these routines as *support* routines. We break these support routines into rough categories, and these are discussed in turn below.

9.1 Initialization

These routines deal with grid/context creation, and processing before the grid/context has been defined.

9.1.1 BLACS_PINFO

BLACS_PINFO(MYPNUM, NPROCS)

MYPNUM (output) INTEGER
An integer between 0 and (NPROCS - 1) which uniquely identifies each process.

NPROCS (output) INTEGER
The number of processes available for BLACS use.

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, NPROCS is the actual number of processes available for use (i.e. NPROWS * NPCOLS $\leq$ NPROCS). In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, NPROCS will be returned as less than one. If a process has been spawned via the keyboard, it will receive MYPNUM of 0, and all other processes will get MYPNUM of -1. This allows the user to distinguish between processes, so that only one reads in data, etc. Only after the virtual machine has been set up (via a call to BLACS_SETUP or SETPVMTIDS) will this routine return the correct values for MYPNUM and NPROCS.

9.1.2 BLACS_SETUP

BLACS_SETUP(MYPNUM, NPROCS)

MYPNUM (output) INTEGER
An integer between 0 and (NPROCS - 1) which uniquely identifies each process.

NPROCS (input/output) INTEGER
On the process spawned from the keyboard (rather than from pvmspawn), this parameter is input, and indicates the number of processes to create when building the virtual machine. For all other processes, it will be output.

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to BLACS_PINFO. The BLACS assume a static system: you start with a given number of processes, and that is all you will ever have. PVM supplies a dynamic system, allowing processes to be added to the system on the fly. BLACS_SETUP is used to actually allocate the virtual machine and spawn processes. It reads in a file called `blacs_setup.dat`, whose first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (PvmTaskDefault), 4 (PvmTaskDebug), 8 (PvmTaskTrace), and 12 (PvmTaskDebug + PvmTaskTrace). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning NPROCS-1 processes to the machines in a round robin fashion. NPROCS is input on the process which has no PVM parent (i.e. MYPNUM=0), and both parameters are output for all processes. Therefore, on PVM systems, the call to BLACS_PINFO informs you that the virtual machine has not been set up, and a call to BLACS_SETUP then sets up the machine and returns the real values for MYPNUM and NPROCS. Note that if the file `blacs_setup.dat` does not exist, the BLACS will prompt the user for the executable name, and processes will be spawned to the current PVM configuration.

9.1.3 BLACS_GRIDINIT

BLACS_GRIDINIT(ICONTXT, ORDER, NPROW, NPCOL)

ICONTXT (input/output) INTEGER
On input, an integer handle indicating the system context to be used in creating the BLACS context. The user may obtain a default system context via a call to BLACS_GET. On output, the integer handle to the created BLACS context.

ORDER (input) CHARACTER*1
Indicates how to map processes to BLACS grid. Choices are:

'R' : Use row-major natural ordering.

'C' : Use column-major natural ordering.

ELSE : Use row-major natural ordering.

NPROW (input) INTEGER
Indicates how many process rows the process grid should contain.

NPCOL (input) INTEGER
Indicates how many process columns the process grid should contain.

All BLACS codes must call this routine, or its companion routine BLACS_GRIDMAP. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system will map into the native machine's process numbering system. Each BLACS grid is contained in a context (its own message passing universe), so that it does not interfere with distributed operations which occur within other grids/contexts. These grid creation routines may be called repeatedly in order to define additional contexts/grids.

The creation of a grid requires input from all processes which are defined to be in it. It is therefore a globally-blocking (sometimes called syncronous) operation (see appendix B for details on blocking) which means that processes belonging to more than one grid will have to agree on which grid formation will be serviced first.

These grid creation routines set up various internals for the BLACS, and so one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already-existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes will be actual processors (hardware), and they are "created" when the user runs his executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine BLACS_SETUP should be used to create the virtual machine. If the PVM user wishes to use a virtual machine already set up using explicit PVM calls, the routine SETPVMTIDS should be used instead of BLACS_SETUP.

This routine creates a simple NPROW x NPCOL process grid. This process grid will use the first NPROW * NPCOL processes, and assign them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, BLACS_GRIDINIT's more complex companion routine BLACS_GRIDMAP must be called instead.

9.1.4 BLACS_GRIDMAP

BLACS_GRIDMAP(ICONTXT, USERMAP, LDUMAP, NPROW, NPCOL)

ICONTXT (input/output) INTEGER
On input, an integer handle indicating the system context to be used in creating the BLACS context. The user may obtain a default system context via a call to BLACS_GET. On output, the integer handle to the created BLACS context.

USERMAP (input) INTEGER array, dimension (LDUMAP, NPCOL)
Input array indicating the process-to-grid mapping.

LDUMAP (input) INTEGER
The leading dimension of the 2D array USERMAP.

NPROW (input) INTEGER
Indicates how many process rows the process grid should contain.

NPCOL (input) INTEGER
Indicates how many process columns the process grid should contain.

All BLACS codes must call this routine, or its companion routine BLACS_GRIDMAP. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system will map into the native machine's process numbering system. Each BLACS grid is contained in a context (its own message passing universe), so that it does not interfere with distributed operations which occur within other grids/contexts. These grid creation routines may be called repeatedly in order to define additional contexts/grids.

The creation of a grid requires input from all processes which are defined to be in it. It is therefore a globally-blocking operation (see appendix B for details on blocking) which means that processes belonging to more than one grid will have to agree on which grid formation will be serviced first.

These grid creation routines set up various internals for the BLACS, and so one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already-existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes will be actual processors (hardware), and they are "created" when the user runs his executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine BLACS_SETUP should be used to create the virtual machine. If the PVM user wishes to use a virtual machine already set up using explicit PVM calls, the routine SETPVMTIDS should be used instead of BLACS_SETUP.

This routine allows the user to map processes to the process grid in an arbitrary manner. USERMAP(i,j) holds the process number of the process to be placed in {i, j} of the process grid. On most distributed systems, this process number will simply by a machine defined number between 0 ... NPROCS-1. For PVM these node numbers will be the PVM TIDS (Task IDs). BLACS_GRIDMAP is not for the inexperienced user – BLACS_GRIDINIT is much simpler. BLACS_GRIDINIT simply performs a GRIDMAP where the first NPROW * NPCOL processes are mapped into the current grid in a row- or column-major natural ordering. BLACS_GRIDMAP allows the experienced user to take advantage of the processors' actual network (i.e. he can map nodes that are physically connected to be neighbors in the BLACS grid, etc.). BLACS_GRIDMAP also opens the way for *multigridding*: the user can separate his nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. BLACS_GRIDMAP also provides the ability to make arbitrary grids or subgrids (e.g., a "nearest neighbor" grid), which can greatly facilitate

operations among groups of processes which do not fall on a row or column of the main process grid.

9.2 Destruction

These routines destroy grids, free resources, etc.

9.2.1 BLACS_FREEBUFF

BLACS_FREEBUFF(ICONTXT, WAIT)

ICONTXT	(input) INTEGER Integer handle indicating the BLACS context.

WAIT	(input) INTEGER
	Whether to wait on non-blocking operations:
	IF (WAIT .EQ. 0) THEN
	Do not wait on operations, free only unused buffers.
	ELSE
	If necessary, wait in order to free all buffers.
	END IF

The BLACS have at least one internal buffer that is used for packing messages (the number of internal buffers varies, depending on which BLACS you are using). On systems where memory is tight, keeping this buffer(s) around may become expensive. Calling this routine will release the BLACS buffer(s). However, the next call to a communication routine which requires packing will cause the buffer to be reallocated.

The parameter WAIT determines whether the BLACS should wait for any non-blocking operations to complete or not. If WAIT = 0, the BLACS will free any buffers that can be freed without waiting. If WAIT is not 0, the BLACS will free all internal buffers, possibly causing the call to block while the BLACS wait for internal non-blocking operations complete.

9.2.2 BLACS_GRIDEXIT

BLACS_GRIDEXIT(ICONTXT)

ICONTXT	(input) INTEGER Integer handle indicating the BLACS context to be freed.

Contexts consume resources, and therefore the user should release them when they are no longer needed. BLACS_GRIDEXIT frees a context. After the freeing of a context, the context no longer exists, and its handle may be re-issued by the BLACS if a new context is defined.

9.2.3 BLACS_ABORT

BLACS_ABORT(ICONTXT, ERRORNUM)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context which is aborting the run.

ERRORNUM (input) INTEGER
User defined integer error number.

When a catastrophic error occurs, the user may need to abort all processes. BLACS_ABORT exists for this reason. Note that both parameters are input, but that BLACS_ABORT uses them only in printing out the error message. The context handle passed in may be anything (i.e., it need not be a valid context handle). This routine kills all BLACS processes, not just those confined to a particular context.

9.2.4 BLACS_EXIT

BLACS_EXIT(CONTINUE)

CONTINUE (input) INTEGER
If CONTINUE is non-zero, it is assumed that the user will continue using the machine after the BLACS are done. Otherwise, it is assumed that no message passing will be done after the BLACS_EXIT call.

This routine should be called when a process has finished all use of the BLACS. It frees all BLACS contexts and releases all memory the BLACS have allocated. CONTINUE indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If CONTINUE is set to 0, then `pvm_exit` will be called; otherwise, it will not. If the user sets CONTINUE not equal to 0, he is indicating that he will be calling explicit PVM send/recvs after the BLACS are done, so that the process cannot tell the virtual machine that it is done. It then becomes the user's responsibility to make sure his code calls `pvm_exit`. PVM users should either call BLACS_EXIT or explicitly call `pvm_exit` to avoid PVM problems.

9.3 Informational and Miscellaneous

These routines return information involving the process grid. Also included here is the barrier routine.

9.3.1 BLACS_GRIDINFO

BLACS_GRIDINFO(ICONTXT, NPROW, NPCOL MYPROW, MYPCOL)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context to be queried.

NPROW (output) INTEGER
On output, the number of process rows in ICONTXT's process grid.

NPCOL (output) INTEGER
On output, the number of process columns in ICONTXT's process grid.

MYPROW (output) INTEGER
On output, the calling process's row coordinate in the process grid.

MYPCOL (output) INTEGER
On output, the calling process's column coordinate in the process grid.

Returns information about the process grid contained in the context whose handle is ICONTXT. If the context handle is invalid, all quantities are returned as -1.

9.3.2 BLACS_PNUM

INTEGER FUNCTION BLACS_PNUM(ICONTXT, PROW, PCOL)

ICONTXT (input) Integer handle indicating the BLACS context to be queried.

PROW (input) INTEGER
The row coordinate of the process whose system process number is to be determined.

PCOL (input) INTEGER
The column coordinate of the process whose system process number is to be determined.

This function returns the system process number (i.e., a task ID for PVM users) of the process at {PROW, PCOL} in the process grid.

9.3.3 BLACS_PCOORD

BLACS_PCOORD(ICONTXT, PNUM, PROW, PCOL)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

PNUM (input) INTEGER
The process number whose coordinates are to be determined. This is the process number of the underlying machine (e.g., it will be a TID for PVM).

PROW (output) INTEGER
On output, the row coordinate of process PNUM in the BLACS grid.

PCOL (output) INTEGER
On output, the column coordinate of process PNUM in the BLACS grid.

Given the system process number (i.e., a task ID for PVM users), returns the row and column coordinates in the BLACS' process grid.

9.3.4 BLACS_BARRIER

BLACS_BARRIER(ICONTXT, SCOPE)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

SCOPE (input) CHARACTER*1
Indicates whether a process row (SCOPE='R'), column ('C'), or entire grid ('A') will participate in barrier.

This routines holds up execution of all processes within the indicated scope until they have all called the routine.

9.4 General purpose

The BLACS have two general purpose routines. They are `BLACS_SET` and `BLACS_GET`. Because they may be called before a grid is created, they are often lumped in with the initialization routines. These routines are used to set and obtain information about various BLACS internals. Some of these internals control general BLACS behavior, and are thus not linked to a particular context. These internals will ignore the parameter `ICONTXT`, which exists so that internals which are tied to a particular context may be operated on.

9.4.1 BLACS_GET

BLACS_GET(ICONTXT, WHAT, VAL)

ICONTXT (input) INTEGER
On WHATs that are tied to a particular context, this is the integer handle indicating the BLACS context to query. Otherwise, it is ignored.

WHAT (input) INTEGER
What BLACS internal information should be returned in VAL. Present options are:

WHAT	Returned in VAL
0	Handle indicating default system context
1	The BLACS message ID range
2	The BLACS debug level
10	Handle indicating the system context used to define the BLACS context whose handle is ICONTXT
11	Number of rings multiring broadcast topology is presently using
12	Number of branches general tree broadcast topology is presently using
13	Number of rings multiring combine topology is presently using
14	Number of branches general tree combine topology is presently using
15	If topologies are being forced to be repeatable, a non-zero is returned. If repeatability is not being enforced, zero is returned (see appendix D for details on repeatability).
16	If topologies are being forced to be heterogeneous coherent, a non-zero is returned. If heterogeneous coherence is not being enforced, zero is returned (see appendix D for details on coherence).

VAL (output) INTEGER ARRAY of variable dimension
The value to which the BLACS internal is presently set. The dimension of VAL is (2) if the message ID range is being returned. For all other queries it is (1).

This routine returns the values the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into BLACS_GRIDINIT or BLACS_GRIDMAP. Some systems, such as MPI, supply their own version of context (in MPI, this corresponds to a communicator). For those users who mix system code with BLACS code, we therefore need to be able to form a BLACS context in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use BLACS_GET to retrieve a default system context which will include all available processes.

Also not tied to a particular BLACS context are the message ID range and the debug level the BLACS were compiled with. For these three values of WHAT, the parameter ICONTXT is not referenced.

The other choices of WHAT are all tied to a particular BLACS context, so the parameter ICONTXT must be a valid BLACS context handle.

9.4.2 BLACS_SET

BLACS_SET(ICONTXT, WHAT, VAL)

ICONTXT (input) INTEGER
On WHATs that are tied to a particular context, this is the integer handle indicating the BLACS context. Otherwise, it is ignored.

WHAT (input) INTEGER
What BLACS internal(s) should be set to VAL. Present options are:

WHAT	VAL determines
1	The BLACS message ID range
11	Number of rings for multiring broadcast topology to use
12	Number of branches for general tree broadcast topology to use
13	Number of rings for multiring combine topology to use
14	Number of branches for general tree combine topology to use
15	Whether topologies should be forced to be repeatable. If this value is 0 (the default) topologies are not required to be repeatable. Any other value requires all used topologies to be repeatable. (see appendix D for details on repeatability).
16	Whether topologies should be forced to be heterogeneous coherent or not. If this value is 0 (the default) topologies are not required to be heterogeneous coherent. Any other value requires all used topologies to be heterogeneous coherent (see appendix D for details on coherence).

VAL (input) INTEGER ARRAY of variable dimension
The value(s) to set internals to. Its specific meaning is dependent on WHAT, as discussed below. Note that for WHAT = 1, the dimension of VAL is (2). Otherwise, it is (1).

Sets BLACS internal defaults. The action taken is dependent upon WHAT, as follows:

1. **Setting the BLACS message ID range**
   If the user wishes to mix the BLACS with other message-passing packages, he may restrict the BLACS to a certain message ID range, which he ensures is not used by the non-BLACS routines. The message ID range must be set before the first call to BLACS_GRIDMAP or BLACS_GRIDINIT. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter ICONTXT is ignored, and VAL is defined as:

   VAL (input) INTEGER array of dimension (2)
   VAL(1): The smallest message ID (also called message type or message tag) the BLACS should use.
   VAL(2): The largest message ID (also called message type or message tag) the BLACS should use.

12. **Set number of rings for TOP = 'M' (multiring broadcast)**
    This quantity is tied to a context, thus ICONTXT is used, and VAL is defined as: VAL(1): The number of rings for multiring topology to use. Valid values are are all nonzero numbers, where negative numbers correspond to decreasing rings, and positive numbers indicate increasing rings. Note that you cannot have more rings

than there are processes in the operation, so if N_p is the number of processes in the operation, then |VAL(1)| > N_p − 1, results in a fully connected topology (i.e., the number of rings will be set to $N_p - 1$).

13. **Set number of branches for TOP = 'T' (general tree broadcast)**
    This quantity is tied to a context, thus ICONTXT is used, and VAL is defined as: VAL(1): The number of branches for general tree topology to use. Valid values are: VAL(1) > 0. Note that you cannot have more branches than there are processes in the operation, so if N_p is the number of processes in the operation, then |VAL(1)| > $N_p - 1$, results in a fully connected topology (i.e., the number of branches will be set to $N_p - 1$).

14. **Set number of rings for TOP = 'M' (multiring combine)**
    This quantity is tied to a context, thus ICONTXT is used, and VAL is defined as: VAL(1): The number of rings for multiring combine topology to use. Valid values are are all nonzero numbers, where negative numbers correspond to decreasing rings, and positive numbers indicate increasing rings. Note that you cannot have more rings than there are processes in the operation, so if N_p is the number of processes in the operation, then |VAL(1)| > N_p − 1, results in a fully connected topology (i.e., the number of rings will be set to $N_p - 1$).

15. **Set number of branches for TOP = 'T' (general tree gather)**
    This quantity is tied to a context, thus ICONTXT is used, and VAL is defined as: VAL(1): The number of branches for general tree topology to use. Valid values are: VAL(1) > 0. Note that you cannot have more branches than there are processes in the operation, so if N_p is the number of processes in the operation, then |VAL(1)| > $N_p - 1$, results in a fully connected topology (i.e., the number of branches will be set to $N_p - 1$).

16. **Set whether topologies must be repeatable or not**
    If this value is set to 0 (the default), topologies are not required to be repeatable. If it is set to any value besides 0, all topologies will be forced to be repeatable (and thus possibly take a performance hit). See Appendix D for details on repeatability.

17. **Set whether topologies must be heterogeneous coherent or not**
    If this value is set to 0 (the default), topologies are not required to be heterogeneous coherent. If it is set to any value besides 0, all topologies will be forced to be heterogeneous coherent (and thus possibly take a performance hit). See Appendix D for details on coherence.

9.5 Unofficial routines

These routines are not part of the BLACS standard, and thus not guaranteed to be in every BLACS implementation. Most of these routines have valid uses, but it is difficult to defend adding them to a message-passing standard. SETPVMTIDS is system specific, and so obviously would not fit into the standard. The timing routines are quite useful, as they allow for system-independent timing, but timing does not have a great deal to do with message passing. Finally, as a service to the user, we allow him to access the BLACS' message ID computation routines. For the user mixing primitive (i.e. system-specific)

message passing with the BLACS, these routines may be convenient. None of these things fit into a message passing standard, and so they are provided by the present BLACS as unofficial service routines.

9.5.1 SETPVMTIDS

SETPVMTIDS(NTASKS, TIDS)

NTASKS	(input) INTEGER The number of PVM tasks the user has spawned.
TIDS	(input) INTEGER array of dimension (NTASKS) This array contains the list of the NTASKS PVM task IDS which will participate in the BLACS.

SETPVMTIDS, as its name implies, is a PVM specific routine. SETPVMTIDS is the advanced PVM user's BLACS_SETUP. BLACS_SETUP may be too restrictive for someone who is using PVM outside the BLACS. For example, they may want to start the main process (process 0,0) via a call to `pvm_spawn`, rather than starting it from the keyboard as BLACS_SETUP requires. SETPVMTIDS requires two parameters from the user. The first is the total number of processes (or tasks) that any BLACS grid will use. Remember that the BLACS is a static system: if you have P processes at the beginning of its execution, you must have those same P processes when the BLACS finish execution. Therefore, the user must set NTASKS to be the largest number of processes he will ever use.

The second argument required by SETPVMTIDS is a list of TIDS in an integer array of at least length NTASKS. All processes require these inputs. This means that in order to use SETPVMTIDS, the PVM user should spawn all of his processes, keeping their TIDS in an integer array, then send that array to all participating processes, and finally have them all call SETPVMTIDS. At this point, he has performed the actions inherent in a BLACS_SETUP call, and he may then proceed to use the BLACS as usual (can make calls to BLACS_PINFO, and then to BLACS_GRIDINIT or BLACS_GRIDMAP, and then proceed with the normal BLACS code).

9.5.2 DCPUTIME

DOUBLE PRECISION FUNCTION DCPUTIME( )

This routine returns time (in seconds) elapsed since an arbitrary starting point. We roughly define CPU time to be the time the processor spends actually executing user code. If CPU time is not available on a given system, `-1.0` is returned.

9.5.3 DWALLTIME

DOUBLE PRECISION FUNCTION DWALLTIME( )

This routine returns time (in seconds) elapsed since an arbitrary starting point. Here we loosely define WALL time to be the time you would figure if you looked at the clock on your wall, began the operation, and then subtracted it from the time your clock showed at the end of the operation. If WALL time is not available on a given system, `-1.0` is returned.

9.5.4 KSENDID

INTEGER FUNCTION KSENDID(ICONTXT, RDEST, CDEST)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

RDEST (input) INTEGER
The row destination of the message that needs an ID.

CDEST (input) INTEGER
The column destination of the message that needs an ID.

Returns a BLACS message ID the user may safely use in primitive send calls.

9.5.5 KRECVID

INTEGER FUNCTION KRECVID(ICONTXT, RSRC, CSRC)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

RSRC (input) INTEGER
The row source of the message that needs an ID.

CSRC (input) INTEGER
The column source of the message that needs an ID.

Returns a BLACS message ID the user may safely use in primitive receive calls.

9.5.6 KBSID

INTEGER FUNCTION KBSID(ICONTXT, SCOPE)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

SCOPE (input) CHARACTER*1
Indicates whether a process row (SCOPE='R'), column ('C'), or entire grid ('A') will participate broadcast.

Returns a BLACS message ID the user may safely use in for the source (destination) of a primitive broadcast (combine).

9.5.7 KBRID

INTEGER FUNCTION KBRID(ICONTXT, SCOPE, RSRC, CSRC)

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context.

SCOPE (input) CHARACTER*1
Indicates whether a process row (SCOPE='R'), column ('C'), or entire grid ('A') will participate broadcast.

RSRC (input) INTEGER
The row source of the broadcast message that needs an ID.

CSRC (input) INTEGER
The column source of the broadcast message that needs an ID.

Returns a BLACS message ID the user may safely use for the destination (contributor) of a primitive broadcast (combine).

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *"LAPACK Users' Guide, Second Edition"*. SIAM, Philadelphia, PA, 1995.

[2] E. Anderson, Z. Bai, C. Bischof, J.W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and A. McKenney. LAPACK: A portable linear algebra library for high-performance computers. Technical Report UT CS-90-105, LAPACK Working Note #20, University of Tennessee, 1990.

[3] M. Barnett, R. Littlefield, D. Payne, and R. A. van de Geijn. Global combine on mesh architectures with wormhole routing. In *7th International Parallel Processing Symposium*, 1993.

[4] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *To appear in Scientific Programming*, 1994. Also available as University of Tennessee LAPACK Working Note #80, UT CS-94-246, 1994.

[5] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. "A Set of Level 3 Basic Linear Algebra Subprograms". *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. "Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs". *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[7] J. Dongarra and E. Grosse. "Distribution of mathematical software via electronic mail". *Communications of the ACM*, 30:403–407, 1987.

[8] J. Dongarra and R. van de Geijn. "Two dimensional Basic Linear Algebra Communication Subprograms". Technical Report UT CS-91-138, LAPACK Working Note #37, University of Tennessee, 1991.

[9] J. Dongarra, R. van de Geijn, and D. Walker. "A Look at Scalable Dense Linear Algebra Librairies". Technical Report UT CS-92-155, LAPACK Working Note #43, University of Tennessee, 1992.

[10] Jack J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Two dimensional basic linear algebra communication subprograms. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 31–40. Elsevier Science Publishers B.V., 1993.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard . *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994. Special issue on MPI. Also available electronically, the url is `ftp://www.netlib.org/mpi/mpi-report.ps`.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. The book is available electronically, the url is `ftp://www.netlib.org/pvm3/book/pvm-book.ps`.

[13] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. "a users' guide to PICL: a portable instrumented communication library". Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, 1990.

[14] Ching-Tien Ho and S. Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 International Conference on Parallel Processing.* IEEE Press, 1986.

[15] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

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

A C Interface to the BLACS

□is	Data operated on is	TYPE is
i	integer	int
s	single precision real	float
d	double precision real	double
c	single precision complex	float
z	double precision complex	double

Table 6: Prefix to C type declaration mapping

Throughout this guide we have been presenting the Fortran 77 interface to the BLACS. The BLACS also have a C interface. In order to avoid name-space conflicts, all C BLACS have C prepended to them (thus DGEBS2D becomes Cdgebs2d, for instance). The only other difference is that C allows the user to pass parameters by value or by address, and the C interface takes advantage of this capability. The names and uses of the parameters remain the same. In particular, the user should note that the C interface BLACS still require the arrays passed in to use column-major storage.

The calling sequences and parameter declarations of the C BLACS are presented below. Here we use the □ to represent the type prefix. Table 6 provides the mapping from the type prefix to the TYPE declaration. This information, coupled with the routine descriptions given in the body of this report, should allow the C programmer to use the BLACS.

A.1 Support Routines

A.1.1 Initialization

```
void Cblacs_pinfo     ( int *mypnum, int *nprocs )
void Cblacs_setup     ( int *mypnum, int *nprocs )
void Cblacs_get       ( int icontxt, int what, int *val )
void Cblacs_set       ( int icontxt, int what, int *val )
void Cblacs_gridinit  ( int *icontxt, char *order, int nprow, int npcol )
void Cblacs_gridmap   ( int *icontxt, int *pmap, int ldpmap, int nprow, int npcol )
```

A.1.2 Destruction

```
void Cblacs_freebuff  ( int icontxt, int wait )
void Cblacs_gridexit  ( int icontxt )
void Cblacs_abort     ( int icontxt, int errornum )
void Cblacs_exit      ( int doneflag )
```

A.1.3 Informational and Miscellaneous

```
void Cblacs_gridinfo  ( int icontxt, int *nprow, int *npcol, int *myprow, int *mypcol )
int  Cblacs_pnum      ( int icontxt, int prow, int pcol )
```

void Cblacs_pcoord (int icontxt, int pnum, int *prow, int *pcol)
void Cblacs_barrier (icontxt, char *scope)

A.1.4 Unofficial

void Csetpvmtids(int ntasks, int *tids)
double Cdcputime ( )
double Cdwalltime ( )
int Cksendid (int icontxt, int rdest, int cdest)
int Ckrecvid (int icontxt, int rsrc, int csrc)
int Ckbsid (int icontxt, char *scope)
int Ckbrid (int icontxt, char *scope, int rsrc, int csrc)

A.2 Point to Point

void C□gesd2d(int icontxt, int m, int n, TYPE *A, int lda, int rdest, int cdest)
void C□gerv2d(int icontxt, int m, int n, TYPE *A, int lda, int rsrc, int csrc)
void C□trsd2d(int icontxt, char *uplo, char *diag, int m, int n, TYPE *A, int lda, int rdest, int cdest)
void C□trrv2d(int icontxt, char *uplo, char *diag, int int n, TYPE *A, int lda, int rsrc, int csrc)

A.3 Broadcasts

void C□gebs2d(int icontxt, char *scope, char *top, int m, int n, TYPE *A, int lda)
void C□gebr2d(int icontxt, char *scope, char *top, int m, int n, TYPE *A, int lda, int rsrc, int csrc)
void C□trbs2d(int icontxt, char *scope, char *top, char *uplo, char *diag, int m, int n, TYPE *A, int lda)
void C□trbr2d(int icontxt, char *uplo, char *diag, int m, int n, TYPE *A, int lda, int rsrc, int csrc)

A.4 Combines

void C□gsum2d(int icontxt, char *scope, char *top, int m, int n, TYPE *A, int lda, int rdest, int cdest)
void C□gamx2d(int icontxt, char *scope, char *top, int m, int n, TYPE *A, int lda, int *RA, int *CA, int RCflag, int rdest, int cdest)
void C□gamn2d(int icontxt, char *scope, char *top, int m, int n, TYPE *A, int lda, int *RA, int *CA, int RCflag, int rdest, int cdest)

B Degrees of Blocking

This appendix provides further discussion of the blocking levels introduced in Section 4.5, and the reasons behind the choices for the BLACS blocking levels. The blocking levels of the BLACS communication routines are:

- vXXSD2D (XX = GE, or TR) : Locally-blocking.
- vXXRV2D (XX = GE, or TR) : Globally-blocking.
- vXXBS2D (XX = GE, or TR) : Globally-blocking.
- vXXBR2D (XX = GE, or TR) : Globally-blocking.
- vGZZZ2D (ZZZ = SUM, AMX or AMN) : Globally-blocking.

B.1 Non-blocking communication

The BLACS do not provide the user with any non-blocking routines. An understanding of this type of communication should still be helpful to the user, however. Therefore, we shall briefly discuss non-blocking communication, and why the BLACS do not explicitly provide this capability.

As previously mentioned, after a non-blocking communication has been posted, the user must probe to determine when the operation has completed. This allows the user to begin an operation, and then do unrelated work until the operation completes (which is determined by the aforementioned probing). In some relatively restricted conditions, this can lead to fairly large performance gains. However, we have found the use of non-blocking communication to be highly error-prone, often leading to non-deterministic or overly complex code. Because of the difficulty in correctly using non-blocking communication, the BLACS, which try to provide an easy-to-program interface, do not explicitly support it. However, whenever it yields increased performance or functionality, the BLACS may use non-blocking communication internally, where its complexity can be shielded from the user.

To give the reader an idea of how non-blocking communication works we will use the example of two processes (for convenience labeled process 0 and process 1) exchanging data (the user buffer is provided by the variable X). This example will be used in each section to illustrate how the various levels of blocking work. The following pseudo-code fragment shows a possible way to perform this swap:

```
IAM = MYPROCESSID()
IF( IAM.EQ.0 ) THEN
   NON_BLOCKING_SEND FROM VARIABLE X TO PROCESS 1
   NON_BLOCKING_RECV INTO VARIABLE TMP FROM PROCESS 1
ELSE IF( IAM .EQ. 1 ) THEN
   NON_BLOCKING_SEND FROM VARIABLE X TO PROCESS 0
   NON_BLOCKING_RECV INTO VARIABLE TMP FROM PROCESS 0
END IF
.
.
```

```
<DO UNRELATED WORK>
.
.
PROBE UNTIL SEND COMPLETES
PROBE UNTIL RECV COMPLETES
X = TMP
```

Even with this simple example, we can see some opportunities for user error. For example, say the user forgets to probe for completion of the send. Then, let us say that process 0's receive completes quickly. Therefore, we overwrite X with process 1's data. Then, when the send is actually completed, process 1 receives its own data back, instead of process 0's data. There are many other ways to go wrong here, and it is for this reason that the BLACS provide no explicit non-blocking operations.

B.2 Locally-blocking

As mentioned before, only send operations may be locally-blocking. Remember that returning from a locally-blocking operation implies the buffer is available for re-use, and will complete even if the complement of the routine has not been posted. It is impossible for a receive to store the send's contents in the buffer (and thus free it for re-use) before the message has been sent, and therefore we see that receives may only be non- or globally-blocking.

We say that a locally-blocking send guarantees completion even if the corresponding receive has not been called. This is actually too strong an assertion. In reality, we guarantee completion up to the limit of available buffer space.

If a locally-blocking send is begun, and the corresponding receive has not been posted, the data to be sent must be buffered so that is not lost when the user's buffer is returned to him. This system buffer may be allocated by the sending process, the destination process, or by the hardware which transports the send. In any case, the amount of buffer space available will always have an upper limit (amount of physical memory free, amount of virtual memory, etc.). When this space is exhausted, the sends will block until enough system buffer space becomes available (i.e., until enough of the outstanding sends are completed by the posting of the corresponding receive). Therefore, while locally-blocking sends allow us greater flexibility than the more restrictive globally-blocking sends we will soon discuss, it is not a good idea to post a great number of them without posting any receives.

As in the non-blocking section, we give an example of two processes swapping data, this time using a locally-blocking send, and a globally-blocking receive.

```
IAM = MYPROCESSID()
IF( IAM.EQ.0 ) THEN
   LOCALLY_BLOCKING_SEND  FROM VARIABLE X TO PROCESS 1
   GLOBALLY_BLOCKING_RECV INTO VARIABLE X FROM PROCESS 1
ELSE IF( IAM .EQ. 1 ) THEN
   LOCALLY_BLOCKING_SEND  FROM VARIABLE X TO PROCESS 0
   GLOBALLY_BLOCKING_RECV INTO VARIABLE X FROM PROCESS 0
END IF
```

B.3 Globally-blocking

Our definition of globally-blocking is a little looser than the common definition. Usually, globally-blocking means that return from an operation insures that the complementary operation has also been called. In other words, return from a globally-blocking send guarantees that the complementary receive has been posted. In this document, however, the term globally-blocking is used to describe any operation that is not guaranteed (see the previous section for a little hedging on using the word "guaranteed" in this context) to return without the complementary post. This means, for instance, that while we must program our code to allow for the possibility that a globally-blocking send does not complete until the corresponding receive is posted, we cannot assume that returning from the send *implies* that the receive has been posted. For example, we can't use one globally-blocking send to synchronize two processes.

The familiar swapping example, this time using globally-blocking sends and receives, is given below.

```
IAM = MYPROCESSID()
IF( IAM.EQ.0 ) THEN
   GLOBALLY_BLOCKING_SEND FROM VARIABLE X TO PROCESS 1
   GLOBALLY_BLOCKING_RECV INTO VARIABLE X FROM PROCESS 1
ELSE IF( IAM .EQ. 1 ) THEN
   TMP = X
   GLOBALLY_BLOCKING_RECV INTO VARIABLE X FROM PROCESS 0
   GLOBALLY_BLOCKING_SEND FROM VARIABLE TMP TO PROCESS 0
END IF
```

Note that process 0 posts a send, followed by a receive, while process 1 posts the receive first, and then the send. If we attempted to use the same pattern as we did with locally-blocking sends (i.e., both processes send and then receive), with a globally-blocking send, we see that 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.

For this simple swap example, having one process reverse the order of its calls is an obvious fix. 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. To be precise, on a system with globally-blocking sends, the following is required in order to show that a code is deadlock free: Let each process be a node in a graph, and let each globally-blocking send create an arc from the sender to the receiver. Let the corresponding receive destroy the arc created by the send. We must never achieve a steady state (all sends/receives which may be reached have been accounted for) where there is a cycle.

For this reason, it was decided that the BLACS would support locally-blocking point to point sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering are used to create locally-blocking sends. The BLACS support globally-blocking point to point receives.

Up till now, we have discussed the blocking levels mainly in terms of point to point message passing, where each operation involves at most two processes. The concept naturally extends to the scoped operations (broadcast and combine), however.

A locally-blocking scoped operation would guarantee to return even if no other processes in the scope called the routine; a globally-blocking scoped operation must be called by all processes before any process is guaranteed to return. All scoped operations in the BLACS are defined as globally-blocking.

The only scoped operation which *could* be programmed as strictly locally-blocking is the broadcast/send operation. However, since only one process in the scope would be calling it, and the others would have to be programmed as globally-blocking, it does not add greatly to the programmability of the code to have locally-blocking broadcast/sends. Further, on some platforms it is possible to achieve considerably better performance if broadcast/sends are allowed to be globally-blocking, and we therefore defined it to fall in line with the other scoped operations as globally-blocking.

C BLACS Error Handling

This section describes the BLACS error handling features. The BLACS error handling behavior may be changed at compile time using the C preprocessor macro `BlacsDebugLvl`. If you are unsure what debug level your BLACS are using, this can be ascertained by call BLACS_GET (see Section 9.4.1 for details).

If the BLACS are compiled with a BLACS debug level of 0, very little error checking is performed. A few critical things will be checked (for instance, BLACS_GRIDINIT will still not allow you to allocate a process grid with more processes than there are available), but for performance reasons, the BLACS will not check most of the parameters.

It is therefore highly recommended that the user link his code to a BLACS library compiled with debug level 1 while debugging his code. BLACS debug level 1 mainly does parameter checking. A few other services are also provided. For instance, the user will be warned if a process sends a message to itself. Having a process send to itself is legal, but it displays very poor performance, and requires enough buffer space that it can occasionally cause hangs for large messages. The BLACS therefore issue a warning when this behavior is detected.

Many times the debug level 0 code will simply hang, and the developer is left without any clue as to what has gone wrong. This may be caused by, for instance, trying to receive from a process which is not in the current context. The debug level 1 BLACS can detect this sort of a user error, and issue a (hopefully helpful) message.

The BLACS issue three types of messages:

1. *BLACS warning*: BLACS detect risky behavior, but attempt to correct or ignore. Warning message is printed, and execution proceeds.

2. *BLACS error*: BLACS detects an error, prints an error message, and kills the machine via a call to BLACS_ABORT.

3. *System error*: The BLACS receive an error message from the underlying system, which is then passed on to the user, and the BLACS kills the machine.

C.1 BLACS Warning and Error Messages

All BLACS warning messages are printed by the internal routine `BlacsWarn`, and all BLACS error messages are printed by the internal routine `BlacsErr`. The only real difference between `BlacsWarn` and `BlacsErr` is that `BlacsErr` calls BLACS_ABORT after the message is printed.

With these central routines handling BLACS error messages, it should be relatively easy for the programmer to modify error handling if the default routines are not adequate for his needs. One particularly annoying problem is that on many systems a print to the screen takes a long time to finish. `BlacsErr` may then kill the machine before the print reaches the screen, and the error message is lost. In this case, the user may wish to make `BlacsErr` wait before killing, or not kill at all, for instance.

BLACS warning messages have the following form:

```
BLACS WARNING '<explanation string>'
from {<p>,<q>}, pnum=<pnum>, Contxt=<ictxt>, on line <#> of file '<fname>'.
```

BLACS error messages have the form:

```
BLACS ERROR '<explanation string>'
from {<p>,<q>}, pnum=<pnum>, Contxt=<ictxt>, on line <#> of file '<fname>'.
```

The meaning of these parameters are:

- `explanation string` This is the message which should help the user track down what is wrong. For example, on an incorrect call to BLACS_GRIDINIT, the user might get: `Process 0 had 2 x 4 grid; correct is 1 x 4.`
- {`p, q`}: The row and column process grid coordinates of the process issuing the warning/error.
- `pnum`: This will be the process number returned in the first argument of BLACS_PINFO.
- `ictxt`: The integer context handle. Please note that this value is not the same across all processes. For instance, process {0, 0} may have `ictxt` = 0 and process {0, 1} have `ictxt` = 1 for the same context. However, the `pnum` and `ictxt` together provide an unambiguous process/context identifier.
- `#`: The line number within the file `fname` which issued the warning.
- `fname`: The file name where the routine which issued the warning/error is located.

Not all of this information may be available at the time an error or warning is issued. For instance, if the error occurs before the creation of the grid, the process grid coordinates will be unavailable. For any value which the BLACS cannot figure out, a -1 is printed to indicate that the value is unknown.

C.1.1 Examples

A few examples should aid in understanding the BLACS warning and error messages.

Example 1

```
BLACS WARNING 'Failure to call BLACS_GET before grid creation makes code non-portable'
from {-1,-1}, pnum=0, Contxt=-1, on line -1 of file 'BLACS_GRIDINIT/BLACS_GRIDMAP'.
```

Here we see that the user has failed to get a system context via a call to BLACS_GET, so that the context handle passed to BLACS_GRIDINIT or BLACS_GRIDMAP is uninitialized. In this case, we were able to detect and correct this, so it is a warning and not an error. If by chance the incoming context had been set to a valid system context, however, we would not have been able to detect the problem, and the new BLACS context would probably be incorrectly defined.

Since this error occurred before the process grid/context was created, we see that the process grid coordinates and the context handle are unavailable, and thus printed as -1. Further, at this point the BLACS do not know which file the user originally called from (BLACS_GRIDINIT or BLACS_GRIDMAP) and thus are also unable to supply the line number.

Example 2

```
BLACS ERROR 'CSRC out of range; CSRC=100, NPCOL=2'
from {0,0}, pnum=0, Contxt=0, on line -1 of file 'igerv2d_.c'.
```

Here we see that process {0, 0} has issued an illegal receive by specifying the the process column source of the message is 100, when there are in fact only 2 process columns. We see that the error is in a call to IGERV2D, and that the line number was unavailable.

C.2 System Error Messages

There are times when the BLACS will receive an error message from the underlying system which is not handled by the BLACS. At this time, the BLACS will print the system error message, and exit. Since these error messages come from the underlying system, their form will necessarily vary depending on which BLACS version is being used. The user may need to obtain a book describing system error messages to understand the message. For example, if the PVM BLACS are being used, a PVM error number will be returned. The PVM quick reference guide, for example, could then be consulted to translate the error number into an understandable error message.

C.2.1 Examples

Example 1

```
libpvm [t40025]: pvm_upkint(): End of buffer
40025: PVM ERROR #-5 on call to pvm_upkint on line 25 of file iunpack00.c.
```

Here we see that the pvm library has printed out an error message saying we've tried to unpack past the end of our buffer. The BLACS have passed back a PVM error number of -5, which is shown to be '`PvmNoData: read past end of buffer`' when looked up on the PVM quick reference guide. Further, we see that the PVM routine in use when the error

occurred was `pvm_upkint`, and that it was called on line 25 of the BLACS internal routine `iunpack00`. This together with some examination of the code in question, revealed that in this case we were trying to receive more data than was sent.

This example is especially well chosen because it illustrates a weakness in the present version of the BLACS error messages. The user is informed that the error occurred in the internal routine `iunpack00`, but not what interface routine originally called `iunpack00`. Because of this lack in the BLACS, the user must examine all possible candidates (in this case, calls to integer point to point receives, broadcast/receives, or combines), for the error. This is a problem that, despite its relative simplicity, has not yet been addressed.

Example 2

```
BLACS ERROR 'MPL error PEMPL17 on call to mpc_recv'
from {0,0}, pnum=0, Contxt=0, on line -1 of file 'Arecv2d00.c'.
```

Here we see that the BLACS are using `BlacsErr` to print a system error message. This particular error might occur when using the MPL BLACS. At this point, the *IBM AIX Parallel Environment Installation and Diagnosis* manual should be consulted. This book indicates that the error `PEMPL17` translates to 'The buffer specified for the operation was too small to hold the received message'.

D Repeatability and coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. The fact the floating point computations are inexact has led us to classify our algorithms according to whether they are *repeatable* and to what degree they guarantee *coherence*. A routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input. A routine is coherent if all processes selected to receive the answer get identical results.

Please note that repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. It is just that inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

In the following sections, we will provide illustrative examples as we examine repeatability and coherence in greater detail.

D.1 Repeatability

Because floating point arithmetic is not exact, it is therefore not truly associative (i.e., (a + b) + c may not be the same as a + (b + c)). The lack of exact arithmetic can cause problems whenever their exists the possibility for reordering of floating point calculations. This problem becomes prevalent in parallel computing due to race conditions in message passing. A simple example should help in understanding the problem. Let us say we have a routine which sums numbers stored on different processes. Let us run this routine on 4

processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0.0, process 1 has the value 1.0, etc.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 then adds them up, and sends the result back to all processes. So, process 0 wants to add a number to 0.0 in the first step. If we order our receives, so that process 0 always receives the message from process 1 first, then 2, and finally 3, we have a *repeatable* algorithm, the result of which is $((0.0 + 1.0) + 2.0) + 3.0$.

However, to get the best parallel performance, it is better not to force a particular ordering, and just have process 0 add the first available number to its value, and continue to do so until all numbers have been added in. If we perform this optimization, a race condition occurs, because the order of the operation is determined by the order in which the messages arrive on process 0, which can be effected by any number of things. This algorithm is not repeatable, because the answer may vary between invocations, even if the input is the same. For instance, one run might produce the sequence $((0.0+1.0)+2.0)+3.0$, while a subsequent run could produce $((0.0 + 2.0) + 1.0) + 3.0$. Both of these results are correct summations of the given numbers, but because of floating point roundoff, they may be different.

Therefore, we see that a routine is *repeatable* if multiple invocations on the same input using the same parallel setup are guaranteed to produce the exact same result.

We classify a routine as not repeatable if multiple invocations on the same input using the same parallel setup may produce different, but equally valid results.

D.2 Coherence

We state that a routine produces coherent output if all processes are guaranteed to have the exact same solution. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, none of our routines are guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms will guarantee coherence only if floating point operations are done the exact same on every node. We call this *homogeneous coherence*: the result will be coherent of the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is *heterogeneous coherence*, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes. Let us go back to our example of summing the process numbers of a 4 process machine for some insight into these levels of coherence.

D.2.1 Example of Incoherence

An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. If we go back to our

example of summing the process numbers, we can demonstrate this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The results each process receives would then be:
Process 0 : $((0.0 + 1.0) + 2.0) + 3.0$
Process 1 : $((1.0 + 2.0) + 3.0) + 0.0$
Process 2 : $((2.0 + 3.0) + 0.0) + 1.0$
Process 3 : $((3.0 + 0.0) + 1.0) + 0.0$

All of these results are equally valid, and all may be different from each other. This algorithm is therefore incoherent. As a side note, notice that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

D.2.2 Example of Homogeneous Coherence

Another way to perform this summation is to have all processes send their data to all other processes, and to ensure we don't have the problem of the previous example, we enforce the natural ordering. Therefore, the answers each node gets is $((0.0 + 1.0) + 2.0) + 3.0$. This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since we have forced an ordering on the addition, this algorithm is repeatable.

D.2.3 Example of Heterogeneous Coherence

In our final example, let us say we have all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can choose any order it wishes and it will give coherent results as long as communication is itself coherent. If we do not force a particular order on the way we do the addition, the algorithm will not be repeatable. If we force a particular order, it will be repeatable.

D.3 Summing it up

We have seen that repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

- **Repeatability**: The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
- **Homogeneous coherence**: All processes selected to possess the result will receive the exact same answer if:
  - Communication does not change the value of the communicated data.
  - All processes perform floating point arithmetic exactly the same.

- **Heterogeneous coherence**: All processes will receive the exact same answer if communication does not change the value of the communicated data.

We have seen that in general, lack of the associative property for floating point calculations may cause both incoherence and/or non-repeatability. We have seen that algorithms that rely on redundant computations are at best homogeneous coherent, and that algorithms in which one process broadcasts the result are heterogeneous coherent. It has been shown that repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they may be ignored in most cases. However, in very specific situations, these issues may become very important. One would not want to have a stopping criteria based on incoherent results, for instance. A user first writing and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run, etc.

In the BLACS, we speak of coherence and repeatability only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. We make the assumption that communication is coherent, which for communication implies that it is also repeatable.

For combine operations, the BLACS allow the user to set flags indicating he wishes combines to be repeatable and/or heterogeneous coherent (see Section 9.4 for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS will restrict the topologies which can be used so that one process figures the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligable to serious depending on the application.

A couple of additional notes are in order. We have discussed incoherence and non-repeatability arising as a result of floating point errors. This might lead the reader to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS will treat integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\sqrt{r^2 + i^2}$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

E Broadcast Topologies

This appendix discusses the broadcast topologies offered by the present BLACS versions in greater detail.

Many factors effect the choice of which topology to use. 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 given below, we use the following symbols: N_p, the number of processors involved in the operation, and T_c, the time for a complete communication (send and receive). Simplified estimates of the time to perform a given algorithm are given below. For a more complete handling of this topic, see [16]

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 indices. 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 (destination) 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. 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 7 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 7 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.

	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	$N_r * T_c$	$\log_2(N_p) * T_c$
MAX TIME	$(\frac{(N_p-1)}{N_r} + N_r - 1) * T_c$	$\log_2(N_p) * T_c$
PIPELINING?	YES	NO

Table 7: Broadcast topology highlights

E.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 cost of the first broadcast is paid, the processors are optimally ordered to perform another ring broadcast. The time each processor incurs for the second broadcast will be roughly T_c, 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 incurred 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. For example, 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 effects 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_c time in the broadcast. However, the last processor in the ring will spend $(N_p - 1) * T_c$ time in algorithm. Figures 3 and 4 respectively show increasing and decreasing ring broadcast.

Unidirectional rings are the most "quiet" algorithms possible: only one processor is sending at a time.

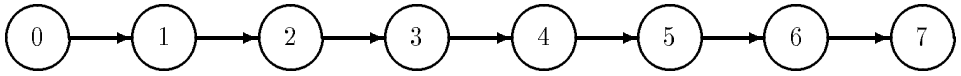


Figure 3: Increasing ring broadcast

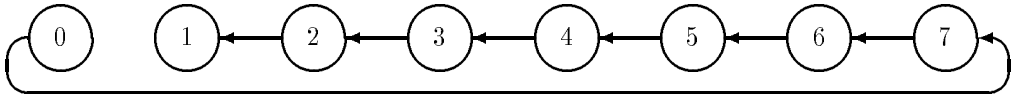


Figure 4: Decreasing ring broadcast

Split Ring The split ring attempts to alleviate the long waiting time inherent in unidirectional rings, without tying up the originating node. Examining Figure 5 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 $(2 * T_c)$ 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.

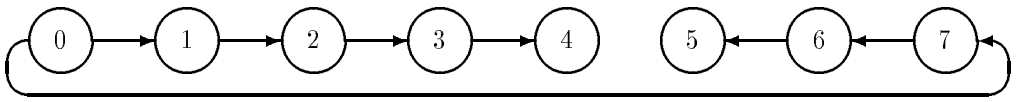


Figure 5: Split ring broadcast

Multiring The multiring algorithm (also referred to as multipath) provides a scalable ring algorithm. By definition, the graph 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) upon which the broadcast is to proceed. The processors participating in the broadcast are then split up into N_r separate increasing or decreasing rings (increasing rings result if BLACS_SET is called with a positive N_r, decreasing rings are used if N_r is set to a negative number). Figure 6 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.

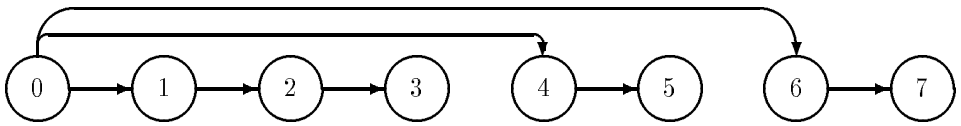


Figure 6: Multiring broadcast with $N_r = 3$

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 $N_r * T_c$. The longest time spent in this algorithm will be roughly $((N_p - 1)/N_r + N_r - 1) * T_c$.

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. The general purpose support routine `BLACS_SET` may be used to do this. Multiring is called by setting `TOP = 'm'`. Here is an example of the recommended way to call the multiring topology:

```
call blacs_set(icontxt, 11, 3)
call dgebs2d(icontxt, 'Row', 'm', m, n, A, lda)
⋮
call dgebs2d(icontxt, 'Column', 'm', 3, 2, work, 5)
```

Notice that `BLACS_SET` 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 paid 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 $T^1 + k * N_r * T_c$.

E.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[3, 14], 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 7.

Hypercube broadcasts are most useful when getting the information out to all processors is more important than saving origin node time. It requires all nodes to spend roughly $T_c * log_2(N_p)$ time in the broadcast. 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 8, 9 and 10 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. The timing analysis for this algorithm is relatively complex, so we do not reproduce it here (analysis for the most common use, $N_b = 1$ is shown in Table 7). See [16] for full details.

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 `BLACS_SET` should be used in the same way as discussed for multiring. An example should clarify this:

```
call blacs_set(icontxt, 11, 2)
call dgebs2d(icontxt, 'Row', 't', m, n, A, lda)
```

This would call the general tree algorithm with $N_b = 2$. The ways to call the general tree broadcast are summarized below.

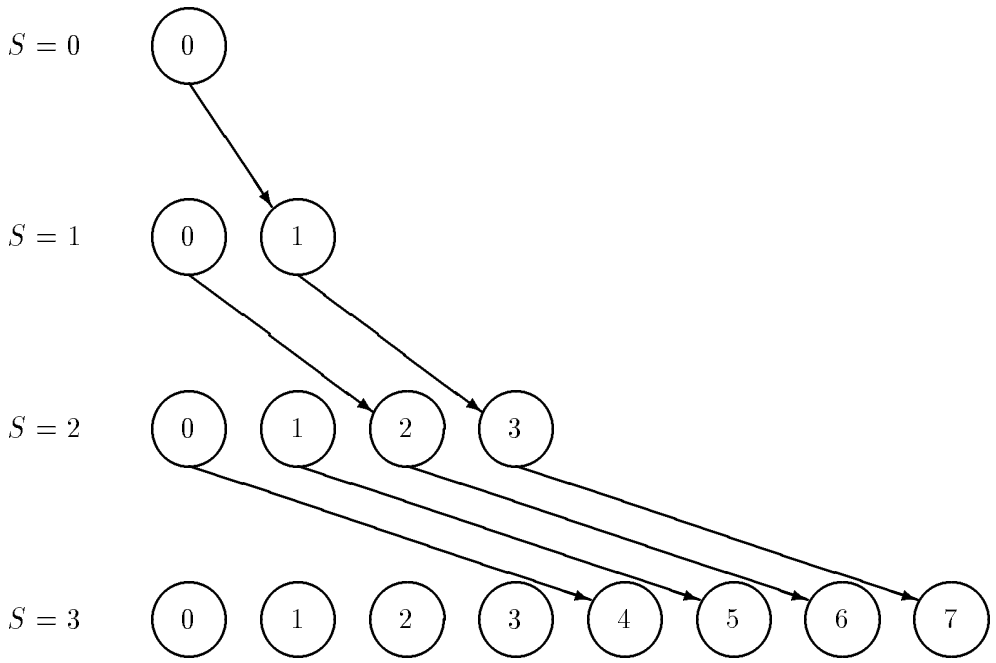


Figure 7: Hypercube broadcast, nearest node first.

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 to `BLACS_SET`.
'f'	perform fully-connected broadcast, i.e. $N_b = Np - 1$

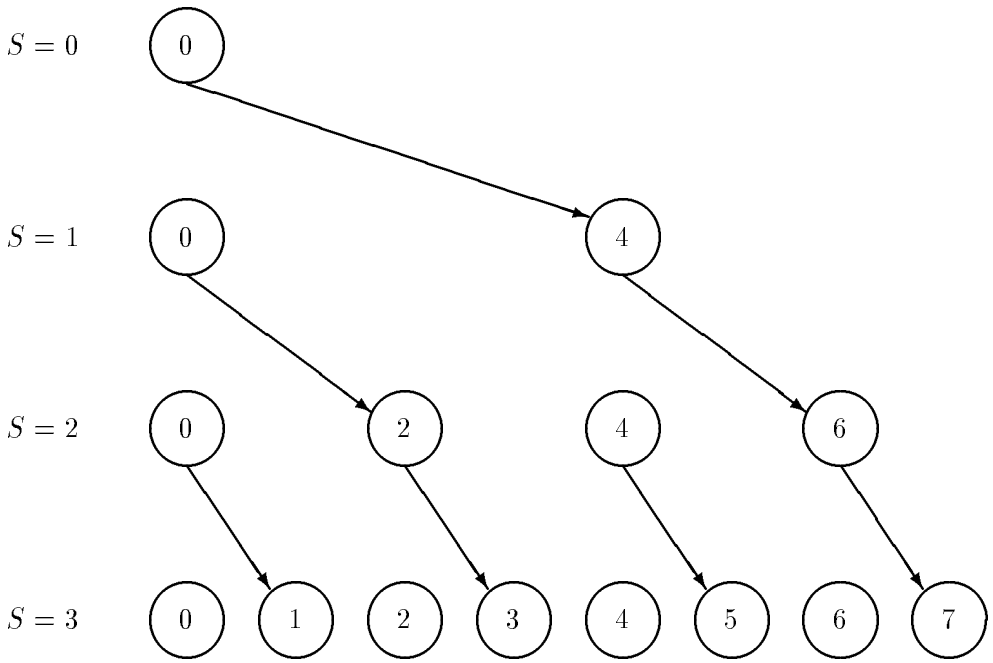


Figure 8: General tree broadcast with $N_b = 1$

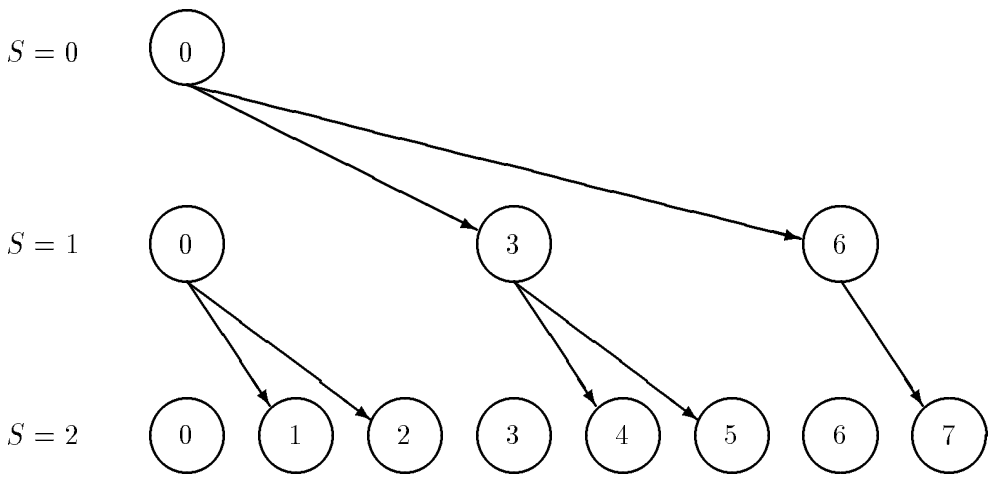


Figure 9: General tree broadcast with $N_b = 2$

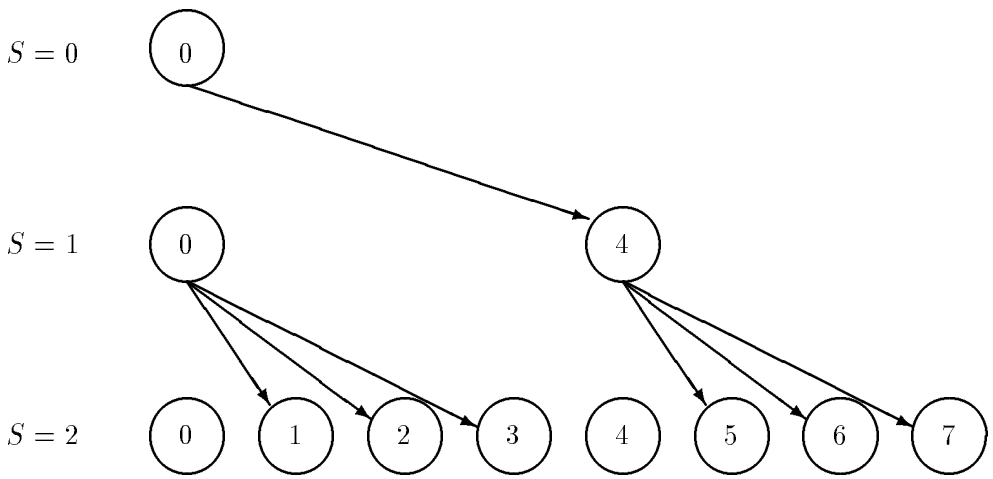


Figure 10: General tree broadcast with $N_b = 3$

F Combine 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.

F.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 appendix E.2, except that communication flows in the opposite direction. Figures 11 and 12 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 `BLACS_SET` 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.

In [16] it is shown that for the presently supported platforms, $N_b = 1$ will usually be the best choice to minimize T_D. It is further demonstrated that $N_b > 1$ broadcasts are typically competitive only for small problem sizes.

With these caveats, we 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 E.2. The longest time any processor would then spend in the algorithm would be $\lceil log_2(N_p) \rceil (2*T_c+T_o)$

This topology is always coherent, and by default is not repeatable. It can be made repeatable by forcing an ordering on the recieves.

F.2 Bidirectional Exchange

This topology is specialized for leave-on-all combines, and therefore, if a leave-on-one combine has been requested, the general tree algorithm with $N_b = 1$ is called instead. It is based on an algorithm presented in [3]. 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 13. 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 $T_D = log_2(N_p) * (2 * T_c + T_o)$, the same as for fan-in/fan-out. However, fan-in/fan-out in general has less link contention, so this topology is only recommended on platforms where sends and receives can be overlapped.

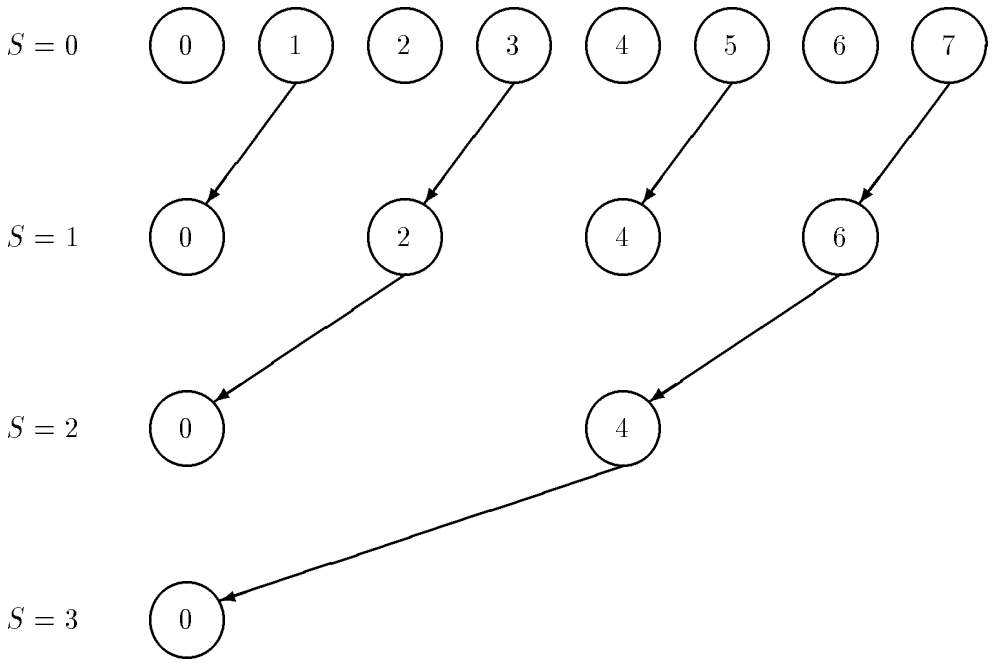


Figure 11: General tree gather with $N_b = 1$

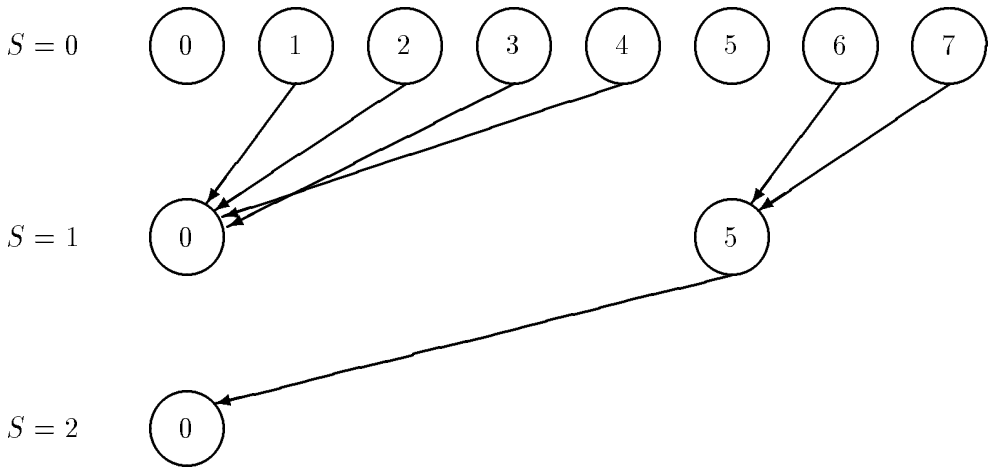


Figure 12: General tree gather with $N_b = 4$

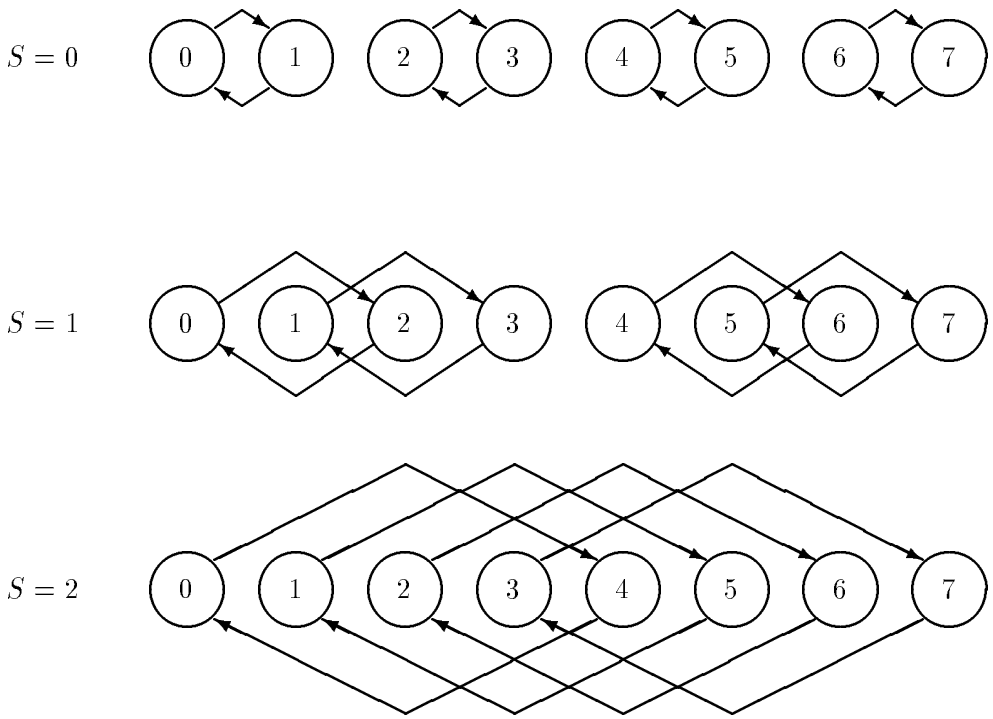


Figure 13: Bidirectional exchange

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 back out to the non-power of two processors. Then, processors will spend roughly $T_D = 2 * T_c + T_o + \lfloor log_2(N_p) \rfloor * (T_c + T_o)$ in the algorithm.

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.

This topology is homogeneous coherent, and repeatable. If the user asks the BLACS to enforce heterogeneous coherence, this topology will not be used.

G Multiring Combine

In this release, only the MPI version has a multiring combine. This topology is much like multiring broadcast: the processes participating in the combine are split up into N_r (number

of rings set by the user) seperate increasing or decreasing rings, which send their partial result to the destination process, where the final result is calculated. If the answer is to be left on all processes, it will be broadcast using the multiring broadcast.

The time to get the answer to one process is $T_D = (\lfloor (N_p - 1)/N_r \rfloor + N_r - 1) * (T_c + T_o)$ If the answer is left on all nodes, the longest time any process spend in the algorithm would be $(\lfloor (N_p - 1)/N_r \rfloor + N_r - 1) * (2 * T_c + T_o)$.

This algorithm can be used to minimize link contention on systems where that is a major concern, and it can display pipelining as well.

Combine pipelining is not as straightforward as broadcast. First, if the answer is left on all processes, the maximal pipe length will be $\lfloor (N_p - 1)/N_r \rfloor$. After this many combines, the broadcast message from previous combines will begin to interfere with new combines.

Mixing pipes formed by broadcasts with those made by combines is not straightforward. In order to use a pipe made by an increasing ring broadcast, for instance, the combine's destination process must be the left neighbor of the broadcast's source. Similarly, the combine destination would need to be the right neighber of a decreasing broadcast source.

H Example Program

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid. For more detailed examples, see the BLACS homepage.

```
      PROGRAM HELLO
*     -- BLACS example code --
*     Written by Clint Whaley 7/26/94
*     Performs a simple check-in type hello world
*     ..
*     .. External Functions ..
      INTEGER BLACS_PNUM
      EXTERNAL BLACS_PNUM
*     ..
*     .. Variable Declaration ..
      INTEGER CONTXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
      INTEGER ICALLER, I, J, HISROW, HISCOL
*
*     Determine my process number and the number of processes in
*     machine
*
      CALL BLACS_PINFO(IAM, NPROCS)
*
*     If in PVM, create virtual machine if it doesn't exist
*
      IF (NPROCS .LT. 1) THEN
         IF (IAM .EQ. 0) THEN
            WRITE(*, 1000)
            READ(*, 2000) NPROCS
         END IF
         CALL BLACS_SETUP(IAM, NPROCS)
      END IF
*
*     Set up process grid that is as close to square as possible
*
      NPROW = INT( SQRT( REAL(NPROCS) ) )
      NPCOL = NPROCS / NPROW
*
*     Get default system context, and define grid
*
      CALL BLACS_GET(0, 0, CONTXT)
      CALL BLACS_GRIDINIT(CONTXT, 'ROW', NPROW, NPCOL)
      CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*     If I'm not in grid, go to end of program
```

```
*
      IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30
*
*     Get my process ID from my grid coordinates
*
      ICALLER = BLACS_PNUM(CONTXT, MYPROW, MYPCOL)
*
*     If I am process {0,0}, receive check-in messages from
*     all nodes
*
      IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

         WRITE(*,*) ' '
         DO 20 I = 0, NPROW-1
            DO 10 J = 0, NPCOL-1

               IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
                  CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
               ENDIF
*
*              Make sure ICALLER is where we think in process grid
*
               CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
               IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
                  WRITE(*,*) 'Grid error!  Halting . . .'
                  STOP
               END IF
               WRITE(*, 3000) I, J, ICALLER

10          CONTINUE
20       CONTINUE
         WRITE(*,*) ' '
         WRITE(*,*) 'All processes checked in.  Run finished.'
*
*     All processes but {0,0} send process ID as a check-in
*
      ELSE
         CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)
      END IF

30    CONTINUE

      CALL BLACS_EXIT(0)

1000  FORMAT('How many processes in machine?')
```

```
2000  FORMAT(I)
3000  FORMAT('Process {',i2,',',i2,'} (node number =',I,
     $       ') has checked in.')

      STOP
      END
```