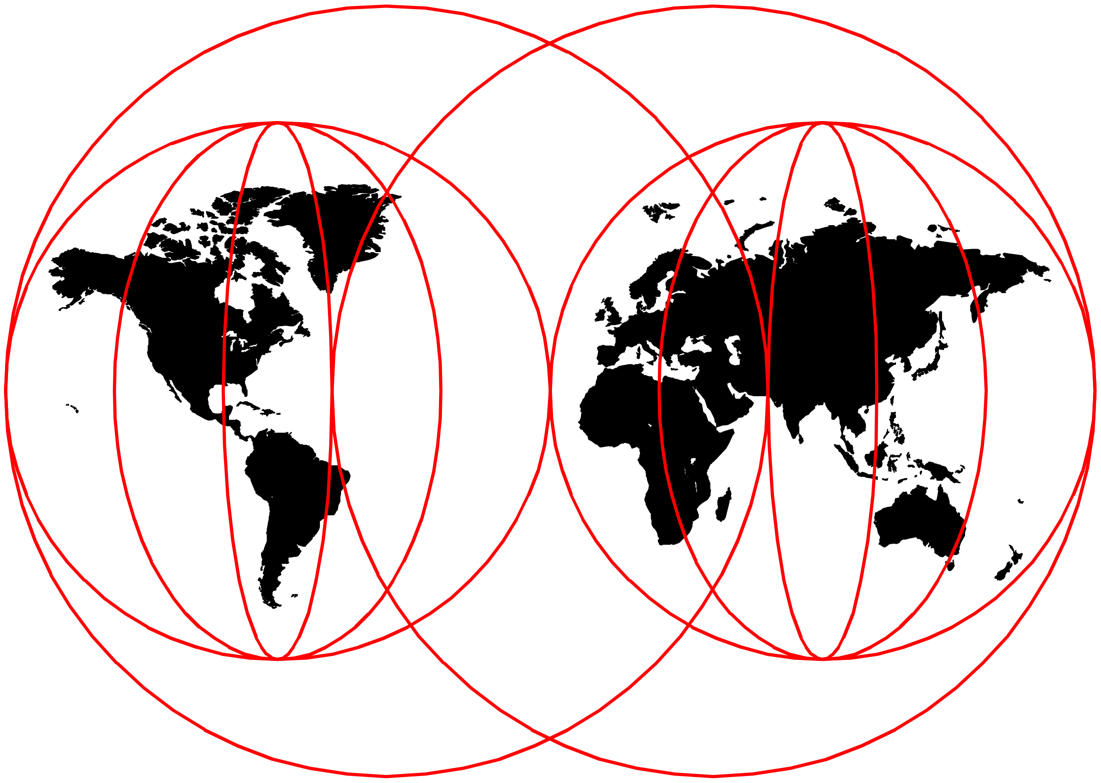
RS/6000 SP: Practical MPI Programming

Yukiya Aoyama
Jun Nakano



International Technical Support Organization

www.redbooks.ibm.com

SG24-5380-00

International Technical Support Organization

SG24-5380-00

RS/6000 SP: Practical MPI Programming

August 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, “Special Notices” on page 207.

First Edition (August 1999)

This edition applies to MPI as is relates to IBM Parallel Environment for AIX Version 2 Release 3 and Parallel System Support Programs 2.4 and subsequent releases.

This redbook is based on an unpublished document written in Japanese. Contact nakanoj@jp.ibm.com for details.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Building 003 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This redbook helps you write MPI (Message Passing Interface) programs that run on distributed memory machines such as the RS/6000 SP. This publication concentrates on the real programs that RS/6000 SP solution providers want to parallelize. Complex topics are explained using plenty of concrete examples and figures.

The SPMD (Single Program Multiple Data) model is the main topic throughout this publication.

The basic architectures of parallel computers, models of parallel computing, and concepts used in the MPI, such as communicator, process rank, collective communication, point-to-point communication, blocking and non-blocking communication, deadlocks, and derived data types are discussed.

Methods of parallelizing programs using distributed data to processes followed by the superposition, pipeline, twisted decomposition, and prefix sum methods are examined.

Individual algorithms and detailed code samples are provided. Several programming strategies described are; two-dimensional finite difference method, finite element method, LU factorization, SOR method, the Monte Carlo method, and molecular dynamics. In addition, the MPMD (Multiple Programs Multiple Data) model is discussed taking coupled analysis and a master/worker model as examples. A section on Parallel ESSL is included.

A brief description of how to use Parallel Environment for AIX Version 2.4 and a reference of the most frequently used MPI subroutines are enhanced with many illustrations and sample programs to make it more readable than the MPI Standard or the reference manual of each implementation of MPI.

We hope this publication will erase of the notion that MPI is too difficult, and will provide an easy start for MPI beginners.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from IBM Japan working at the RS/6000 Technical Support Center, Tokyo.

Yukiya Aoyama has been involved in technical computing since he joined IBM Japan in 1982. He has experienced vector tuning for 3090 VF, serial tuning for RS/6000, and parallelization on RS/6000 SP. He holds a B.S. in physics from Shimane University, Japan.

Jun Nakano is an IT Specialist from IBM Japan. From 1990 to 1994, he was with the IBM Tokyo Research Laboratory and studied algorithms. Since 1995, he has been involved in benchmarks of RS/6000 SP. He holds an M.S. in physics from the University of Tokyo. He is interested in algorithms, computer architectures, and operating systems. He is also a coauthor of the redbook, *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*.

This project was coordinated by:

Scott Vetter
International Technical Support Organization, Austin Center

Thanks to the following people for their invaluable contributions to this project:

Anshul Gupta
IBM T. J. Watson Research Center

Danny Shieh
IBM Austin

Yoshinori Shimoda
IBM Japan

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 221 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an internet note to `redbook@us.ibm.com`

Chapter 1. Introduction to Parallel Programming

This chapter provides brief descriptions of the architectures that support programs running in parallel, the models of parallel programming, and an example of parallel processing.

1.1 Parallel Computer Architectures

You can categorize the architecture of parallel computers in terms of two aspects: whether the memory is physically centralized or distributed, and whether or not the address space is shared. Table 1 provides the relationships of these attributes.

Table 1. Categorization of Parallel Architectures

	Shared Address Space	**Individual Address Space**
Centralized memory	SMP (Symmetric Multiprocessor)	N/A
Distributed memory	NUMA (Non-Uniform Memory Access)	MPP (Massively Parallel Processors)

SMP (Symmetric Multiprocessor) architecture uses shared system resources such as memory and I/O subsystem that can be accessed equally from all the processors. As shown in Figure 1, each processor has its own cache which may have several levels. SMP machines have a mechanism to maintain coherency of data held in local caches. The connection between the processors (caches) and the memory is built as either a bus or a crossbar switch. For example, the POWER3 SMP node uses a bus, whereas the RS/6000 model S7A uses a crossbar switch. A single operating system controls the SMP machine and it schedules processes and threads on processors so that the load is balanced.

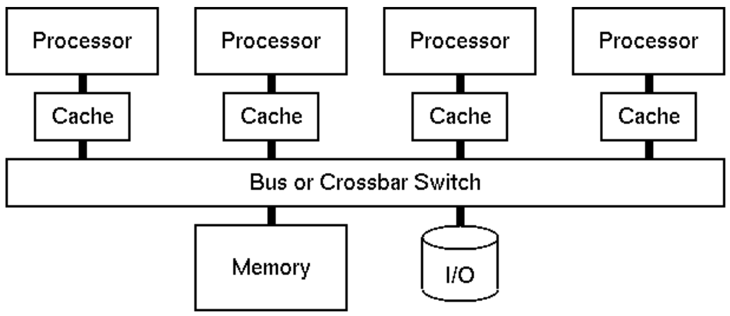


Figure 1. SMP Architecture

MPP (Massively Parallel Processors) architecture consists of nodes connected by a network that is usually high-speed. Each node has its own processor, memory, and I/O subsystem (see Figure 2 on page 2). The operating system is running on each node, so each node can be considered a workstation. The RS/6000 SP fits in this category. Despite the term *massively*, the number of nodes is not necessarily large. In fact, there is no criteria. What makes the situation more complex is that each node can be an SMP node (for example, POWER3 SMP node) as well as a uniprocessor node (for example, 160 MHz POWER2 Superchip node).

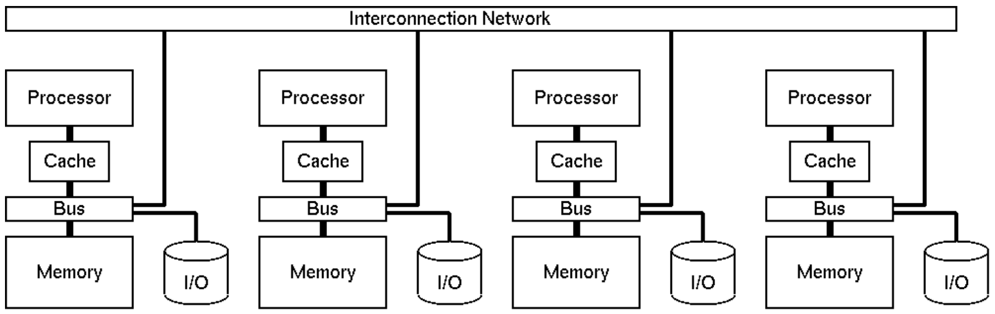


Figure 2. MPP Architecture

NUMA (Non-Uniform Memory Access) architecture machines are built on a similar hardware model as MPP, but it typically provides a shared address space to applications using a hardware/software directory-based protocol that maintains cache coherency. As in an SMP machine, a single operating system controls the whole system. The memory latency varies according to whether you access local memory directly or remote memory through the interconnect. Thus the name *non-uniform memory access.* The RS/6000 series has not yet adopted this architecture.

1.2 Models of Parallel Programming

The main goal of parallel programming is to utilize all the processors and minimize the elapsed time of your program. Using the current software technology, there is no software environment or layer that absorbs the difference in the architecture of parallel computers and provides a single programming model. So, you may have to adopt different programming models for different architectures in order to balance performance and the effort required to program.

1.2.1 SMP Based

Multi-threaded programs are the best fit with SMP architecture because threads that belong to a process share the available resources. You can either write a multi-thread program using the POSIX threads library (pthreads) or let the compiler generate multi-thread executables. Generally, the former option places the burdeon on the programmer, but when done well, it provides good performance because you have complete control over how the programs behave. On the other hand, if you use the latter option, the compiler automatically parallelizes certain types of DO loops, or else you must add some directives to tell the compiler what you want it to do. However, you have less control over the behavior of threads. For details about SMP features and thread coding techniques using XL Fortran, see *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, SG24-5155.

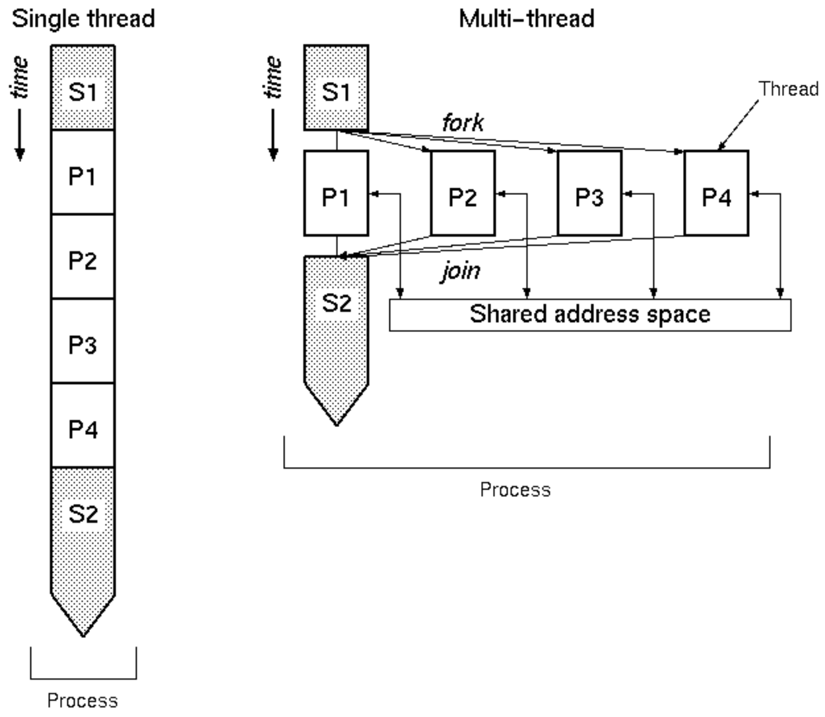


Figure 3. Single-Thread Process and Multi-Thread Process

In Figure 3, the single-thread program processes S1 through S2, where S1 and S2 are inherently sequential parts and P1 through P4 can be processed in parallel. The multi-thread program proceeds in the *fork-join model*. It first processes S1, and then the first thread forks three threads. Here, the term *fork* is used to imply the creation of a thread, not the creation of a process. The four threads process P1 through P4 in parallel, and when finished they are joined to the first thread. Since all the threads belong to a single process, they share the same address space and it is easy to reference data that other threads have updated. Note that there is some overhead in forking and joining threads.

1.2.2 MPP Based on Uniprocessor Nodes (Simple MPP)

If the address space is not shared among nodes, parallel processes have to transmit data over an interconnecting network in order to access data that other processes have updated. HPF (High Performance Fortran) may do the job of data transmission for the user, but it does not have the flexibility that hand-coded message-passing programs have. Since the class of problems that HPF resolves is limited, it is not discussed in this publication.

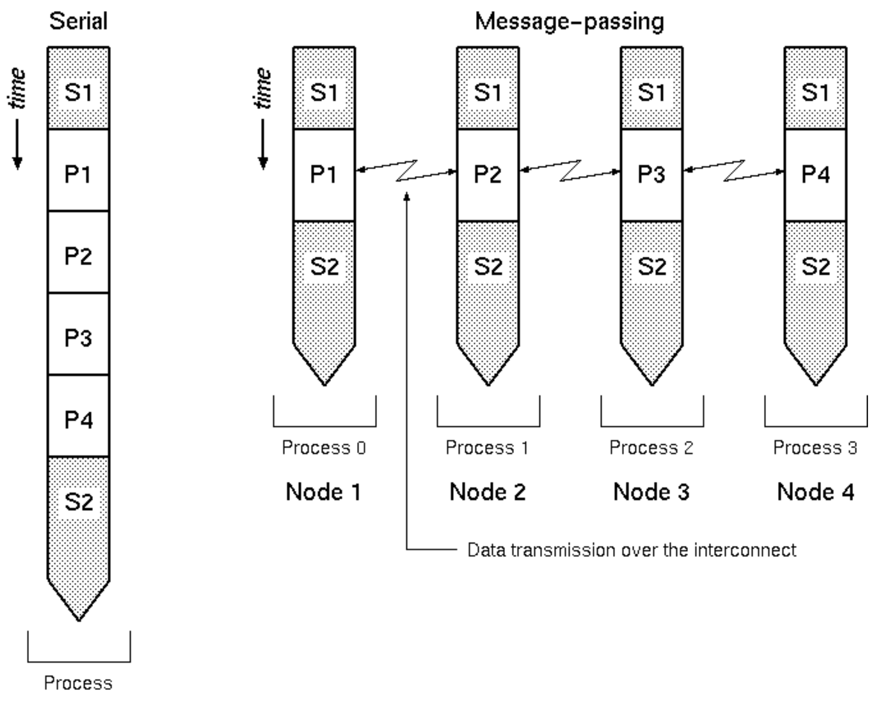


Figure 4. Message-Passing

Figure 4 illustrates how a message-passing program runs. One process runs on each node and the processes communicate with each other during the execution of the parallelizable part, P1-P4. The figure shows links between processes on the adjacent nodes only, but each process communicates with all the other processes in general. Due to the communication overhead, work load unbalance, and synchronization, time spent for processing each of P1-P4 is generally longer in the message-passing program than in the serial program. All processes in the message-passing program are bound to S1 and S2.

1.2.3 MPP Based on SMP Nodes (Hybrid MPP)

An RS/6000 SP with SMP nodes makes the situation more complex. In the hybrid architecture environment you have the following two options.

Multiple Single-Thread Processes per Node

In this model, you use the same parallel program written for simple MPP computers. You just increase the number of processes according to how many processors each node has. Processes still communicate with each other by message-passing whether the message sender and receiver run on the same node or on different nodes. The key for this model to be successful is that the intranode message-passing is optimized in terms of communication latency and bandwidth.

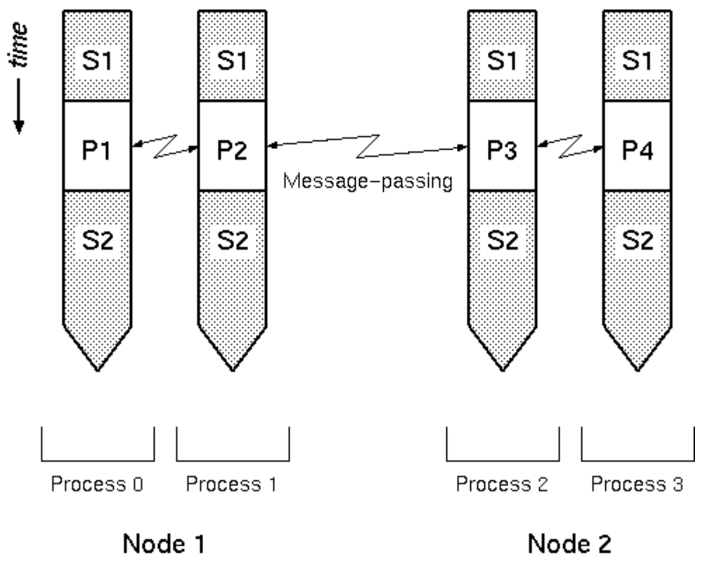


Figure 5. Multiple Single-Thread Processes Per Node

Parallel Environment Version 2.3 and earlier releases only allow one process to use the high-speed protocol (*User Space protocol*) per node. Therefore, you have to use IP for multiple processes, which is slower than the User Space protocol. In Parallel Environment Version 2.4, you can run up to four processes using User Space protocol per node. This functional extension is called MUSPPA (Multiple User Space Processes Per Adapter). For communication latency and bandwidth, see the paragraph beginning with "Performance Figures of Communication" on page 6.

One Multi-Thread Process Per Node

The previous model (multiple single-thread processes per node) uses the same program written for simple MPP, but a drawback is that even two processes running on the same node have to communicate through message-passing rather than through shared memory or memory copy. It is possible for a parallel run-time environment to have a function that automatically uses shared memory or memory copy for intranode communication and message-passing for internode communication. Parallel Environment Version 2.4, however, does not have this automatic function yet.

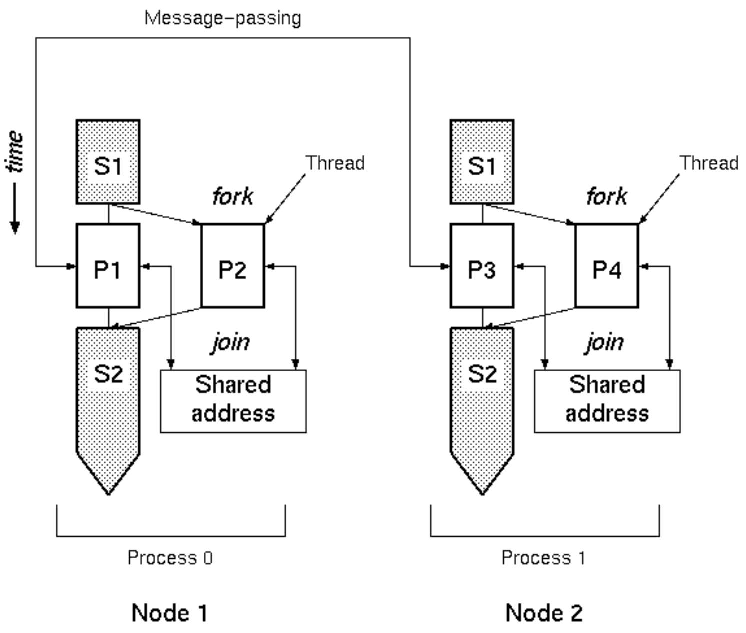


Figure 6. One Multi-Thread Process Per Node

To utilize the shared memory feature of SMP nodes, run one multi-thread process on each node so that intranode communication uses shared memory and internode communication uses message-passing. As for the multi-thread coding, the same options described in 1.2.1, “SMP Based” on page 2 are applicable (user-coded and compiler-generated). In addition, if you can replace the parallelizable part of your program by a subroutine call to a multi-thread parallel library, you do not have to use threads. In fact, Parallel Engineering and Scientific Subroutine Library for AIX provides such libraries.

Note

Further discussion of MPI programming using multiple threads is beyond the scope of this publication.

Performance Figures of Communication

Table 2 shows point-to-point communication latency and bandwidth of User Space and IP protocols on POWER3 SMP nodes. The software used is AIX 4.3.2, PSSP 3.1, and Parallel Environment 2.4. The measurement was done using a Pallas MPI Benchmark program. Visit `http://www.pallas.de/pages/pmb.htm` for details.

Table 2. Latency and Bandwidth of SP Switch (POWER3 Nodes)

Protocol	Location of two processes	Latency	Bandwidth
User Space	On different nodes	22 μ sec	133 MB/sec
	On the same node	37 μ sec	72 MB/sec

Protocol	Location of two processes	Latency	Bandwidth
IP	On different nodes	159 μ sec	57 MB/sec
	On the same node	119 μ sec	58 MB/sec

Note that when you use User Space protocol, both latency and bandwidth of intranode communication is not as good as internode communication. This is partly because the intranode communication is not optimized to use memory copy at the software level for this measurement. When using SMP nodes, keep this in mind when deciding which model to use. If your program is not multi-threaded and is communication-intensive, it is possible that the program will run faster by lowering the degree of parallelism so that only *one* process runs on each node neglecting the feature of *multiple* processors per node.

1.3 SPMD and MPMD

When you run multiple processes with message-passing, there are further categorizations regarding how many different programs are cooperating in parallel execution. In the SPMD (Single Program Multiple Data) model, there is only one program and each process uses the same executable working on different sets of data (Figure 7 (a)). On the other hand, the MPMD (Multiple Programs Multiple Data) model uses different programs for different processes, but the processes collaborate to solve the same problem. Most of the programs discussed in this publication use the SPMD style. Typical usage of the MPMD model can be found in the master/worker style of execution or in the coupled analysis, which are described in 4.7, “MPMD Models” on page 137.

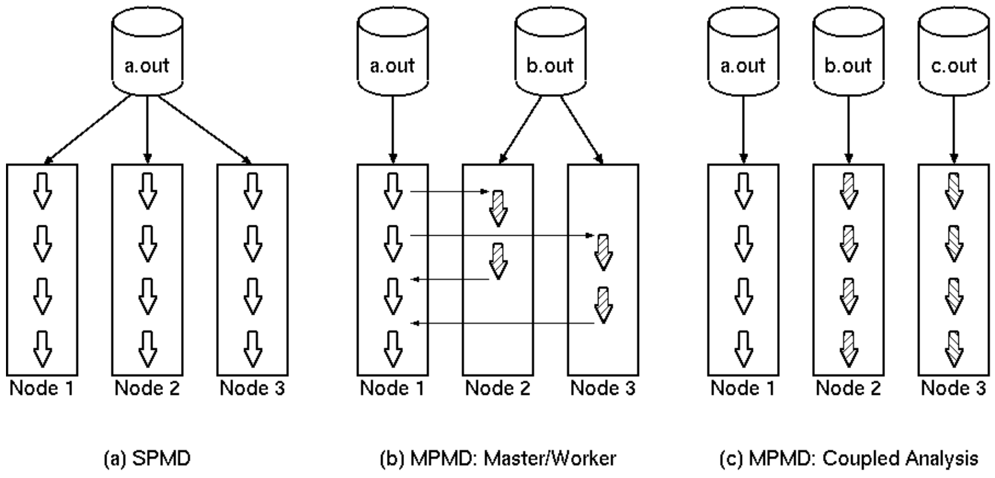


Figure 7. SPMD and MPMD

Figure 7 (b) shows the master/worker style of the MPMD model, where `a.out` is the master program which dispatches jobs to the worker program, `b.out`. There are several workers serving a single master. In the coupled analysis (Figure 7 (c)), there are several programs (`a.out`, `b.out`, and `c.out`), and each program does a different task, such as structural analysis, fluid analysis, and thermal analysis. Most of the time, they work independently, but once in a while, they exchange data to proceed to the next time step.

In the following figure, the way an SPMD program works and why message-passing is necessary for parallelization is introduced.

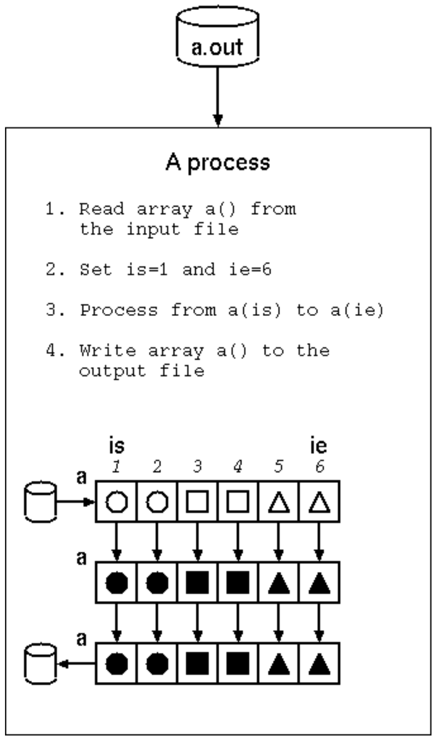
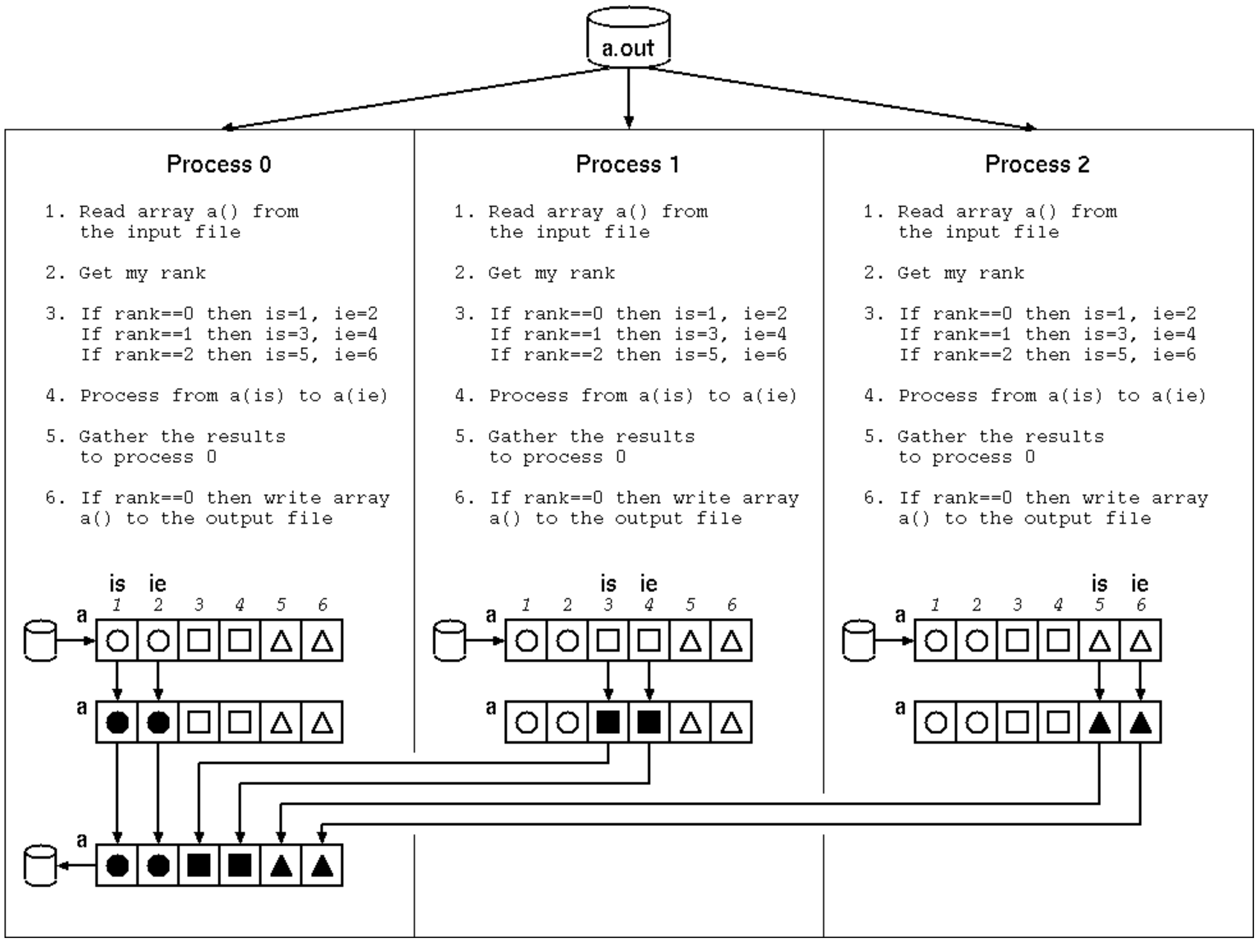

a.out
A process
1. Read array a() from the input file
2. Set is=1 and ie=6
3. Process from a(is) to a(ie)
4. Write array a() to the output file
is
ie
a
1 2 3 4 5 6
a
a

Figure 8. A Sequential Program

Figure 8 shows a sequential program that reads data from a file, does some computation on the data, and writes the data to a file. In this figure, white circles, squares, and triangles indicate the initial values of the elements, and black objects indicate the values after they are processed. Remember that in the SPMD model, all the processes execute the same program. To distinguish between processes, each process has a unique integer called *rank*. You can let processes behave differently by using the value of rank. Hereafter, the process whose rank is *r* is referred to as process *r*. In the parallelized program in Figure 9 on page 9, there are three processes doing the job. Each process works on one third of the data, therefore this program is expected to run three times faster than the sequential program. This is the very benefit that you get from parallelization.



```
1. Read array a() from
   the input file

2. Get my rank

3. If rank==0 then is=1, ie=2
   If rank==1 then is=3, ie=4
   If rank==2 then is=5, ie=6

4. Process from a(is) to a(ie)

5. Gather the results
   to process 0

6. If rank==0 then write array
   a() to the output file
```

Figure 9. An SPMD Program

In Figure 9, all the processes read the array in Step 1 and get their own rank in Step 2. In Steps 3 and 4, each process determines which part of the array it is in charge of, and processes that part. After all the processes have finished in Step 4, none of the processes have all of the data, which is an undesirable side effect of parallelization. It is the role of message-passing to consolidate the processes separated by the parallelization. Step 5 gathers all the data to a process and that process writes the data to the output file.

To summarize, keep the following two points in mind:

- The purpose of parallelization is to reduce the time spent for computation. Ideally, the parallel program is *p* times faster than the sequential program, where *p* is the number of processes involved in the parallel execution, but this is not always achievable.
- Message-passing is the tool to consolidate what parallelization has separated. It should not be regarded as the parallelization itself.

The next chapter begins a voyage into the world of parallelization.

Chapter 2. Basic Concepts of MPI

In this chapter, the basic concepts of the MPI such as communicator, point-to-point communication, collective communication, blocking/non-blocking communication, deadlocks, and derived data types are described. After reading this chapter, you will understand how data is transmitted between processes in the MPI environment, and you will probably find it easier to write a program using MPI rather than TCP/IP.

2.1 What is MPI?

The Message Passing Interface (MPI) is a standard developed by the Message Passing Interface Forum (MPIF). It specifies a portable interface for writing message-passing programs, and aims at practicality, efficiency, and flexibility at the same time. MPIF, with the participation of more than 40 organizations, started working on the standard in 1992. The first draft (Version 1.0), which was published in 1994, was strongly influenced by the work at the IBM T. J. Watson Research Center. MPIF has further enhanced the first version to develop a second version (MPI-2) in 1997. The latest release of the first version (Version 1.2) is offered as an update to the previous release and is contained in the MPI-2 document. For details about MPI and MPIF, visit `http://www.mpi-forum.org/`. The design goal of MPI is quoted from “MPI: A Message-Passing Interface Standard (Version 1.1)” as follows:

- *Design an application programming interface (not necessarily for compilers or a system implementation library).*
- *Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.*
- *Allow for implementations that can be used in a heterogeneous environment.*
- *Allow convenient C and Fortran 77 bindings for the interface.*
- *Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.*
- *Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.*
- *Define an interface that can be implemented on many vendor’s platforms, with no significant changes in the underlying communication and system software.*
- *Semantics of the interface should be language independent.*
- *The interface should be designed to allow for thread-safety.*

The standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies

- Bindings for Fortran 77 and C
- Environmental management and inquiry
- Profiling interface

The IBM Parallel Environment for AIX (PE) Version 2 Release 3 accompanying with Parallel System Support Programs (PSSP) 2.4 supports MPI Version 1.2, and the IBM Parallel Environment for AIX Version 2 Release 4 accompanying with PSSP 3.1 supports MPI Version 1.2 and some portions of MPI-2. The MPI subroutines supported by PE 2.4 are categorized as follows:

Table 3. MPI Subroutines Supported by PE 2.4

Type	Subroutines	Number
Point-to-Point	MPI_SEND, MPI_RECV, MPI_WAIT,...	35
Collective Communication	MPI_BCAST, MPI_GATHER, MPI_REDUCE,...	30
Derived Data Type	MPI_TYPE_CONTIGUOUS, MPI_TYPE_COMMIT,...	21
Topology	MPI_CART_CREATE, MPI_GRAPH_CREATE,...	16
Communicator	MPI_COMM_SIZE, MPI_COMM_RANK,...	17
Process Group	MPI_GROUP_SIZE, MPI_GROUP_RANK,...	13
Environment Management	MPI_INIT, MPI_FINALIZE, MPI_ABORT,...	18
File	MPI_FILE_OPEN, MPI_FILE_READ_AT,...	19
Information	MPI_INFO_GET, MPI_INFO_SET,...	9
IBM Extension	MPE_IBCAST, MPE_IGATHER,...	14

You do not need to know all of these subroutines. When you parallelize your programs, only about a dozen of the subroutines may be needed. Appendix B, “Frequently Used MPI Subroutines Illustrated” on page 161 describes 33 frequently used subroutines with sample programs and illustrations. For detailed descriptions of MPI subroutines, see *MPI Programming and Subroutine Reference Version 2 Release 4,* GC23-3894.

2.2 Environment Management Subroutines

This section shows what an MPI program looks like and explains how it is executed on RS/6000 SP. In the following program, each process writes the number of the processes and its rank to the standard output. Line numbers are added for the explanation.

env.f

```
1       PROGRAM env
2       INCLUDE 'mpif.h'
3       CALL MPI_INIT(ierr)
4       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
5       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
6       PRINT *,'nprocs =',nprocs,'myrank =',myrank
7       CALL MPI_FINALIZE(ierr)
8       END
```

Note that the program is executed in the SPMD (Single Program Multiple Data) model. All the nodes that run the program, therefore, need to see the same executable file with the same path name, which is either shared among nodes by NFS or other network file systems, or is copied to each node's local disk.

Line 2 includes mpif.h, which defines MPI-related parameters such as MPI_COMM_WORLD and MPI_INTEGER. For example, MPI_INTEGER is an integer whose value is 18 in Parallel Environment for AIX. All Fortran procedures that use MPI subroutines have to include this file. Line 3 calls MPI_INIT for initializing an MPI environment. MPI_INIT must be called once and only once before calling any other MPI subroutines. In Fortran, the return code of every MPI subroutine is given in the last argument of its subroutine call. If an MPI subroutine call is done successfully, the return code is 0; otherwise, a non zero value is returned. In Parallel Environment for AIX, without any user-defined error handler, a parallel process ends abnormally if it encounters an MPI error: PE prints error messages to the standard error output and terminates the process. Usually, you do not check the return code each time you call MPI subroutines. The subroutine MPI_COMM_SIZE in line 4 returns the number of processes belonging to the communicator specified in the first argument. A *communicator* is an identifier associated with a group of processes. MPI_COMM_WORLD defined in mpif.h represents the group consisting of all the processes participating in the parallel job. You can create a new communicator by using the subroutine MPI_COMM_SPLIT. Each process in a communicator has its unique *rank*, which is in the range `0..size-1` where `size` is the number of processes in that communicator. A process can have different ranks in each communicator that the process belongs to. MPI_COMM_RANK in line 5 returns the rank of the process within the communicator given as the first argument. In line 6, each process prints the number of all processes and its rank, and line 7 calls MPI_FINALIZE. MPI_FINALIZE terminates MPI processing and no other MPI call can be made afterwards. Ordinary Fortran code can follow MPI_FINALIZE. For details of MPI subroutines that appeared in this sample program, see B.1, “Environmental Subroutines” on page 161.

Suppose you have already decided upon the node allocation method and it is configured appropriately. (Appendix A, “How to Run Parallel Jobs on RS/6000 SP” on page 155 shows you the detail.) Now you are ready to compile and execute the program as follows. (Compile options are omitted.)

```
$ mpxlf env.f
** env   === End of Compilation 1 ===
1501-510  Compilation successful for file env.f.
$ export MP_STDOUTMODE=ordered
$ export MP_LABELIO=yes
$ a.out -procs 3
  0: nprocs = 3 myrank = 0
  1: nprocs = 3 myrank = 1
  2: nprocs = 3 myrank = 2
```

For compiling and linking MPI programs, use the `mpxlf` command, which takes care of the paths for include files and libraries for you. For example, mpif.h is located at /usr/lpp/ppe.poe/include, but you do not have to care about it. The environment variables MP_STDOUTMODE and MP_LABELIO control the stdout and stderr output from the processes. With the setting above, the output is sorted by increasing order of ranks, and the rank number is added in front of the output from each process.

Although each process executes the same program in the SPMD model, you can make the behavior of each process different by using the value of the rank. This is where the parallel speed-up comes from; each process can operate on a different part of the data or the code concurrently.

2.3 Collective Communication Subroutines

Collective communication allows you to exchange data among a group of processes. The communicator argument in the collective communication subroutine calls specifies which processes are involved in the communication. In other words, all the processes belonging to that communicator must call the same collective communication subroutine with matching arguments. There are several types of collective communications, as illustrated below.

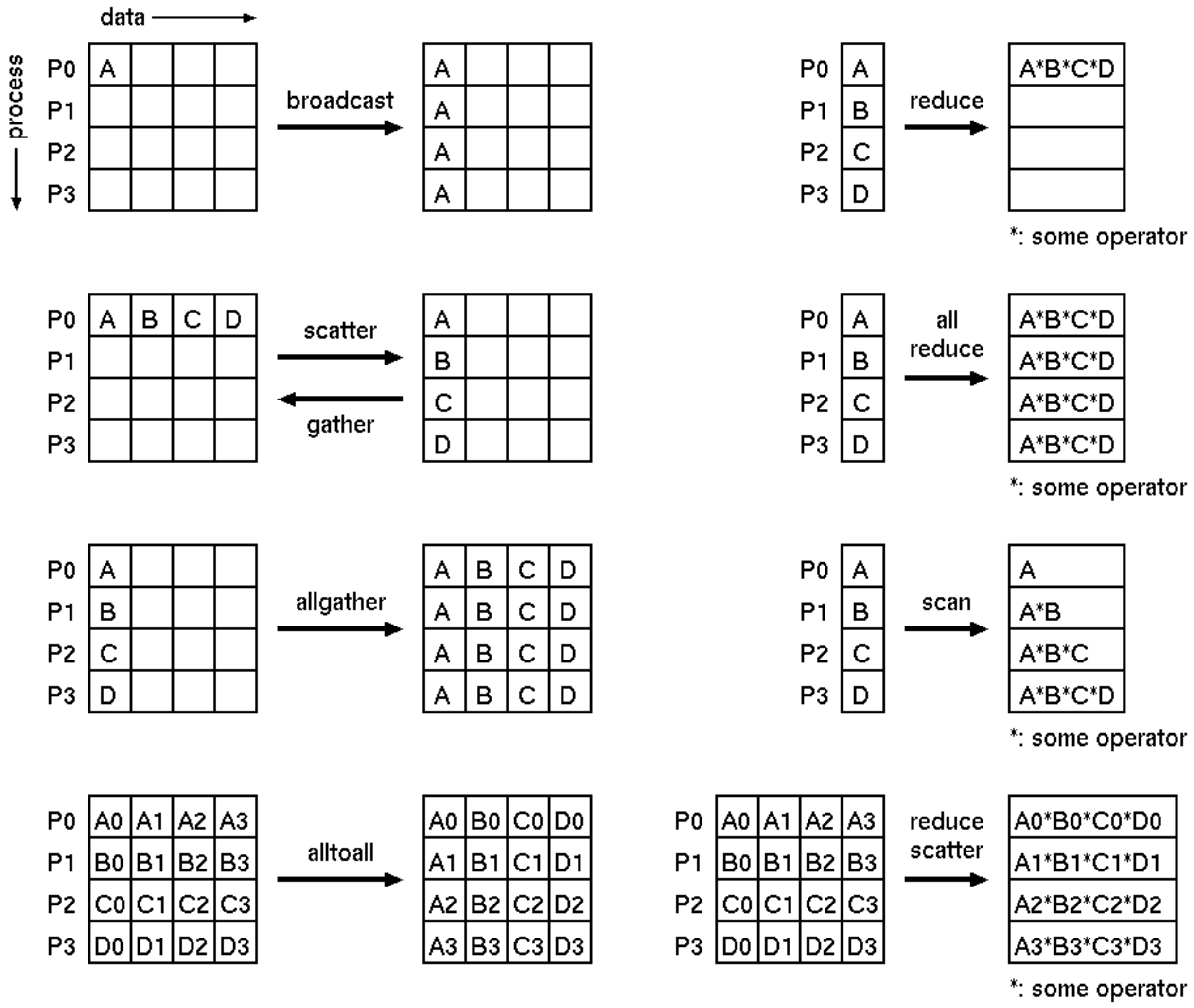


Figure 10. Patterns of Collective Communication

Some of the patterns shown in Figure 10 have a variation for handling the case where the length of data for transmission is different among processes. For example, you have subroutine MPI_GATHERV corresponding to MPI_GATHER.

Table 4 shows 16 MPI collective communication subroutines that are divided into four categories.

Table 4. MPI Collective Communication Subroutines

Category	Subroutines
1. One buffer	**MPI_BCAST**
2. One send buffer and one receive buffer	**MPI_GATHER**, MPI_SCATTER, **MPI_ALLGATHER**, MPI_ALLTOALL, MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV
3. Reduction	**MPI_REDUCE**, **MPI_ALLREDUCE**, MPI_SCAN, MPI_REDUCE_SCATTER
4. Others	MPI_BARRIER, MPI_OP_CREATE, MPI_OP_FREE

The subroutines printed in boldface are used most frequently. MPI_BCAST, MPI_GATHER, and MPI_REDUCE are explained as representatives of the main three categories.

All of the MPI collective communication subroutines are blocking. For the explanation of blocking and non-blocking communication, see 2.4.1, “Blocking and Non-Blocking Communication” on page 23. IBM extensions to MPI provide non-blocking collective communication. Subroutines belonging to categories 1, 2, and 3 have IBM extensions corresponding to non-blocking subroutines such as MPE_IBCAST, which is a non-blocking version of MPI_BCAST.

2.3.1 MPI_BCAST

The subroutine MPI_BCAST broadcasts the message from a specific process called *root* to all the other processes in the communicator given as an argument. (See also B.2.1, “MPI_BCAST” on page 163.)

bcast.f

```
 1       PROGRAM bcast
 2       INCLUDE 'mpif.h'
 3       INTEGER imsg(4)
 4       CALL MPI_INIT(ierr)
 5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 7       IF (myrank==0) THEN
 8         DO i=1,4
 9           imsg(i) = i
10         ENDDO
11       ELSE
12         DO i=1,4
13           imsg(i) = 0
14         ENDDO
15       ENDIF
16       PRINT *,'Before:',imsg
17       CALL MP_FLUSH(1)
18       CALL MPI_BCAST(imsg, 4, MPI_INTEGER,
19      &               0, MPI_COMM_WORLD, ierr)
20       PRINT *,'After :',imsg
21       CALL MPI_FINALIZE(ierr)
22       END
```

In bcast.f, the process with rank=0 is chosen as the root. The root stuffs an integer array imsg with data, while the other processes initialize it with zeroes. MPI_BCAST is called in lines 18 and 19, which broadcasts four integers from the root process (its rank is 0, the fourth argument) to the other processes in the communicator MPI_COMM_WORLD. The triplet (imsg, 4, MPI_INTEGER) specifies the address of the buffer, the number of elements, and the data type of the elements. Note the different role of imsg in the root process and in the other processes. On the root process, imsg is used as the send buffer, whereas on non-root processes, it is used as the receive buffer. MP_FLUSH in line 17 flushes the standard output so that the output can be read easily. MP_FLUSH is not an MPI subroutine and is only included in IBM Parallel Environment for AIX. The program is executed as follows:

```
$ a.out -procs 3
  0: Before: 1 2 3 4
  1: Before: 0 0 0 0
  2: Before: 0 0 0 0
  0: After : 1 2 3 4
  1: After : 1 2 3 4
  2: After : 1 2 3 4
```

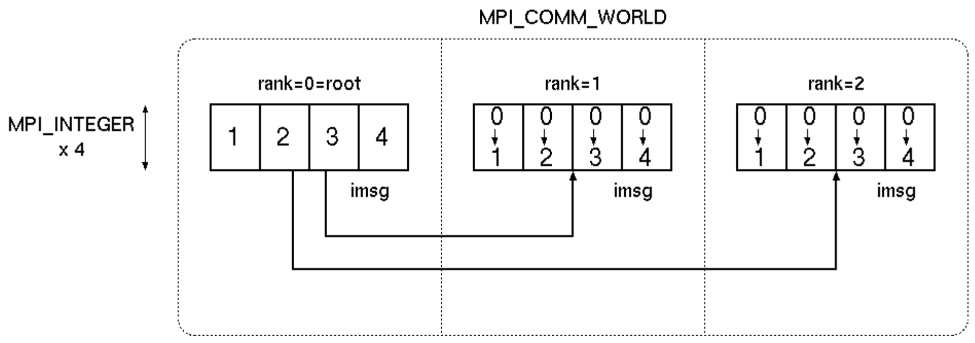


Figure 11. MPI_BCAST

Descriptions of MPI data types and communication buffers follow.

MPI subroutines recognize data types as specified in the MPI standard. The following is a description of MPI data types in the Fortran language bindings.

Table 5. MPI Data Types (Fortran Bindings)

MPI Data Types	Description (Fortran Bindings)
MPI_INTEGER1	1-byte integer
MPI_INTEGER2	2-byte integer
MPI_INTEGER4, MPI_INTEGER	4-byte integer
MPI_REAL4, MPI_REAL	4-byte floating point
MPI_REAL8, MPI_DOUBLE_PRECISION	8-byte floating point
MPI_REAL16	16-byte floating point
MPI_COMPLEX8, MPI_COMPLEX	4-byte float real, 4-byte float imaginary
MPI_COMPLEX16, MPI_DOUBLE_COMPLEX	8-byte float real, 8-byte float imaginary

MPI Data Types	Description (Fortran Bindings)
MPI_COMPLEX32	16-byte float real, 16-byte float imaginary
MPI_LOGICAL1	1-byte logical
MPI_LOGICAL2	2-byte logical
MPI_LOGICAL4, MPI_LOGICAL	4-byte logical
MPI_CHARACTER	1-byte character
MPI_BYTE, MPI_PACKED	N/A

You can combine these data types to make more complex data types called *derived data types*. For details, see 2.5, “Derived Data Types” on page 28.

As line 18 of `bcast.f` shows, the send buffer of the root process and the receive buffer of non-root processes are referenced by the same name. If you want to use a different buffer name in the receiving processes, you can rewrite the program as follows:

```
IF (myrank==0) THEN
  CALL MPI_BCAST(imsg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
ELSE
  CALL MPI_BCAST(jmsg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
ENDIF
```

In this case, the contents of `imsg` of process 0 are sent to `jmsg` of the other processes. Make sure that the amount of data transmitted matches between the sending process and the receiving processes.

2.3.2 MPI_GATHER

The subroutine MPI_GATHER transmits data from all the processes in the communicator to a single receiving process. (See also B.2.5, “MPI_GATHER” on page 169 and B.2.6, “MPI_GATHERV” on page 171.)

gather.f

```
 1       PROGRAM gather
 2       INCLUDE 'mpif.h'
 3       INTEGER irecv(3)
 4       CALL MPI_INIT(ierr)
 5       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 6       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 7       isend = myrank + 1
 8       CALL MPI_GATHER(isend, 1, MPI_INTEGER,
 9      &                irecv, 1, MPI_INTEGER,
10      &                0, MPI_COMM_WORLD, ierr)
11       IF (myrank==0) THEN
12         PRINT *,'irecv =',irecv
13       ENDIF
14       CALL MPI_FINALIZE(ierr)
15       END
```

In this example, the values of `isend` of processes 0, 1, and 2 are 1, 2, and 3 respectively. The call of MPI_GATHER in lines 8-10 gathers the value of `isend` to a receiving process (process 0) and the data received are copied to an integer array `irecv` in increasing order of rank. In lines 8 and 9, the triplets `(isend, 1,` MPI_INTEGER) and (irecv, 1, MPI_INTEGER) specify the address of the send/receive buffer, the number of elements, and the data type of the elements. Note that in line 9, the number of elements received from each process by the root process (in this case, 1) is given as an argument. This is not the total number of elements received at the root process.

```
  $ a.out -procs 3
    0: irecv = 1 2 3
```

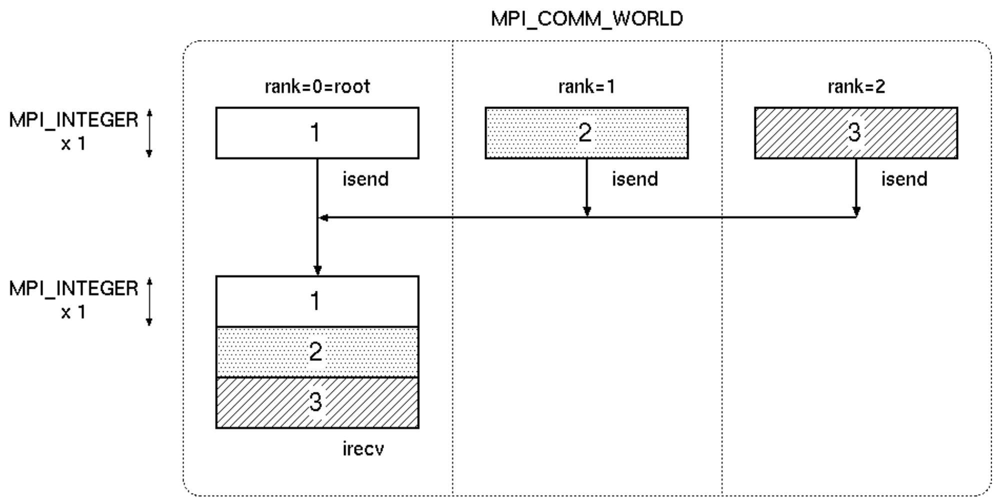


Figure 12. MPI_GATHER

Important

The memory locations of the send buffer (isend) and the receive buffer (irecv) must not overlap. The same restriction applies to all the collective communication subroutines that use send and receive buffers (categories 2 and 3 in Table 4 on page 15).

In MPI-2, this restriction is partly removed: You can use the send buffer as the receive buffer by specifying MPI_IN_PLACE as the first argument of MPI_GATHER at the root process. In such a case, sendcount and sendtype are ignored at the root process, and the contribution of the root to the gathered array is assumed to be in the correct place already in the receive buffer.

When you use MPI_GATHER, the length of the message sent from each process must be the same. If you want to gather different lengths of data, use MPI_GATHERV instead.

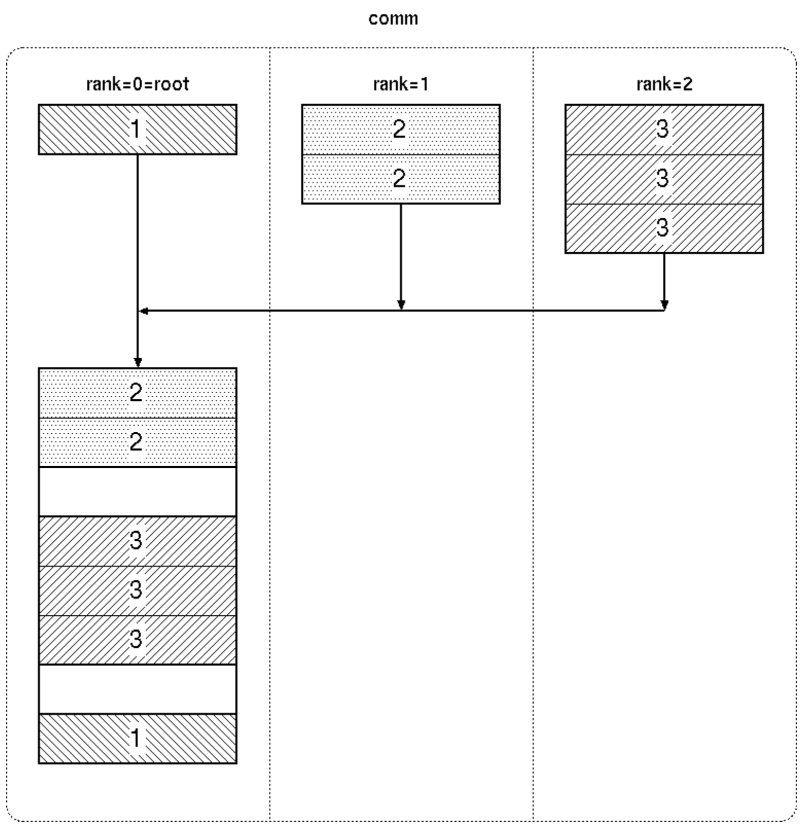


Figure 13. MPI_GATHERV

As Figure 13 shows, MPI_GATHERV gathers messages with different sizes and you can specify the displacements that the gathered messages are placed in the receive buffer. Like MPI_GATHER, subroutines MPI_SCATTER, MPI_ALLGATHER, and MPI_ALLTOALL have corresponding “V” variants, namely, MPI_SCATTERV, MPI_ALLGATHERV, and MPI_ALLTOALLV.

2.3.3 MPI_REDUCE

The subroutine MPI_REDUCE does reduction operations such as summation of data distributed over processes, and brings the result to the root process. (See also B.2.11, “MPI_REDUCE” on page 180.)

reduce.f

```
 1        PROGRAM reduce
 2        INCLUDE 'mpif.h'
 3        REAL a(9)
 4        CALL MPI_INIT(ierr)
 5        CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 6        CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 7        ista = myrank * 3 + 1
 8        iend = ista + 2
 9        DO i=ista,iend
10          a(i) = i
11        ENDDO
12        sum = 0.0
13        DO i=ista,iend
```

```
14          sum = sum + a(i)
15        ENDDO
16        CALL MPI_REDUCE(sum, tmp, 1, MPI_REAL, MPI_SUM, 0,
17       &                MPI_COMM_WORLD, ierr)
18        sum = tmp
19        IF (myrank==0) THEN
20          PRINT *,'sum =',sum
21        ENDIF
22        CALL MPI_FINALIZE(ierr)
23        END
```

The program above calculates the sum of a floating-point array `a(i) (i=1..9)`. It is assumed that there are three processes involved in the computation, and each process is in charge of one third of the array `a()`. In lines 13-15, a partial sum (`sum`) is calculated by each process, and in lines 16-17, these partial sums are added and the result is sent to the root process (process 0). Instead of nine additions, each process does three additions plus one global sum. As is the case with MPI_GATHER, the send buffer and the receive buffer cannot overlap in memory. Therefore, another variable, `tmp`, had to be used to store the global sum of `sum`. The fifth argument of MPI_REDUCE, MPI_SUM, specifies which reduction operation to use, and the data type is specified as MPI_REAL. The MPI provides several common operators by default, where MPI_SUM is one of them, which are defined in mpif.h. See Table 6 on page 21 for the list of operators. The following output and figure show how the program is executed.

```
$ a.out -procs 3
  0: sum = 45.00000000
```

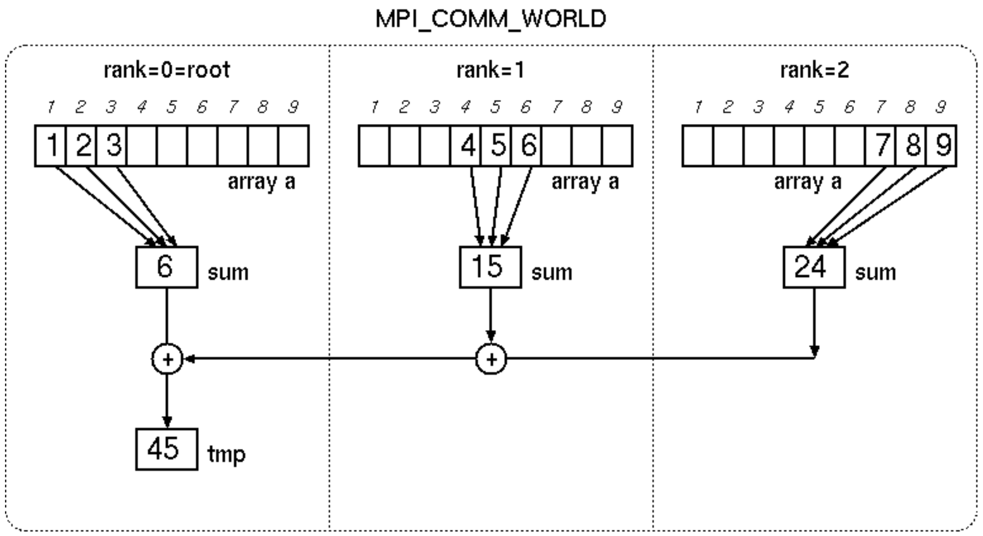


Figure 14. MPI_REDUCE (MPI_SUM)

When you use MPI_REDUCE, be aware of rounding errors that MPI_REDUCE may produce. In floating-point computations with finite accuracy, you have $(a+b)+c \neq a+(b+c)$ in general. In `reduce.f`, you wanted to calculate the sum of the array `a()`. But since you calculate the partial sum first, the result may be different from what you get using the serial program.

Sequential computation:

```
a(1) + a(2) + a(3) + a(4) + a(5) + a(6) + a(7) + a(8) + a(9)
```

Parallel computation:

```
[a(1) + a(2) + a(3)] + [a(4) + a(5) + a(6)] + [a(7) + a(8) + a(9)]
```

Moreover, in general, you need to understand the order that the partial sums are added. Fortunately, in PE, the implementation of MPI_REDUCE is such that you always get the same result if you execute MPI_REDUCE with the same arguments using the same number of processes.

Table 6. Predefined Combinations of Operations and Data Types

Operation	Data type
MPI_SUM (sum), MPI_PROD (product)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX
MPI_MAX (maximum), MPI_MIN (minimum)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
MPI_MAXLOC (max value and location), MPI_MINLOC (min value and location)	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_LAND (logical AND), MPI_LOR (logical OR), MPI_LXOR (logical XOR)	MPI_LOGICAL
MPI_BAND (bitwise AND), MPI_BOR (bitwise OR), MPI_BXOR (bitwise XOR)	MPI_INTEGER, MPI_BYTE

MPI_MAXLOC obtains the value of the maximum element of an array and its location at the same time. If you are familiar with XL Fortran intrinsic functions, MPI_MAXLOC can be understood as MAXVAL and MAXLOC combined. The data type MPI_2INTEGER in Table 6 means two successive integers. In the Fortran bindings, use a one-dimensional integer array with two elements for this data type. For real data, MPI_2REAL is used, where the first element stores the maximum or the minimum value and the second element is its location converted to real. The following is a serial program that finds the maximum element of an array and its location.

```
PROGRAM maxloc_s
INTEGER n(9)
DATA n /12, 15, 2, 20, 8, 3, 7, 24, 52/
imax = -999
DO i = 1, 9
  IF (n(i) > imax) THEN
    imax = n(i)
    iloc = i
  ENDIF
ENDDO
PRINT *, 'Max =', imax, 'Location =', iloc
END
```

The preceding program is parallelized for three-process execution as follows:

```
PROGRAM maxloc_p
INCLUDE 'mpif.h'
INTEGER n(9)
```

```
      INTEGER isend(2), irecv(2)
      DATA n /12, 15, 2, 20, 8, 3, 7, 24, 52/
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      ista = myrank * 3 + 1
      iend = ista + 2
      imax = -999
      DO i = ista, iend
        IF (n(i) > imax) THEN
          imax = n(i)
          iloc = i
        ENDIF
      ENDDO
      isend(1) = imax
      isend(2) = iloc
      CALL MPI_REDUCE(isend, irecv, 1, MPI_2INTEGER,
     &                MPI_MAXLOC, 0, MPI_COMM_WORLD, ierr)
      IF (myrank == 0) THEN
        PRINT *, 'Max =', irecv(1), 'Location =', irecv(2)
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Note that local maximum (`imax`) and its location (`iloc`) is copied to an array `isend(1:2)` before reduction.

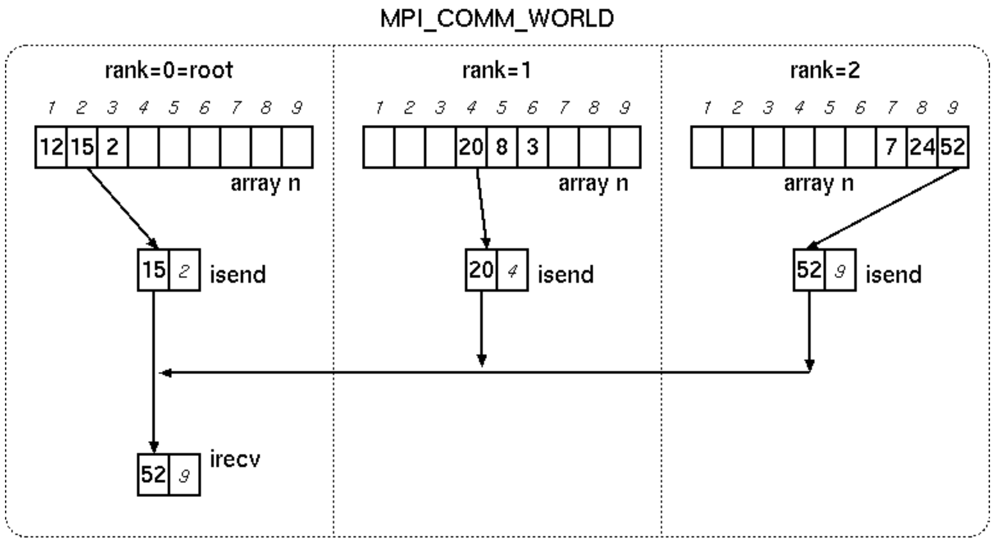


Figure 15. MPI_REDUCE (MPI_MAXLOC)

The output of the program is shown below.

```
$ a.out -procs 3
  0: Max = 52 Location = 9
```

If none of the operations listed in Table 6 on page 21 meets your needs, you can define a new operation with MPI_OP_CREATE. Appendix B.2.15, “MPI_OP_CREATE” on page 187 shows how to define “MPI_SUM” for MPI_DOUBLE_COMPLEX and “MPI_MAXLOC” for a two-dimensional array.

2.4 Point-to-Point Communication Subroutines

When you use point-to-point communication subroutines, you should know about the basic notions of blocking and non-blocking communication, as well as the issue of deadlocks.

2.4.1 Blocking and Non-Blocking Communication

Even when a single message is sent from process 0 to process 1, there are several steps involved in the communication. At the sending process, the following events occur one after another.

1. The data is copied to the *user buffer* by the user.
2. The user calls one of the MPI send subroutines.
3. The system copies the data from the user buffer to the system buffer.
4. The system sends the data from the system buffer to the destination process.

The term *user buffer* means scalar variables or arrays used in the program. The following occurs during the receiving process:

1. The user calls one of the MPI receive subroutines.
2. The system receives the data from the source process and copies it to the system buffer.
3. The system copies the data from the system buffer to the user buffer.
4. The user uses the data in the user buffer.

Figure 16 on page 24 illustrates the above steps.

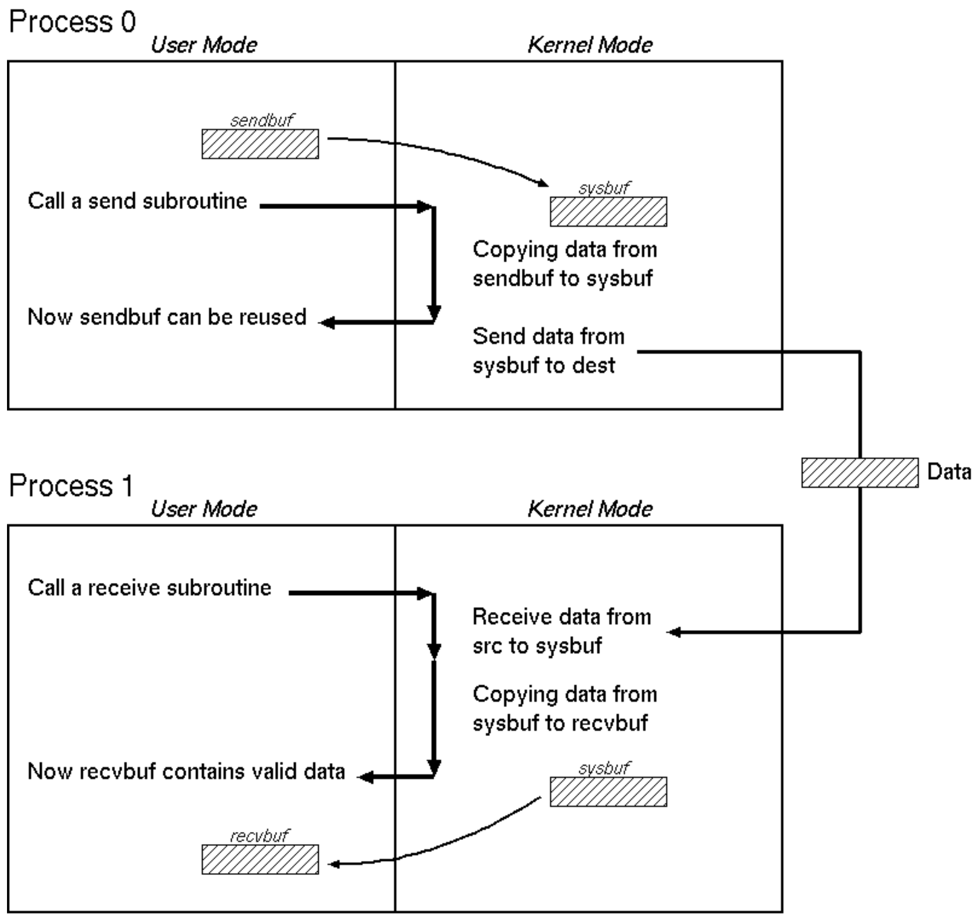

Process 0
User Mode
Kernel Mode
sendbuf
sysbuf
Call a send subroutine
Copying data from sendbuf to sysbuf
Now sendbuf can be reused
Send data from sysbuf to dest
Data
Process 1
User Mode
Kernel Mode
Call a receive subroutine
Receive data from src to sysbuf
Copying data from sysbuf to recvbuf
Now recvbuf contains valid data
sysbuf
recvbuf

Figure 16. Data Movement in the Point-to-Point Communication

As Figure 16 shows, when you send data, you cannot or should not reuse your buffer until the system copies data from user buffer to the system buffer. Also when you receive data, the data is not ready until the system completes copying data from a system buffer to a user buffer. In MPI, there are two modes of communication: blocking and non-blocking. When you use blocking communication subroutines such as MPI_SEND and MPI_RECV, the program will not return from the subroutine call until the copy to/from the system buffer has finished. On the other hand, when you use non-blocking communication subroutines such as MPI_ISEND and MPI_IRECV, the program immediately returns from the subroutine call. That is, a call to a non-blocking subroutine only indicates that the copy to/from the system buffer is initiated and it is not assured that the copy has completed. Therefore, you have to make sure of the completion of the copy by MPI_WAIT. If you use your buffer before the copy completes, incorrect data may be copied to the system buffer (in case of non-blocking send), or your buffer does not contain what you want (in case of non-blocking receive). For the usage of point-to-point subroutines, see B.3, “Point-to-Point Communication Subroutines” on page 189.

Why do you use non-blocking communication despite its complexity? Because non-blocking communication is generally faster than its corresponding blocking communication. Some hardware may have separate co-processors that are dedicated to communication. On such hardware, you may be able to hide the latency by computation. In other words, you can do other computations while the system is copying data back and forth between user and system buffers.

2.4.2 Unidirectional Communication

When you send a message from process 0 to process 1, there are four combinations of MPI subroutines to choose from depending on whether you use a blocking or non-blocking subroutine for sending or receiving data.

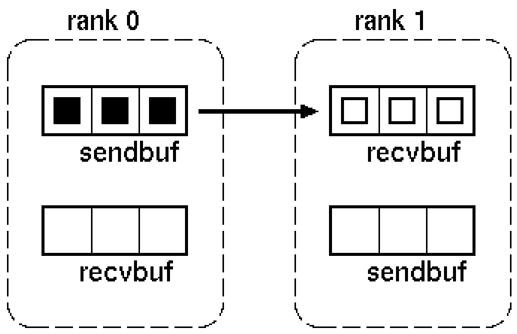


Figure 17. Point-to-Point Communication

Written explicitly, the four combinations are the following:

Blocking send and blocking receive

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus, ierr)
ENDIF
```

Non-blocking send and blocking receive

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus, ierr)
ENDIF
```

Blocking send and non-blocking receive

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_IRECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ENDIF
```

Non-blocking send and non-blocking receive

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_IRECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq, ierr)
ENDIF
CALL MPI_WAIT(ireq, istatus, ierr)
```

Note that you can place MPI_WAIT anywhere after the call of non-blocking subroutine and before reuse of the buffer.

2.4.3 Bidirectional Communication

When two processes need to exchange data with each other, you have to be careful about deadlocks. When a deadlock occurs, processes involved in the deadlock will not proceed any further. Deadlocks can take place either due to the incorrect order of send and receive, or due to the limited size of the system buffer. In Figure 18, process 0 and process 1 call the send subroutine once and the receive subroutine once.

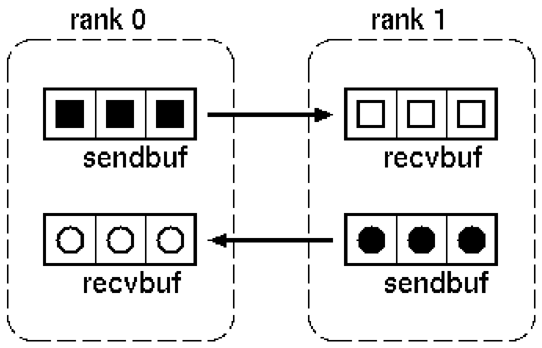


Figure 18. Duplex Point-to-Point Communication

There are essentially three cases depending on the order of send and receive subroutines called by both processes.

Case 1 Both processes call the send subroutine first, and then receive.

Case 2 Both processes call the receive subroutine first, and then send.

Case 3 One process calls send and receive subroutines in this order, and the other calls in the opposite order.

For each case, there are further options based on your use of blocking or non-blocking subroutines.

Case 1. Send first and then receive

Consider the following code:

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_RECV(recvbuf, ...)
ENDIF
```

Remember that the program returns from MPI_SEND when the copy from `sendbuf` to the system buffer has finished. As long as the system buffer is larger than `sendbuf`, the program ends normally. What if the system buffer is not large enough to hold all the data in `sendbuf`? Process 1 is supposed to receive data from process 0, and that part of process 0's system buffer that has been received by process 1 can be reused for other data. Then the uncopied data fills up this space. This cycle repeats until all the data in `sendbuf` of process 0 has copied to the system buffer. And only at that time, the program returns from MPI_SEND. In the previous program example, process 1 does the same thing as process 0: it waits for process 0 to receive the data.

However process 0 does not reach the MPI_RECV statement until process 1 receives data, which leads to a deadlock. Since MPI_ISEND immediately followed by MPI_WAIT is logically equivalent to MPI_SEND, the following code also gives rise to a deadlock if `sendbuf` is larger than the system buffer. The situation does not change if you use MPI_IRECV instead of MPI_RECV.

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_WAIT(ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_WAIT(ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
ENDIF
```

On the other hand, the following code is free from deadlock because the program immediately returns from MPI_ISEND and starts receiving data from the other process. In the meantime, data transmission is completed and the calls of MPI_WAIT for the completion of send at both processes do not lead to a deadlock.

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ENDIF
```

Case 2. Receive first and then send

The following code leads to a deadlock regardless of how much system buffer you have.

```
IF (myrank==0) THEN
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_SEND(sendbuf, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_SEND(sendbuf, ...)
ENDIF
```

If you use MPI_ISEND instead of MPI_SEND, deadlock still occurs. On the other hand, the following code can be safely executed.

```
IF (myrank==0) THEN
  CALL MPI_IRECV(recvbuf, ..., ireq, ...)
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_IRECV(recvbuf, ..., ireq, ...)
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ENDIF
```

Case 3. One process sends and receives; the other receives and sends

It is always safe to order the calls of MPI_(I)SEND and MPI_(I)RECV so that a send subroutine call at one process and a corresponding receive subroutine call at the other process appear in matching order.

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, ...)
  CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_SEND(sendbuf, ...)
ENDIF
```

In this case, you can use either blocking or non-blocking subroutines.

Considering the previous options, performance, and the avoidance of deadlocks, it is recommended to use the following code.

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq1, ...)
  CALL MPI_IRECV(recvbuf, ..., ireq2, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq1, ...)
  CALL MPI_IRECV(recvbuf, ..., ireq2, ...)
ENDIF
CALL MPI_WAIT(ireq1, ...)
CALL MPI_WAIT(ireq2, ...)
```

2.5 Derived Data Types

As will be discussed in 3.1, “What is Parallelization?” on page 41, if the total amount of data transmitted between two processes is the same, you should transmit it a fewer number of times. Suppose you want to send non-contiguous data to another process. For the purpose of a fewer number of data transmissions, you can first copy the non-contiguous data to a contiguous buffer, and then send it at one time. On the receiving process, you may have to unpack the data and copy it to proper locations. This procedure may look cumbersome, but, MPI provides mechanisms, called *derived data types*, to specify more general, mixed, and non-contiguous data. While it is convenient, the data transmissions using derived data types might result in lower performance than the manual coding of packing the data, transmitting, and unpacking. For this reason, when you use derived data types, be aware of the performance impact.

2.5.1 Basic Usage of Derived Data Types

Suppose you want to send array elements `a(4)`, `a(5)`, `a(7)`, `a(8)`, `a(10)` and `a(11)` to another process. If you define a derived data type `itype1` as shown in Figure 19 on page 29, just send one data of type `itype1` starting at `a(4)`. In this figure, empty slots mean that they are neglected in data transmission. Alternatively, you can send three data of type `itype2` starting at `a(4)`.

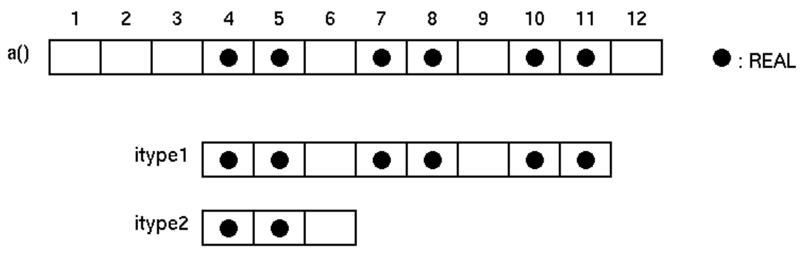


Figure 19. Non-Contiguous Data and Derived Data Types

If you are sending to process `idst`, the actual code will either look like:

```
CALL MPI_SEND(a(4), 1, itype1, idst, itag, MPI_COMM_WORLD, ierr)
```

or

```
CALL MPI_SEND(a(4), 3, itype2, idst, itag, MPI_COMM_WORLD, ierr)
```

Now, the construction of these complex data types is examined. (See also B.4, “Derived Data Types” on page 197.)

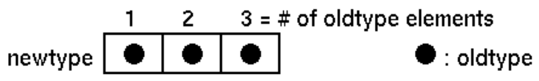


Figure 20. MPI_TYPE_CONTIGUOUS

First, MPI_TYPE_CONTIGUOUS is used to define a data type representing the contiguous occurrence of an existing data type, which can be either a derived data type or a basic MPI data type such as MPI_INTEGER or MPI_REAL.

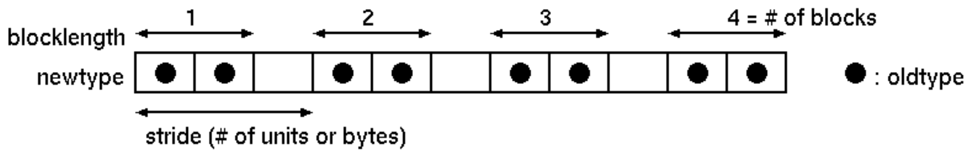


Figure 21. MPI_TYPE_VECTOR/MPI_TYPE_HVECTOR

By using MPI_TYPE_(H)VECTOR, you can repeat an existing data type by placing blanks in between.

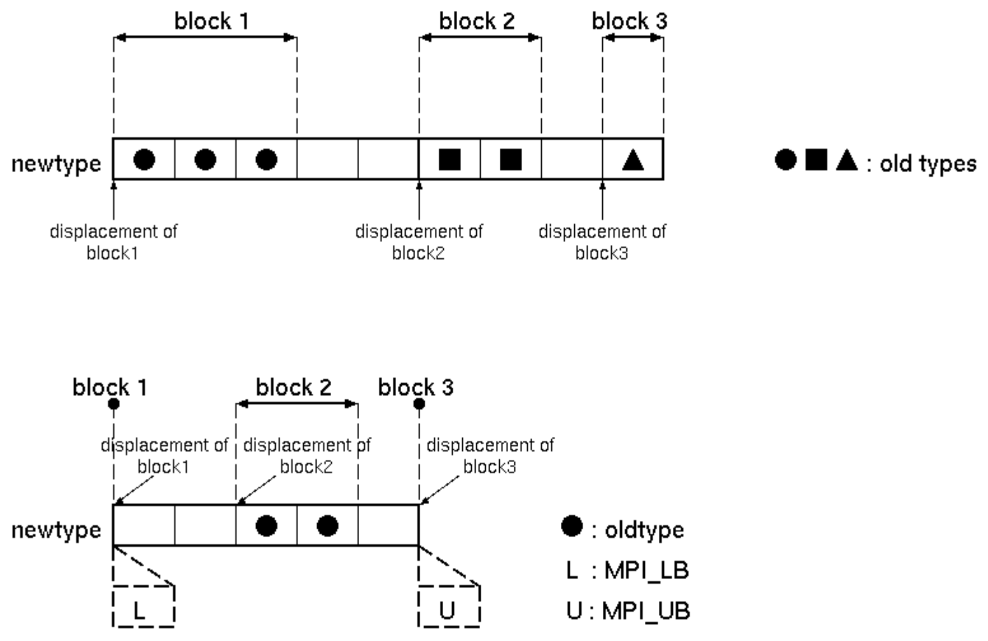


Figure 22. MPI_TYPE_STRUCT

MPI_TYPE_STRUCT allows you to combine multiple data types into one. When you want to put empty slots at the beginning or at the end of the new data type, put one object of MPI_LB or MPI_UB in calling MPI_TYPE_STRUCT. These pseudo data types occupy no space, that is, the size of MPI_LB and MPI_UB is zero.

After you have defined a new data type, register the new data type by MPI_TYPE_COMMIT.

2.5.2 Subroutines to Define Useful Derived Data Types

In parallelizing programs, you often need to send and receive elements of a submatrix that are not contiguous in memory. This section provides four utility subroutines to define the most typical derived data types that you might want to use in your programs. These subroutines are not part of the MPI library and are not supported by IBM.

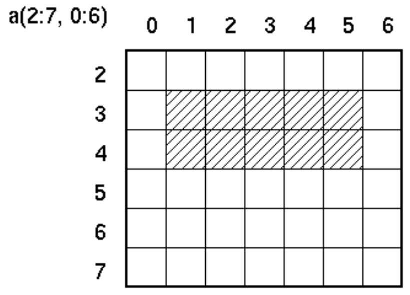


Figure 23. A Submatrix for Transmission

Suppose you want to send data in the shaded region in Figure 23 on page 30. By using the utility subroutine para_type_block2a or para_type_block2 described below, the code becomes very neat:

```
CALL para_type_block2a(2, 7, 2, 5, MPI_REAL, itype)
CALL MPI_SEND(a(3,1), 1, itype, idst, itag, MPI_COMM_WORLD, ierr)
```

or

```
CALL para_type_block2(2, 7, 0, 3, 4, 1, 5, MPI_REAL, itype2)
CALL MPI_SEND(a, 1, itype2, idst, itag, MPI_COMM_WORLD, ierr)
```

The meanings of parameters are clarified in the following sections. The source code for these subroutines is also included, as well as three dimensional versions. When you use the data type defined by para_type_block2a, specify the initial address of the submatrix, `a(3,1)`, as the address of the send buffer. On the other hand, in using para_type_block2, specify the initial address of the matrix, `a` or `a(2,0)`, as the address of the send buffer.

2.5.2.1 Utility Subroutine: para_type_block2a

Usage

```
CALL para_type_block2a(imin, imax, ilen, jlen, ioldtype,
                       inewtype)
```

Parameters

INTEGER imin	The minimum coordinate value of the first dimension
INTEGER imax	The maximum coordinate value of the first dimension
INTEGER ilen	The length of the rectangle in the first dimension
INTEGER jlen	The length of the rectangle in the second dimension
INTEGER ioldtype	The data type of the elements in the rectangular area
INTEGER inewtype	The newly defined derived data type

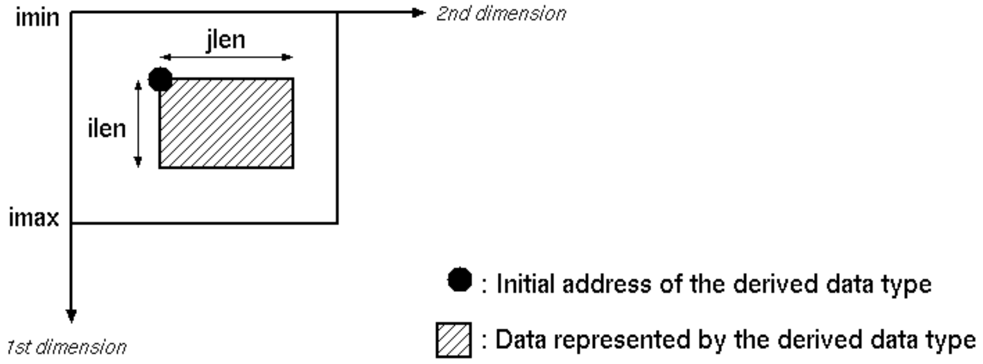

imin
jlen
2nd dimension
ilen
imax
1st dimension
● : Initial address of the derived data type
▨ : Data represented by the derived data type

Figure 24. Utility Subroutine para_type_block2a

Source code

```
      SUBROUTINE para_type_block2a(imin, imax, ilen, jlen,
     &                             ioldtype, inewtype)
      INCLUDE 'mpif.h'
      CALL MPI_TYPE_VECTOR(jlen, ilen, imax - imin + 1,
     &                      ioldtype, inewtype, ierr)
      CALL MPI_TYPE_COMMIT(inewtype, ierr)
      END
```

2.5.2.2 Utility Subroutine: para_type_block2

Usage

```
CALL para_type_block2(imin, imax, jmin, ista, iend, jsta, jend,
                      ioldtype, inewtype)
```

Parameters

INTEGER imin	The minimum coordinate value of the first dimension
INTEGER imax	The maximum coordinate value of the first dimension
INTEGER jmin	The minimum coordinate value of the second dimension
INTEGER ista	The minimum coordinate value of the rectangular area in the first dimension
INTEGER iend	The maximum coordinate value of the rectangular area in the first dimension
INTEGER jsta	The minimum coordinate value of the rectangular area in the second dimension
INTEGER jend	The maximum coordinate value of the rectangular area in the second dimension
INTEGER ioldtype	The data type of the elements in the rectangular area
INTEGER inewtype	The newly defined derived data type

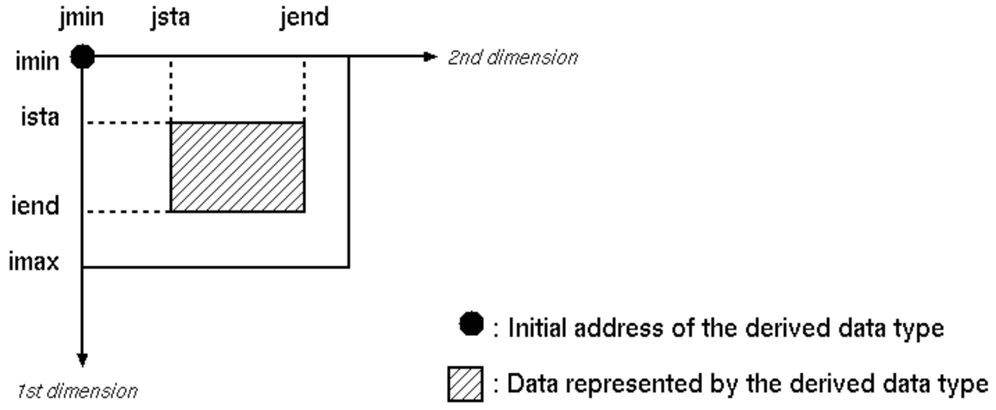

jmin
jsta
jend
imin
2nd dimension
ista
iend
imax
1st dimension
● : Initial address of the derived data type
▨ : Data represented by the derived data type

Figure 25. Utility Subroutine para_type_block2

Source code

```
      SUBROUTINE para_type_block2(imin, imax, jmin,
     &                            ista, iend, jsta, jend,
     &                            ioldtype, inewtype)
      INCLUDE 'mpif.h'
      INTEGER iblock(2), idisp(2), itype(2)
      CALL MPI_TYPE_EXTENT(ioldtype, isize, ierr)
      ilen = iend - ista + 1
      jlen = jend - jsta + 1
      CALL MPI_TYPE_VECTOR(jlen, ilen, imax - imin + 1,
     &                     ioldtype, itemp, ierr)
      iblock(1) = 1
      iblock(2) = 1
      idisp(1) = 0
      idisp(2) = ((imax-imin+1) * (jsta-jmin) + (ista-imin)) * isize
      itype(1) = MPI_LB
      itype(2) = itemp
      CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype, inewtype, ierr)
      CALL MPI_TYPE_COMMIT(inewtype, ierr)
      END
```

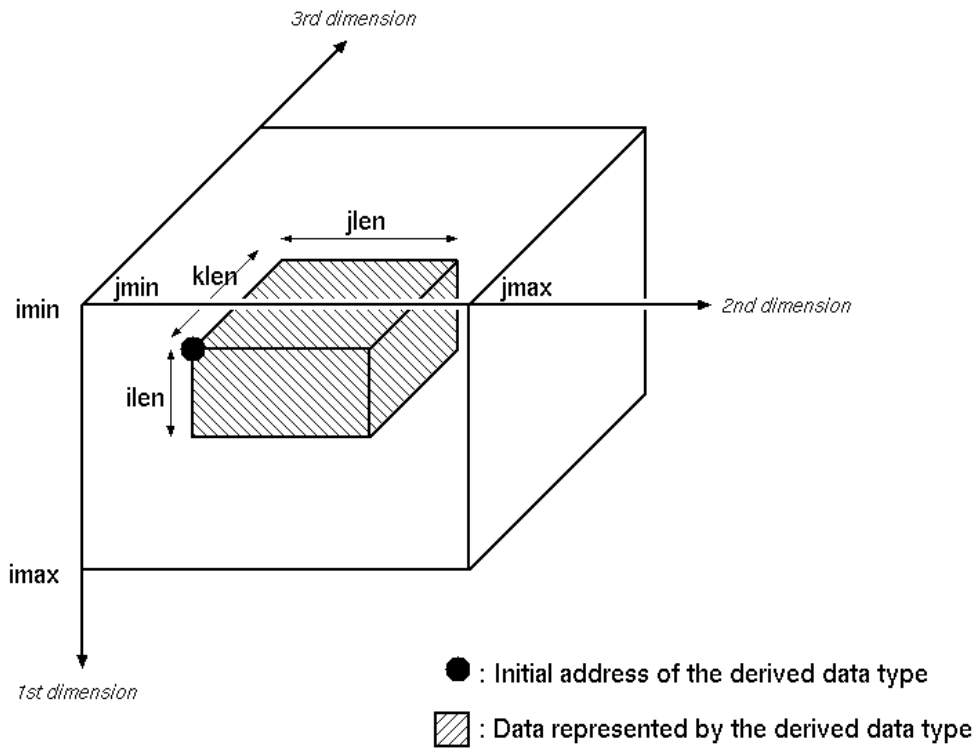
2.5.2.3 Utility Subroutine: para_type_block3a

Usage

```
CALL para_type_block3a(imin, imax, jmin, jmax, ilen, jlen,
                       klen, ioldtype, inewtype)
```

Parameters

INTEGER imin	The minimum coordinate value of the first dimension
INTEGER imax	The maximum coordinate value of the first dimension
INTEGER jmin	The minimum coordinate value of the second dimension
INTEGER jmax	The maximum coordinate value of the second dimension
INTEGER ilen	The length of the rectangular solid in the first dimension
INTEGER jlen	The length of the rectangular solid in the second dimension
INTEGER klen	The length of the rectangular solid in the third dimension
INTEGER ioldtype	The data type of the elements in the rectangular solid
INTEGER inewtype	The newly defined derived data type



3rd dimension
jlen
klen
jmin
imin
jmax
2nd dimension
ilen
imax
1st dimension
● : Initial address of the derived data type
▨ : Data represented by the derived data type

Figure 26. Utility Subroutine para_type_block3a

Source code

```
      SUBROUTINE para_type_block3a(imin, imax, jmin, jmax,
     &                             ilen, jlen, klen,
     &                             ioldtype,inewtype)
      INCLUDE 'mpif.h'
      CALL MPI_TYPE_EXTENT(ioldtype, isize, ierr)
      CALL MPI_TYPE_VECTOR(jlen, ilen, imax - imin + 1,
     &                     ioldtype, itemp, ierr)
      idist = (imax - imin + 1) * (jmax - jmin + 1) * isize
      CALL MPI_TYPE_HVECTOR(klen, 1, idist, itemp, inewtype, ierr)
      CALL MPI_TYPE_COMMIT(inewtype, ierr)
      END
```

2.5.2.4 Utility Subroutine: para_type_block3

Usage

```
CALL para_type_block3(imin, imax, jmin, jmax, kmin, ista, iend,
                      jsta, jend, ksta, kend, ioldtype, inewtype)
```

Parameters

INTEGER imin	The minimum coordinate value of the first dimension
INTEGER imax	The maximum coordinate value of the first dimension

INTEGER jmin	The minimum coordinate value of the second dimension
INTEGER jmax	The maximum coordinate value of the second dimension
INTEGER kmin	The minimum coordinate value of the third dimension
INTEGER ista	The minimum coordinate value of the rectangular solid in the first dimension
INTEGER iend	The maximum coordinate value of the rectangular solid in the first dimension
INTEGER jsta	The minimum coordinate value of the rectangular solid in the second dimension
INTEGER jend	The maximum coordinate value of the rectangular solid in the second dimension
INTEGER ksta	The minimum coordinate value of the rectangular solid in the third dimension
INTEGER kend	The maximum coordinate value of the rectangular solid in the third dimension
INTEGER ioldtype	The data type of the elements in the rectangular solid
INTEGER inewtype	The newly defined derived data type

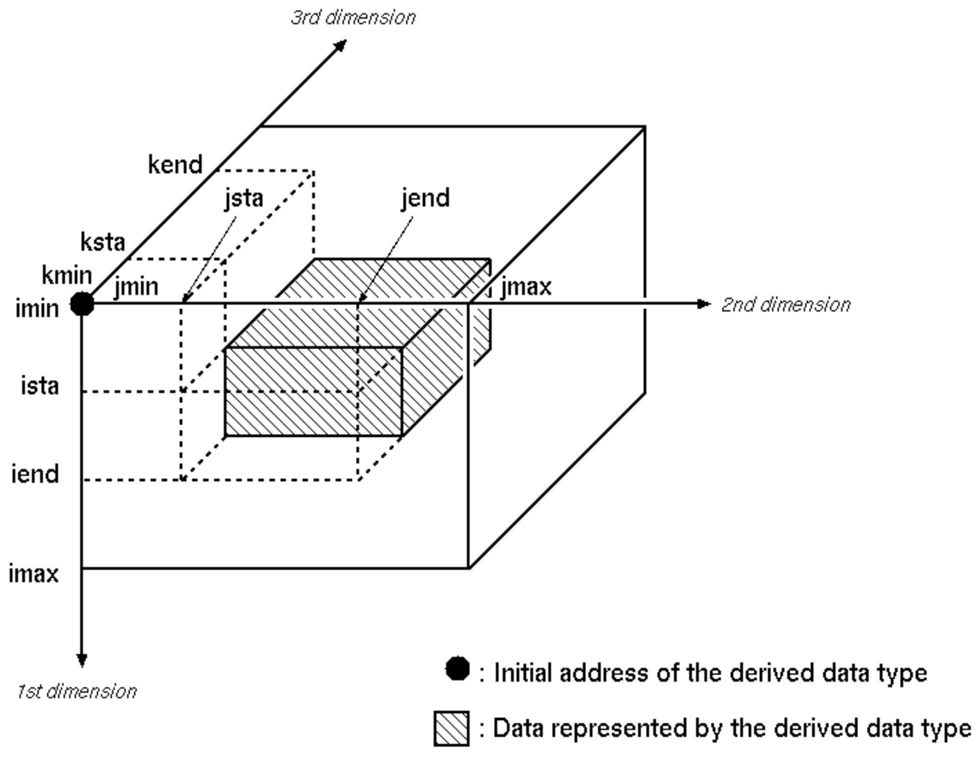

3rd dimension
kend
jsta
jend
ksta
kmin
jmin
jmax
imin
2nd dimension
ista
iend
imax
1st dimension
● : Initial address of the derived data type
▨ : Data represented by the derived data type

Figure 27. Utility Subroutine para_type_block3

Source code

```
      SUBROUTINE para_type_block3(imin, imax, jmin, jmax, kmin,
     &           ista, iend, jsta, jend, ksta, kend,
     &           ioldtype, inewtype)
      INCLUDE 'mpif.h'
      INTEGER iblock(2), idisp(2), itype(2)
      CALL MPI_TYPE_EXTENT(ioldtype, isize, ierr)
      ilen = iend - ista + 1
      jlen = jend - jsta + 1
      klen = kend - ksta + 1
      CALL MPI_TYPE_VECTOR(jlen, ilen, imax - imin + 1,
     &                     ioldtype, itemp, ierr)
      idist = (imax-imin+1) * (jmax-jmin+1) * isize
      CALL MPI_TYPE_HVECTOR(klen, 1, idist, itemp, itemp2, ierr)
      iblock(1) = 1
      iblock(2) = 1
      idisp(1) = 0
      idisp(2) = ((imax-imin+1) * (jmax-jmin+1) * (ksta-kmin)
     &          + (imax-imin+1) * (jsta-jmin) + (ista-imin)) * isize
      itype(1) = MPI_LB
      itype(2) = itemp2
      CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype, inewtype, ierr)
      CALL MPI_TYPE_COMMIT(inewtype, ierr)
      END
```

2.6 Managing Groups

In most of the MPI programs, all processes work together to solve a single problem. In other words, the communicator MPI_COMM_WORLD is often used in calling communication subroutines. However, it is possible that a problem consists of several subproblems and each problem can be solved concurrently. This type of application can be found in the category of coupled analysis. MPI allows you to create a new group as a subset of an existing group. Specifically, use MPI_COMM_SPLIT for this purpose.

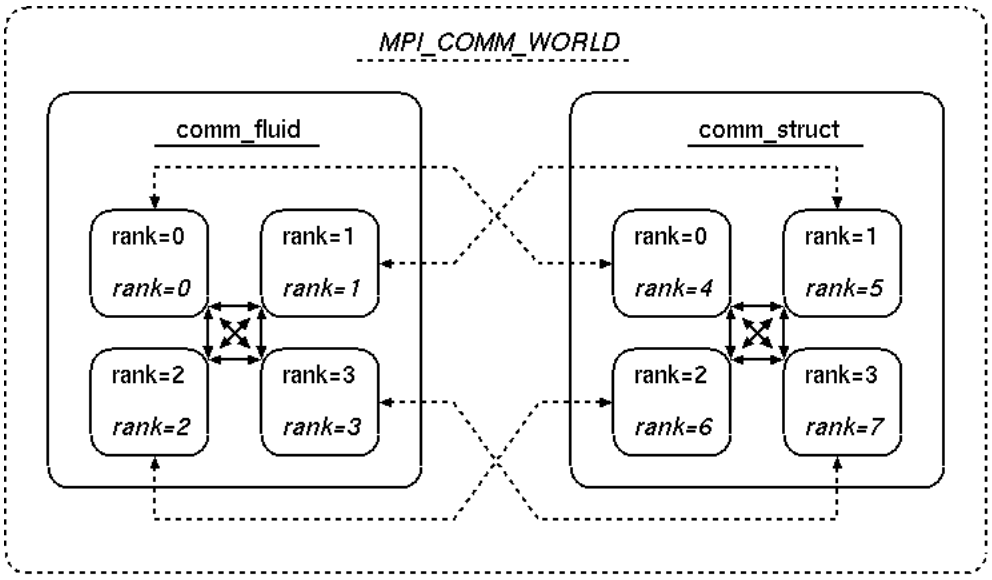


Figure 28. Multiple Communicators

Consider a problem with a fluid dynamics part and a structural analysis part, where each part can be computed in parallel most of the time. You can create new communicators for each part, say comm_fluid and comm_struct (see Figure

28). Processes participating in the fluid part or the structure part communicate with each other within the communicator comm_fluid or comm_struct, respectively (solid arrows in the figure). When both parts need to exchange results, processes on both sides can communicate in the communicator MPI_COMM_WORLD (the dashed arrows). In Figure 28 on page 36 ranks within MPI_COMM_WORLD are printed in italic, and ranks within comm_fluid or comm_struct are in boldface.

2.7 Writing MPI Programs in C

When you write an MPI program in the C language, there are some points you should be aware of.

- Include mpi.h instead of mpif.h.
- C is case-sensitive. All of the MPI functions have the form *MPI_Function*, where MPI and I, C, and so on are in upper-case as in MPI_Init, MPI_Comm_size. Constants defined in mpi.h are all in upper-case such as MPI_INT, MPI_SUM, and MPI_COMM_WORLD.
- Those arguments of an MPI function call that specify the address of a buffer have to be given as pointers.
- Return code of an MPI function call is given as an integer return value of that function.
- Data types in the C semantics are defined in a more individual way. The following is a partial list.

MPI_Status	Status object
MPI_Request	Request handle
MPI_Datatype	Data type handle
MPI_Op	Handle for reduction operation

  In Fortran, all the above objects and handles are just defined as an integer or an array of integers.

- Predefined MPI data types in C are different from Fortran bindings, as listed in Table 7.

Table 7. MPI Data Types (C Bindings)

MPI Data Types	Description (C Bindings)
MPI_CHAR	1-byte character
MPI_UNSIGNED_CHAR	1-byte unsigned character
MPI_SIGNED_CHAR	1-byte signed character
MPI_SHORT	2-byte integer
MPI_INT, MPI_LONG	4-byte integer
MPI_UNSIGNED_SHORT	2-byte unsigned integer
MPI_UNSIGNED, MPI_UNSIGNED_LONG	4-byte unsigned integer
MPI_FLOAT	4-byte floating point
MPI_DOUBLE, MPI_LONG_DOUBLE	8-byte floating point

MPI Data Types	Description (C Bindings)
MPI_UNSIGNED_LONG_LONG	8-byte unsigned integer
MPI_LONG_LONG_INT	8-byte integer
MPI_WCHAR	Wide (2-byte) unsigned character

Table 8 shows predefined combinations of reduction operations and data types in C language.

Table 8. Predefined Combinations of Operations and Data Types (C Language)

Operation	Data type
MPI_SUM (sum), MPI_PROD (product), MPI_MAX (maximum), MPI_MIN (minimum)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE
MPI_MAXLOC (max value and location), MPI_MINLOC (min value and location)	MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT, MPI_LONG_DOUBLE_INT
MPI_LAND (logical AND), MPI_LOR (logical OR), MPI_LXOR (logical XOR)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
MPI_BAND (bitwise AND), MPI_BOR (bitwise OR), MPI_BXOR (bitwise XOR)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_BYTE

Data types used for MPI_MAXLOC and MPI_MINLOC are essentially C structures as shown in Table 9.

Table 9. Data Types for Reduction Functions (C Language)

Data Type	Description (C structure)
MPI_FLOAT_INT	{MPI_FLOAT, MPI_INT}
MPI_DOUBLE_INT	{MPI_DOUBLE, MPI_INT}
MPI_LONG_INT	{MPI_LONG, MPI_INT}
MPI_2INT	{MPI_INT, MPI_INT}
MPI_SHORT_INT	{MPI_SHORT, MPI_INT}
MPI_LONG_DOUBLE_INT	{MPI_LONG_DOUBLE, MPI_INT}

The following is a sample C program.

sample.c

```
#include <mpi.h>

void main(int argc, char **argv)
{
    int         nprocs, myrank, tag, rc;
    float       sendbuf, recvbuf;
    MPI_Request req;
    MPI_Status  status;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    tag = 1;
    if (myrank == 0) {
        sendbuf = 9.0;
        rc = MPI_Send(&sendbuf, 1, MPI_FLOAT,
                      1, tag, MPI_COMM_WORLD);
    } else if (myrank == 1) {
        rc = MPI_Irecv(&recvbuf, 1, MPI_FLOAT,
                       0, tag, MPI_COMM_WORLD, &req);
        rc = MPI_Wait(&req, &status);
        printf("recvbuf = %f\n", recvbuf);
    }
    rc = MPI_Finalize();
}
```

The above program is compiled and executed as follows.

```
$ mpcc sample.c
$ ./a.out -procs 2
  1:recvbuf = 9.000000
```

Chapter 3. How to Parallelize Your Program

In Chapter 2, “Basic Concepts of MPI” on page 11, you learned the most fundamental MPI subroutines. In this chapter, you will see how these subroutines are used in parallelizing programs.

3.1 What is Parallelization?

You parallelize your program in order to run the program faster. How much faster will the parallel program run? Let us begin with Amdahl’s law. Suppose that in terms of running time, a fraction p of your program can be parallelized and that the remaining $1-p$ cannot be parallelized. In the ideal situation, if you execute the program using n processors, the parallel running time will be

$$1-p+p/n$$

of the serial running time. This is a direct consequence of Amdahl’s law applied to an ideal case of parallel execution. For example, if 80% of your program can be parallelized and you have four processors, the parallel running time will be $1-0.8+0.8/4 = 0.4$, that is, 40% of the serial running time as shown in Figure 29.

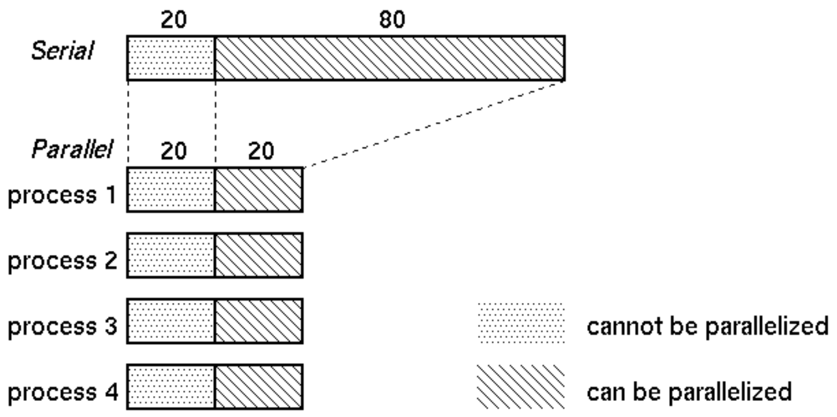


Figure 29. Parallel Speed-up: An Ideal Case

Because 20% of your program cannot be parallelized, you get only 2.5 times speed-up although you use four processors. For this program, the parallel running time is never shorter than 20% of the serial running time (five times speed-up) even if you have infinitely many processors. Amdahl’s law tells you that it is important to identify the fraction of your program that can be parallelized and to maximize it. Figure 30 on page 42 shows the upper bound of parallel speed-up ($1/(1-p)$) for various values of p.

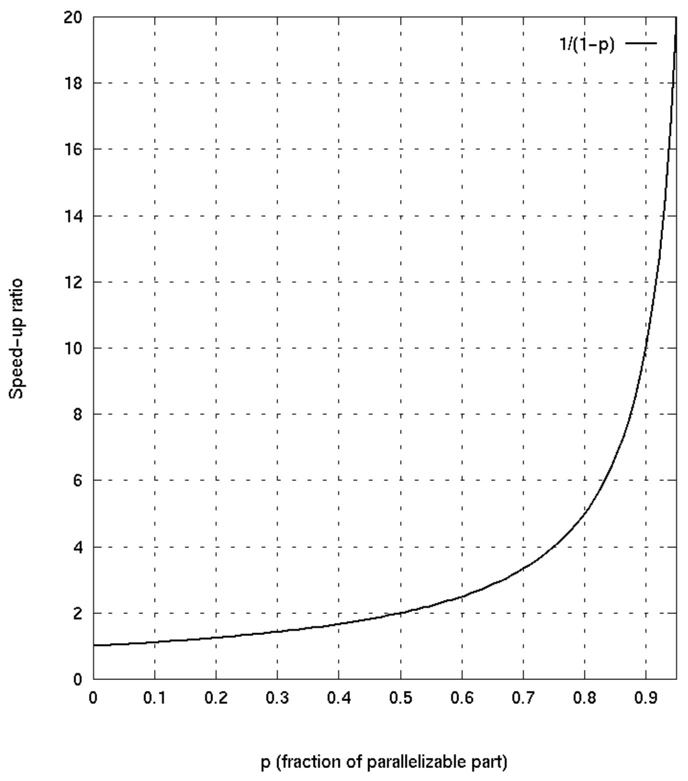


Figure 30. The Upper Bound of Parallel Speed-Up

The above argument is too simplified to be applied to real cases. When you run a parallel program, there is a communication overhead and a workload imbalance among processes in general. Therefore, the running time will be as Figure 31 shows.

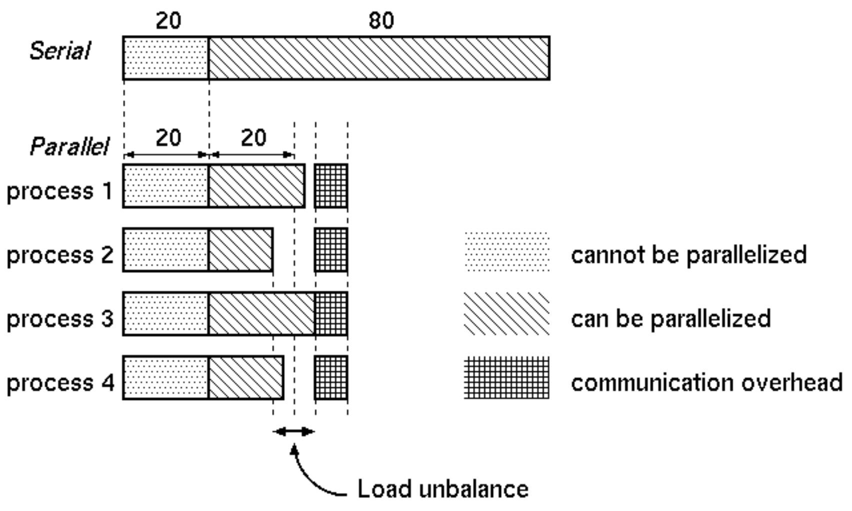


Figure 31. Parallel Speed-Up: An Actual Case

To summarize, you should follow these guidelines:

1. Increase the fraction of your program that can be parallelized.

2. Balance the workload of parallel processes.
3. Minimize the time spent for communication.

You may need to change the algorithm of your program in order to increase the parallelizable part. For example, an ordinary SOR (Successive Over-Relaxation) method has a tight dependence that prevents you from parallelizing it efficiently. But, if you rewrite the code to use the red-black SOR method, the code becomes well fit to be parallelized efficiently. See 4.4, “SOR Method” on page 120 for details.

What can you do to balance the workload of processes? Several measures can be taken according to the nature of the program. Changing the distribution of matrices from block distribution to cyclic distribution is one of them. See 3.4.1, “Block Distribution” on page 54 and 3.4.2, “Cyclic Distribution” on page 56 for examples.

The communication time is expressed as follows:

$$\text{Communication time} = \text{Latency} + \frac{\text{Message size}}{\text{Bandwidth}}$$

The latency is the sum of sender overhead, receiver overhead, and time of flight, which is the time for the first bit of the message to arrive at the receiver. This formula is illustrated in Figure 32, where the inverse of the gradient of the line gives the bandwidth.

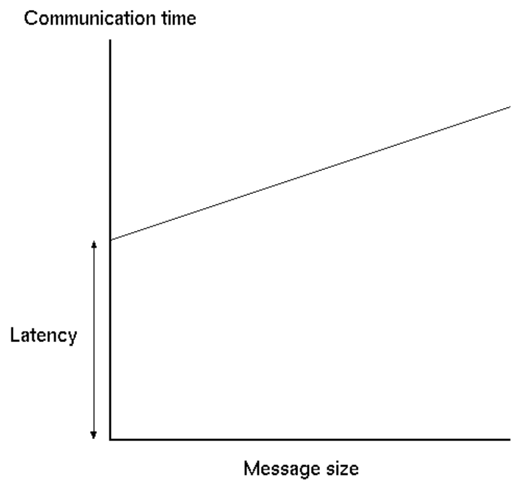


Figure 32. The Communication Time

Using this formula, the effective bandwidth is calculated as follows:

$$\text{Effective bandwidth} = \frac{\text{Message size}}{\text{Communication time}} = \frac{\text{Bandwidth}}{1 + \frac{\text{Latency} \cdot \text{Bandwidth}}{\text{Message size}}}$$

The effective bandwidth approaches the network bandwidth when the message size grows toward infinity. It is clear that the larger the message is, the more efficient the communication becomes.

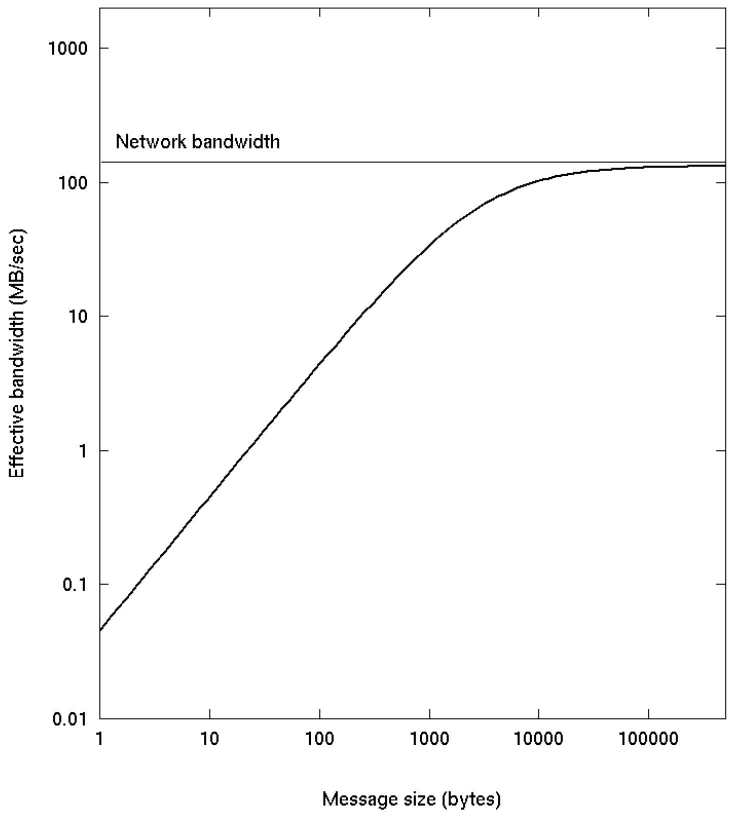


Figure 33. The Effective Bandwidth

Figure 33 is a plot of the effective bandwidth for a network with 22μsec latency and 133 MB/sec bandwidth, which corresponds to User Space protocol over SP switch network using POWER3 nodes. If the message size is greater than 100 KB, the effective bandwidth is close to the network bandwidth.

The following two strategies show how you can decrease the time spent for communication.

Strategy 1. Decrease the amount of data transmitted

Suppose that in a program employing the finite difference method, a matrix is divided into three chunks in block distribution fashion and they are processed separately by three processes in parallel.

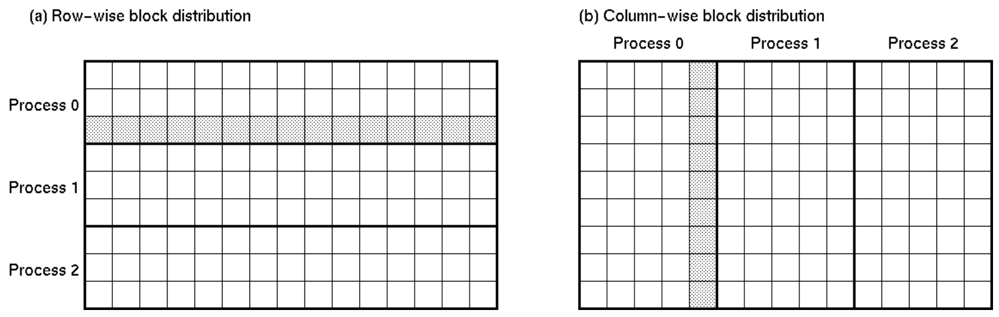

(a) Row-wise block distribution
Process 0
Process 1
Process 2
(b) Column-wise block distribution
Process 0
Process 1
Process 2

Figure 34. Row-Wise and Column-Wise Block Distributions

Figure 34 shows two ways of distributing the matrix: row-wise and column-wise block distributions. In the finite difference method, each matrix element depends on the value of its neighboring elements. So, process 0 has to send the data in the shaded region to process 1 as the computation proceeds. If the computational time is the same for both distributions, you should use the column-wise block distribution in order to minimize the communication time because the submatrix of the column-wise distribution has a smaller intersection than that of the row-wise distribution in this case.

Strategy 2. Decrease the number of times that data is transmitted

Suppose that in Figure 35, process 0 has to send the matrix elements in the shaded region to process 1. In Fortran, multi-dimensional arrays are stored in column-major order, that is, the array `a(i,j)` is stored in the order of `a(1,1)`, `a(2,1)`, `a(3,1)`,..., `a(N,1)`, `a(2,1)`, `a(2,2)`, `a(3,2)`,..., `a(N,N)`. Therefore the matrix elements that process 0 is going to send to process 1 (`a(4,1)`, `a(4,2)`, `a(4,3)`,..., `a(4,N)`) are not contiguous in memory.

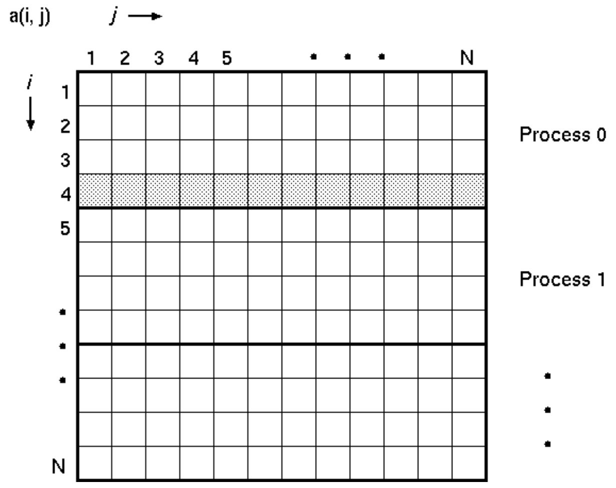


Figure 35. Non-Contiguous Boundary Elements in a Matrix

If you call an MPI subroutine N times separately for each matrix element, the communication overhead will be unacceptably large. Instead, you should first copy the matrix elements to a contiguous memory location, and then call the MPI subroutine once. Generally, the time needed for copying the data is much smaller than the communication latency. Alternatively, you can define a derived data type to let MPI pack the elements but it may not be optimal in terms of performance. See 2.5, “Derived Data Types” on page 28 for details.

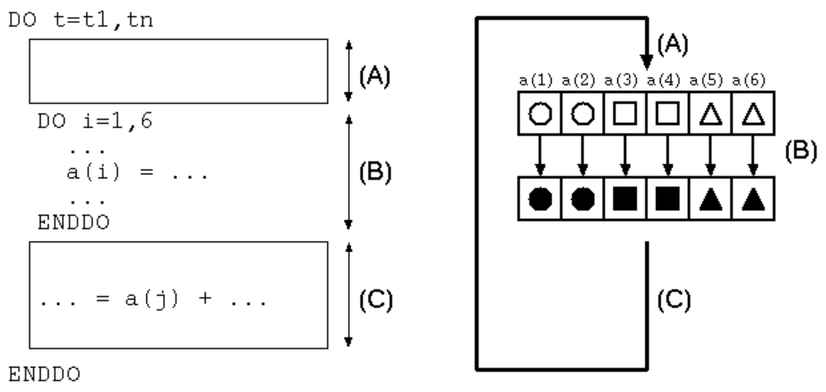
3.2 Three Patterns of Parallelization

For symmetric multiprocessor (SMP) machines, there are compilers that can parallelize programs automatically or with the aid of compiler directives given by the user. The executables generated by such compilers run in parallel using multiple threads, and those threads can communicate with each other by use of shared address space without explicit message-passing statements. When you use the automatic parallelization facility of a compiler, you usually do not have to worry about how and which part of the program is parallelized. The downside of the automatic parallelization is that your control over parallelization is limited. On the other hand, think about parallelizing your program using MPI and running it on massively parallel processors (MPP) such as RS/6000 SP or clusters of RS/6000 workstations. In this case, you have complete freedom about how and where to parallelize, where there is parallelism at all. But it is *you* who has to decide how to parallelize your program and add some code that explicitly transmits messages between the processes. You are responsible for the parallelized program to run correctly. Whether the parallelized program performs well or not often depends on your decision about how and which part of the program to parallelize.

Although they are not complete, the following three patterns show typical ways of parallelizing your code from a global point of view.

Pattern 1. Partial Parallelization of a DO Loop

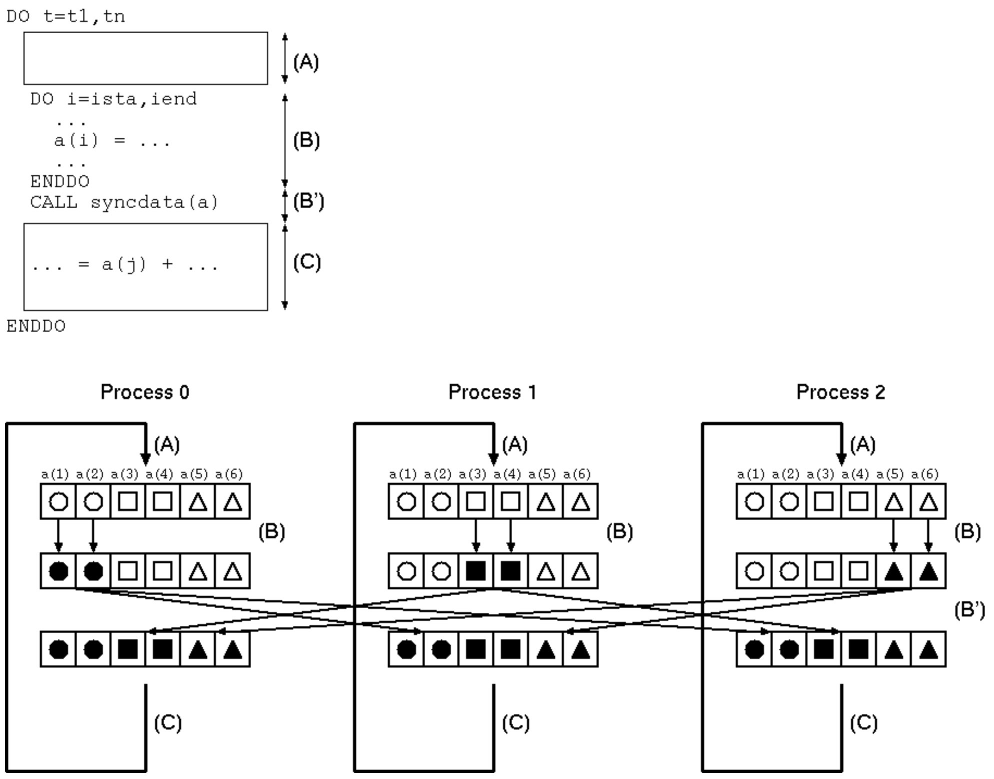
In some programs, most of the CPU time is consumed by a very small part of the code. Figure 36 shows a code with an enclosing DO loop of `t` that ticks time steps. Suppose that the inner DO loop (B) spends most of the CPU time and the parts (A) and (C) contain a large number of lines but do not spend much time. Therefore, you don’t get much benefit from parallelizing (A) and (C). It is reasonable to parallelize only the inner DO loop (B). However, you should be careful about the array `a()`, because it is updated in (B) and is referenced in (C). In this figure, black objects (circles, squares, and triangles) indicate that they have updated values.



```
DO t=t1,tn

  DO i=1,6
     ...
     a(i) = ...
     ...
  ENDDO

  ... = a(j) + ...

ENDDO
```

Figure 36. Pattern 1: Serial Program

Figure 37 shows how the code is parallelized and executed by three processes. The iterations of the inner DO loop are distributed among processes and you can expect parallel speed-up from here. Since the array `a()` is referenced in part (C), the updated values of `a()` are distributed by `syncdata` in (B’). Section 3.5.3.2, “Synchronizing Data” on page 73 gives you the implementation of `syncdata`. Note that you don’t have to care about which part of the array `a()` is used by which process later, because after (B’) every process has up-to-date values of `a()`. On the other hand, you may be doing more data transmissions than necessary.



```
DO t=t1,tn

  DO i=ista,iend
    ...
    a(i) = ...
    ...
  ENDDO
  CALL syncdata(a)

 ... = a(j) + ...

ENDDO
```

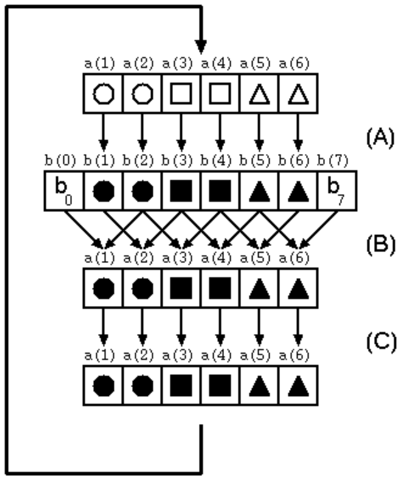
Figure 37. Pattern 1: Parallelized Program

By using this method, you can minimize the workload of parallelization. In order for this method to be effective, however, the inner DO loop should account for a considerable portion of the running time and the communication overhead due to `syncdata` should be negligible.

Pattern 2. Parallelization Throughout a DO Loop

In programs using the finite difference method, you often see that within the outer DO loop that is ticking time steps, there are several DO loops that almost equally contribute to the total running time, as shown in Figure 38 on page 48. If you are to synchronize data among processes every time after each DO loop, the communication overhead might negate the benefit of parallelization. In this case, you need to parallelize all the inner DO loops and minimize the amount of messages exchanged in order to get a reasonable parallel speed-up.

```
DO t=t1,tn
  DO i=1,6
    b(i) = b(i) + a(i)          (A)
  ENDDO
  DO i=1,6
    a(i) = b(i-1) + b(i+1)      (B)
  ENDDO
  DO i=1,6
    a(i) = a(i) + 1.0           (C)
  ENDDO
ENDDO
```

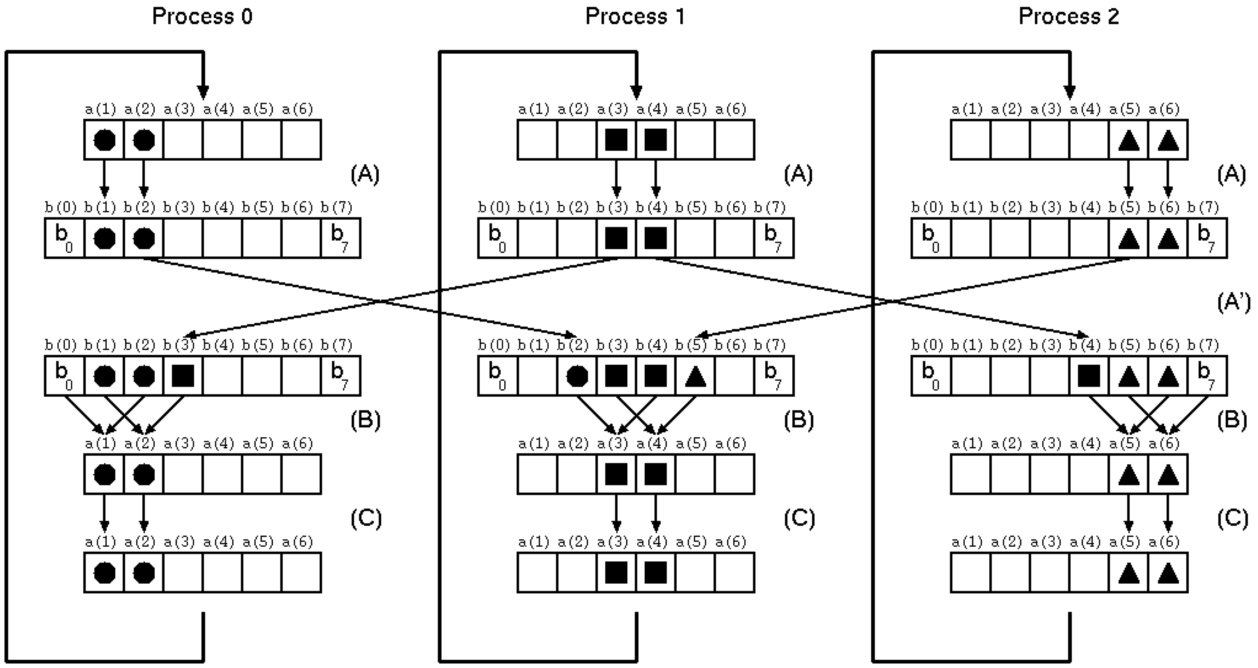


a(1) a(2) a(3) a(4) a(5) a(6)
(A)
b(0) b(1) b(2) b(3) b(4) b(5) b(6) b(7)
b_0
b_7
(B)
a(1) a(2) a(3) a(4) a(5) a(6)
(C)
a(1) a(2) a(3) a(4) a(5) a(6)

Figure 38. Pattern 2: Serial Program

In the parallelized program in Figure 39 on page 49, the iterations of DO loops are distributed among processes. That is, in each DO loop, a process executes the statements only for its assigned range of the iteration variable ($ista \leq i \leq iend$). Each process does not need to know the values of arrays `a()` and `b()` outside this range except for loop (B) where adjacent values of array `b()` are necessary to compute `a()`. So, it is necessary and sufficient to exchange data of the boundary element with neighboring processes after loop (A). The subroutine `shift` is assumed to do the job. The implementation of `shift` is shown in 3.5.2, “One-Dimensional Finite Difference Method” on page 67. In this program, the values of `b(0)` and `b(7)` are fixed and not updated.

```
DO t=t1,tn
  DO i=ista,iend
    b(i) = b(i) + a(i)          (A)
  ENDDO
  CALL shift(b)                 (A')
  DO i=ista,iend
    a(i) = b(i-1) + b(i+1)      (B)
  ENDDO
  DO i=ista,iend
    a(i) = a(i) + 1.0           (C)
  ENDDO
ENDDO
```



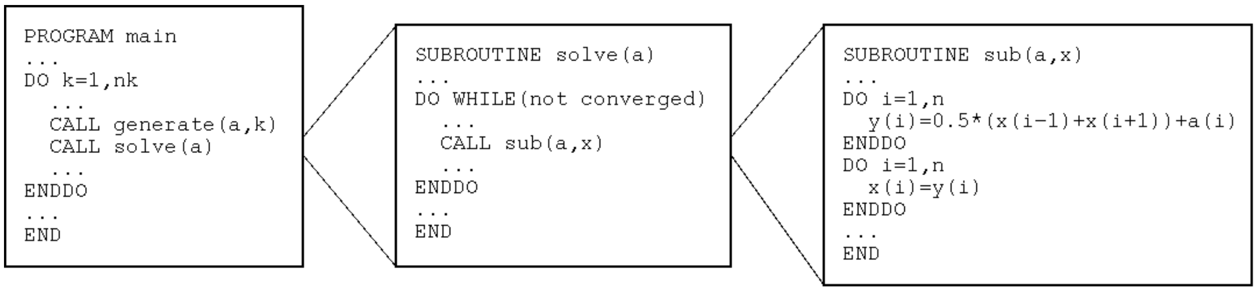
Process 0
Process 1
Process 2
a(1) a(2) a(3) a(4) a(5) a(6)
b(0) b(1) b(2) b(3) b(4) b(5) b(6) b(7)
(A)
(A')
(B)
(C)

Figure 39. Pattern 2: Parallel Program

Although the workload of rewriting the code is large compared with Pattern 1, you will get the desired speed-up.

Pattern 3. Coarse-Grained versus Fine-Grained Parallelization

A program sometimes has parallelism at several depth levels of the scoping unit. Figure 40 on page 50 shows a program, which calls subroutine `solve` for an independent set of input array `a()`. Suppose that subroutine `sub` is the hot spot of this program. This program has parallelism in DO loops in the main program and in subroutine `sub`, but not in subroutine `solve`. Whether you parallelize the program in the main program or in subroutine `sub` is a matter of granularity of parallelization.



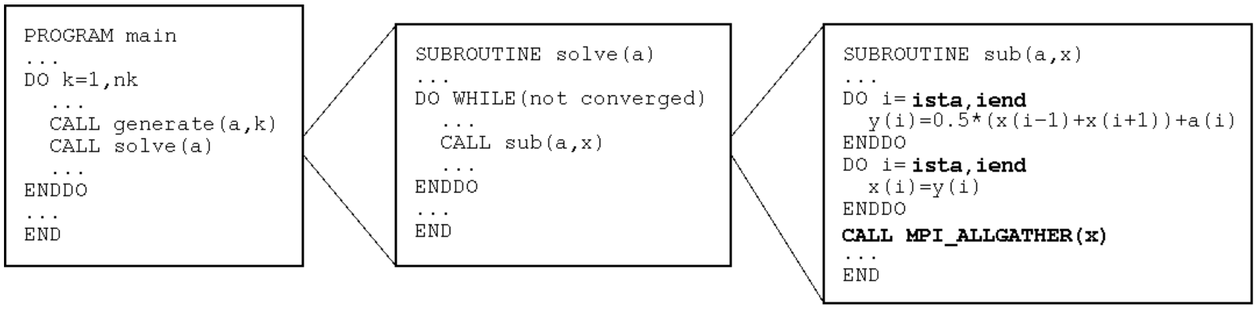
```
PROGRAM main
...
DO k=1,nk
  ...
  CALL generate(a,k)
  CALL solve(a)
  ...
ENDDO
...
END
```

```
SUBROUTINE solve(a)
...
DO WHILE(not converged)
  ...
  CALL sub(a,x)
  ...
ENDDO
...
END
```

```
SUBROUTINE sub(a,x)
...
DO i=1,n
  y(i)=0.5*(x(i-1)+x(i+1))+a(i)
ENDDO
DO i=1,n
  x(i)=y(i)
ENDDO
...
END
```

Figure 40. Pattern 3: Serial Program

If you parallelize subroutine `sub` as in Figure 41, you need to add extra code (MPI_ALLGATHER) to keep consistency and to rewrite the range of iteration variables in all the DO loops in `sub`. But the workload of each process will be fairly balanced because of the fine-grained parallelization.



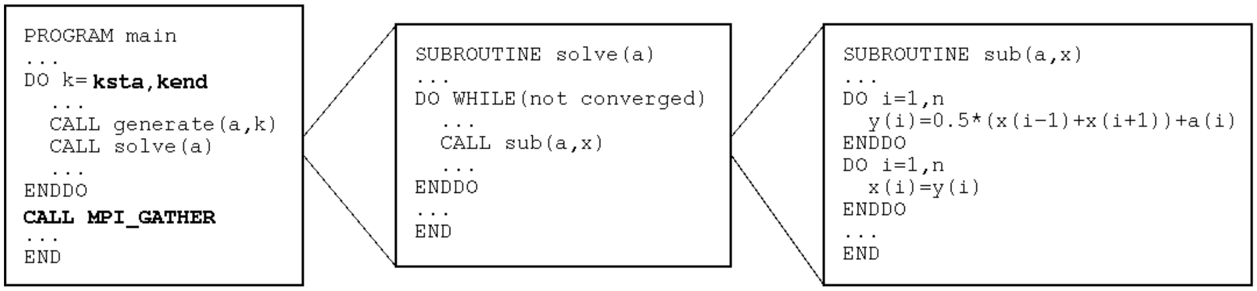
```
PROGRAM main
...
DO k=1,nk
  ...
  CALL generate(a,k)
  CALL solve(a)
  ...
ENDDO
...
END
```

```
SUBROUTINE solve(a)
...
DO WHILE(not converged)
  ...
  CALL sub(a,x)
  ...
ENDDO
...
END
```

```
SUBROUTINE sub(a,x)
...
DO i= ista,iend
  y(i)=0.5*(x(i-1)+x(i+1))+a(i)
ENDDO
DO i= ista,iend
  x(i)=y(i)
ENDDO
CALL MPI_ALLGATHER(x)
...
END
```

Figure 41. Pattern 3: Parallelized at the Innermost Level

On the other hand, if you parallelize the DO loop in the main program as in Figure 42, less statements need to be rewritten. However, since the work is distributed to processes in coarse-grained fashion, there might be more load unbalance between processes. For example, the number of iterations needed for the solution to converge in `solve` may vary considerably from problem to problem.



```
PROGRAM main
...
DO k= ksta,kend
  ...
  CALL generate(a,k)
  CALL solve(a)
  ...
ENDDO
CALL MPI_GATHER
...
END
```

```
SUBROUTINE solve(a)
...
DO WHILE(not converged)
  ...
  CALL sub(a,x)
  ...
ENDDO
...
END
```

```
SUBROUTINE sub(a,x)
...
DO i=1,n
  y(i)=0.5*(x(i-1)+x(i+1))+a(i)
ENDDO
DO i=1,n
  x(i)=y(i)
ENDDO
...
END
```

Figure 42. Pattern 3: Parallelized at the Outermost Level

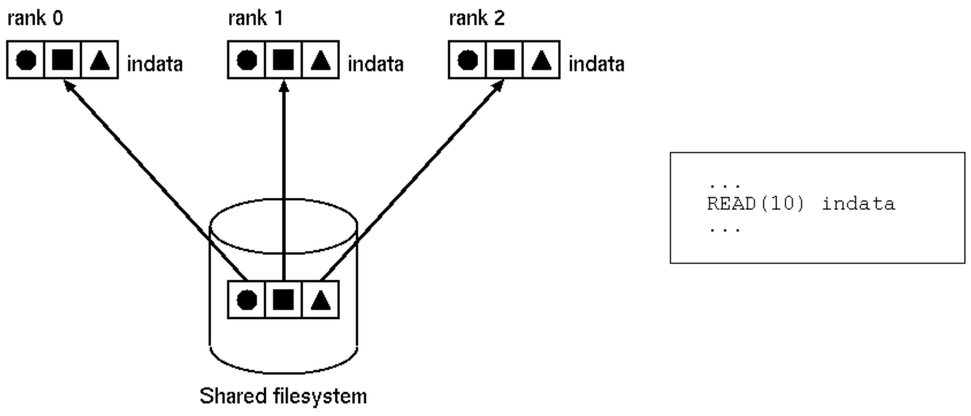
Generally, it is recommended to adopt coarse-grained parallelization if possible, as long as the drawback due to the load imbalance is negligible.

3.3 Parallelizing I/O Blocks

This section describes typical method used to parallelize a piece of code containing I/O operations. For better performance, you may have to prepare files and the underlying file systems appropriately.

Input 1. All the processes read the input file on a shared file system

The input file is located on a shared file system and each process reads data from the same file. For example, if the file system is an NFS, it should be mounted across a high speed network, but, even so, there will be I/O contention among reading processes. If you use GPFS (General Parallel File System), you might distribute the I/O across underlying GPFS server nodes. Note that unless you modify the code, the input file has to be accessed by the processes with the same path name.

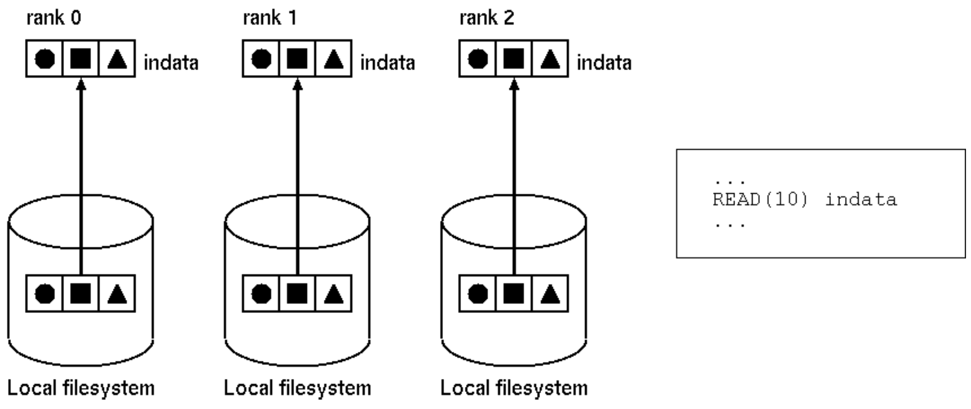


```
...
READ(10) indata
...
```

Figure 43. The Input File on a Shared File System

Input 2. Each process has a local copy of the input file

Before running the program, copy the input file to each node so that the parallel processes can read the file locally. This method gives you better performance than reading from a shared file system, at the cost of more disk space used and the additional work of copying the file.

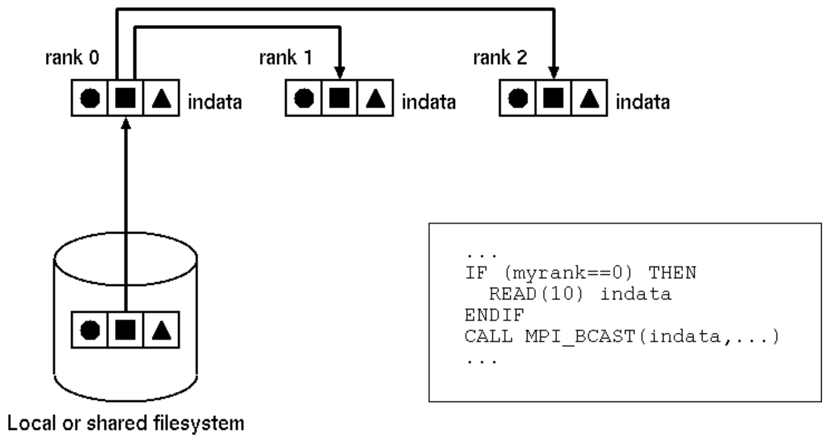


```
...
READ(10) indata
...
```

Figure 44. The Input File Copied to Each Node

Input 3. One process reads the input file and distributes it to the other processes

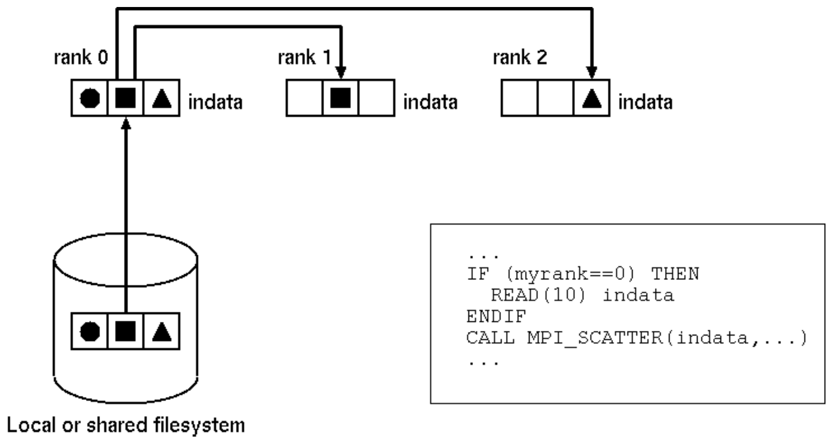
The input file is read by one process, and that process distributes the data to the other processes by using MPI_BCAST, for instance.



```
...
IF (myrank==0) THEN
  READ(10) indata
ENDIF
CALL MPI_BCAST(indata,...)
...
```

Figure 45. The Input File Read and Distributed by One Process

In all three cases of parallelized input, you can modify the code so that each process reads (or receives) the minimum amount of data that is necessary, as shown in Figure 46, for example.



```
...
IF (myrank==0) THEN
  READ(10) indata
ENDIF
CALL MPI_SCATTER(indata,...)
...
```

Figure 46. Only the Necessary Part of the Input Data is Distributed

In the program, MPI_SCATTER is called instead of MPI_BCAST.

Output 1. Standard output

In Parallel Environment for AIX, standard output messages of all the processes are displayed by default at the terminal which started the parallel process. You can modify the code:

```
...
PRINT *, '=== Job started ==='
...
```

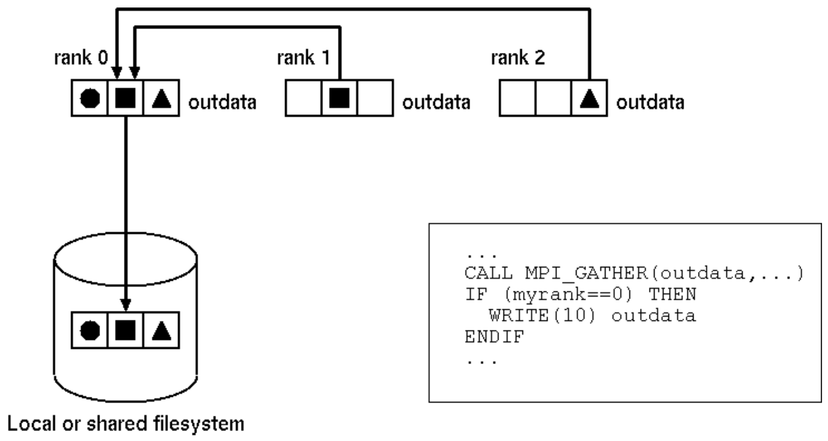
as

```
...
IF (myrank == 0) THEN
  PRINT *, '=== Job started ==='
ENDIF
...
```

so that only process 0 will write to the standard output. Alternatively, you can set the environment variable MP_STDOUTMODE as 0 to get the same effect. See A.5, "Standard Output and Standard Error" on page 158 for the usage of MPI_STDOUTMODE.

Output 2. One process gathers data and writes it to a local file

First, one of the processes gathers data from the other processes. Then that process writes the data to a file.



```
...
CALL MPI_GATHER(outdata,...)
IF (myrank==0) THEN
  WRITE(10) outdata
ENDIF
...
```

Figure 47. One Process Gathers Data and Writes It to a Local File

Output 3. Each process writes its data sequentially to a file on a shared file system

The output file is on a shared file system. If each process writes its portion of data individually and simultaneously to the same file, the contents of the file may be corrupted. The data have to be written to the file sequentially, process by process.

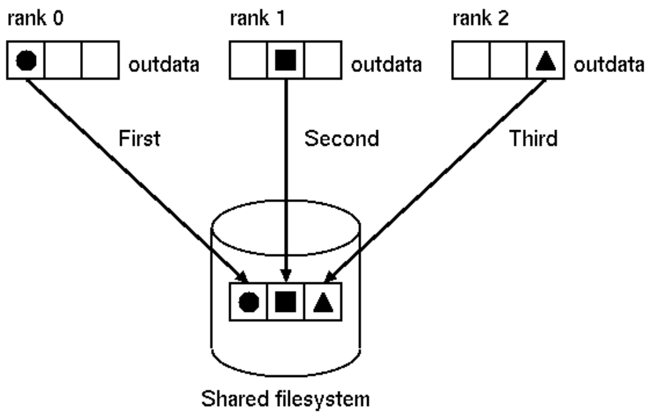


Figure 48. Sequential Write to a Shared File

The outline of the program is as follows:

```
...
DO irank = 0, nprocs - 1
  CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)
  IF (irank == myrank) THEN
    IF (myrank == 0) THEN
      OPEN(10, FILE='output')
    ELSE
      OPEN(10, FILE='output', POSITION='APPEND')
    ENDIF
    WRITE(10) (outdata(i), i=ista,iend)
    CLOSE(10)
  ENDIF
ENDDO
...
```

The subroutine MPI_BARRIER is used to synchronize processes. The range of array `output()` held by each process is assumed to be `ista..iend`.

3.4 Parallelizing DO Loops

In almost all of the scientific and technical programs, the hot spots are likely to be found in DO loops. Thus parallelizing DO loops is one of the most important challenges when you parallelize your program. The basic technique of parallelizing DO loops is to distribute iterations among processes and to let each process do its portion in parallel. Usually, the computations within a DO loop involves arrays whose indices are associated with the loop variable. Therefore distributing iterations can often be regarded as dividing arrays and assigning chunks (and computations associated with them) to processes.

3.4.1 Block Distribution

In block distribution, the iterations are divided into *p* parts, where *p* is the number of processes to be executed in parallel. The iterations in each part is consecutive in terms of the loop variable. For example, if four processes will execute a DO loop of 100 iterations, process 0 executes iterations 1-25, process 1 does 26-50, process 2 does 51-75, and process 3 does 76-100. If the number of iterations, *n*, is not divisible by the number of processes, *p*, there are several ways you can adopt in distributing *n* iterations to *p* processes. Suppose when you divide *n* by *p*, the quotient is *q* and the remainder is *r*, that is, $n = p \times q + r$. For example, in the case of $n = 14$ and $p = 4$, *q* is 3 and *r* is 2. One way to distribute iterations in such cases is as follows.

Processes `0..r-1` are assigned $q + 1$ iterations each.

The other processes are assigned *q* iterations.

This distribution corresponds to expressing *n* as $n = r(q+1) + (p-r)q$.

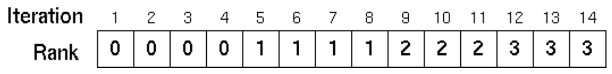


Figure 49. Block Distribution

Figure 49 depicts how 14 iterations are block-distributed to four processes using the above method. In parallelizing a program, it is convenient to provide a utility subroutine that calculates the range of iterations of a particular process. Of course, this is not an MPI subroutine.

Usage

```
CALL para_range(n1, n2, nprocs, irank, ista, iend)
```

Parameters

INTEGER n1	The lowest value of the iteration variable (IN)
INTEGER n2	The highest value of the iteration variable (IN)
INTEGER nprocs	The number of processes (IN)
INTEGER irank	The rank for which you want to know the range of iterations (IN)
INTEGER ista	The lowest value of the iteration variable that processes irank executes (OUT)
INTEGER iend	The highest value of the iteration variable that processes irank executes (OUT)

The following is the actual code.

```
      SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
      iwork1 = (n2 - n1 + 1) / nprocs
      iwork2 = MOD(n2 - n1 + 1, nprocs)
      ista = irank * iwork1 + n1 + MIN(irank, iwork2)
      iend = ista + iwork1 - 1
      IF (iwork2 > irank) iend = iend + 1
      END
```

Obviously, other ways of block distribution are also possible. The most important one among them is the way used in Parallel Engineering and Scientific Subroutine Library for AIX. For details, see *“Distributing Your Data”* in *Parallel Engineering and Scientific Subroutine Library for AIX Guide and Reference,* SA22-7273.

```
      SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
      iwork = (n2 - n1) / nprocs + 1
      ista = MIN(irank * iwork + n1, n2 + 1)
      iend = MIN(ista + iwork - 1, n2)
      END
```

This subroutine distributes 14 iterations to four processes, as shown in Figure 50, which is slightly different from Figure 49.

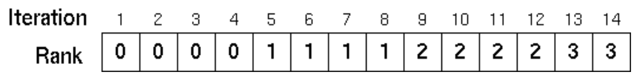


Figure 50. Another Block Distribution

Note that in this method, it may happen that some processes are assigned no iterations at all. Since MPI subroutines return errors when you specify negative integers as the number of elements to send or receive, special care is taken in calculating `ista` so that the number of elements held by a process is always given by `iend - ista + 1`.

Take a look at the following example. The original serial program computes the sum of the elements of an array `a()`.

```
PROGRAM main
PARAMETER (n = 1000)
DIMENSION a(n)
DO i = 1, n
  a(i) = i
ENDDO
sum = 0.0
DO i = 1, n
  sum = sum + a(i)
ENDDO
PRINT *,'sum =',sum
END
```

The parallelized program divides the iterations in the block distribution manner, and each process computes the partial sum of the array in the range calculated by the subroutine para_range. The partial sums are finally added together by the subroutine MPI_REDUCE.

```
 PROGRAM main
 INCLUDE 'mpif.h'
 PARAMETER (n = 1000)
 DIMENSION a(n)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 CALL para_range(1, n, nprocs, myrank, ista, iend)
 DO i = ista, iend
   a(i) = i
 ENDDO
 sum = 0.0
 DO i = ista, iend
   sum = sum + a(i)
 ENDDO
 CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,
&                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
 sum = ssum
 IF (myrank == 0) PRINT *,'sum =',sum
 CALL MPI_FINALIZE(ierr)
 END
```

In using the block distribution, depending on the underlying physical model, the workload may be significantly unbalanced among processes. The cyclic distribution described in the next section may work for such cases.

3.4.2 Cyclic Distribution

In cyclic distribution, the iterations are assigned to processes in a round-robin fashion. The simplest way to rewrite a DO loop is:

```
DO i = n1, n2
  computation
ENDDO
```

Or as follows:

```
      DO i = n1 + myrank, n2, nprocs
        computation
      ENDDO
```

Figure 51 shows how 14 iterations are assigned to four processes in cyclic distribution.

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

Figure 51. Cyclic Distribution

The program used to calculate the sum of an array `a()` in the previous section can be parallelized by the cyclic distribution as well.

```
      PROGRAM main
      INCLUDE 'mpif.h'
      PARAMETER (n = 1000)
      DIMENSION a(n)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i = 1 + myrank, n, nprocs
        a(i) = i
      ENDDO
      sum = 0.0
      DO i = 1 + myrank, n, nprocs
        sum = sum + a(i)
      ENDDO
      CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,
     &                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
      sum = ssum
      PRINT *,'sum =',sum
      CALL MPI_FINALIZE(ierr)
      END
```

Note that, in general, the cyclic distribution incurs more cache misses than the block distribution because of non-unit stride access to matrices within a loop. In which case should you use the cyclic distribution? One example is the LU factorization where the cyclic distribution provides more balanced workload for processes than the block distribution does. See 4.3, “LU Factorization” on page 116 for details. Other examples are molecular dynamics and the distinct element method, where the amount of computation may deviate a lot from particle to particle and from iteration to iteration.

If there are several DO loops with different iteration ranges, you can provide a mapping array `map()` and add a conditional statement in loops instead of adjusting the lower bound of each DO loop.

```
      INTEGER map(n)
      DO i = 1, n
        map(i) = MOD(i - 1, nprocs)
      ENDDO
      DO i = n1, n2
        IF (map(i) == myrank) THEN
```

```
    computation
  ENDIF
ENDDO
DO i = n3, n4
  IF (map(i) == myrank) THEN
    computation
  ENDIF
ENDDO
```

The downside of this method is that if the computation per iteration is small, the IF statement in the loop may prevent the compiler from aggressive optimization and the performance may be degraded considerably.

3.4.3 Block-Cyclic Distribution

The block-cyclic distribution is, as the name suggests, a natural generalization of the block distribution and the cyclic distribution. The iterations are partitioned into equally sized chunks (except for the last chunk) and these chunks are assigned to processes in a round-robin fashion. When you use the block-cyclic distribution, the “block” nature gives you less cache misses than the pure cyclic distribution, and the “cyclic” nature provides better load balance for certain classes of algorithms than the pure block distribution.

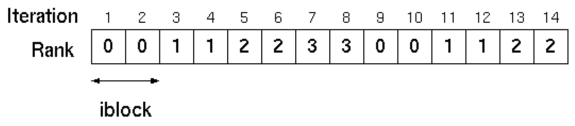


Figure 52. Block-Cyclic Distribution

Figure 52 is an example of the block-cyclic distribution of 14 iterations to four processes with a block size of two. In general, a loop:

```
DO i = n1, n2
  computation
ENDDO
```

is rewritten as

```
DO ii = n1 + myrank * iblock, n2, nprocs * iblock
  DO i = ii, MIN(ii + iblock - 1, n2)
    computation
  ENDDO
ENDDO
```

where the block size, `iblock`, is assumed to be given somewhere before the loop. You can use a mapping array instead as described in 3.4.2, “Cyclic Distribution” on page 56.

3.4.4 Shrinking Arrays

When you use block distribution, cyclic distribution, or block-cyclic distribution, it is often sufficient for each process to have only a part of the array(s) and to do the computation associated with that part. If a process needs data that another process holds, there has to be some transmission of the data between the processes. This subject is covered in 3.5, “Parallelization and Message-Passing” on page 66. On a parallel hardware with distributed memory architecture such as

RS/6000 SP, each node has physical memory of its own and processes running on different nodes do not share the address space. Therefore the memory utilization is improved by letting each process use only the necessary and sufficient amount of memory. This technique is referred to as *shrinking arrays* or *shrunk arrays* in this publication. Put in another way, shrinking arrays allows you to use larger memory for your parallel job.

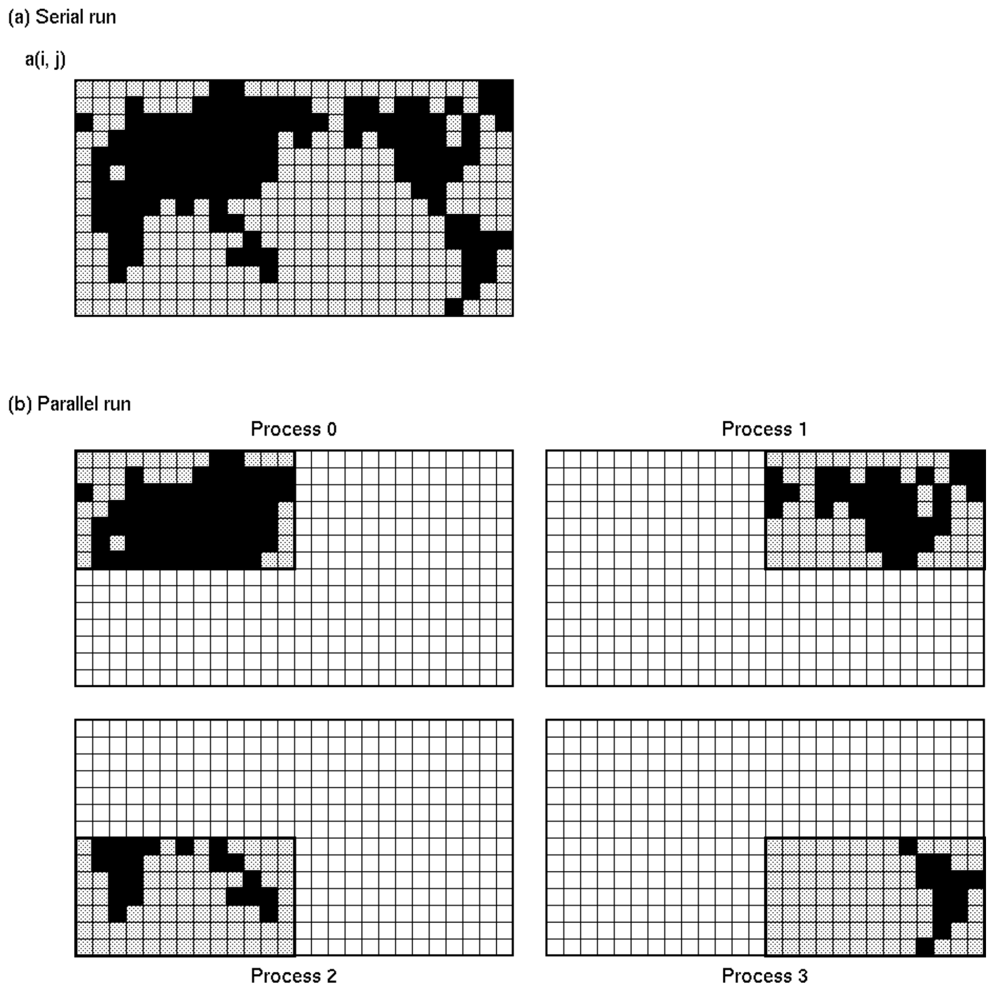


Figure 53. The Original Array and the Unshrunken Arrays

In Figure 53 (a), array `a()` is used for simulating some physical phenomena on the surface of the earth. Suppose that the size of the matrix of Figure 53 (a) just fits in the physical memory of a node. You run the simulation in parallel using four processes using four nodes. If you don't shrink arrays (Figure 53 (b)), the resolution of the simulation is the same as that of the serial run due to the limitation of memory size.

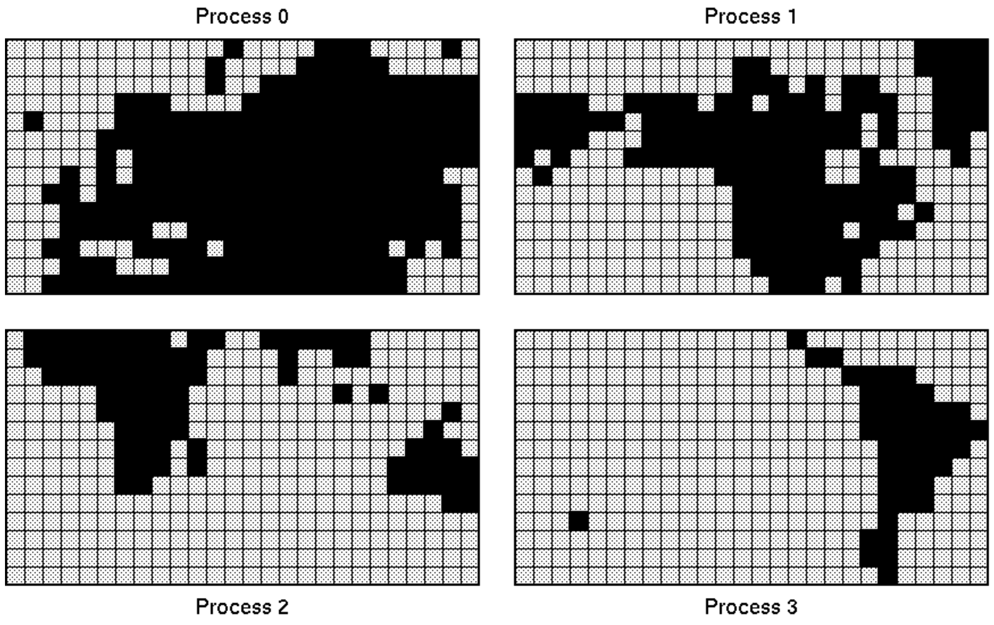


Figure 54. The Shrunk Arrays

When you shrink arrays, processes do not have to store data outside their assigned region. Therefore, each process can use more memory for its portion. As illustrated in Figure 54, the resolution of the simulation can be quadrupled by shrinking arrays in this case. Or, in other cases, you may want to extend the area to be computed with the same resolution. The technique of shrinking arrays allows you to use more memory for parallel processing.

Massively Parallel Processor (MPP) computers are able to use this technique to the fullest because the total amount of memory grows as you add more nodes. On the other hand, shared memory computers (SMP: Symmetric Multiprocessor) have comparatively small maximum memory sizes, which limits the size of the problem that you can solve.

If the size of arrays and the number of processes are known beforehand, you can explicitly write a program to shrink arrays. If this it is not the case, it is convenient to exploit the ALLOCATE statement, which dynamically provides storage at execution time.

```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n1 = 1, n2 = 1000)
REAL, ALLOCATABLE :: a(:)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(n1, n2, nprocs, myrank, ista, iend)
ALLOCATE (a(ista:iend))
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)
```

```
      ENDDO
      DEALLOCATE (a)
      CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL,
     &                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
      sum = ssum
      PRINT *,'sum =',sum
      CALL MPI_FINALIZE(ierr)
      END
```

In the above sample program, each process consumes memory only for its portion of the block-distributed array `a()`.

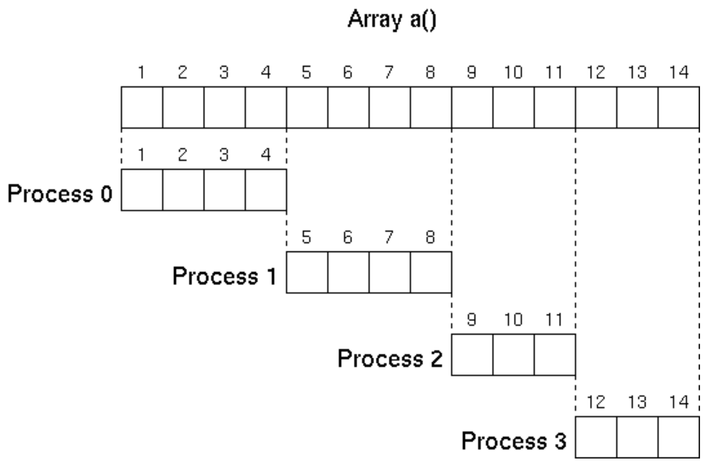


Figure 55. Shrinking an Array

Figure 55 exemplifies how each process holds its own data. Process 0, for example, has only four elements of the array `a()` instead of the entire array.

3.4.5 Parallelizing Nested Loops

In this section, only doubly nested loops are considered for parallelization. The arguments given here, however, are easily generalized to more highly nested loops. Only block distribution is discussed in this section.

When you parallelize nested loops, there are two factors that you always have to keep in mind:

1. Minimize cache misses
2. Minimize communication

According to the above two guidelines, decide how to parallelize the loops. Remember that in Fortran, multi-dimensional arrays are stored in memory in column-major order. (See "Strategy 2. Decrease the number of times that data is transmitted" on page 45.) In the case of C language, the order is the other way round. Therefore, the elements of a two-dimensional array is stored in memory in the order shown in Figure 56 on page 62.

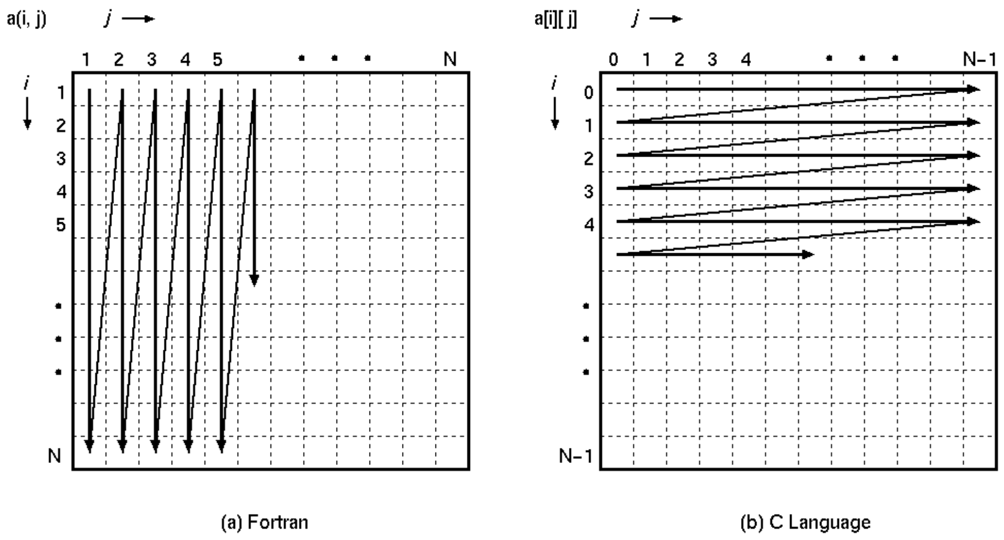


Figure 56. How a Two-Dimensional Array is Stored in Memory

It is more efficient to access arrays successively in the order they are stored in memory than to access them irrelevantly to that order. This rule is an ABC of technical computing. Consider the following loops.

Loop A

```
DO j = 1, n
  DO i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  ENDDO
ENDDO
```

Loop B

```
DO i = 1, n
  DO j = 1, n
    a(i,j) = b(i,j) + c(i,j)
  ENDDO
ENDDO
```

The larger number of cache misses due to a large stride makes Loop B much slower than Loop A. When you parallelize Loop A, the same rule applies.

Loop A1

```
DO j = jsta, jend
  DO i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  ENDDO
ENDDO
```

Loop A2

```
DO j = 1, n
  DO i = ista, iend
    a(i,j) = b(i,j) + c(i,j)
  ENDDO
ENDDO
```

As far as Loop A is concerned, you are advised to parallelize the outer loop as in Loop A1 due to fewer cache misses. See Figure 57.

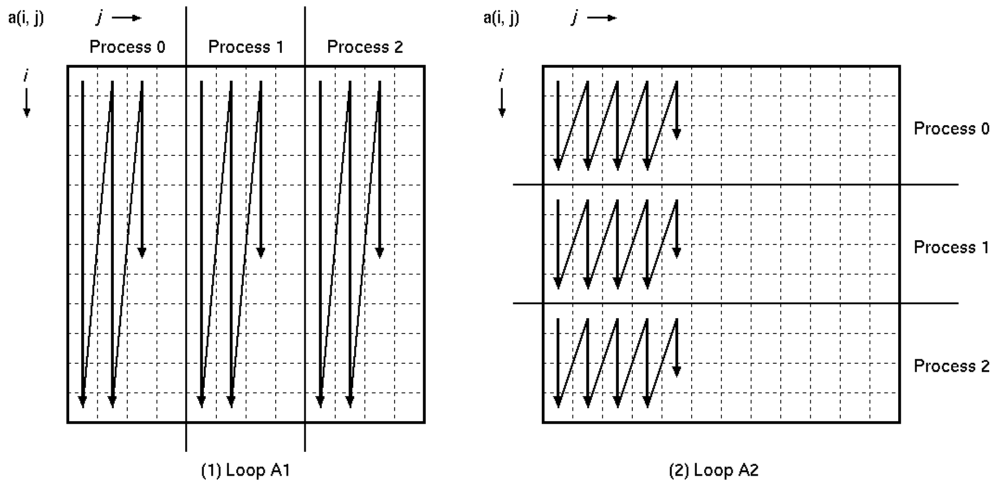


Figure 57. Parallelization of a Doubly-Nested Loop: Memory Access Pattern

Next, take into consideration the amount of data that needs to be transmitted. In Loop A, the computation is local in that the data necessary for the computation are distributed to processes in the same way the loop iterations are divided. Consider the following loop:

Loop C

```
DO j = 1, n
  DO i = 1, n
    a(i,j) = b(i,j-1) + b(i,j+1)
  ENDDO
ENDDO
```

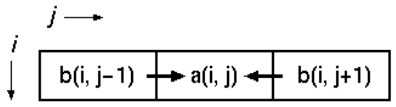


Figure 58. Dependence in Loop C

Since Loop C has a dependence on neighboring elements on the same row, if you parallelize the outer loop (column-wise distribution), you need to exchange data on the boundary between processes. In Figure 59, suppose that the process `r` has valid data for matrix elements `a(i,j)` and `b(i,j)` only in the range $1 \le i \le n$ and $jsta \le j \le jend$. Then, when you calculate `a(i,jsta)`, you need to have the value of `b(i,jsta-1)` transmitted from process `r-1`, and when you calculate `a(i,jend)`, you need `b(i,jend+1)` from process `r+1`. In total, process `r` has to receive `2n` elements from neighboring processes, and send `2n` elements back to them in return.

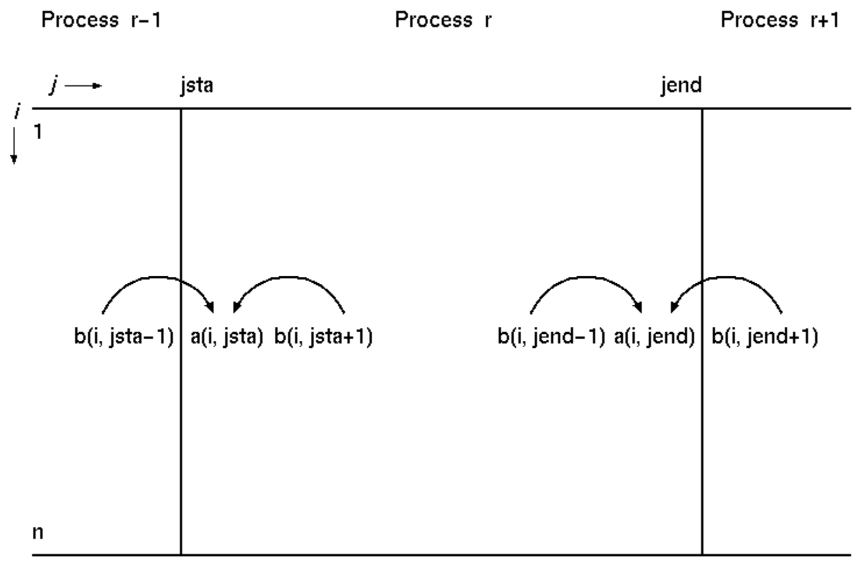


Figure 59. Loop C Block-Distributed Column-Wise

On the other hand, if you parallelize the inner loop (row-wise distribution), communication is not necessary. Since, in general, the communication latency is much larger than the memory latency, you should parallelize the inner loop in this case. Compare this with the discussion about Loop A1 and Loop A2.

Loop C has dependence in only one dimension. What if a loop has dependence in both dimensions?

Loop D

```
DO j = 1, n
  DO i = 1, m
    a(i,j) = b(i-1,j) + b(i,j-1) + b(i,j+1) + b(i+1,j)
  ENDDO
ENDDO
```

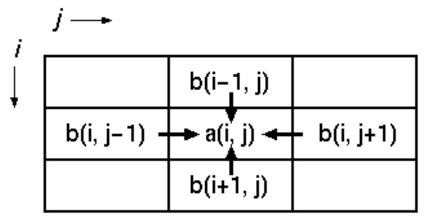


Figure 60. Dependence in Loop D

In such cases, the range of iterations also matters in deciding your strategy. Suppose `m` is greater than `n`. As you saw in Figure 59, the number of elements that must be transmitted, if any, is proportional to the length of boundary between processes. In Figure 61 on page 65, the elements of matrix `b(i,j)` that have to be transmitted are marked as shaded boxes. It is evident that due to the assumption of `m` and `n`, you should parallelize the inner loop (row-wise distribution) in this case.

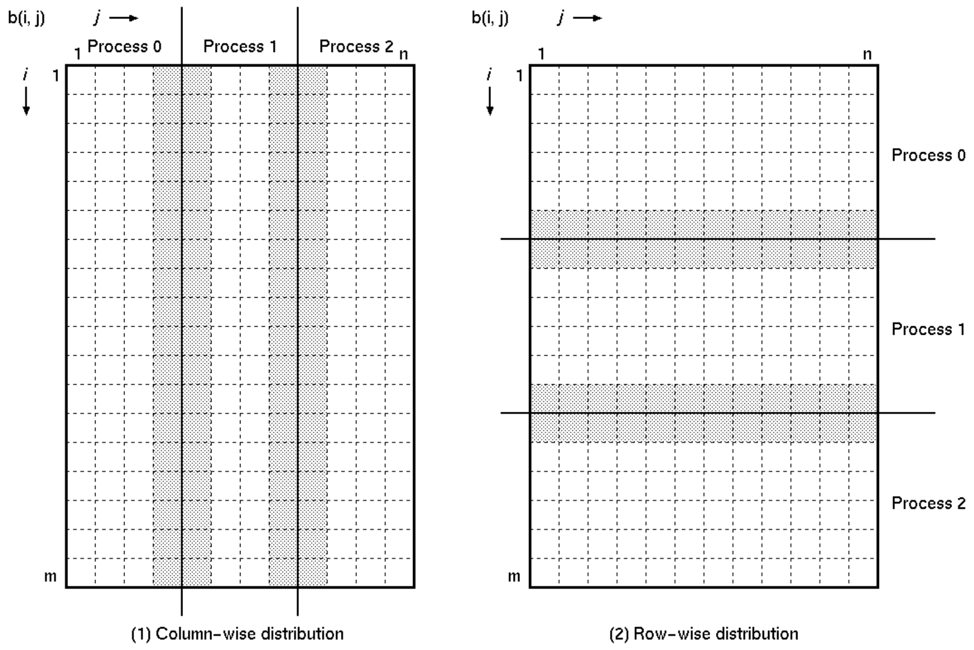


Figure 61. Loop D Block-Distributed (1) Column-Wise and (2) Row-Wise

All the doubly-nested loops in this section so far are parallelized either in the inner or in the outer loop. However, you can parallelize both loops as follows:

Loop E

```
DO j = jsta, jend
  DO i = ista, iend
    a(i,j) = b(i-1,j) + b(i,j-1) + b(i,j+1) + b(i+1,j)
  ENDDO
ENDDO
```

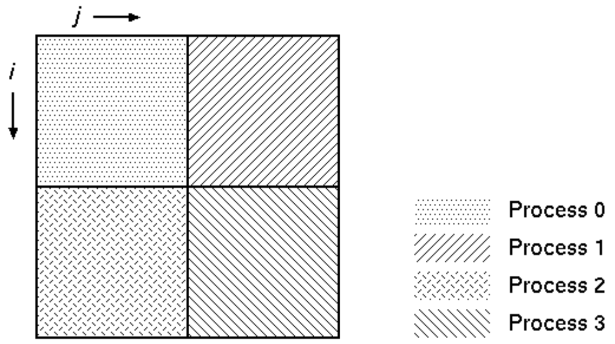


Figure 62. Block Distribution of Both Dimensions

Figure 62 is an example of distributing a matrix to four processes by dividing both dimensions into two. There is an obvious restriction: the number of processes has to be a compound number. In other words, it has to be expressed as a product of integers greater than 1.

When the number of processes is expressed in several ways as a product of two integers, you should be careful in choosing among those options. Generally, choose a distribution so that the rectangle area for each process becomes as close to square as possible. This is based on the following observation. The amount of data for transmission is proportional to the perimeter of the rectangle except for the boundary processes, and the perimeter is minimized in square as long as the area of the rectangle is the same.

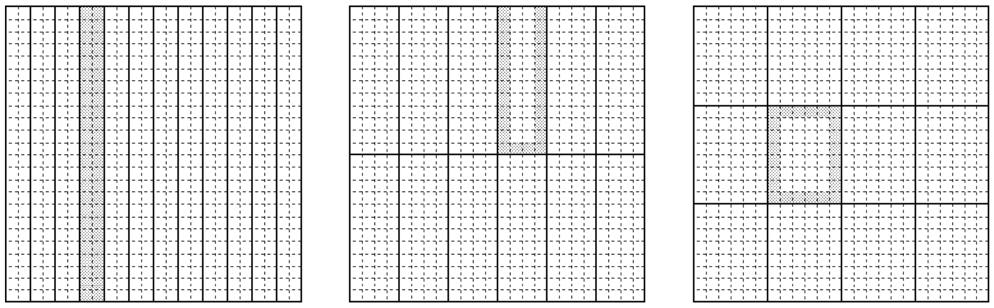


Figure 63. The Shape of Submatrices and Their Perimeter

Figure 63 shows three ways of distributing a 24x24 matrix to 12 processes. From left to right, the matrix is divided into 1x12, 2x6, and 3x4 blocks. And the maximum number of elements to be transmitted per process is, 48, 26, and 24, respectively.

3.5 Parallelization and Message-Passing

As you saw in 3.4, “Parallelizing DO Loops” on page 54, the focus of parallelization is dividing the amount of computation by *p* (the number of processes) and speeding up the computation. In doing so, a process may not have all the data it needs, or it may have incorrect or false data because they lie outside the assigned part of that process and the process does not care about them. Message-passing can be viewed as a job of fixing this situation, which you do not do voluntarily but are obliged to do reluctantly. But do not worry. There are not an overwhelming number of variations of situations where you have to use message-passing. In this section are the typical cases and their solutions that are applicable to wide range of programs.

3.5.1 Reference to Outlier Elements

Consider parallelizing the following code by block distribution.

```
...
REAL a(9), b(9)
...
DO i = 1, 9
  a(i) = i
ENDDO
DO i = 1, 9
  b(i) = b(i) * a(1)
ENDDO
...
```

The second loop references a specific element `a(1)`, and all processes need to get the right value.

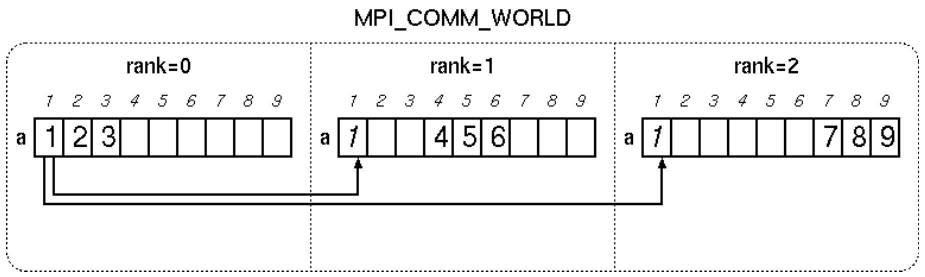


Figure 64. Reference to an Outlier Element

Figure 64 shows a parallel execution by three processes. Since processes 1 and 2 do not have a valid entry for `a(1)`, it has to be broadcasted by process 0.

```
...
REAL a(9), b(9)
...
DO i = ista, iend
  a(i) = i
ENDDO
CALL MPI_BCAST(a(1), 1, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
DO i = ista, iend
  b(i) = b(i) * a(1)
ENDDO
...
```

This type of data transmission is seen in the LU factorization, where the process that is holding the pivot column broadcasts that column to the other processes. See 4.3, “LU Factorization” on page 116 for detail.

3.5.2 One-Dimensional Finite Difference Method

It is not an exaggeration to call the finite difference method (FDM) the *king* of technical computing because the FDM and its variations are the most frequently used method to solve partial differential equations. This section discusses one-dimensional FDM in its most simplified form. Two-dimensional FDM is discussed in 4.1, “Two-Dimensional Finite Difference Method” on page 99.

The following program abstracts the essence of one-dimensional FDM, where coefficients and the enclosing loop for convergence are omitted.

```
PROGRAM main
IMPLICIT REAL*8(a-h,o-z)
PARAMETER (n = 11)
DIMENSION a(n), b(n)
DO i = 1, n
  b(i) = i
ENDDO
DO i = 2, n-1
  a(i) = b(i-1) + b(i+1)
ENDDO
END
```

Executed in serial, the above program computes the array `a()` from left to right in Figure 65 on page 68. In this figure, the data dependence is shown as arrows. The same kind of program was discussed in “Pattern 2. Parallelization

Throughout a DO Loop" on page 47, where the details of message-passing was hidden in a presumed subroutine `shift`. Here, explicit code of `shift` is given.

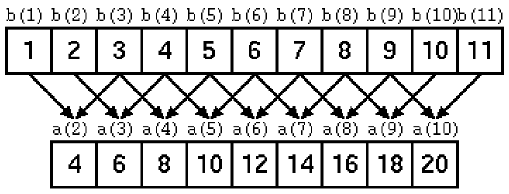

b(1) b(2) b(3) b(4) b(5) b(6) b(7) b(8) b(9) b(10) b(11)
1 2 3 4 5 6 7 8 9 10 11
a(2) a(3) a(4) a(5) a(6) a(7) a(8) a(9) a(10)
4 6 8 10 12 14 16 18 20

Figure 65. Data Dependence in One-Dimensional FDM

The following program is a parallelized version that distributes data in the block distribution fashion and does the necessary data transmissions. The program does not exploit the technique of shrinking arrays. The numbers on the left are not part of the program and are only for reference.

```
 1      PROGRAM main
 2      INCLUDE 'mpif.h'
 3      IMPLICIT REAL*8(a-h,o-z)
 4      PARAMETER (n = 11)
 5      DIMENSION a(n), b(n)
 6      INTEGER istatus(MPI_STATUS_SIZE)
 7      CALL MPI_INIT(ierr)
 8      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 9      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
10      CALL para_range(1, n, nprocs, myrank, ista, iend)
11      ista2 = ista
12      iend1 = iend
13      IF (myrank == 0)          ista2 = 2
14      IF (myrank == nprocs - 1) iend1 = n - 1
15      inext = myrank + 1
16      iprev = myrank - 1
17      IF (myrank == nprocs-1) inext = MPI_PROC_NULL
18      IF (myrank == 0)        iprev = MPI_PROC_NULL
19      DO i = ista, iend
20        b(i) = i
21      ENDDO
22      CALL MPI_ISEND(b(iend),   1, MPI_REAL8, inext, 1, MPI_COMM_WORLD, isend1, ierr)
23      CALL MPI_ISEND(b(ista),   1, MPI_REAL8, iprev, 1, MPI_COMM_WORLD, isend2, ierr)
24      CALL MPI_IRECV(b(ista-1), 1, MPI_REAL8, iprev, 1, MPI_COMM_WORLD, irecv1, ierr)
25      CALL MPI_IRECV(b(iend+1), 1, MPI_REAL8, inext, 1, MPI_COMM_WORLD, irecv2, ierr)
26      CALL MPI_WAIT(isend1, istatus, ierr)
27      CALL MPI_WAIT(isend2, istatus, ierr)
28      CALL MPI_WAIT(irecv1, istatus, ierr)
29      CALL MPI_WAIT(irecv2, istatus, ierr)
30      DO i = ista2, iend1
31        a(i) = b(i-1) + b(i+1)
32      ENDDO
33      CALL MPI_FINALIZE(ierr)
34      END
```

Lines 2, 7, 8, and 9 are for the initialization of MPI processes. Since non-blocking communication subroutines and consequently MPI_WAIT are used later, a declaration of a status array, `istatus`, is also necessary (line 6). Subroutine para_range (3.4.1, "Block Distribution" on page 54) calculates lower and upper bounds of the iteration variable (line 10). Since the second loop in the original program has range `2..n-1` instead of `1..n`, another pair of bounds, `ista2` and `iend1`, are prepared for the second loop (lines 11-14). Consider why you cannot use para_range for calculating `ista2` and `iend1`. In communicating with adjacent processes, processes `0` and `nprocs-1` have only one neighbor, while the other processes have two. You can handle this discrepancy by adding IF statements that treat processes `0` and `nprocs-1` specially. But in the above program, send and receive subroutines are used symmetrically for all processes by specifying

MPI_PROC_NULL as the source or the destination if no transmission is necessary (lines 17 and 18).

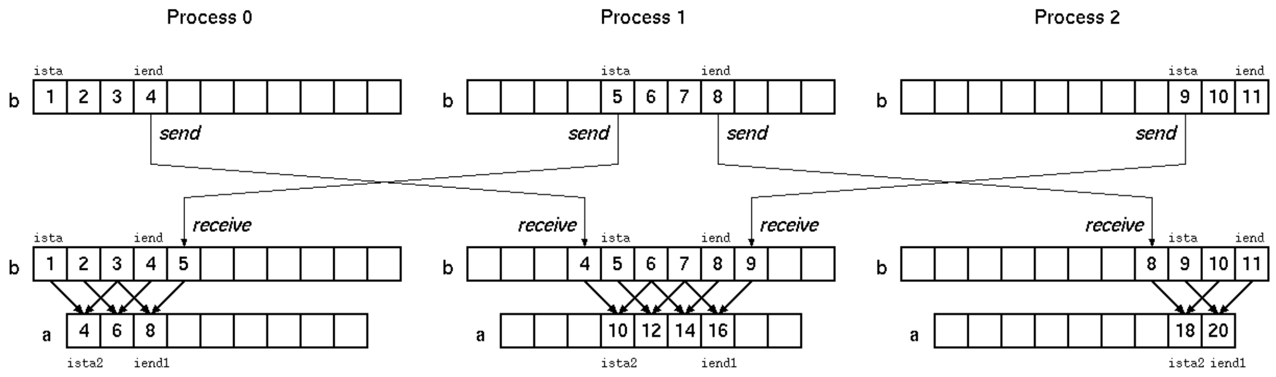


Figure 66. Data Dependence and Movements in the Parallelized FDM

In one-dimensional FDM, the data to be transmitted is on the boundary of a one-dimensional array, which are *points*. Likewise, in two-dimensional FDM, data on *lines* are transmitted, and in three-dimensional FDM, data on *planes* are transmitted.

Some FDM programs have cyclic boundary conditions imposed by the underlying physical model. Typically, you see code as shown in the following:

```
DO i = 1, n
  a(i) = ...
ENDDO
a(1) = a(n)
```

Do not overlook the last statement in parallelizing the code.

```
 INTEGER istatus(MPI_STATUS_SIZE)
...
DO i = ista, iend
  a(i) = ...
ENDDO
IF (myrank == nprocs - 1) THEN
  CALL MPI_SEND(a(n), 1, MPI_REAL8, 0, 1, MPI_COMM_WORLD, ierr)
ELSEIF (myrank == 0) THEN
  CALL MPI_RECV(a(1), 1, MPI_REAL8, nprocs - 1, 1, MPI_COMM_WORLD, istatus, ierr)
ENDIF
```

Of course, you can choose the combination of non-blocking subroutines and MPI_WAIT.

3.5.3 Bulk Data Transmissions

This section deals with the cases where data to be transmitted does not necessarily lie on the boundary between processes. Rather, each process sends the whole data assigned to it to a certain process (gather), or processes exchange data simultaneously to have the up-to-date elements over whole arrays (allgather), or you change the block distribution from column-wise to row-wise and reassign the matrix elements to processes.

3.5.3.1 Gathering Data to One Process

Sometimes (usually near the end of a parallel program), you might need to gather data computed separately by parallel processes to a single process. The following four cases are possible depending on whether the data held by a process is contiguous in memory or not and whether the send and receive buffers overlap or not.

1. Contiguous data; send and receive buffers do not overlap
2. Contiguous data; send and receive buffers overlap
3. Non-contiguous data; send and receive buffers do not overlap
4. Non-contiguous data; send and receive buffers overlap

Case 3 is not described because it is part of the understanding of cases 1, 2, and 4.

In the following discussion, two-dimensional arrays are considered. If the data to be transmitted are not contiguous in memory, the utility subroutines that define derived data types (2.5, "Derived Data Types" on page 28) are used. In using MPI_GATHER and its derivatives (MPI_GATHERV, MPI_ALLGATHER, and MPI_ALGATHERV), the send and receive buffers may not overlap in memory, so you have to resort to point-to-point communication subroutines if they overlap. In MPI-2, the situation has changed though (See 2.3.2, "MPI_GATHER" on page 17).

Case 1. Contiguous data; send and receive buffers do not overlap

You only need MPI_GATHERV for this case. Array elements in Figure 67 are chosen deliberately so that they indicate the displacement counts measured from the beginning of the matrix.

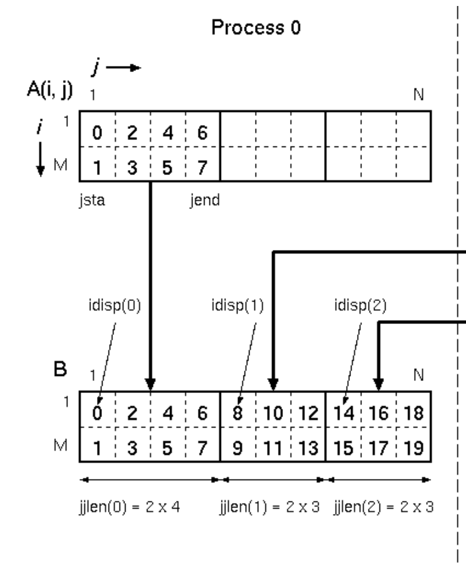
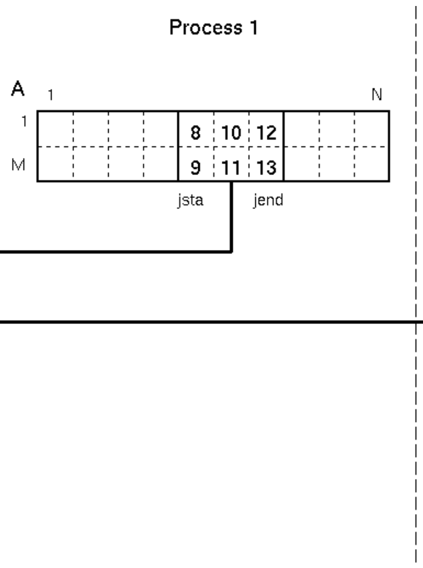
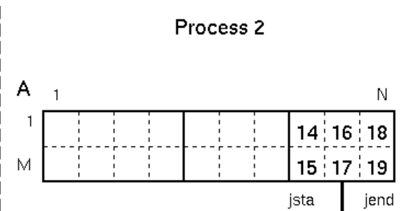


Figure 67. Gathering an Array to a Process (Contiguous; Non-Overlapping Buffers)

The parallelized code segment is given below.

```
REAL a(m,n), b(m,n)
INTEGER, ALLOCATABLE :: idisp(:), jjlen(:)
...
ALLOCATE (idisp(0:nprocs-1), jjlen(0:nprocs-1))
DO irank = 0, nprocs - 1
  CALL para_range(1, n, nprocs, irank, jsta, jend)
```

```
      jjlen(irank) = m * (jend - jsta + 1)
      idisp(irank) = m * (jsta - 1)
    ENDDO
    CALL para_range(1, n, nprocs, myrank, jsta, jend)
    ...
    CALL MPI_GATHERV(a(1,jsta), jjlen(myrank), MPI_REAL,
   &     b, jjlen, idisp, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
    DEALLOCATE (idisp, jjlen)
    ...
```

Case 2. Contiguous data; send and receive buffers overlap

When the send and receive buffers overlap in memory, you cannot use MPI_GATHERV. Use point-to-point communication subroutines instead.

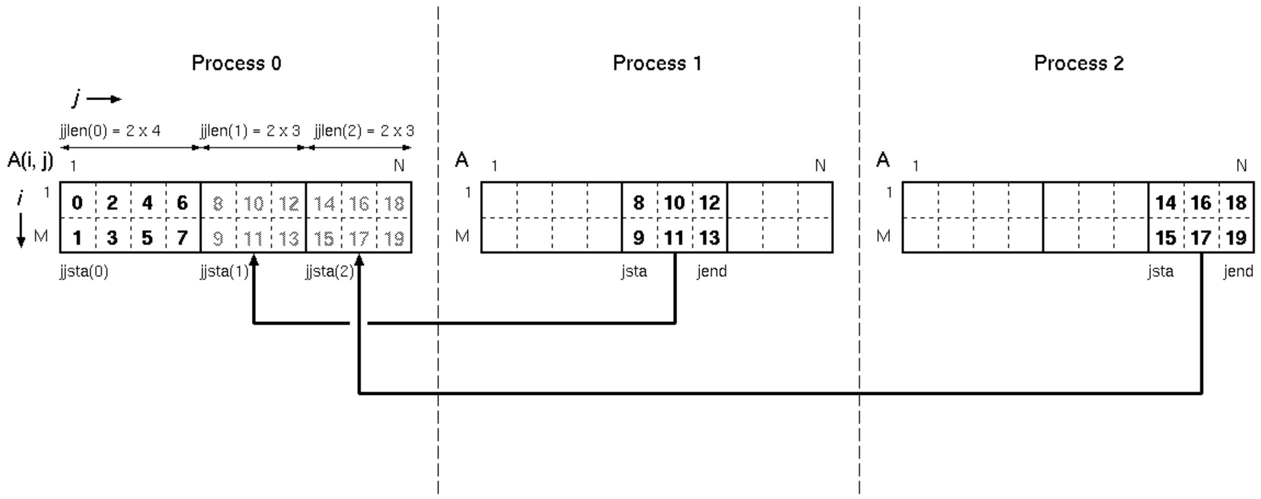


Figure 68. Gathering an Array to a Process (Contiguous; Overlapping Buffers)

```
    REAL a(m,n)
    INTEGER, ALLOCATABLE :: jjsta(:), jjlen(:), iireq(:)
    INTEGER istatus(MPI_STATUS_SIZE)
    ...
    ALLOCATE (jjsta(0:nprocs-1))
    ALLOCATE (jjlen(0:nprocs-1))
    ALLOCATE (iireq(0:nprocs-1))
    DO irank = 0, nprocs - 1
      CALL para_range(1, n, nprocs, irank, jsta, jend)
      jjsta(irank) = jsta
      jjlen(irank) = m * (jend - jsta + 1)
    ENDDO
    CALL para_range(1, n, nprocs, myrank, jsta, jend)
    ...
    IF (myrank == 0) THEN
      DO irank = 1, nprocs - 1
        CALL MPI_IRECV(a(1,jjsta(irank)),jjlen(irank),MPI_REAL,
   &          irank, 1, MPI_COMM_WORLD, iireq(irank),ierr)
      ENDDO
      DO irank = 1, nprocs - 1
        CALL MPI_WAIT(iireq(irank), istatus, ierr)
      ENDDO
    ELSE
```

```
        CALL MPI_ISEND(a(1,jsta), jjlen(myrank), MPI_REAL,
     &                 0, 1, MPI_COMM_WORLD, ireq, ierr)
        CALL MPI_WAIT(ireq, istatus, ierr)
      ENDIF
      DEALLOCATE (jjsta, jjlen, iireq)
      ...
```

Case 3. Non-contiguous data; send and receive buffers do not overlap

This case is trivial after reading Case 4.

Case 4. Non-contiguous data; send and receive buffers overlap

When you divide a two-dimensional array by rows, each process holds elements that are not contiguous in memory. In the sample program, the utility subroutine para_type_block2 given in 2.5.2.2, “Utility Subroutine: para_type_block2” on page 32 is used. Note that you might get better performance by packing and sending the data manually.

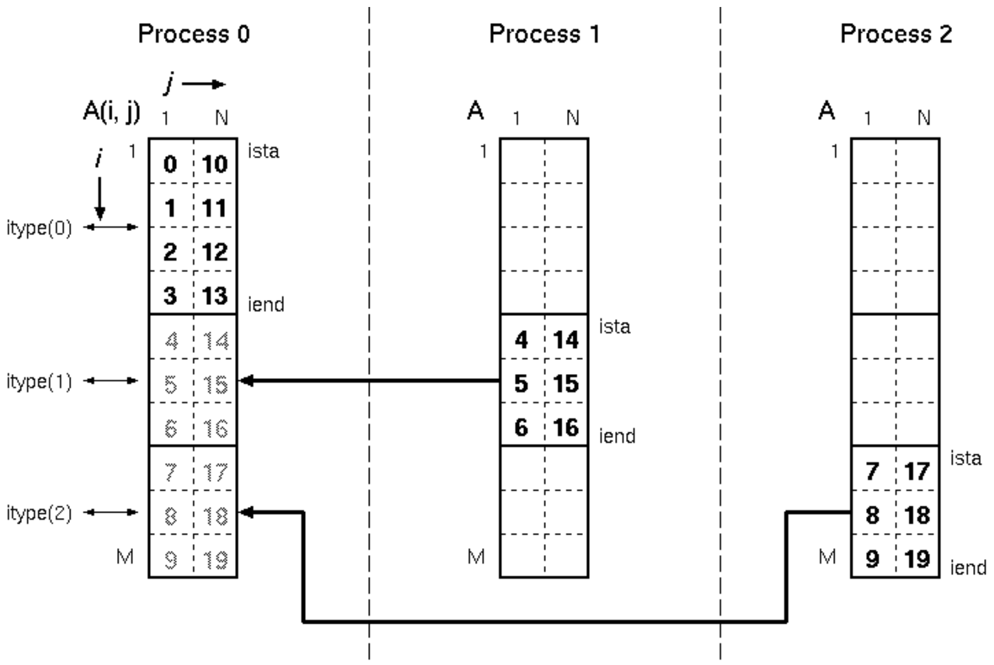


Figure 69. Gathering an Array to a Process (Non-Contiguous; Overlapping Buffers)

```
      REAL a(m,n)
      INTEGER, ALLOCATABLE :: itype(:), iireq(:)
      INTEGER istatus(MPI_STATUS_SIZE)
      ...
      ALLOCATE (itype(0:nprocs-1), iireq(0:nprocs-1))
      DO irank = 0, nprocs - 1
        CALL para_range(1, m, nprocs, irank, ista, iend)
        CALL para_type_block2(1, m, 1, ista, iend, 1, n,
     &                        MPI_REAL, itype(irank))
      ENDDO
      CALL para_range(1, m, nprocs, myrank, ista, iend)
      ...
      IF (myrank == 0) THEN
        DO irank = 1, nprocs - 1
      CALL MPI_IRECV(a, 1, itype(irank), irank,
&                    1, MPI_COMM_WORLD, iireq(irank), ierr)
    ENDDO
    DO irank = 1, nprocs - 1
      CALL MPI_WAIT(iireq(irank), istatus, ierr)
    ENDDO
  ELSE
    CALL MPI_ISEND(a, 1, itype(myrank), 0,
&                  1, MPI_COMM_WORLD, ireq, ierr)
    CALL MPI_WAIT(ireq, istatus, ierr)
  ENDIF
  DEALLOCATE (itype, iireq)
  ...
```

3.5.3.2 Synchronizing Data

Sometimes you need to let all the parallel processes have up-to-date data at a certain point in the program. This happens when you use the method mentioned in “Pattern 1. Partial Parallelization of a DO Loop” on page 46, or when you are debugging your parallel program (Table 84 on page 92). As in 3.5.3.1, “Gathering Data to One Process” on page 69, there are four cases depending on whether the data for transmission are contiguous or not and whether the send and receive buffers overlap in memory or not. To avoid repetition of the same idea, non-contiguous cases are not discussed.

Case 1. Synchronizing data; send and receive buffers do not overlap

Only one call of the subroutine MPI_ALLGATHERV suffices for handling this case.

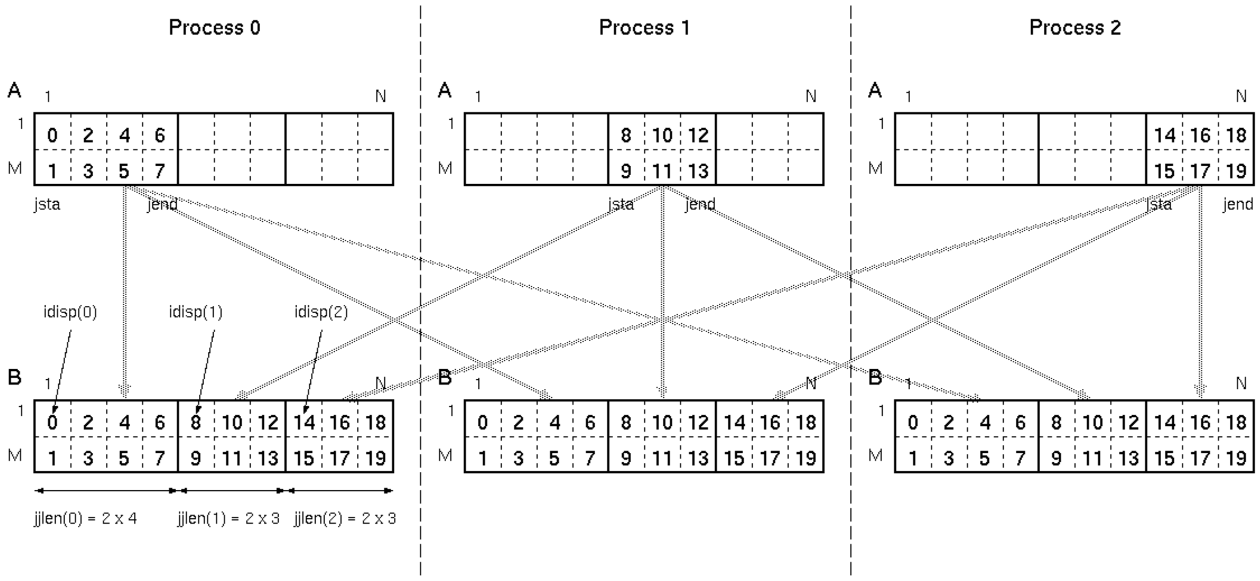


Figure 70. Synchronizing Array Elements (Non-Overlapping Buffers)

```
  REAL a(m,n), b(m,n)
  INTEGER, ALLOCATABLE :: idisp(:), jjlen(:)
  ...
  ALLOCATE (idisp(0:nprocs-1), jjlen(0:nprocs-1))
  DO irank = 0, nprocs - 1
```

```
      CALL para_range(1, n, nprocs, irank, jsta, jend)
      jjlen(irank) = m * (jend - jsta + 1)
      idisp(irank) = m * (jsta - 1)
    ENDDO
    CALL para_range(1, n, nprocs, myrank, jsta, jend)
    ...
    CALL MPI_ALLGATHERV(a(1,jsta), jjlen(myrank), MPI_REAL,
   &     b, jjlen, idisp, MPI_REAL, MPI_COMM_WORLD, ierr)
    DEALLOCATE (idisp, jjlen)
    ...
```

Case 2. Synchronizing data; send and receive buffers overlap

Since you cannot use MPI_ALLGATHERV in this case, you have to broadcast as many times as there are processes.

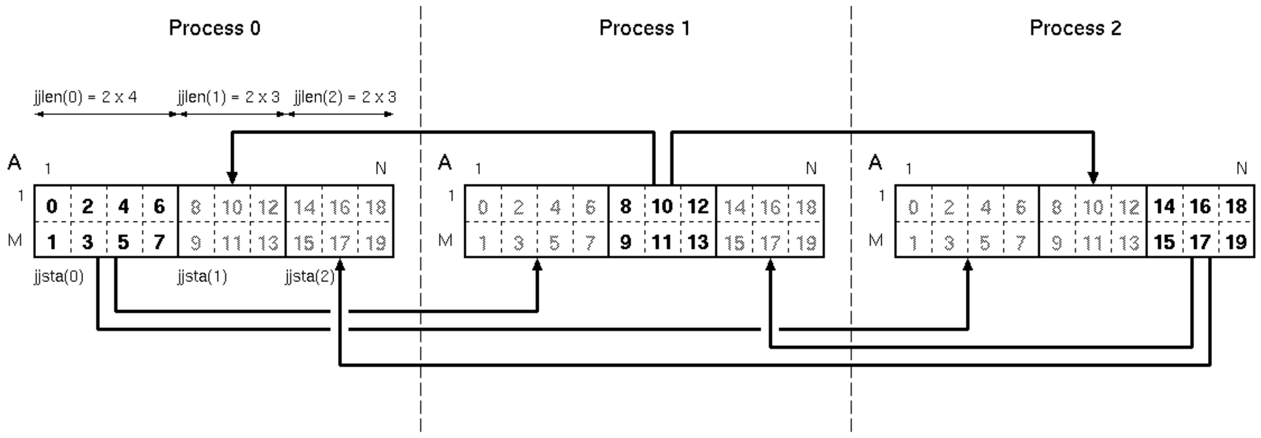


Figure 71. Synchronizing Array Elements (Overlapping Buffers)

```
    REAL a(m,n)
    INTEGER, ALLOCATABLE :: jjsta(:), jjlen(:)
    ...
    ALLOCATE (jjsta(0:nprocs-1), jjlen(0:nprocs-1))
    DO irank = 0, nprocs - 1
      CALL para_range(1, n, nprocs, irank, jsta, jend)
      jjsta(irank) = jsta
      jjlen(irank) = m * (jend - jsta + 1)
    ENDDO
    CALL para_range(1, n, nprocs, myrank, jsta, jend)
    ...
    DO irank = 0, nprocs - 1
      CALL MPI_BCAST(a(1,jjsta(irank)), jjlen(irank), MPI_REAL,
   &                 irank, MPI_COMM_WORLD, ierr)
    ENDDO
    DEALLOCATE (jjsta, jjlen)
    ...
```

The sample code given here uses MPI_BCAST, but often you get better performance by using MPE_IBCAST at the cost of portability. MPE_IBCAST, described in B.2.2, “MPE_IBCAST (IBM Extension)” on page 164, is a non-blocking version of MPI_BCAST, which is included in the IBM extensions to the MPI standard. If you use MPE_IBCAST, the italic part of the code should be rewritten as follows.

```
      DO irank = 0, nprocs - 1
        CALL MPE_IBCAST(a(1,jjsta(irank)),jjlen(irank),MPI_REAL,
     &       irank, MPI_COMM_WORLD, iireq(irank), ierr)
      ENDDO
      DO irank = 0, nprocs - 1
        CALL MPI_WAIT(iireq(irank), istatus, ierr)
      ENDDO
```

The integer arrays `iireq()` and `istatus()` are assumed to be allocated appropriately. Don't forget to call MPI_WAIT to complete data transmissions since you are using non-blocking subroutines.

3.5.3.3 Transposing Block Distributions

Suppose that you need to change the distribution of a two-dimensional matrix from column-wise block distribution to row-wise block distribution during the computation.

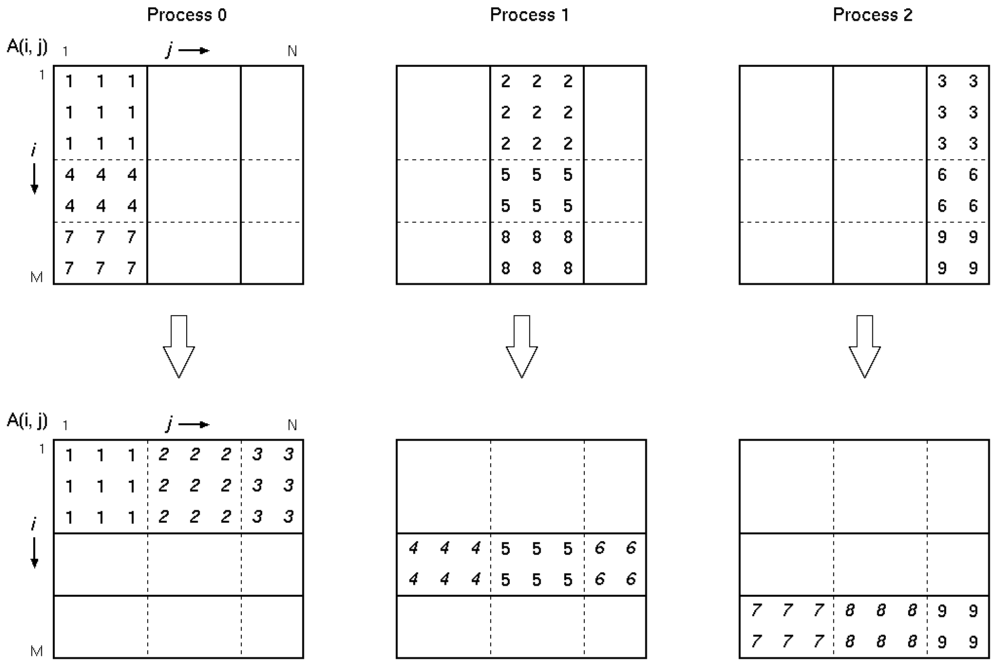


Figure 72. Transposing Block Distributions

In Figure 72, matrix `a()` is assigned to three processes in the column-wise block distribution at first. Then you change the distribution to row-wise for some reason. This case happens when you parallelize two-dimensional FFT code, for instance. The boundary lines of both distributions divide the matrix into nine blocks, and it is easier to consider the data transmission in terms of these blocks. To deal with this communication pattern, it is natural to define derived data types for each block as shown in Figure 73 on page 76.

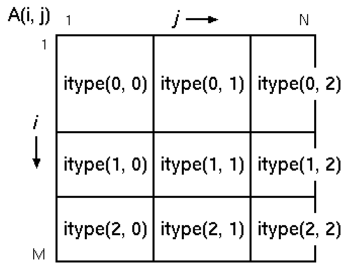


Figure 73. Defining Derived Data Types

You can use the utility subroutine described in 2.5.2.2, “Utility Subroutine: para_type_block2” on page 32 for defining derived data types. The following code does the job.

```
      ...
      INCLUDE 'mpif.h'
      PARAMETER (m=7, n=8)
      REAL a(m,n)
      PARAMETER (ncpu=3)
      INTEGER itype(0:ncpu-1, 0:ncpu-1)
      INTEGER ireq1(0:ncpu-1), ireq2(0:ncpu-1)
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO jrank = 0, nprocs-1
        CALL para_range(1, n, nprocs, jrank, jsta, jend)
        DO irank = 0, nprocs-1
          CALL para_range(1, m, nprocs, irank, ista, iend)
          CALL para_type_block2
     &         (1, m, 1, ista, iend, jsta, jend, MPI_REAL, itemp)
          itype(irank, jrank) = itemp
        ENDDO
      ENDDO
      CALL para_range(1, m, nprocs, myrank, ista, iend)
      CALL para_range(1, n, nprocs, myrank, jsta, jend)
      ...
      DO irank = 0, nprocs-1
        IF (irank /= myrank) THEN
          CALL MPI_ISEND(a, 1, itype(irank, myrank),
     &         irank, 1, MPI_COMM_WORLD, ireq1(irank), ierr)
          CALL MPI_IRECV(a, 1, itype(myrank, irank),
     &         irank, 1, MPI_COMM_WORLD, ireq2(irank), ierr)
        ENDIF
      ENDDO
      DO irank = 0, nprocs-1
        IF (irank /= myrank) THEN
          CALL MPI_WAIT(ireq1(irank), istatus, ierr)
          CALL MPI_WAIT(ireq2(irank), istatus, ierr)
        ENDIF
      ENDDO
      ...
```

Even if the source matrix and the target matrix do not overlap in memory, you cannot use MPI_ALLTOALLV in this case.

3.5.4 Reduction Operations

When given as a textbook exercise, it is quite easy and straightforward to parallelize reduction operations. However in real programs, reduction operations are likely to be overlooked when data distribution, data dependences, and other items are receiving your attention.

```
      ...
      sum1 = 0.0
      sum2 = 0.0
      amax = 0.0
      DO i = 1, n
        a(i) = a(i) + b(i)
        c(i) = c(i) + d(i)
        e(i) = e(i) + f(i)
        g(i) = g(i) + h(i)
        x(i) = x(i) + y(i)
        sum1 = sum1 + a(i)
        sum2 = sum2 + c(i)
        IF (a(i) > amax) amax = a(i)
      ENDDO
      DO i = 1, n
        g(i) = g(i) * sum1 + sum2
      ENDDO
      PRINT *, amax ...
```

The above code is parallelized as follows.

```
      ...
      REAL works(2), workr(2)
      sum1 = 0.0
      sum2 = 0.0
      amax = 0.0
      DO i = ista, iend
        a(i) = a(i) + b(i)
        c(i) = c(i) + d(i)
        e(i) = e(i) + f(i)
        g(i) = g(i) + h(i)
        x(i) = x(i) + y(i)
        sum1 = sum1 + a(i)
        sum2 = sum2 + c(i)
        IF (a(i) > amax) amax = a(i)
      ENDDO
      works(1) = sum1
      works(2) = sum2
      CALL MPI_ALLREDUCE(works, workr, 2, MPI_REAL,
     &                   MPI_SUM, MPI_COMM_WORLD, ierr)
      sum1 = workr(1)
      sum2 = workr(2)
      CALL MPI_REDUCE(amax, aamax, 1, MPI_REAL,
     &                MPI_MAX, 0, MPI_COMM_WORLD, ierr)
      amax = aamax
      DO i = ista, iend
        g(i) = g(i) * sum1 + sum2
      ENDDO
      IF (myrank == 0) THEN
        PRINT *, amax
      ENDIF
      ...
```

Since every process needs the value of `sum1` and `sum2`, you have to use MPI_ALLREDUCE rather than MPI_REDUCE. On the other hand, since `amax` is necessary to only one process, MPI_REDUCE is sufficient for it. Note that variables `sum1` and `sum2` are copied into the array `works` in order to reduce the number of calls of the MPI_ALLREDUCE subroutine.

3.5.5 Superposition

Usually, the subroutines MPI_REDUCE and MPI_ALLREDUCE are used to gather data distributed over processes and to do some computation on the way. They are also useful for just gathering data where no computation is meant in the first place. Let's see how they work by looking at a sample program.

Suppose you have a matrix distributed to processes in the row-wise cyclic distribution. Each process has reserved memory for storing the whole matrix rather than the shrunken one. Each process does some computation on the matrix elements assigned to it, and you need to gather the data to process 0 at some time.

An orthodox method to gather data in this case is to define derived data types for non-contiguous data held by each task and send the data to process 0. Or, on the sending process, you first pack the non-contiguous data to a contiguous location and then send the packed data to process 0. On the receiving process, you receive the data, unpack, and copy them to proper locations. But both methods might look cumbersome to you. Consider the MPI_REDUCE subroutine. Make sure the matrix elements that lie outside the assigned part of the process are set to zero.

```
      REAL a(n,n), aa(n,n)
      ...
      DO j = 1, n
        DO i = 1, n
          a(i,j) = 0.0
        ENDDO
      ENDDO
      DO j = 1, n
        DO i = 1 + myrank, n, nprocs
          a(i,j) = computation
        ENDDO
      ENDDO
      CALL MPI_REDUCE(a, aa, n*n, MPI_REAL, MPI_SUM, 0,
     &                MPI_COMM_WORLD, ierr)
      ...
```

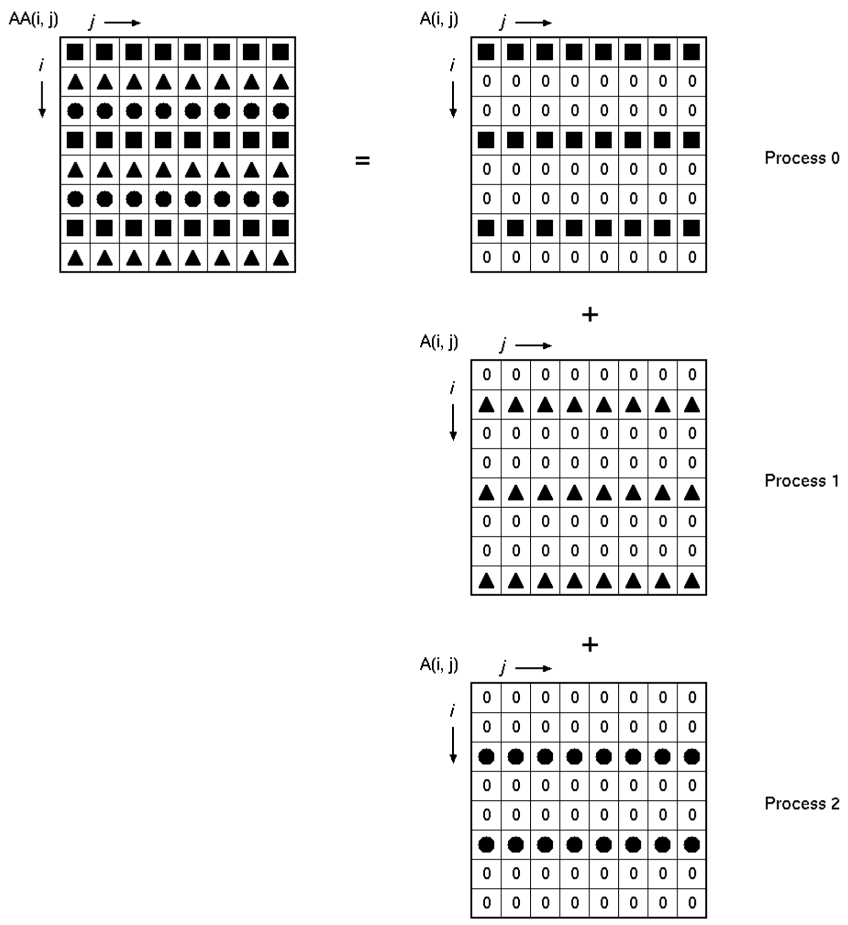


Figure 74. Superposition

Figure 74 illustrates how the parallel program gathers data in the case of `n=8` and `nprocs=3`. Note that although it is easy to implement the superposition method, you should not use it when the communication overhead is a concern because each process sends *n*n* elements instead of *n*n/nprocs* elements.

3.5.6 The Pipeline Method

The pipeline method is a technique to parallelize a loop that has loop-carried dependences. The following is a typical loop that has a flow dependence, that is, each iteration has to be executed strictly in order.

```
DO i = 1, n
  x(i) = x(i-1) + 1.0
ENDDO
```

If you expand the loop into flat statements, it is easy to see what a flow dependence means:

```
x(1) = x(0) + 1.0        (iteration 1)
x(2) = x(1) + 1.0        (iteration 2)
x(3) = x(2) + 1.0        (iteration 3)
...
```

Iteration 2 uses the value of `x(1)` calculated in iteration 1; thus iteration 2 has to be executed after iteration 1 has finished, and so on.

The following program has a flow dependence that prevents you from parallelization by naively using block distribution or cyclic distribution.

```
PROGRAM main
PARAMETER (mx = ..., my = ...)
DIMENSION x(0:mx,0:my)
...
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j) + x(i-1,j) + x(i,j-1)
  ENDDO
ENDDO
...
```

This kind of loop is seen in the forward elimination and the backward substitution of ICCG (Incomplete Cholesky Conjugate Gradient) method that solves linear equations with a symmetric band diagonal matrix. The dependence involved in program `main` is illustrated in Figure 75 (a).

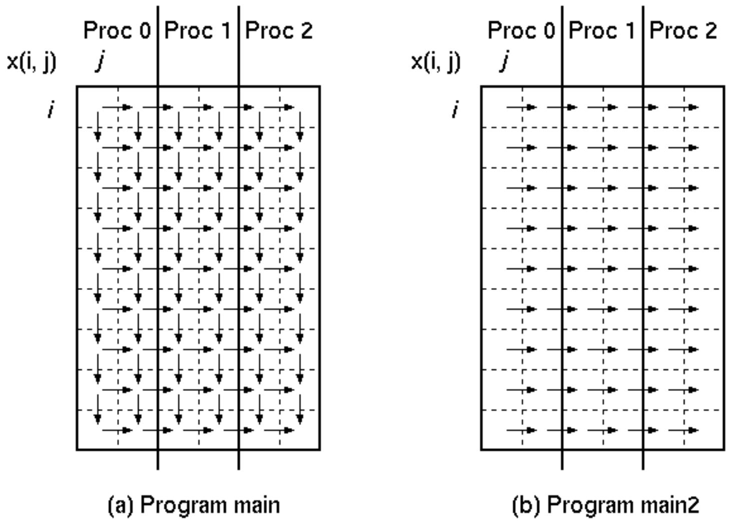


Figure 75. Data Dependences in (a) Program main and (b) Program main2

Note that parallelizing the following loop is quite different from parallelizing program `main` because this loop has no loop-carried dependences. You must make sure that each process has the data it needs, especially the values of `x(i,j)` lying just outside the boundary. Then each process can start calculating `y(i,j)` in parallel.

```
DO j = 1, my
  DO i = 1, mx
    y(i,j) = x(i,j) + x(i-1,j) + x(i,j-1)
  ENDDO
ENDDO
```

If you parallelize program `main` by column-wise block distribution as shown in Figure 75 (a), each process needs to get its left neighbor's boundary elements. In addition, the process has to use those boundary elements *after* its left neighbor process has updated them.

Program main2 below has less complex dependence as shown in Figure 75 (b). If you don't have to consider another part of the program, you should divide the matrix in row-wise block distribution so that the flow dependence is confined within each process. But sometimes you may have to use column-wise block distribution considering the performance of another part of the program.

```
PROGRAM main2
PARAMETER (mx = ..., my = ...)
DIMENSION x(mx,0:my)
...
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j) + x(i,j-1)
  ENDDO
ENDDO
...
```

The following discusses how program main is parallelized using the pipeline method. Suppose that you have three processes and matrix x() is assigned to processes using column-wise block distribution (Figure 76 on page 82 (a)). The idea is that within each process you divide the block by the row into even smaller blocks. In the figure, a block size of two (iblock=2) is chosen, and the blocks with the same number are executed in parallel as you see below.

Because of the dependence, each process cannot update the matrix elements at once. Instead, at first, process 0 does the computation of the first small block (block 1 in the left-most column), which provides the data necessary for process 1 to do the computation of block 2 in the middle column. While process 0 is working on block 1, processes 1 and 2 are just waiting, that is, the degree of parallelism is one.

Next, process 0 sends the boundary elements of block 1 to process 1. The number of elements to be sent is iblock. Then process 0 works on block 2 in the left-most column, and at the same time, process 1 does block 2 in the middle column. Therefore, the degree of parallelism is two. Likewise, in the next step, after process 0 sends iblock boundary elements of block 2 to process 1, and process 1 sends iblock boundary elements of block 2 to process 2, all the three processes start processing block 3 lying in each column. At this time, the degree of parallelism becomes three.

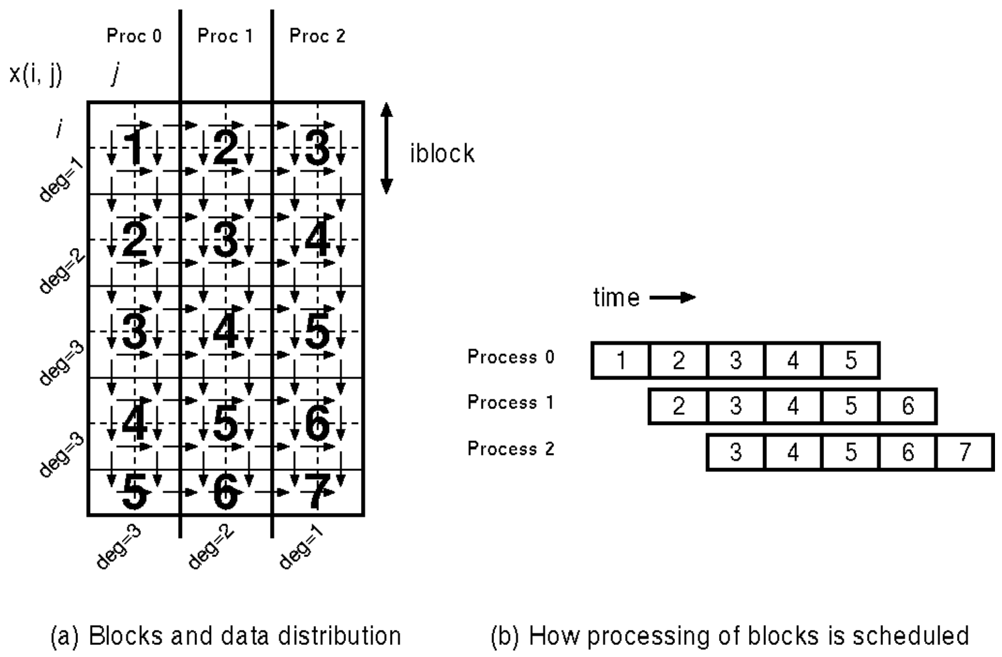


Figure 76. The Pipeline Method

After blocks 5 are processed, the degree of parallelism decreases to two (block 6) and then to one (block 7). The following is the parallelized version of program `main`. Note the way the processes proceed in the pipelined fashion. Each process waits for data to be sent from the left neighbor, does computation, and then sends data to the right neighbor. By use of MPI_PROC_NULL as the destination, process 0 and process `nprocs-1` are treated appropriately. In particular, process 0 can start its job without waiting for data to be delivered to it.

```
      PROGRAM mainp
      INCLUDE 'mpif.h'
      PARAMETER (mx = ..., my = ...)
      DIMENSION x(0:mx,0:my)
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(1, my, nprocs, myrank, jsta, jend)
      inext = myrank + 1
      IF (inext == nprocs) inext = MPI_PROC_NULL
      iprev = myrank - 1
      IF (iprev == -1) iprev = MPI_PROC_NULL
      iblock = 2
      ...
      DO ii = 1, mx, iblock
        iblklen = MIN(iblock, mx - ii + 1)
        CALL MPI_IRECV(x(ii, jsta - 1), iblklen, MPI_REAL,
     &                 iprev, 1, MPI_COMM_WORLD, ireqr, ierr)
        CALL MPI_WAIT(ireqr, istatus, ierr)
        DO j = jsta, jend
          DO i = ii, ii + iblklen - 1
            x(i,j) = x(i,j) + x(i-1,j) + x(i,j-1)
          ENDDO
        ENDDO
        CALL MPI_ISEND(x(ii,jend), iblklen, MPI_REAL,
     &                 inext, 1, MPI_COMM_WORLD, ireqs, ierr)
        CALL MPI_WAIT(ireqs, istatus, ierr)
      ENDDO
      ...
```

The above program is best understood by putting yourself in the place of the process and by thinking what you have to do, rather than by playing the role of the conductor who orchestrates the processes. One remark about the program: the calls of MPI_WAIT immediately follow MPI_IRECV and MPI_ISEND, which is logically equivalent to using blocking subroutines. Is there any danger of deadlocks? (See "Case 2. Receive first and then send" on page 27.) The answer is *no* , because there is always one process (process 0) that does not receive but only sends.

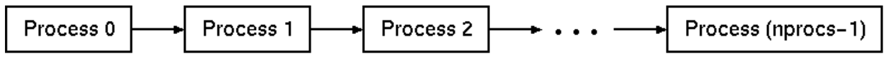


Figure 77. Data Flow in the Pipeline Method

In Figure 77, data transmissions are completed from left to right. In an open string topology like this, there will not be a deadlock as long as one-directional communication is concerned. On the other hand, in a closed string topology, deadlock may occur even in the case of one-directional communication. (See 3.5.7, "The Twisted Decomposition" on page 83.)

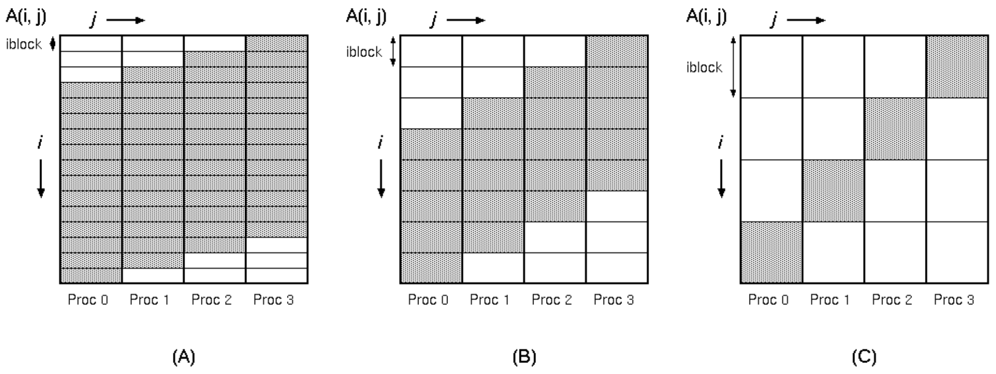


Figure 78. Block Size and the Degree of Parallelism in Pipelining

In choosing the block size (`iblock`), you should be aware of the trade-off between the number of data transmissions and the average degree of parallelism. In Figure 78, the shaded blocks mean that they can be processed by the maximum degree of parallelism. Apparently, smaller block size implies a higher degree of parallelism. On the other hand, the smaller the block size is, the more often the data transmissions will take place, thus higher communication overhead due to latency. You may have to find an optimal block size by trial and error.

3.5.7 The Twisted Decomposition

In the following program, Loop A is flow dependent on the first dimension of matrix `x()`, and Loop B on the second dimension (See also Figure 79 (a) and (b)). The two types of loops appear in the program alternately, which can be seen in the ADI (alternating direction implicit) method. Whether you distribute the matrix row-wise or column-wise, you cannot escape from the dependences on both dimensions.

```
      PROGRAM main
      PARAMETER (mx = ..., my = ...)
      REAL x(0:mx,0:my)
      ...
! Loop A
      DO j = 1, my
        DO i = 1, mx
          x(i,j) = x(i,j) + x(i-1,j)
        ENDDO
      ENDDO
! Loop B
      DO j = 1, my
        DO i = 1, mx
          x(i,j) = x(i,j) + x(i,j-1)
        ENDDO
      ENDDO
      ...
      END
```

One way to parallelize this program is to use the pipeline method described in 3.5.6, “The Pipeline Method” on page 79, but the average degree of parallelism might not meet your performance requirement. Of course, you can increase the degree of parallelism by using a smaller block size in the pipeline method, but it makes the communication overhead increase. The twisted decomposition method described below allows all processes to start computations at once without any stand-by time for some processes, thereby it provides better load balance than the pipeline method. The downside is that it is difficult and time-consuming to write the program because the data is distributed to processes in a complex way.

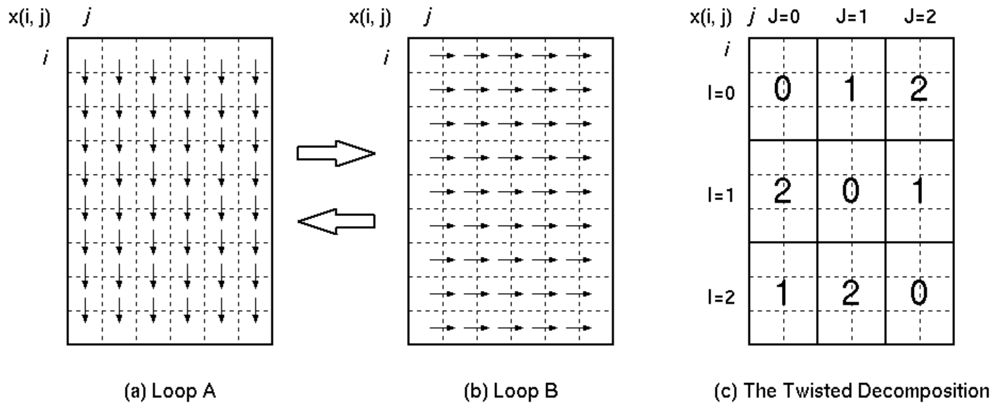


Figure 79. The Twisted Decomposition

In applying the twisted decomposition method, rows and columns of the two-dimensional array `x()` are divided into `nprocs` blocks, thus making `nprocs`^{2} blocks, where `nprocs` is the number of processes. Apply the block coordinate system `(I, J)` in order to identify the location of blocks (Figure 79 (c)). In the twisted decomposition, block `(I, J)` is assigned to process `(I - J + nprocs)` modulo `nprocs`. In Loop A, all processes do computation of the row `I=0`, then `I=1`, and finally `I=2`. In Loop B, all processes do computation of the column `J=0`, then `J=1`, and finally `J=2`. Before proceeding to the next block, each process sends and receives boundary elements to/from neighboring processes. Note that every row and every column in the block coordinate system contains all ranks so that the degree of parallelism is always the number of processes. The following is the parallelized version of the program.

```
      PROGRAM main
      INCLUDE 'mpif.h'
      INTEGER istatus(MPI_STATUS_SIZE)
      INTEGER, ALLOCATABLE :: is(:), ie(:), js(:), je(:)
      PARAMETER (mx = ..., my = ..., m = ...)
      DIMENSION x(0:mx,0:my)
      DIMENSION bufs(m),bufr(m)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      ALLOCATE (is(0:nprocs-1), ie(0:nprocs-1))
      ALLOCATE (js(0:nprocs-1), je(0:nprocs-1))
      DO ix = 0, nprocs - 1
        CALL para_range(1, mx, nprocs, ix, is(ix), ie(ix))
        CALL para_range(1, my, nprocs, ix, js(ix), je(ix))
      ENDDO
      inext = myrank + 1
      IF (inext == nprocs) inext = 0
      iprev = myrank - 1
      IF (iprev == -1) iprev = nprocs - 1
      ...
! Loop A
      DO ix = 0, nprocs - 1
        iy = MOD(ix + myrank, nprocs)
        ista = is(ix)
        iend = ie(ix)
        jsta = js(iy)
        jend = je(iy)
        jlen = jend - jsta + 1
        IF (ix /= 0) THEN
          CALL MPI_IRECV(bufr(jsta), jlen, MPI_REAL, inext, 1,
     &                   MPI_COMM_WORLD, ireqr, ierr)
          CALL MPI_WAIT(ireqr, istatus, ierr)
          CALL MPI_WAIT(ireqs, istatus, ierr)
          DO j = jsta, jend
            x(ista-1,j) = bufr(j)
          ENDDO
        ENDIF
        DO j = jsta, jend
          DO i = ista, iend
            x(i,j) = x(i,j) + x(i-1,j)
          ENDDO
        ENDDO
        IF (ix /= nprocs - 1) THEN
          DO j = jsta, jend
            bufs(j) = x(iend,j)
          ENDDO
          CALL MPI_ISEND(bufs(jsta), jlen, MPI_REAL, iprev, 1,
     &                   MPI_COMM_WORLD, ireqs, ierr)
        ENDIF
      ENDDO
! Loop B
      DO iy = 0, nprocs - 1
        ix = MOD(iy + nprocs - myrank, nprocs)
        ista = is(ix)
        iend = ie(ix)
        jsta = js(iy)
        jend = je(iy)
        ilen = iend - ista + 1
        IF (iy /= 0) THEN
          CALL MPI_IRECV(x(ista,jsta-1), ilen, MPI_REAL, iprev, 1,
     &                   MPI_COMM_WORLD, ireqr, ierr)
          CALL MPI_WAIT(ireqr, istatus, ierr)
          CALL MPI_WAIT(ireqs, istatus, ierr)
        ENDIF
        DO j = jsta, jend
          DO i = ista, iend
            x(i,j) = x(i,j) + x(i,j-1)
          ENDDO
        ENDDO
        IF (iy /= nprocs - 1) THEN
          CALL MPI_ISEND(x(ista,jend), ilen, MPI_REAL, inext, 1,
     &                   MPI_COMM_WORLD, ireqs, ierr)
        ENDIF
      ENDDO
      DEALLOCATE (is, ie)
      DEALLOCATE (js, je)
      ...
      CALL MPI_FINALIZE(ierr)
      END
```

Be careful about the location of the MPI_WAIT statements (shown in bold face in the program) corresponding to MPI_ISEND, which avoids deadlock. First, confirm that data transmissions take place correctly among processes. Consider Loop B, for example. The structure of Loop B is:

```
! Loop B
      DO iy = 0, nprocs - 1
        ...
        IF (iy /= 0) THEN
          Receive
          Wait for receive to complete
          Wait for send to complete
        ENDIF
        ...
        IF (iy /= nprocs - 1) THEN
          Send
        ENDIF
      ENDDO
```

In this example, there is a chain where a certain process always receives data from a specific process that points to it, as shown in Figure 80, and always sends data to a specific process which is pointed to by that process in the figure.

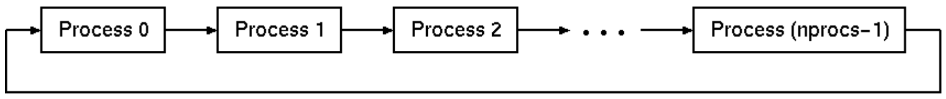
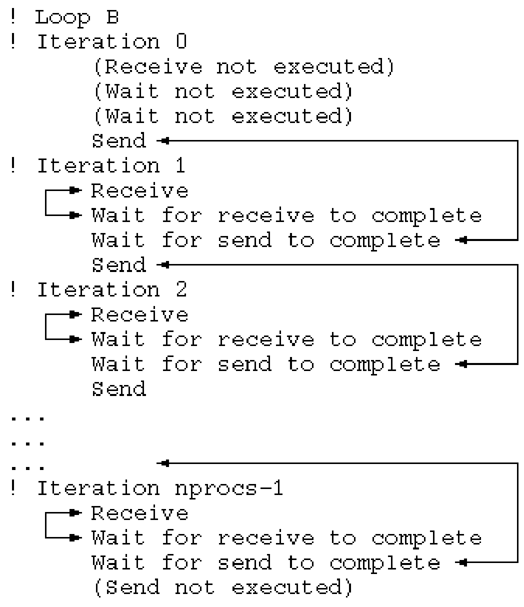


Figure 80. Data Flow in the Twisted Decomposition Method

The above loop is expanded into flat statements as shown in Figure 81. It is easy to see that the program works correctly and that deadlocks will not occur.



```
! Loop B
! Iteration 0
      (Receive not executed)
      (Wait not executed)
      (Wait not executed)
      Send
! Iteration 1
      Receive
      Wait for receive to complete
      Wait for send to complete
      Send
! Iteration 2
      Receive
      Wait for receive to complete
      Wait for send to complete
      Send
...
...
...
! Iteration nprocs-1
      Receive
      Wait for receive to complete
      Wait for send to complete
      (Send not executed)
```

Figure 81. Loop B Expanded

Next, consider what happens if MPI_WAIT is called just after MPI_ISEND. In that case, essentially every process calls blocking subroutines MPI_SEND and MPI_RECV, in this order, which may cause deadlock due to a shortage of the system buffer for sending the data. See the arguments in “Case 1. Send first and then receive” on page 26. The root cause of the problem is that all processes do blocking send and then blocking receive in this order and that the topology of processes is closed in terms of data flow (see Figure 80). These two conditions constitute the possible deadlocks. Note that in the pipeline method (refer 3.5.6, “The Pipeline Method” on page 79), the latter condition is not satisfied, that is, the process topology is open.

3.5.8 Prefix Sum

In the pipeline method and the twisted decomposition method, the loops were nested. Therefore, you have a chance to parallelize. But, what if a non-nested loop has a loop-carried dependence? In one dimension, the pipeline method is only a serial processing.

```
      PROGRAM main
      PARAMETER (n = ...)
      REAL a(0:n), b(n)
      ...
      DO i = 1, n
        a(i) = a(i-1) + b(i)
      ENDDO
      ...
```

The dependence involved in the above program is illustrated in Figure 82 on page 88. You cannot get parallel speed-up just by distributing matrices `a()`.

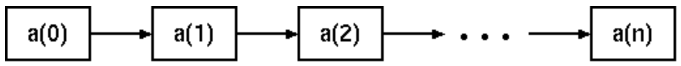


Figure 82. Loop-Carried Dependence in One Dimension

The above program can be parallelized by the technique of Prefix Sum. Here is how it works.

On the exit of the loop, the array `a()` has the following values:

```
a(1) = a(0) + b(1)
a(2) = a(0) + b(1) + b(2)
a(3) = a(0) + b(1) + b(2) + b(3)
...
a(n) = a(0) + b(1) + b(2) + b(3) + ... + b(n)
```

So, the loop is equivalent to calculating $a_k = a_0 + \sum_{1 \le i \le k} b_i$ for $1 \le k \le n$, thus the name of prefix sum.

The operation does not need to be addition. In general, a loop with the structure:

```
    DO i = 1, n
      a(i) = a(i-1) op b(i)
    ENDDO
```

where *op* is a binary operator, can be parallelized by the prefix sum technique, if the combination of the data type of `a()` and the operator *op* is one of the predefined combinations for MPI_REDUCE (or MPI_SCAN). See Table 11 on page 181 for the list of combinations. Allowable combinations may be extended by including user-defined operators (“MPI_OP_CREATE” on page 187) but further discussion is omitted here.

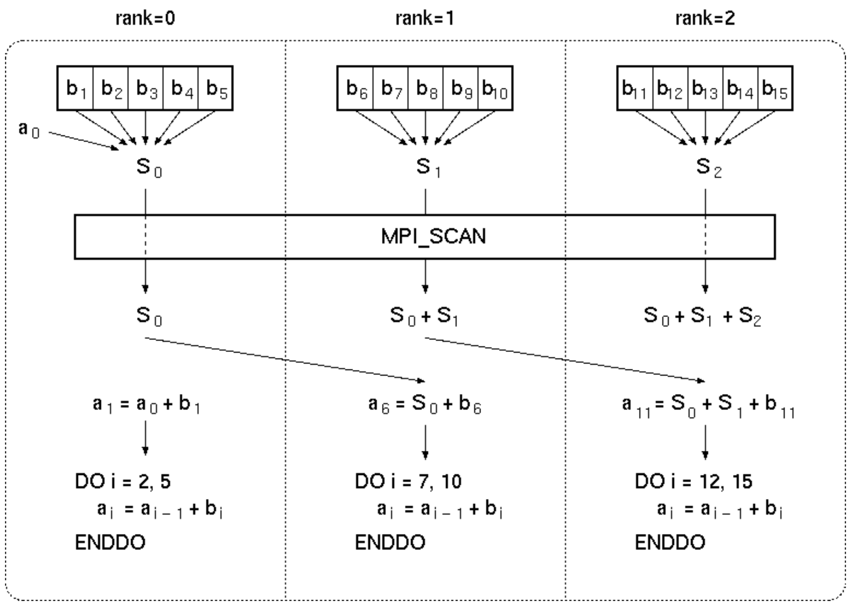


Figure 83. Prefix Sum

In the prefix sum method, the arrays `a()` and `b()` are block-distributed to processes and each process computes a partial sum of the array `b()`. The prefix sum of these partial sums is obtained by the subroutine MPI_SCAN, and it is used in calculating the starting element of the array by each process. Figure 83 shows a desirable behavior of processes and data, where the prefix sum obtained by MPI_SCAN is shifted among processes. But in the following program, this data transmission is replaced by a local calculation. For example, the process with rank 1 in Figure 83 has the values of S_1 and (S_0+S_1) after MPI_SCAN, so instead of getting the value of S_0 from process 0, process 1 calculates it as (S_0+S_1)-S_1.

```
       PROGRAM main
       INCLUDE 'mpif.h'
       PARAMETER (n = ...)
       REAL a(0:n), b(n)
       CALL MPI_INIT(ierr)
       CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
       CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
       CALL para_range(1, n, nprocs, myrank, ista, iend)
       ...
       sum = 0.0
       DO i = ista, iend
         sum = sum + b(i)
       ENDDO
       IF (myrank == 0) THEN
         sum = sum + a(0)
       ENDIF
       CALL MPI_SCAN(sum, ssum, 1, MPI_REAL, MPI_SUM,
      &              MPI_COMM_WORLD, ierr)
       a(ista) = b(ista) + ssum - sum
       IF (myrank == 0) THEN
         a(ista) = a(ista) + a(0)
       ENDIF
       DO i = ista+1, iend
         a(i) = a(i-1) + b(i)
       ENDDO
       ...
```

Note that in the parallelized program, the total number of additions is roughly doubled compared with the serial program because there are two DO loops, one before and after the call of MPI_SCAN. Therefore, if the number of processes is small, the prefix sum method is not effective.

3.6 Considerations in Parallelization

So far, the basic techniques of parallelizing programs have been discussed, concentrating on local structures such as DO loops. In the remainder of this chapter, advice and tips are given in the global point of view of parallelization.

3.6.1 Basic Steps of Parallelization

You may want to parallelize an existing serial program, or you may want to write a parallel program from scratch. Here are the basic steps required to parallelize a serial program.

1. Tune the serial program

Before parallelizing the program, tune the hot spots. Hot spots can be identified by the `prof`, `gprof`, or `tprof` command. For more information on serial tuning, see *AIX Version 4 Optimization and Tuning Guide for FORTRAN, C, and C++*, SC09-1705, *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, SG24-5155, or K. Dowd and C. Severance, *High Performance Computing (2nd Edition)*, O'Reilly (1998).

2. Consider the outline of parallelization and the target performance

   Get the profile of the tuned serial program. If the program has a sharp profile, that is, only a small part of your program is consuming most of the CPU time, it might be sufficient to parallelize only that part. On the other hand, if the program has a flat profile, that is, there are no predominant routines and CPU time is consumed evenly throughout the program, you may have to parallelize the overall program. Remind yourself of the patterns of parallelization described in 3.2, "Three Patterns of Parallelization" on page 46. At the same time, estimate by using the Amdahl's law how much faster the program becomes when parallelized (3.1, "What is Parallelization?" on page 41), and check if the estimated running time meets your requirements. For this purpose, you have to know how much of the program can be run in parallel and how much communication overhead will be introduced by parallelization. Some experiments may have to be done in order to get the latter figure.

3. Determine your strategy for parallelization

   - For nested loops, which loop to parallelize and how? (Block distribution, cyclic distribution, and so forth.)
   - Which scalar variables and arrays must be transmitted
   - Whether to shrink arrays or not
   - Whether to replace part of the program by Parallel ESSL subroutines

4. Parallelize the program

   Use the techniques provided previously in this chapter. The following describes two more techniques; The use of MODULE statements of Fortran 90 and an incremental parallelization method.

MODULE statements

Suppose there are a number of subroutines that are involved in parallelization. By using MODULE statements, you can simplify the passing of arguments related to parallelization. For example, consider the serial program below.

```
PROGRAM main
...
n = ...
CALL sub1(n)
CALL sub2(n)
...
END

SUBROUTINE sub1(n)
...
DO i = 1, n
  ...
ENDDO
...
END
```

```
SUBROUTINE sub2(n)
...
DO i = 1, n
  ...
ENDDO
...
END
```

This program is parallelized using a module as follows.

```
MODULE para
INCLUDE 'mpif.h'
INTEGER nprocs, myrank, ista, iend
INTEGER istatus(MPI_STATUS_SIZE)
END

PROGRAM main
USE para
...
n = ...
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, ista, iend)
CALL sub1(n)
CALL sub2(n)
...
CALL MPI_FINALIZE(ierr)
END

SUBROUTINE sub1(n)
USE para
...
DO i = ista, iend
  ...
ENDDO
...
END

SUBROUTINE sub2(n)
USE para
...
DO i = ista, iend
  ...
ENDDO
...
END
```

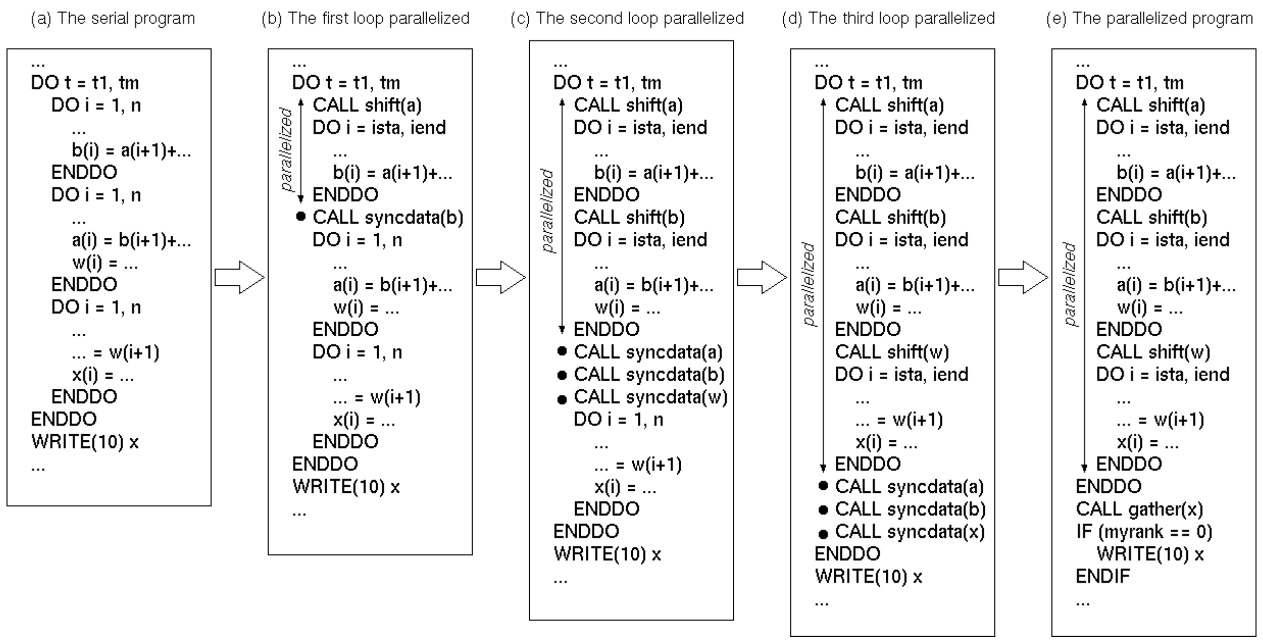
An advantage of modules is that, unlike COMMON blocks, modules can contain allocatable arrays. IMPLICIT declaration in a MODULE block is only effective to the variables within the block. It is not inherited to the procedures that use the MODULE block. In the program, the MODULE block should appear before the procedures that use it. The USE statement can only appear prior to all other statements except for comment lines in the procedure. See the Fortran language reference for details.

Incremental parallelization

Suppose that you are going to parallelize the code below. The outer DO loop is ticking the time steps, which encloses three DO loops that contribute to the running time almost equally. After the time-ticking loop, the contents of array `x()` is written to a file. This kind of code is often seen in the finite difference method. Assume that there is no parallelism in the outer loop so that you have to parallelize all the inner loops as in “Pattern 2. Parallelization Throughout a DO Loop” on page 47. Array `w()` is only used as a working area and does not carry data beyond time steps. Also assume that if the arrays are block-distributed to processes, the data dependence is such that only boundary elements need to be exchanged between adjacent processes.

```
...
DO t = t1, tm
  DO i = 1, n
    ...
    b(i) = a(i+1) + ...
  ENDDO
  DO i = 1, n
    ...
    a(i) = b(i+1) + ...
    w(i) = ...
  ENDDO
  DO i = 1, n
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
ENDDO
WRITE(10) x
...
```

A skillful programmer could parallelize the inner loops all at once and the parallelized program might give the correct answer. But if the answer is found to be wrong, it is quite difficult to debug the code. As an alternative approach, an incremental method is proposed below.


(a) The serial program
(b) The first loop parallelized
(c) The second loop parallelized
(d) The third loop parallelized
(e) The parallelized program
parallelized
parallelized
parallelized
parallelized

(a) The serial program
```
...
DO t = t1, tm
  DO i = 1, n
    ...
    b(i) = a(i+1)+...
  ENDDO
  DO i = 1, n
    ...
    a(i) = b(i+1)+...
    w(i) = ...
  ENDDO
  DO i = 1, n
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
ENDDO
WRITE(10) x
...
```

(b) The first loop parallelized
```
...
DO t = t1, tm
  CALL shift(a)
  DO i = ista, iend
    ...
    b(i) = a(i+1)+...
  ENDDO
● CALL syncdata(b)
  DO i = 1, n
    ...
    a(i) = b(i+1)+...
    w(i) = ...
  ENDDO
  DO i = 1, n
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
ENDDO
WRITE(10) x
...
```

(c) The second loop parallelized
```
...
DO t = t1, tm
  CALL shift(a)
  DO i = ista, iend
    ...
    b(i) = a(i+1)+...
  ENDDO
  CALL shift(b)
  DO i = ista, iend
    ...
    a(i) = b(i+1)+...
    w(i) = ...
  ENDDO
● CALL syncdata(a)
● CALL syncdata(b)
● CALL syncdata(w)
  DO i = 1, n
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
ENDDO
WRITE(10) x
...
```

(d) The third loop parallelized
```
...
DO t = t1, tm
  CALL shift(a)
  DO i = ista, iend
    ...
    b(i) = a(i+1)+...
  ENDDO
  CALL shift(b)
  DO i = ista, iend
    ...
    a(i) = b(i+1)+...
    w(i) = ...
  ENDDO
  CALL shift(w)
  DO i = ista, iend
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
● CALL syncdata(a)
● CALL syncdata(b)
● CALL syncdata(x)
ENDDO
WRITE(10) x
...
```

(e) The parallelized program
```
...
DO t = t1, tm
  CALL shift(a)
  DO i = ista, iend
    ...
    b(i) = a(i+1)+...
  ENDDO
  CALL shift(b)
  DO i = ista, iend
    ...
    a(i) = b(i+1)+...
    w(i) = ...
  ENDDO
  CALL shift(w)
  DO i = ista, iend
    ...
    ... = w(i+1)
    x(i) = ...
  ENDDO
ENDDO
CALL gather(x)
IF (myrank == 0)
  WRITE(10) x
ENDIF
...
```

Figure 84. Incremental Parallelization

Instead of parallelizing all the loops at once, parallelize one loop at a time from top to bottom. After the parallelized loop, do some message-passing in order to make sure that in the unparallelized part of the program all processes have the up-to-date data.

In Figure 84 on page 92, these temporary message-passing statements are marked with a bullet, which will be removed as the parallelization proceeds. Take a look at Figure 84 (b), where the first loop is parallelized by the block distribution. By the time the first loop is executed, array `a()` is modified in the previous time step. Therefore, a presupposed subroutine `shift` is called before entering the first loop, which is not a temporary modification. After the loop, each process has the valid elements of array `b()` only in the assigned range. Since later in the same time step a process may need elements of `b()` calculated by another process, the array is synchronized among processes by use of a presupposed subroutine `syncdata`. Note that `syncdata` appears just after the parallelized part. For the implementation of `syncdata`, see 3.5.3.2, "Synchronizing Data" on page 73.

The incremental steps continues in this way until you parallelize all the loops (Figure 84 (b)-(d)). Note that you can check the correctness of the program after any step of this method by compiling and running the program. After you have parallelized all the inner DO loops, add a statement that gathers array `x()` to process 0 after the outer DO loop (Figure 84 (e)). Run the parallelized program then check the answer.

3.6.2 Trouble Shooting

The following describes typical symptoms and a list of things that might solve the problem.

Parallelized program does not start

- Are the environment variables and the command line options correct for parallel execution? (See Appendix A, "How to Run Parallel Jobs on RS/6000 SP" on page 155.)
- Is the communication network up and running? If you are using RS/6000 SP, has the SP switch network started, or are any nodes fenced?
- Is the job control subsystem working? On RS/6000 SP, has the root user issued `jm_start` (before Parallel Environment 2.3), or is LoadLeveler correctly configured and running?

Parallelized program ends abnormally

- Have you included `mpif.h` in the program that uses MPI subroutines?
- Are the arguments of all the MPI subroutine calls correct? Is the argument for the Fortran return code (`ierr`) missing?
- Have you defined status objects, which may be used in MPI_RECV, MPI_WAIT, and so on, as INTEGER istatus(MPI_STATUS_SIZE)?
- You can get more run-time information by raising the value of the MP_INFOLEVEL environment variable. As mentioned in A.5, "Standard Output and Standard Error" on page 158, the default value of MP_INFOLEVEL is 1, so specify 2 or larger.

Parallelized program starts but becomes inactive

- If the input from the standard input (stdin) exceeds 32 KB, set `MP_HOLD_STDIN=yes`. Otherwise, the program halts while reading from stdin.
- Are there any communication deadlocks? See the arguments in 2.4.3, “Bidirectional Communication” on page 26.

Parallelized program gives wrong answer

- Have you parallelized a loop that does not have parallelism? A very simple check, although not sufficient, is to see if the program gives the same answer when the iterations are reversed in the loop in question.
- If your executable is located on a local file system in the distributed environment, have you copied the latest executable to each node? You might be using an old executable with the same name.
- Are arrays distributed correctly to processes? For example, if the block distribution is used, check `ista` and `iend` for all processes. To show the standard output of all processes together with their ranks, specify `MP_LABELIO=yes` and `MP_STDOUTMODE=ordered` or `unordered`. See also A.5, “Standard Output and Standard Error” on page 158.
- If you are using non-blocking communication subroutines such as MPI_ISEND and MPI_IRECV, have you issued MPI_WAIT before reusing the communication buffer?
- Some numerical errors may arise due to parallelization because the order of computations might have changed. For example, if you use MPI_REDUCE to get a sum of local variables, the result may be different (within rounding errors) from what you get when you calculate it serially.
- Unless you pay special attention in parallelization, a program using random numbers will produce a different answer when parallelized. That’s because the sequence of random numbers may be different from what the serial program accesses. See also 4.5, “Monte Carlo Method” on page 131.

3.6.3 Performance Measurements

This section discusses how to measure the performance of parallel programs and how to evaluate them.

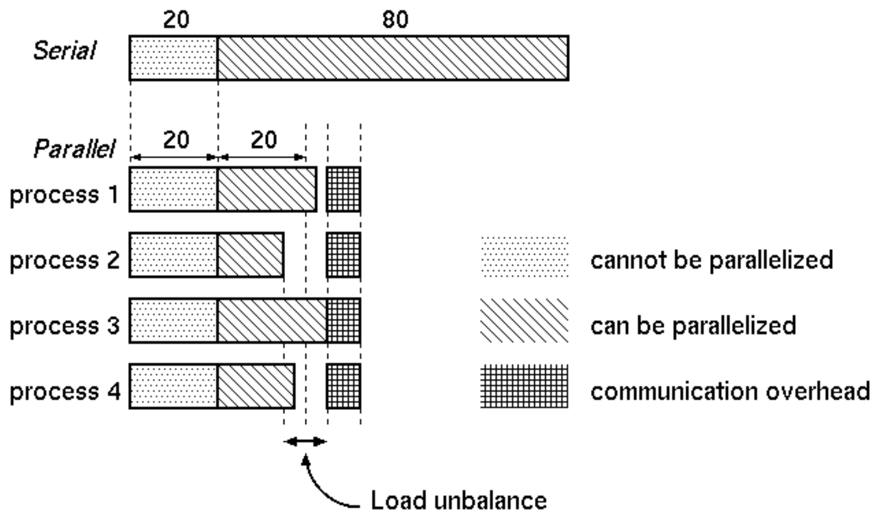


Figure 85. Parallel Speed-Up: An Actual Case

The above figure is a copy of Figure 31 on page 42. You need an index to measure how well parallel programs perform. In production applications, what matters is the elapsed time, so the following index is often used.

$$\text{Speed-up}(p) = \frac{\text{Elapsed time of the serial program}}{\text{Elapsed time of the parallel program with } p \text{ processes}}$$

Elapsed time can be measured by the built-in shell command `time` or the AIX command `timex`. Note that the dividend in the formula is not the elapsed time of the parallel program run with one process. Even executed with one process, parallel programs have overhead for initializing environment, such as calculating the range of loop variables.

In the ideal case, $\text{Speed-up}(p) = p$, but because part of the program might not be parallelized and the parallelized program has communication overhead (see Figure 85), the speed-up is usually less than *p*.

Important

Note that the speed-up ratio defined above is relative to the hardware specification that you are using. On different hardware, the ratio of the computation-dependent part and communication-dependent part in Figure 85 may change. For instance, consider a hardware that has the same CPU but a network that is twice as slow. In such a case, the speed-up ratio also varies, even if you run the same serial and parallel programs.

The previous note suggests that the speed-up ratio is not an absolute measure. Here is another situation. Suppose you have tuned a serial program `ser.f` and got `sertune.f`. Both programs are completely parallelizable and you have `para.f` for `ser.f` and `paratune.f` for `sertune.f`. Suppose that the serial tuning does not change the amount of communication of the parallelized program, so `para.f` and `paratune.f` involve the same amount of communication overhead. Therefore the impact of communication overhead is larger in `paratune.f` and the speed-up ratio is worse.

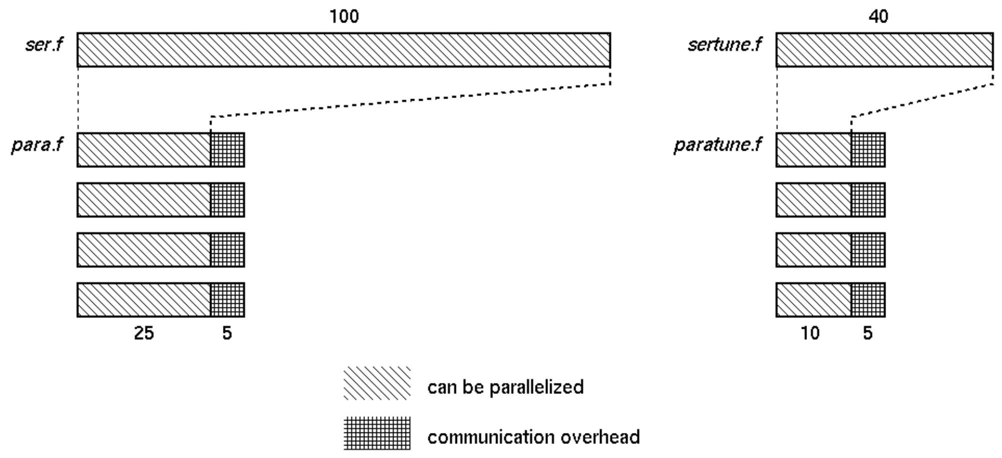


Figure 86. Speed-Up Ratio for Original and Tuned Programs

Figure 86 shows a concrete example. You tuned `ser.f` faster by a factor of 2.5. Both `ser.f` and `sertune.f` are completely parallelized, but with the same communication overhead. The speed-up ratio of a four-process run for `ser.f/para.f` is 100/(25+5)=3.33, whereas that of `sertune.f/paratune.f` is 40/(10+5)=2.67. The lesson from this example is, “Don’t focus on the speed-up ratio you’ve got. Stick to the elapsed time!”

When measuring the elapsed time of a parallel program, be aware that if the parallel executable is located on a shared file system, the time for loading the executable may vary significantly from process to process due to I/O and network contention.

When you are parallelizing a program, you may want to know the elapsed time of a particular section of the program. If your concern is the time spent by the parallel job as a whole, do as the following program shows.

```
PROGRAM main
REAL*8 elp1, elp2, rtc
...
CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)
elp1 = rtc()
This section is measured.
CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)
elp2 = rtc()
PRINT *, 'Elapsed time (sec) =', elp2 - elp1
...
END
```

The `rtc` intrinsic function of XL Fortran returns a REAL*8 value of the number of seconds since the initial value of the machine real-time clock.

If you are interested in the load balance of the processes, you have to measure the elapsed time for each process:

```
PROGRAM main
REAL*8 elp1, elp2, rtc
```

```
...
CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)
elp1 = rtc()
This section is measured.
elp2 = rtc()
PRINT *, 'Elapsed time (sec) =', elp2 - elp1
...
END
```

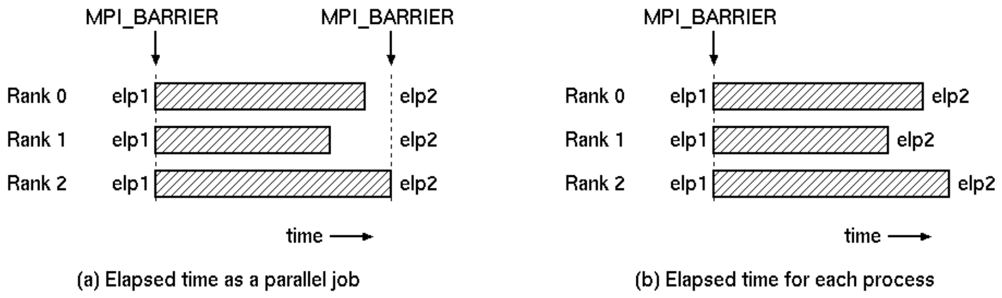


Figure 87. Measuring Elapsed Time

Figure 87 illustrates how each of the above time-measuring jobs work.

Chapter 4. Advanced MPI Programming

This chapter covers parallelizing programs that have more complex dependences and are closer to production applications than the programs that were discussed previously in this publication. As long as space permits, the source code is presented so that you can use parts of the code when you need to parallelize your own programs.

4.1 Two-Dimensional Finite Difference Method

Programs using the two-dimensional finite difference method (FDM) can be parallelized by generalizing the case discussed in Chapter 3.5.2, “One-Dimensional Finite Difference Method” on page 67. You have to decide how to distribute the matrices considering the amount of data transmitted and the continuity of matrix elements for transmission. See Chapter 3.4.5, “Parallelizing Nested Loops” on page 61 for details. The following program is considered for parallelization in this section. Different values of `m` and `n` may be chosen for each method of distribution.

```
PROGRAM main
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 6, n = 9)
DIMENSION a(m,n), b(m,n)
DO j = 1, n
  DO i = 1, m
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
DO j = 2, n - 1
  DO i = 2, m - 1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
END
```

The above program is a skeleton of the two-dimensional FDM, where coefficients and the enclosing loop for convergence are omitted. Data dependences exist in both dimensions. For a one-dimensional FDM, you usually decide whether to use column-wise block distribution or row-wise block distribution based on the shape of the matrix, because the distribution decides the amount of data exchanged between adjacent processes, which you want to minimize.

For a two-dimensional FDM, three ways of distributing the arrays are possible. Namely, column-wise block distribution, row-wise block distribution, and block distribution in both dimensions.

4.1.1 Column-Wise Block Distribution

In column-wise block distribution, the boundary elements between processes are contiguous in memory. In the parallelized program, a utility subroutine `para_range` (Chapter 3.4.1, “Block Distribution” on page 54) is called to calculate the range `jsta..jend` of each process. The program does not use the technique of a shrinking array. When you use it, each process needs to allocate `a(1:m, jsta-1:jend+1)` to accommodate data sent from neighboring processes. More strictly, `a(1:m, MAX(1,jsta-1):MIN(n,jend+1))` would be enough.

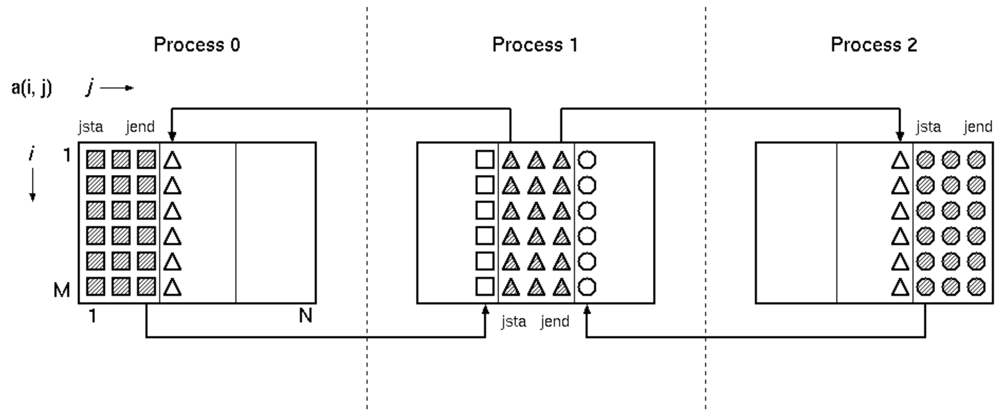


Figure 88. Two-Dimensional FDM: Column-Wise Block Distribution

```
PROGRAM main
INCLUDE 'mpif.h'
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 6, n = 9)
DIMENSION a(m,n), b(m,n)
INTEGER istatus(MPI_STATUS_SIZE)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, jsta, jend)
jsta2 = jsta
jend1 = jend
IF (myrank == 0)          jsta2 = 2
IF (myrank == nprocs - 1) jend1 = n - 1
inext = myrank + 1
iprev = myrank - 1
IF (myrank == nprocs - 1) inext = MPI_PROC_NULL
IF (myrank == 0)          iprev = MPI_PROC_NULL
DO j = jsta, jend
  DO i = 1, m
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
CALL MPI_ISEND(a(1,jend)  ,m,MPI_REAL8,inext,1,MPI_COMM_WORLD,isend1,ierr)
CALL MPI_ISEND(a(1,jsta)  ,m,MPI_REAL8,iprev,1,MPI_COMM_WORLD,isend2,ierr)
CALL MPI_IRECV(a(1,jsta-1),m,MPI_REAL8,iprev,1,MPI_COMM_WORLD,irecv1,ierr)
CALL MPI_IRECV(a(1,jend+1),m,MPI_REAL8,inext,1,MPI_COMM_WORLD,irecv2,ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
DO j = jsta2, jend1
  DO i = 2, m - 1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

If there are several matrices that have the same dependence as `a()`, you should write a subroutine for the data transmission and replace the statements printed in italics in the above program by a call to that subroutine.

4.1.2 Row-Wise Block Distribution

In row-wise block distribution, the boundary elements between processes are not contiguous in memory. You should either use derived data types representing the boundary elements, or write code for packing data, sending/receiving it, and unpacking it. The former option may produce a clean code that is easy to read and maintain, but as mentioned in Chapter 2.5, “Derived Data Types” on page 28, using derived data types may result in less performance than the hand-coded program. In the sample program that follows, the hand-coded option is chosen.

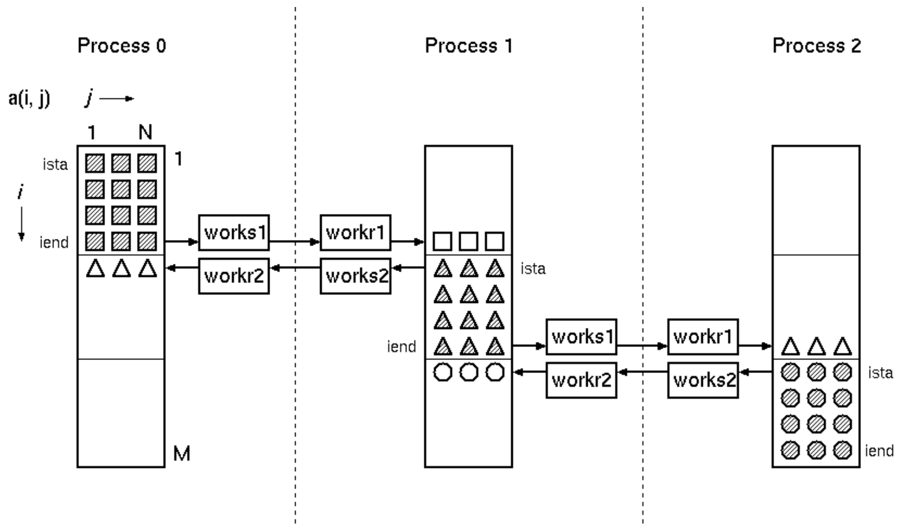


Figure 89. Two-Dimensional FDM: Row-Wise Block Distribution

```
PROGRAM main
INCLUDE 'mpif.h'
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 12, n = 3)
DIMENSION a(m,n), b(m,n)
DIMENSION works1(n), workr1(n), works2(n), workr2(n)
INTEGER istatus(MPI_STATUS_SIZE)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, m, nprocs, myrank, ista, iend)
ista2 = ista
iend1 = iend
IF (myrank == 0)          ista2 = 2
IF (myrank == nprocs - 1) iend1 = m-1
inext = myrank + 1
iprev = myrank - 1
IF (myrank == nprocs - 1) inext = MPI_PROC_NULL
IF (myrank == 0)          iprev = MPI_PROC_NULL
DO j = 1, n
  DO i = ista, iend
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
IF (myrank /= nprocs - 1) THEN
  DO j = 1, n
    works1(j) = a(iend,j)
  ENDDO
ENDIF
IF (myrank /= 0) THEN
  DO j = 1, n
    works2(j) = a(ista,j)
  ENDDO
ENDIF
CALL MPI_ISEND(works1,n,MPI_REAL8,inext,1,MPI_COMM_WORLD,isend1,ierr)
CALL MPI_ISEND(works2,n,MPI_REAL8,iprev,1,MPI_COMM_WORLD,isend2,ierr)
CALL MPI_IRECV(workr1,n,MPI_REAL8,iprev,1,MPI_COMM_WORLD,irecv1,ierr)
CALL MPI_IRECV(workr2,n,MPI_REAL8,inext,1,MPI_COMM_WORLD,irecv2,ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
IF (myrank /= 0) THEN
  DO j = 1, n
    a(ista-1,j) = workr1(j)
  ENDDO
ENDIF
IF (myrank /= nprocs - 1) THEN
  DO j = 1, n
    a(iend+1,j) = workr2(j)
  ENDDO
ENDIF
DO j = 2, n - 1
  DO i = ista2, iend1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

4.1.3 Block Distribution in Both Dimensions (1)

As illustrated in Figure 63 on page 66, the amount of data transmitted might be minimized when you divide the matrix in both dimensions. Therefore, you must take into account the size of the matrix and the number of processes to find the best way to divide the matrix. In using this distribution, it is often the case that you know the number of processes in advance. In the sample program, the number of processes is assumed to be nine, and the program explicitly uses this value.

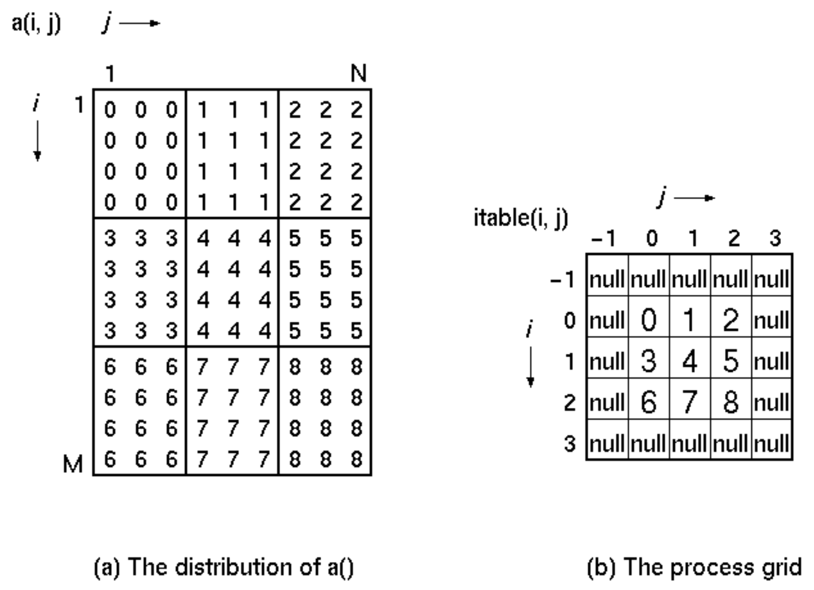


Figure 90. Two-Dimensional FDM: The Matrix and the Process Grid

Matrix `a()` is distributed to nine processes as shown in Figure 90 (a), where the number in the matrix indicates the rank of the process that the element belongs to. For looking up adjacent processes quickly, a process grid matrix `itable()` is prepared. In Figure 90 (b), null stands for MPI_PROC_NULL, which means that if a message is directed to it, actual data transmission will not take place. In the program, each process has its coordinate `(myranki,myrankj)` in the process grid. For example, the coordinate of process 7 is `(2,1)`. Each process can find its neighbors by accessing `itable(myranki±1 , myrankj±1 )`.

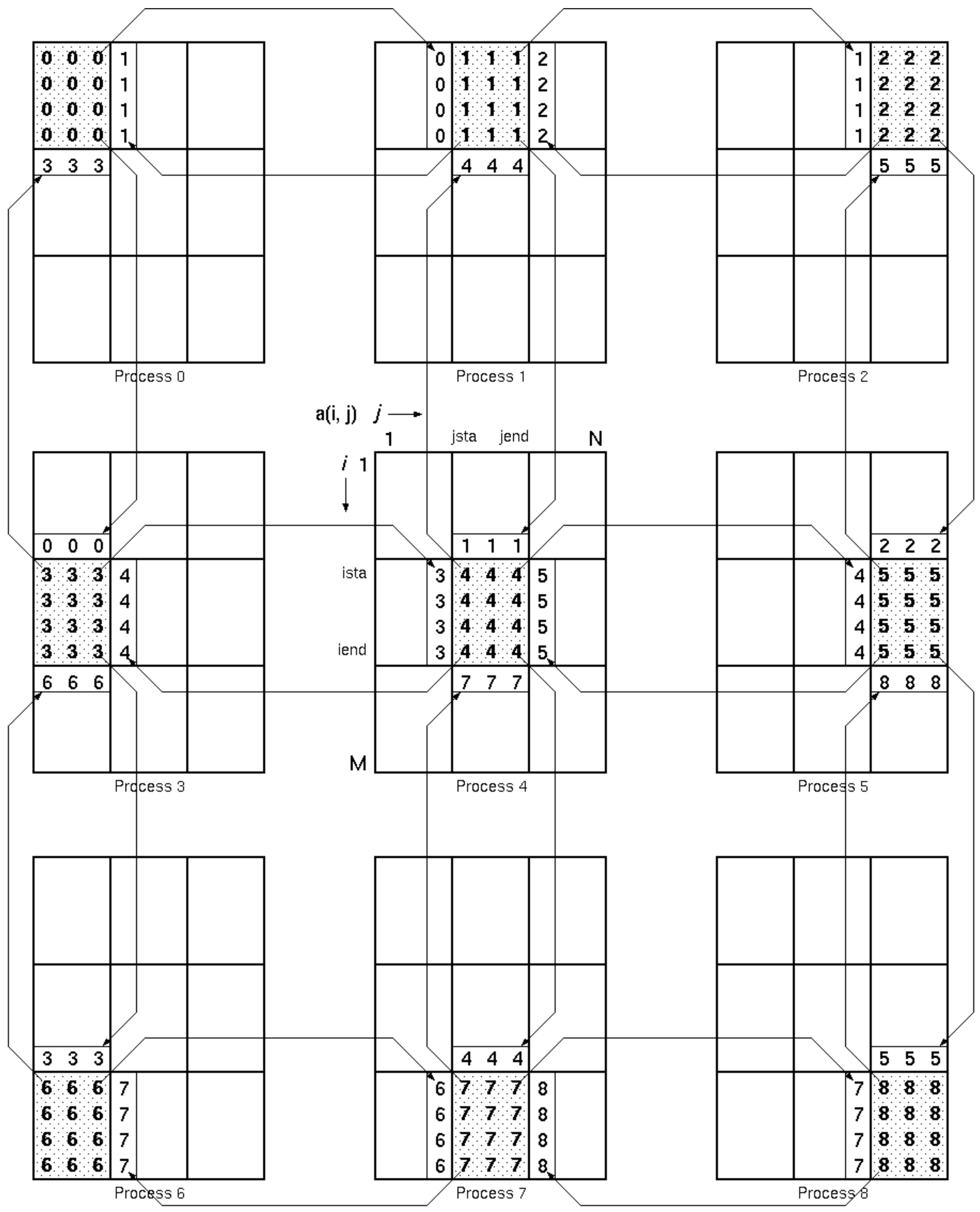


Figure 91. Two-Dimensional FDM: Block Distribution in Both Dimensions (1)

Figure 91 illustrates how boundary elements are exchanged among processes.

```
PROGRAM main
INCLUDE 'mpif.h'
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 12, n = 9)
DIMENSION a(m,n), b(m,n)
DIMENSION works1(n), workr1(n), works2(n), workr2(n)
INTEGER istatus(MPI_STATUS_SIZE)
```

```
PARAMETER (iprocs = 3, jprocs = 3)
INTEGER itable(-1:iprocs, -1:jprocs)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (nprocs /= iprocs * jprocs) THEN
  PRINT *,'=== Error ==='
  STOP
ENDIF
DO j = -1, jprocs
  DO i = -1, iprocs
    itable(i,j) = MPI_PROC_NULL
  ENDDO
ENDDO
irank = 0
DO i = 0, iprocs - 1
  DO j = 0, jprocs - 1
    itable(i,j) = irank
    IF (myrank == irank) THEN
      myranki = i
      myrankj = j
    ENDIF
    irank = irank + 1
  ENDDO
ENDDO
CALL para_range(1, n, jprocs, myrankj, jsta, jend)
jsta2 = jsta
jend1 = jend
IF (myrankj == 0)          jsta2 = 2
IF (myrankj == jprocs - 1) jend1 = n - 1
CALL para_range(1, m, iprocs, myranki, ista, iend)
ista2 = ista
iend1 = iend
IF (myranki == 0)          ista2 = 2
IF (myranki == iprocs - 1) iend1 = m - 1
ilen  = iend - ista + 1
jlen  = jend - jsta + 1
jnext = itable(myranki,     myrankj + 1)
jprev = itable(myranki,     myrankj - 1)
inext = itable(myranki + 1, myrankj)
iprev = itable(myranki - 1, myrankj)
DO j = jsta, jend
  DO i = ista, iend
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
IF (myranki /= iprocs - 1) THEN
  DO j = jsta, jend
    works1(j) = a(iend,j)
  ENDDO
ENDIF
IF (myranki /= 0) THEN
  DO j = jsta, jend
    works2(j) = a(ista,j)
  ENDDO
ENDIF
CALL MPI_ISEND(a(ista,jend),ilen,MPI_REAL8,jnext,1,MPI_COMM_WORLD,isend1,ierr)
CALL MPI_ISEND(a(ista,jsta),ilen,MPI_REAL8,jprev,1,MPI_COMM_WORLD,isend2,ierr)
CALL MPI_ISEND(works1(jsta),jlen,MPI_REAL8,inext,1,MPI_COMM_WORLD,jsend1,ierr)
CALL MPI_ISEND(works2(jsta),jlen,MPI_REAL8,iprev,1,MPI_COMM_WORLD,jsend2,ierr)
CALL MPI_IRECV(a(ista,jsta-1),ilen,MPI_REAL8,jprev,1,MPI_COMM_WORLD,irecv1,ierr)
CALL MPI_IRECV(a(ista,jend+1),ilen,MPI_REAL8,jnext,1,MPI_COMM_WORLD,irecv2,ierr)
CALL MPI_IRECV(workr1(jsta)  ,jlen,MPI_REAL8,iprev,1,MPI_COMM_WORLD,jrecv1,ierr)
CALL MPI_IRECV(workr2(jsta)  ,jlen,MPI_REAL8,inext,1,MPI_COMM_WORLD,jrecv2,ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(jsend1, istatus, ierr)
CALL MPI_WAIT(jsend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
CALL MPI_WAIT(jrecv1, istatus, ierr)
CALL MPI_WAIT(jrecv2, istatus, ierr)
IF (myranki /= 0) THEN
  DO j = jsta, jend
    a(ista-1,j) = workr1(j)
  ENDDO
ENDIF
IF (myranki /= iprocs - 1) THEN
```

```
      DO j = jsta, jend
        a(iend+1,j) = workr2(j)
      ENDDO
    ENDIF
    DO j = jsta2, jend1
      DO i = ista2, iend1
        b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
      ENDDO
    ENDDO
    CALL MPI_FINALIZE(ierr)
    END
```

Note that in the vertical data transmissions, non-contiguous matrix elements are first copied to buffers and then transmitted.

4.1.4 Block Distribution in Both Dimensions (2)

The following program has more complex dependence, that is, the value of `b(i,j)` is calculated from eight neighbors including four diagonals.

```
      PROGRAM main
      IMPLICIT REAL*8 (a-h,o-z)
      PARAMETER (m = 12, n = 9)
      DIMENSION a(m,n), b(m,n)
      DO j = 1, n
        DO i = 1, m
          a(i,j) = i + 10.0 * j
        ENDDO
      ENDDO
      DO j = 2, n - 1
        DO i = 2, m - 1
          b(i,j) = a(i-1,j  ) + a(i,  j-1) + a(i,  j+1) + a(i+1,j  )
     &          + a(i-1,j-1) + a(i+1,j-1) + a(i-1,j+1) + a(i+1,j+1)
        ENDDO
      ENDDO
      END
```

The dependence is illustrated in Figure 92.

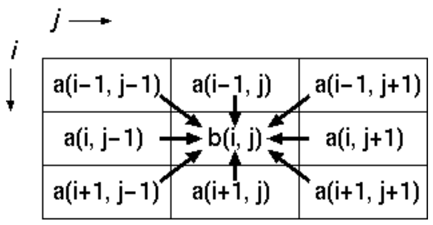


Figure 92. Dependence on Eight Neighbors

If the matrices are divided in both dimensions, a process needs data of cater-cornered processes. You can add four sends and four receives in the program given in 4.1.3, “Block Distribution in Both Dimensions (1)” on page 102, but here the diagonal elements are copied in two steps without calling additional communication subroutines. Figure 93 on page 106 shows how boundary elements are gathered to process 4.

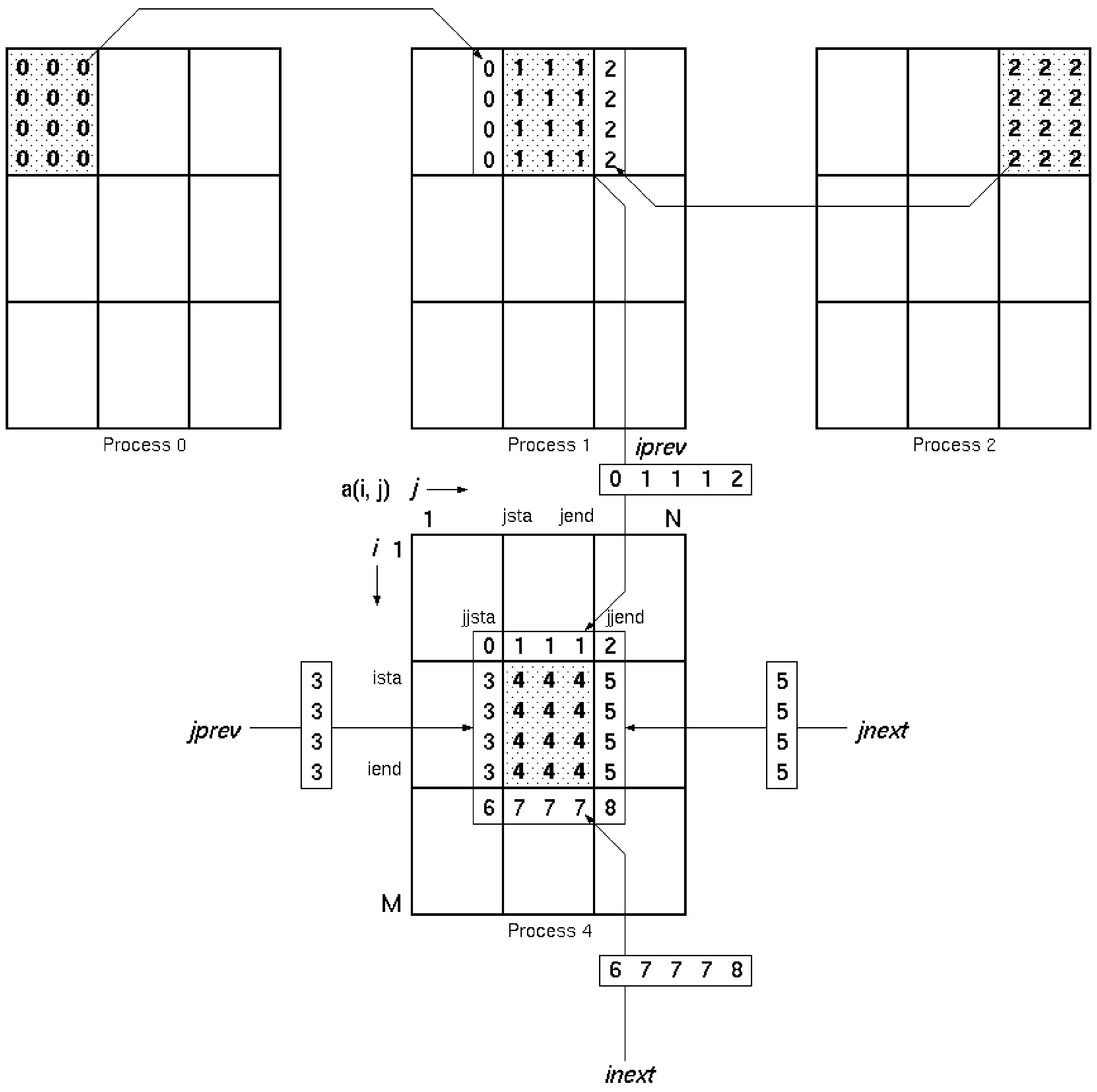


Figure 93. Two-Dimensional FDM: Block Distribution in Both Dimensions (2)

In the parallelized program, horizontal data transmissions take place first and then vertical transmissions follow.

```
      PROGRAM main
      INCLUDE 'mpif.h'
      IMPLICIT REAL*8 (a-h,o-z)
      PARAMETER (m = 12, n = 9)
      DIMENSION a(m,n), b(m,n)
      DIMENSION works1(n), workr1(n), works2(n), workr2(n)
      INTEGER istatus(MPI_STATUS_SIZE)
      PARAMETER (iprocs = 3, jprocs = 3)
      INTEGER itable(-1:iprocs, -1:jprocs)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (nprocs /= iprocs * jprocs) THEN
```

```
  PRINT *,'=== Error ==='
  STOP
ENDIF
DO j = -1, jprocs
  DO i = -1, iprocs
    itable(i,j) = MPI_PROC_NULL
  ENDDO
ENDDO
irank = 0
DO i = 0, iprocs - 1
  DO j = 0, jprocs - 1
    itable(i,j) = irank
    IF (myrank == irank) THEN
      myranki = i
      myrankj = j
    ENDIF
    irank = irank + 1
  ENDDO
ENDDO
CALL para_range(1, n, jprocs, myrankj, jsta, jend)
jsta2 = jsta
jend1 = jend
IF (myrankj == 0)          jsta2 = 2
IF (myrankj == jprocs - 1) jend1 = n - 1
CALL para_range(1, m, iprocs, myranki, ista, iend)
ista2 = ista
iend1 = iend
IF (myranki == 0)          ista2 = 2
IF (myranki == iprocs - 1) iend1 = m - 1
jjsta = MAX(1, ista - 1)
jjend = MIN(n, jend + 1)
ilen = iend  - ista  + 1
jlen = jjend - jjsta + 1
jnext = itable(myranki,     myrankj + 1)
jprev = itable(myranki,     myrankj - 1)
inext = itable(myranki + 1, myrankj)
iprev = itable(myranki - 1, myrankj)
DO j = jsta, jend
  DO i = ista, iend
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
CALL MPI_ISEND(a(ista,jend),ilen,MPI_REAL8,jnext,1,MPI_COMM_WORLD,isend1,ierr)
CALL MPI_ISEND(a(ista,jsta),ilen,MPI_REAL8,jprev,1,MPI_COMM_WORLD,isend2,ierr)
CALL MPI_IRECV(a(ista,jsta-1),ilen,MPI_REAL8,jprev,1,MPI_COMM_WORLD,irecv1,ierr)
CALL MPI_IRECV(a(ista,jend+1),ilen,MPI_REAL8,jnext,1,MPI_COMM_WORLD,irecv2,ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
IF (myranki /= iprocs - 1) THEN
  DO j = jjsta, jjend
    works1(j) = a(iend,j)
  ENDDO
ENDIF
IF (myranki /= 0) THEN
  DO j = jjsta, jjend
    works2(j) = a(ista,j)
  ENDDO
ENDIF
CALL MPI_ISEND(works1(jjsta),jlen,MPI_REAL8,inext,1,MPI_COMM_WORLD,jsend1,ierr)
CALL MPI_ISEND(works2(jjsta),jlen,MPI_REAL8,iprev,1,MPI_COMM_WORLD,jsend2,ierr)
CALL MPI_IRECV(workr1(jjsta),jlen,MPI_REAL8,iprev,1,MPI_COMM_WORLD,jrecv1,ierr)
CALL MPI_IRECV(workr2(jjsta),jlen,MPI_REAL8,inext,1,MPI_COMM_WORLD,jrecv2,ierr)
CALL MPI_WAIT(jsend1, istatus, ierr)
CALL MPI_WAIT(jsend2, istatus, ierr)
CALL MPI_WAIT(jrecv1, istatus, ierr)
CALL MPI_WAIT(jrecv2, istatus, ierr)
IF (myranki /= 0) THEN
  DO j = jjsta, jjend
    a(ista-1,j) = workr1(j)
  ENDDO
ENDIF
IF (myranki /= iprocs - 1) THEN
  DO j = jjsta, jjend
    a(iend+1,j) = workr2(j)
  ENDDO
ENDIF
```

```
 DO j = jsta2, jend1
   DO i = ista2, iend1
     b(i,j) = a(i-1,j  ) + a(i,  j-1) + a(i,  j+1) + a(i+1,j  )
&           + a(i-1,j-1) + a(i+1,j-1) + a(i-1,j+1) + a(i+1,j+1)
   ENDDO
 ENDDO
 END
```

The three-dimensional FDM would be even more complex than the above program, which is beyond the scope of this publication.

4.2 Finite Element Method

In the finite element method (FEM), the data dependence is much more irregular than the finite difference method, so it is generally harder to parallelize. The implicit method for FEM problems has a hot spot in solving linear equations with a sparse symmetric matrix. In this section, the explicit method of FEM is considered for parallelization.

The following is an example program that mimics data dependence of an FEM solver using the explicit method.

```
...
PARAMETER(iemax = 12, inmax = 21)
REAL*8 ve(iemax), vn(inmax)
INTEGER index(4,iemax)
...
DO ie = 1, iemax
  ve(ie) = ie * 10.0
ENDDO
DO in = 1, inmax
  vn(in) = in * 100.0
ENDDO
DO itime = 1, 10
  DO ie = 1, iemax
    DO j = 1, 4
      vn(index(j,ie)) = vn(index(j,ie)) + ve(ie)
    ENDDO
  ENDDO
  DO in = 1, inmax
    vn(in) = vn(in) * 0.25
  ENDDO
  DO ie = 1, iemax
    DO j = 1, 4
      ve(ie) = ve(ie) + vn(index(j,ie))
    ENDDO
  ENDDO
  DO ie = 1, iemax
    ve(ie) = ve(ie) * 0.25
  ENDDO
ENDDO
PRINT *,'Result',vn,ve
...
```

The geometry of the model is illustrated in Figure 94 on page 109. There are 12 elements (represented by boxes) and each element has four nodes (circles) adjacent to it. The method described in this section is applicable to more irregular meshes as well. In the above program, `ve()` and `vn()` represent some quantity at elements and at nodes, respectively. Within the enclosing time step loop of `itime`, there are four loops, which manipulate and update values at nodes and elements. In lines 13-17, the value of `ve()` is added to `vn()` at adjacent nodes (Figure 94 (a)). Then the value of `vn()` is updated locally in lines 18-20 (Figure 94 (b)). This value of `vn()` is used to update `ve()` in turn in lines 21-25 (Figure 94 (c)). At the end of the time step, the value of `ve()` is updated locally in lines 26-28 (Figure 94 (d)).

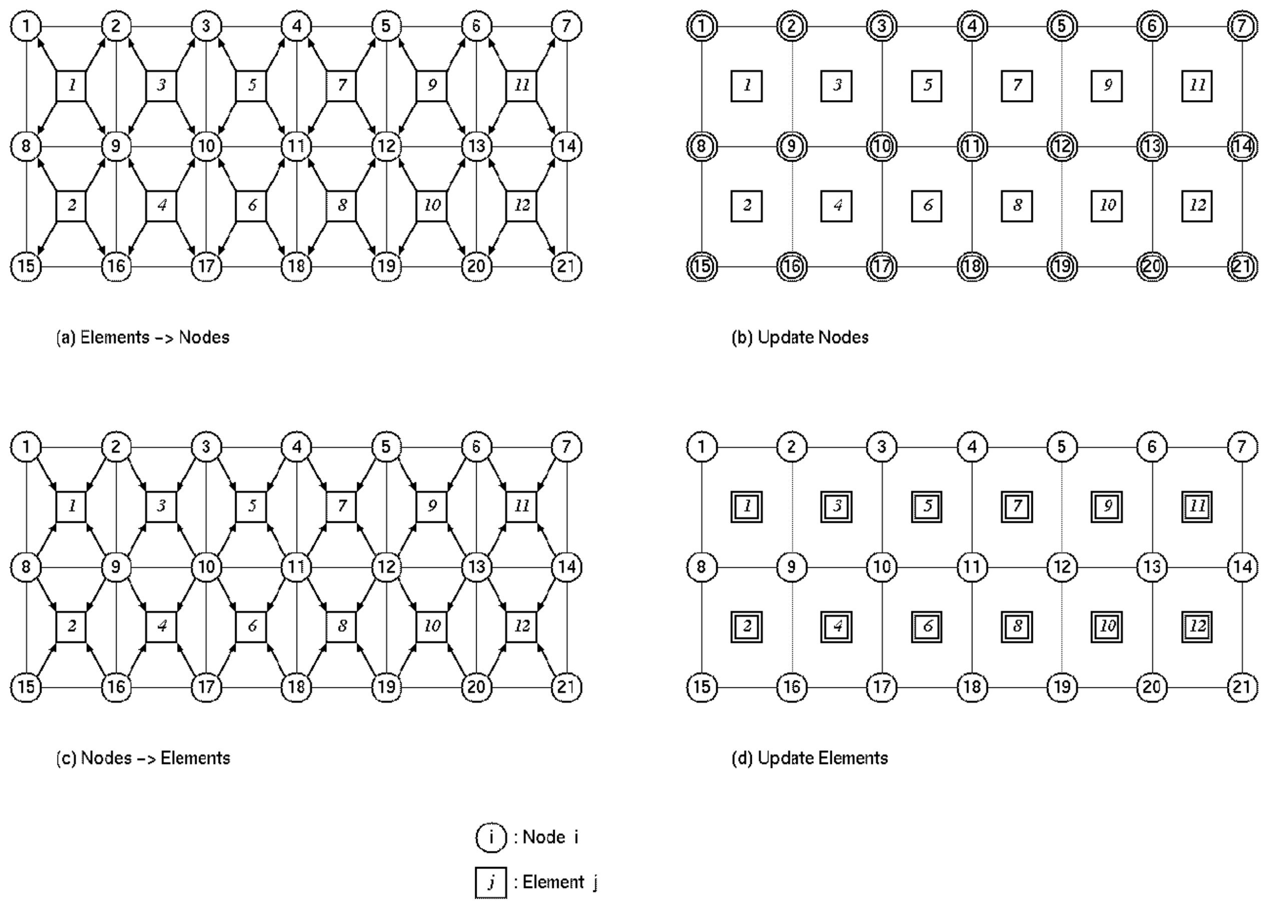


Figure 94. Finite Element Method: Four Steps within a Time Step

To look up adjacent nodes for each element, a matrix `index()` is prepared. The value of `index(j,k)` indicates the `k`-th neighbor node of element `j`, where `k=1..4`.

$$
index = \begin{bmatrix}
1 & 8 & 2 & 9 & 3 & 10 & 4 & 11 & 5 & 12 & 6 & 13 \\
2 & 9 & 3 & 10 & 4 & 11 & 5 & 12 & 6 & 13 & 7 & 14 \\
9 & 16 & 10 & 17 & 11 & 18 & 12 & 19 & 13 & 20 & 14 & 21 \\
8 & 15 & 9 & 16 & 10 & 17 & 11 & 18 & 12 & 19 & 13 & 20
\end{bmatrix}
$$

The strategy used to distribute data to processes is illustrated in Figure 95 on page 110, where the number of processes is supposed to be three. Each process is in charge of four elements and the nodes adjacent to them. As for a node that lies on the boundary between two processes, pick up the process which has the lower rank and designate it as the primary process for that node. The other process is treated as secondary for that node. In the figure, nodes for which a process is secondary are marked with a dashed circle. For example, for node 3, process 0 is primary and process 1 is secondary. Note that whether a process is primary or secondary depends on the boundary node concerned. For instance, process 1 is primary for node 12, but is secondary for node 10. In general, some boundary nodes might have more than one secondary processes.

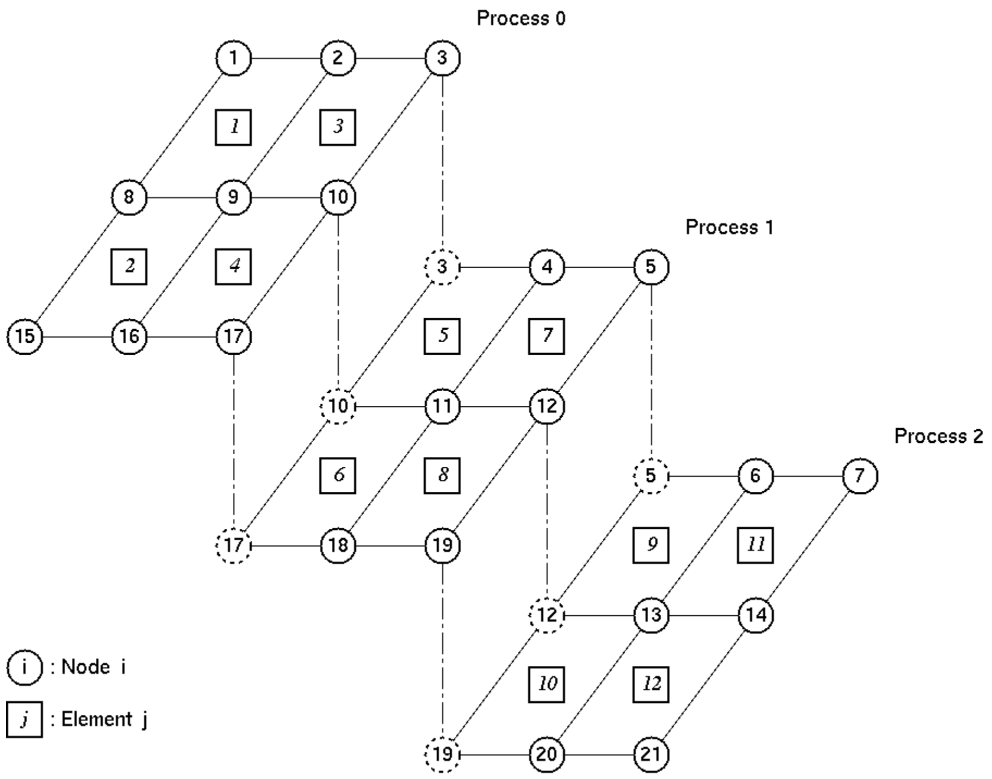


Figure 95. Assignment of Elements and Nodes to Processes

To work with boundary nodes, the following data structure is employed.

Process `i` has two-dimensional arrays `ibwork1` and `ibwork2`. The `j`-th column of `ibwork1` is a list of boundary nodes between processes `i` and `j`, where process `j` is primary. The number of such boundary nodes is given by `ibcnt1(j)`. The `j`-th column of `ibwork2` is a list of boundary nodes between processes `i` and `j`, where process `i` is primary. The number of such boundary nodes is given by `ibcnt2(j)`. By the definition of `ibwork1` and `ibwork2`, the `j`-th column of process `i`'s `ibwork1` (`ibwork2`) is identical with the `i`-th column of process `j`'s `ibwork2` (`ibwork1`). The values of `ibwork1` and `ibwork2` corresponding to the distribution in Figure 95 are shown in Figure 96 on page 111.

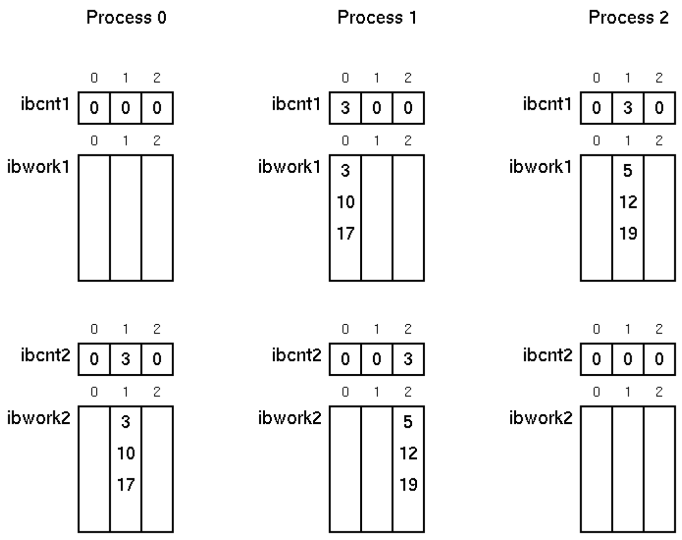


Figure 96. Data Structures for Boundary Nodes

In short, on a secondary process, `ibwork1` is used to exchange boundary node data with the primary process, and on the primary process, `ibwork2` is used to exchange boundary node data with secondary processes.

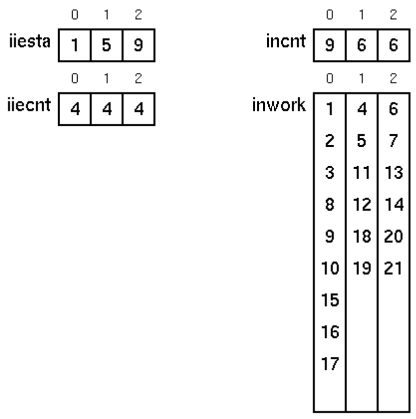


Figure 97. Data Structures for Data Distribution

Also, other data structures are prepared for storing information of data distribution (see Figure 97). Since elements are block distributed, nodes are not necessarily distributed to processes regularly. The array `inwork()` stores node numbers that are maintained primarily by each process.

Here is a rough explanation of the parallelized program. After the initialization of MPI, the data structure is prepared, and element array `ve()` and node array `vn()` are also initialized. The array `index()` is assumed to be given.

```
      ...
      PARAMETER(iemax = 12, inmax = 21)
      REAL*8 ve(iemax), vn(inmax)
      INTEGER index(4,iemax)
      INCLUDE 'mpif.h'
      PARAMETER (ncpu = 3)
      INTEGER nprocs, myrank
      INTEGER istatus(MPI_STATUS_SIZE)
      INTEGER itemp(inmax,0:ncpu-1)
      INTEGER incnt(0:ncpu-1), inwork(inmax,0:ncpu-1)
      INTEGER ibcnt1(0:ncpu-1), ibwork1(inmax,0:ncpu-1)
      INTEGER ibcnt2(0:ncpu-1), ibwork2(inmax,0:ncpu-1)
      INTEGER iiesta(0:ncpu-1), iiecnt(0:ncpu-1)
      INTEGER ireqs(0:ncpu-1), ireqr(0:ncpu-1)
      DIMENSION bufs(inmax,0:ncpu-1), bufr(inmax,0:ncpu-1)
      ...
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO irank = 0, nprocs - 1
        DO i = 1, inmax
          itemp(i,irank) = 0
        ENDDO
        ibcnt1(irank) = 0
        ibcnt2(irank) = 0
        incnt(irank) = 0
      ENDDO
      DO irank = 0, nprocs - 1
        CALL para_range(1, iemax, nprocs, irank, iesta, ieend)
        iiesta(irank) = iesta
        iiecnt(irank) = ieend - iesta + 1
        DO ie = iesta, ieend
          DO j = 1, 4
            itemp(index(j,ie),irank) = 1
          ENDDO
        ENDDO
      ENDDO
      CALL para_range(1, iemax, nprocs, myrank, iesta, ieend)
      DO i = 1, inmax
        iflg = 0
        DO irank = 0, nprocs - 1
          IF (itemp(i,irank) == 1) THEN
            IF (iflg == 0) THEN
              iflg = 1
              iirank = irank
            ELSE
              itemp(i,irank) = 0
              IF (irank == myrank) THEN
                ibcnt1(iirank) = ibcnt1(iirank) + 1
                ibwork1(ibcnt1(iirank),iirank) = i
              ELSEIF (iirank == myrank) THEN
                ibcnt2(irank) = ibcnt2(irank) + 1
                ibwork2(ibcnt2(irank),irank) = i
              ENDIF
            ENDIF
          ENDIF
        ENDDO
      ENDDO
      DO irank = 0, nprocs - 1
        DO i = 1, inmax
          IF (itemp(i,irank) == 1) THEN
            incnt(irank) = incnt(irank) + 1
            inwork(incnt(irank),irank) = i
          ENDIF
        ENDDO
      ENDDO
      DO ie = iesta, ieend
        ve(ie) = ie * 10.0
      ENDDO
      DO i = 1, incnt(myrank)
        in = inwork(i,myrank)
        vn(in) = in * 100.0
      ENDDO
```

Now, the main time-ticking loop starts. First, node array `vn()` is updated using element array `ve()` in Figure 98 on page 113. For a boundary node `k`, secondary processes first clear the value of `vn(k)` to zero so that they calculate only the difference contributed by the local elements and then send that difference to the primary process (see Figure 99 on page 114).

```
      DO itime = 1, 10
        DO irank = 0, nprocs - 1
          DO i = 1, ibcnt1(irank)
            vn(ibwork1(i,irank)) = 0.0
          ENDDO
        ENDDO
        DO ie = iesta, ieend
          DO j = 1, 4
           vn(index(j,ie)) = vn(index(j,ie)) + ve(ie)
          ENDDO
        ENDDO
        DO irank = 0, nprocs - 1
          DO i = 1, ibcnt1(irank)
            bufs(i,irank) = vn(ibwork1(i,irank))
          ENDDO
        ENDDO
        DO irank = 0, nprocs - 1
          IF (ibcnt1(irank) /= 0)
     &      CALL MPI_ISEND(bufs(1,irank),ibcnt1(irank),MPI_REAL8,irank,1,
     &                     MPI_COMM_WORLD,ireqs(irank),ierr)
          IF (ibcnt2(irank) /= 0)
     &      CALL MPI_IRECV(bufr(1,irank),ibcnt2(irank),MPI_REAL8,irank,1,
     &                     MPI_COMM_WORLD,ireqr(irank),ierr)
        ENDDO
        DO irank = 0, nprocs - 1
          IF (ibcnt1(irank) /= 0) CALL MPI_WAIT(ireqs(irank),istatus,ierr)
          IF (ibcnt2(irank) /= 0) CALL MPI_WAIT(ireqr(irank),istatus,ierr)
        ENDDO
        DO irank = 0, nprocs - 1
          DO i = 1, ibcnt2(irank)
            vn(ibwork2(i,irank)) = vn(ibwork2(i,irank)) + bufr(i,irank)
          ENDDO
        ENDDO
```

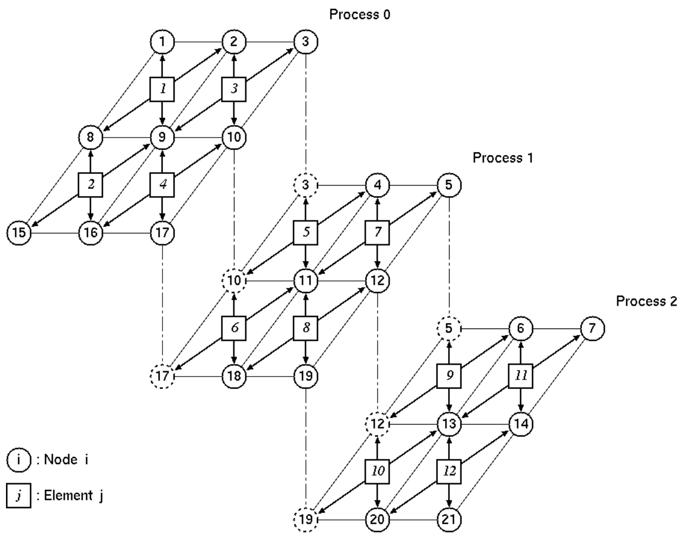


Figure 98. Contribution of Elements to Nodes Are Computed Locally

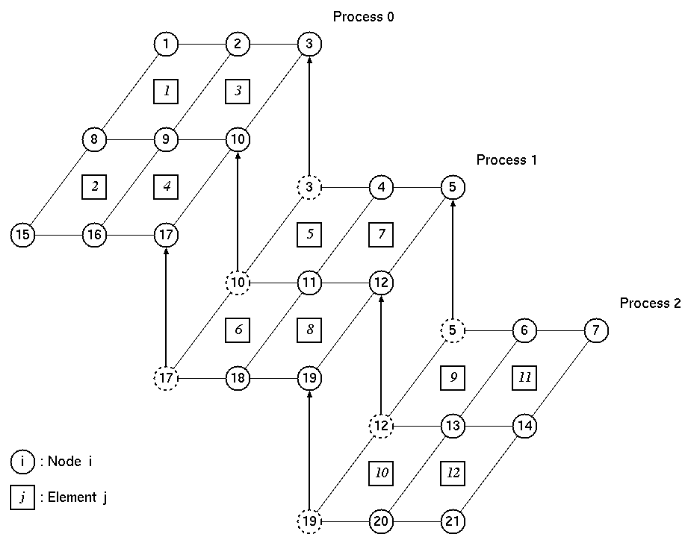


Figure 99. Secondary Processes Send Local Contribution to Primary Processes

Node array `vn()` is updated on primary processes and the new value is sent to secondary processes (see Figure 100 on page 115).

```
      DO i = 1, incnt(myrank)
        in = inwork(i,myrank)
        vn(in) = vn(in) * 0.25
      ENDDO
      DO irank = 0, nprocs - 1
        DO i = 1, ibcnt2(irank)
          bufs(i,irank) = vn(ibwork2(i,irank))
        ENDDO
      ENDDO
      DO irank = 0, nprocs - 1
        IF (ibcnt2(irank) /= 0)
    &     CALL MPI_ISEND(bufs(1,irank),ibcnt2(irank),MPI_REAL8,irank,1,
    &                    MPI_COMM_WORLD,ireqs(irank),ierr)
        IF (ibcnt1(irank) /= 0)
    &     CALL MPI_IRECV(bufr(1,irank),ibcnt1(irank),MPI_REAL8,irank,1,
    &                    MPI_COMM_WORLD,ireqr(irank),ierr)
      ENDDO
      DO irank = 0, nprocs - 1
        IF (ibcnt2(irank) /= 0) CALL MPI_WAIT(ireqs(irank),istatus,ierr)
        IF (ibcnt1(irank) /= 0) CALL MPI_WAIT(ireqr(irank),istatus,ierr)
      ENDDO
      DO irank = 0, nprocs - 1
        DO i = 1, ibcnt1(irank)
          vn(ibwork1(i,irank)) = bufr(i,irank)
        ENDDO
      ENDDO
```

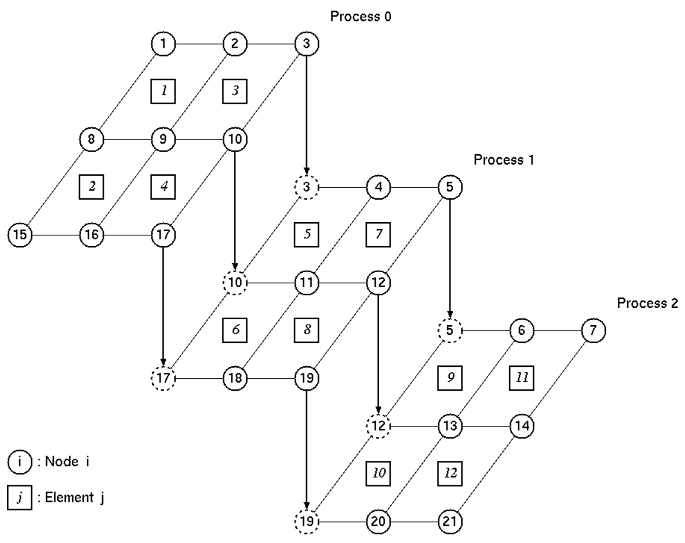


Figure 100. Updated Node Values Are Sent from Primary to Secondary

Element array `ve()` is updated (see Figure 101 on page 115) using node array `vn()`.

```
  DO ie = iesta, ieend
    DO j=1,4
      ve(ie) = ve(ie) + vn(index(j,ie))
    ENDDO
  ENDDO
  DO ie = iesta, ieend
    ve(ie) = ve(ie) * 0.25
  ENDDO
ENDDO
```

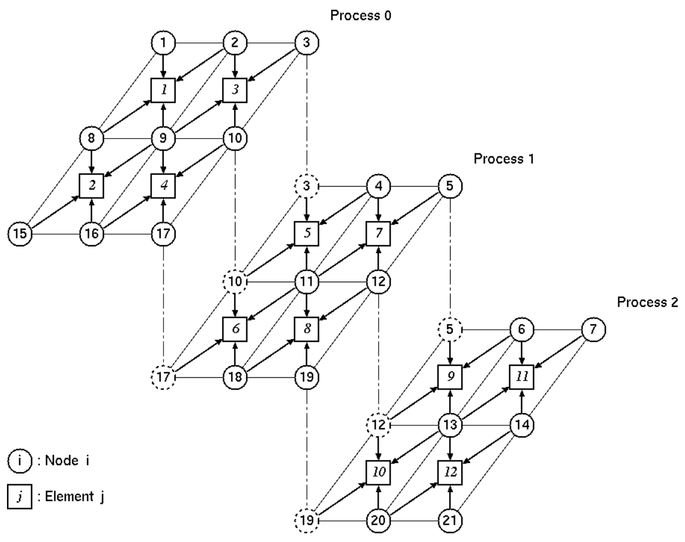


Figure 101. Contribution of Nodes to Elements Are Computed Locally

Finally, after the last time step, arrays `ve()` and `vn()` are collected at process 0.

```
DO i = 1, incnt(myrank)
  bufs(i,myrank) = vn(inwork(i,myrank))
ENDDO
IF (myrank == 0) THEN
```

```
        DO irank = 1, nprocs - 1
          CALL MPI_IRECV(bufr(1,irank),incnt(irank),MPI_REAL8,irank,1,
     &                   MPI_COMM_WORLD,ireqs(irank),ierr)
        ENDDO
        DO irank = 1, nprocs - 1
        CALL MPI_WAIT(ireqs(irank),istatus,ierr)
      ENDDO
      ELSE
        CALL MPI_ISEND(bufs(1,myrank),incnt(myrank),MPI_REAL8,0,1,
     &                 MPI_COMM_WORLD,ireqr,ierr)
        CALL MPI_WAIT(ireqr,istatus,ierr)
      ENDIF
      IF (myrank == 0) THEN
        DO irank = 1, nprocs - 1
          DO i = 1, incnt(irank)
            vn(inwork(i,irank)) = bufr(i,irank)
          ENDDO
        ENDDO
      ENDIF
      IF (myrank == 0) THEN
        DO irank = 1, nprocs - 1
          CALL MPI_IRECV(ve(iiesta(irank)),iiecnt(irank),MPI_REAL8,irank,1,
     &                   MPI_COMM_WORLD,ireqs(irank),ierr)
        ENDDO
        DO irank = 1, nprocs - 1
        CALL MPI_WAIT(ireqs(irank),istatus,ierr)
        ENDDO
      ELSE
        CALL MPI_ISEND(ve(iesta),ieend-iesta+1,MPI_REAL8,0,1,
     &                 MPI_COMM_WORLD,ireqr,ierr)
        CALL MPI_WAIT(ireqr,istatus,ierr)
      ENDIF
      CALL MPI_FINALIZE(ierr)
      PRINT *,'Result',vn,ve
      ...
```

Note that the problem solved in this section is rather special in that the numbering of elements are such that when elements are block-distributed to processes, it makes a moderately small number of boundary nodes between processes. In addition, on irregular meshes, the number of assigned nodes might not be well balanced among processes, if you divide data evenly in terms of number of elements.

4.3 LU Factorization

The LU factorization is a popular method for solving linear equations with dense matrices. Parallel ESSL for AIX has subroutines for LU factorization which outperform hand-coded solutions in most cases. (See Chapter 4.8, “Using Parallel ESSL” on page 139 for details.) So, regard this section as an example of one of the methodologies of parallelization. The following program solves a system of linear equations: *Ax=b*. It first factorizes the matrix *A* into a product of a lower triangular matrix (*L*) and a upper triangular matrix (*U*). Matrices *L* and *U* overwrite the original matrix. Then in the forward elimination and in the backward substitution, the solution vector *x* overwrites *b*. Pivoting and loop-unrolling are not considered in order to maintain simplicity.

```
      PROGRAM main
      PARAMETER (n = ...)
      REAL a(n,n)
      ...
! LU factorization
      DO k = 1, n-1
        DO i = k+1, n
          a(i,k) = a(i,k) / a(k,k)
        ENDDO
```

```
          DO j = k+1, n
            DO i = k+1, n
              a(i,j) = a(i,j) - a(i,k) * a(k,j)
            ENDDO
          ENDDO
        ENDDO
! Forward elimination
        DO i = 2, n
          DO j = 1, i - 1
            b(i) = b(i) - a(i,j) * b(j)
          ENDDO
        ENDDO
! Backward substitution
        DO i = n, 1, -1
          DO j = i + 1, n
            b(i) = b(i) - a(i, j) * b(j)
          ENDDO
          b(i) = b(i) / a(i,i)
        ENDDO
        END
        ...
```

As the execution proceeds, the part of the matrix which is processed in the LU factorization shrinks as follows: $\{a(i,j) | k+1 \le i, j \le n\}$ where `k` increases from `1` to `n-1`.

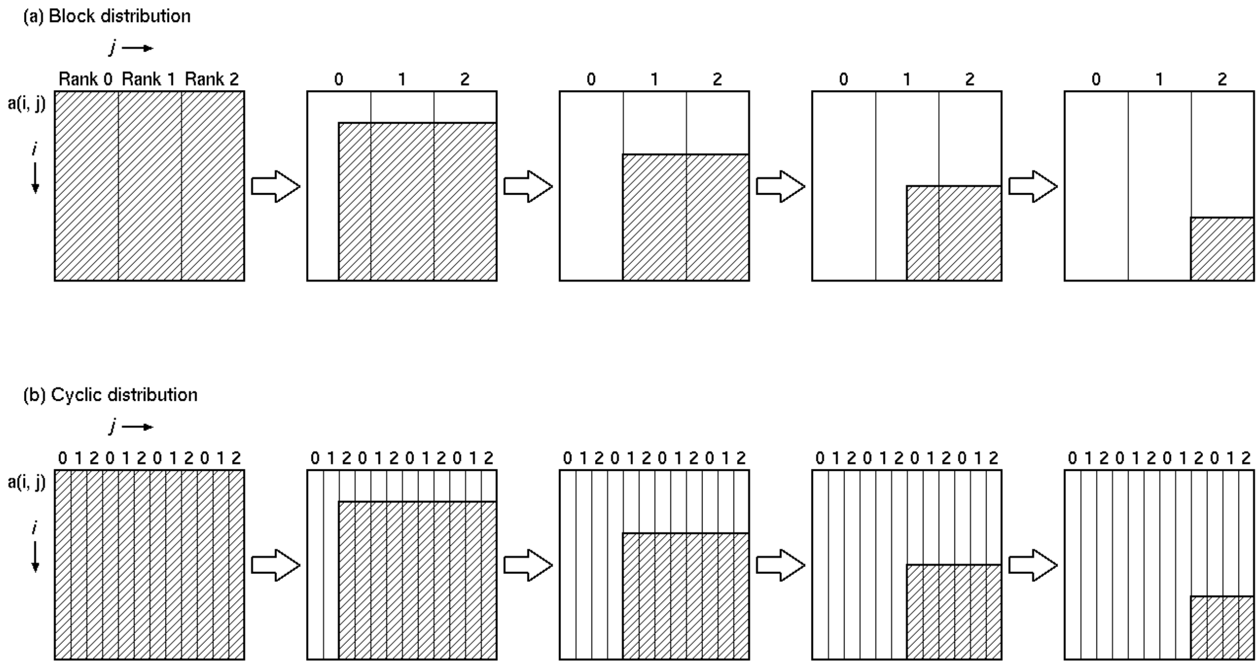


Figure 102. Data Distributions in LU Factorization

Because of this access pattern of LU factorization, when the matrix is block-distributed, the workload will not be balanced among processes in the later iterations of `k`, as Figure 102 (a) shows. On the other hand, cyclic distribution provides a good load balance (see Figure 102 (b)).

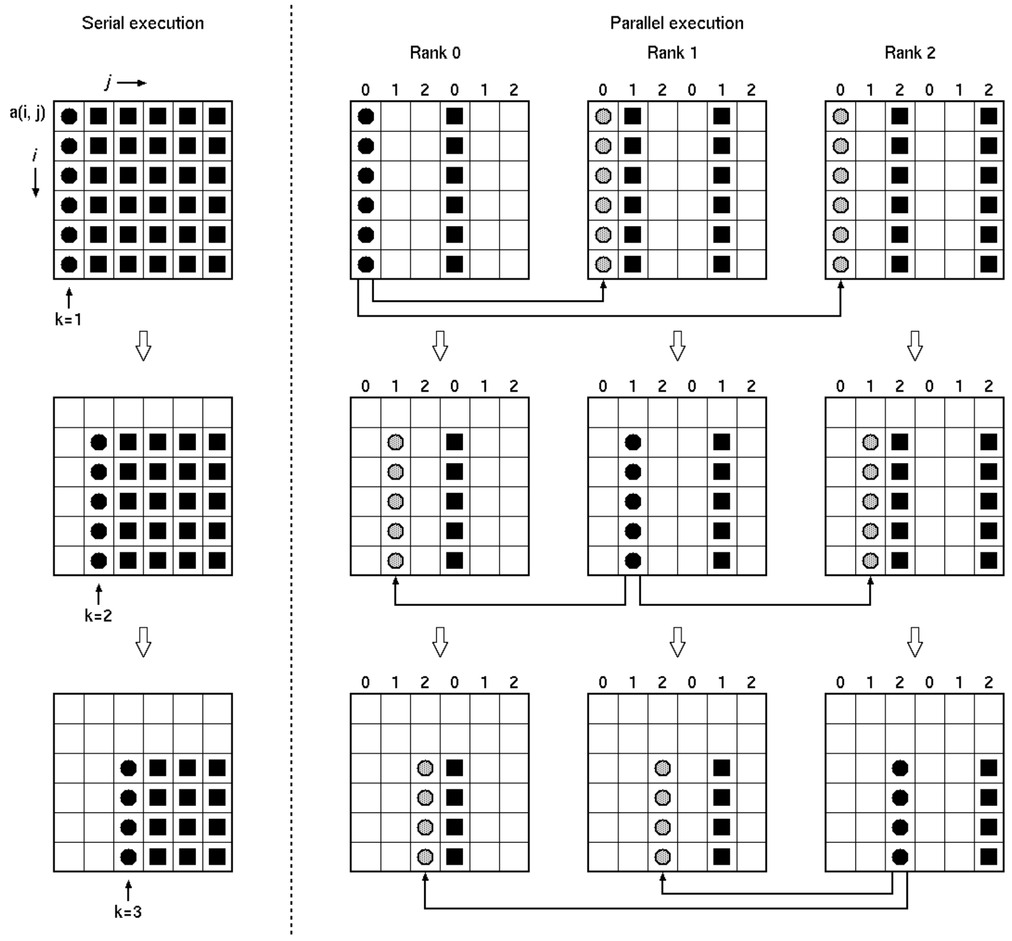


Figure 103. First Three Steps of LU Factorization

Figure 103 illustrates the first three steps (`k=1,2,3`) of LU factorization. In the serial execution, matrix elements marked with black circles are used to update elements marked with black boxes. In the parallel execution, since the matrix is distributed cyclic, the process that holds black circles varies from iteration to iteration. In each iteration, black circles are broadcasted from the process that has them to the other processes, and all the processes start updating their own part. In the parallelized program, cyclic distribution is implemented by use of a mapping array (See Chapter 3.4.2, “Cyclic Distribution” on page 56). In iteration `k`, the pivot is owned by process `map(k)`.

```
      PROGRAM main
      INCLUDE 'mpif.h'
      PARAMETER (n = ...)
      REAL a(n,n)
      INTEGER map(n)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i = 1, n+1
        map(i) = MOD(i-1, nprocs)
      ENDDO
      ...
! LU factorization
```

```
      DO k = 1, n
        IF (map(k) == myrank) THEN
          DO i = k+1, n
            a(i,k) = a(i,k) / a(k,k)
          ENDDO
        ENDIF
        CALL MPI_BCAST(a(k,k), n-k+1, MPI_REAL, map(k),
     &                 MPI_COMM_WORLD, ierr)
        DO j = k+1, n
          IF (map(j) == myrank) THEN
            DO i = k+1, n
              a(i,j) = a(i,j) - a(i,k) * a(k,j)
            ENDDO
          ENDIF
        ENDDO
      ENDDO
! Forward elimination
      DO i = 2, n
        s = 0.0
        DO j = 1 + myrank, i - 1, nprocs
          s = s + a(i,j) * b(j)
        ENDDO
        CALL MPI_ALLREDUCE(s, ss, 1, MPI_REAL, MPI_SUM,
     &                     MPI_COMM_WORLD, ierr)
        b(i) = b(i) - ss
      ENDDO
! Backward substitution
      DO i = n, 1, -1
        s = 0.0
        IF (map(i+1) <= myrank) THEN
          ii = i + 1 + myrank - map(i+1)
        ELSE
          ii = i + 1 + myrank - map(i+1) + nprocs
        ENDIF
        DO j = ii, n, nprocs
          s = s + a(i, j) * b(j)
        ENDDO
        CALL MPI_ALLREDUCE(s, ss, 1, MPI_REAL, MPI_SUM,
     &                     MPI_COMM_WORLD, ierr)
        b(i) = (b(i) - ss) / a(i,i)
      ENDDO
      END
      ...
```

Since `map(n+1)` is referenced in the backward substitution, the range of `map()` has to be `1..n+1`. The backward substitution also needs `a(i,i)`, so the range of the loop variable `k` in the LU factorization is modified without affecting the outcome of the LU factorization, and the diagonal elements are broadcast at the time of LU factorization.

If you incrementally parallelize the program (see “Incremental parallelization” on page 92), you can add the following code to synchronize the value of matrix `a()` among processes between factorization and forward elimination or between forward elimination and backward substitution in the process of parallelization.

```
      DO j = 1, n
        IF (map(j) == myrank) THEN
          DO i = 1, n
          tmp(i,j) = a(i,j)
        ENDDO
      ELSE
        DO i = 1, n
          tmp(i,j) = 0.0
        ENDDO
      ENDIF
    ENDDO
    CALL MPI_ALLREDUCE(tmp, a, n*n, MPI_REAL, MPI_SUM,
   &                    MPI_COMM_WORLD, ierr)
```

This code uses the technique of superposition (Chapter 3.5.5, “Superposition” on page 78).

4.4 SOR Method

The following program solves a two-dimensional Laplace equation using the successive over-relaxation (SOR) method. The outermost loop is for the convergence of `x()`, and the inner loops regarding `i` and `j` are for updating the value of `x()`. The convergence is accelerated with the over-relaxation parameter omega.

```
    PROGRAM sor
    PARAMETER (mmax = 6, nmax = 9)
    PARAMETER (m = mmax - 1, n = nmax - 1)
    REAL x(0:mmax, 0:nmax)
    ...
    DO k = 1, 300
      err1 = 0.0
      DO j = 1, n
        DO i = 1, m
          temp = 0.25 * (x(i,j-1) + x(i-1,j)
   &                    + x(i+1,j) + x(i,j+1)) - x(i,j)
          x(i,j) = x(i,j) + omega * temp
          IF (abs(temp) > err1) err1 = abs(temp)
        ENDDO
      ENDDO
      IF (err1 <= eps) exit
    ENDDO
    ...
    END
```

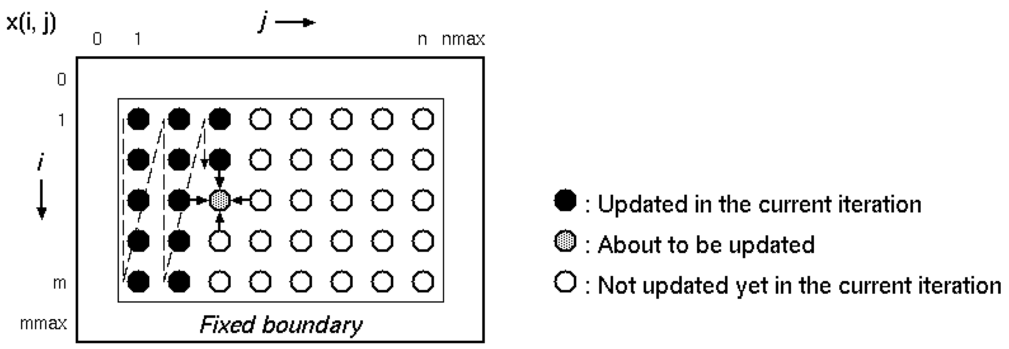

x(i, j)
j
0
1
n
nmax
0
1
i
m
mmax
Fixed boundary
● : Updated in the current iteration
◍ : About to be updated
○ : Not updated yet in the current iteration

Figure 104. SOR Method: Serial Run

Figure 104 on page 120 shows data dependence of the SOR program. In updating x(i,j), its four neighbors are referenced. However, due to the loop structure, when x(i,j) is updated, x(i,j-1) and x(i-1,j) are already updated in that iteration, whereas x(i+1,j) and x(i,j+1) are not. Therefore, there is a strict restriction in the order of update, which is shown in dashed arrows in the figure.

You could use the pipeline method (Chapter 3.5.6, “The Pipeline Method” on page 79) to parallelize this program, but, usually, an alternative method called red-black SOR is used.

4.4.1 Red-Black SOR Method

In the red-black SOR, the order of computation is modified from the original SOR method, which means the steps required for convergence may change and the result may be different within the precision that is imposed by the variable eps. First, each matrix element is conceptually colored with red or black as follows.

If i+j is even, x(i,j) is red. (Shown as circles in Figure 105)

If i+j is odd, x(i,j) is black. (Shown as boxes in Figure 105)

The red-black SOR updates red elements and black elements alternately. Note that the four neighbors of a red element are all black, and vice versa. So, red (black) elements can be updated independently of other red (black) elements, and, thereby, the red-black SOR can be efficiently parallelized.

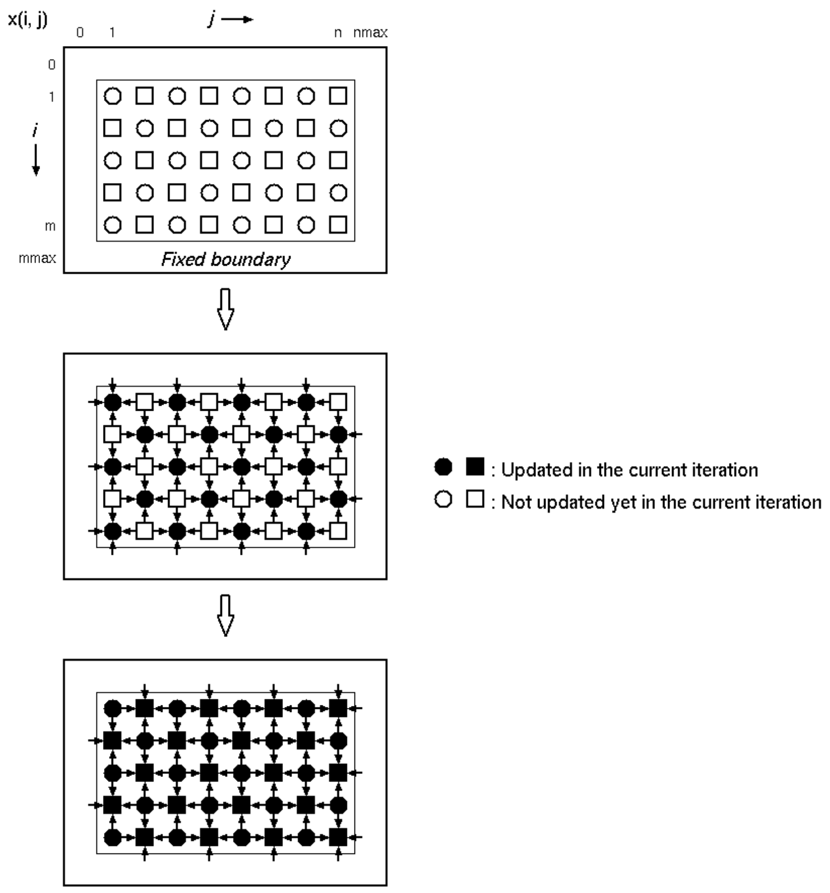

x(i, j)
j
0
1
n
nmax
0
1
i
m
mmax
Fixed boundary
● ■ : Updated in the current iteration
○ □ : Not updated yet in the current iteration

Figure 105. Red-Black SOR Method

The program of the red-black SOR method is as follows.

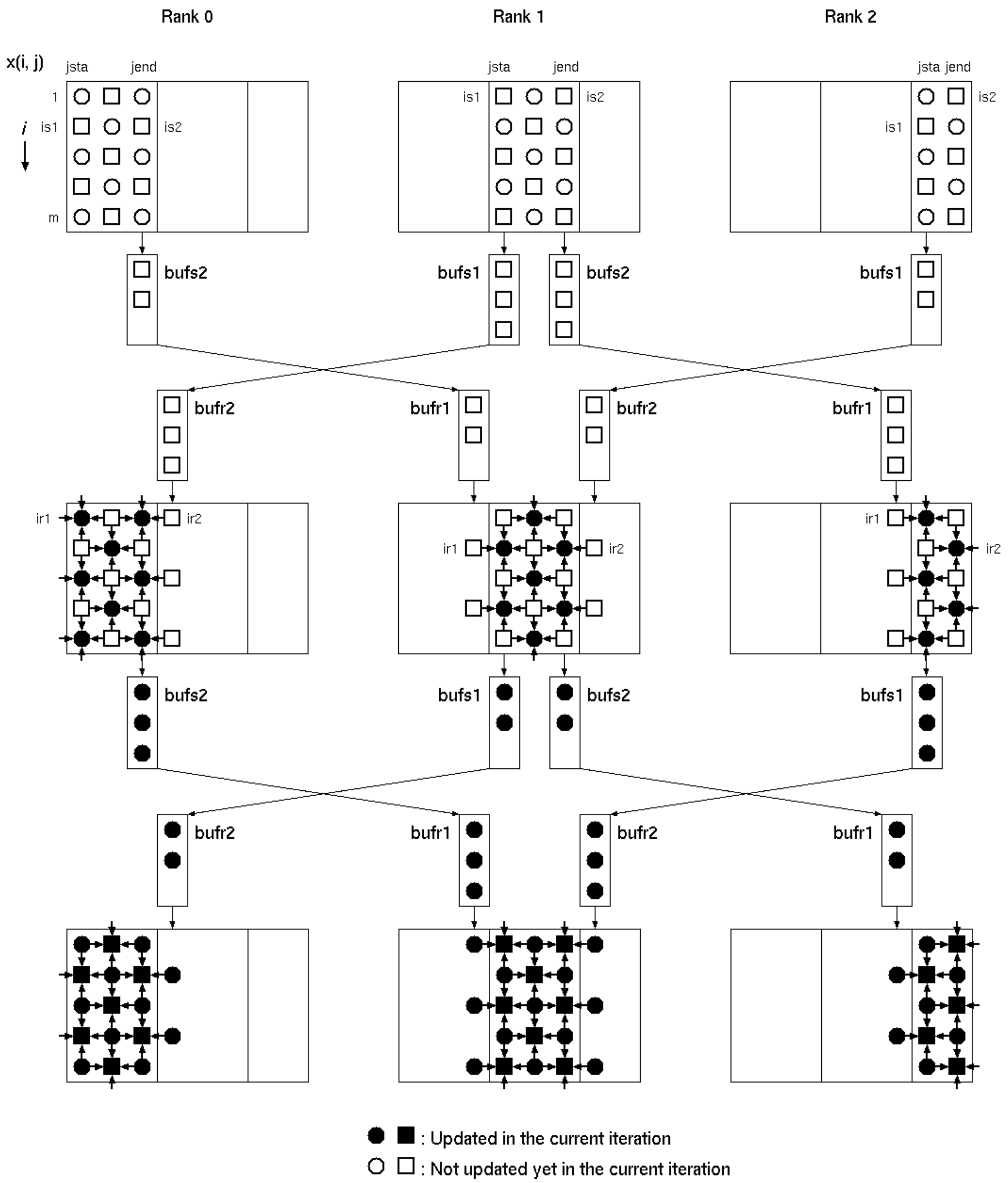
```
      PROGRAM red_black
      PARAMETER (mmax = 6, nmax = 9)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      REAL x(0:mmax, 0:nmax)
      ...
      DO k = 1, 300
        err1 = 0.0
        DO j = 1, n
          DO i = MOD(j+1,2) + 1, m, 2
            temp = 0.25 * (x(i,j-1) + x(i-1,j)
     &                   + x(i+1,j) + x(i,j+1)) - x(i,j)
            x(i,j) = x(i,j) + omega * temp
            IF (abs(temp) > err1) err1 = abs(temp)
          ENDDO
        ENDDO
        DO j = 1, n
          DO i = MOD(j,2) + 1, m, 2
            temp = 0.25 * (x(i,j-1) + x(i-1,j)
     &                   + x(i+1,j) + x(i,j+1)) - x(i,j)
            x(i,j) = x(i,j) + omega * temp
            IF (abs(temp) > err1) err1 = abs(temp)
          ENDDO
        ENDDO
         IF (err1 <= eps) exit
      ENDDO
      ...
      END
```

Figure 106 on page 123 illustrates how the red-black SOR works with three processes. Matrix `x()` is block-distributed by columns. First, each process packs black (shown as boxes in the figure) boundary elements to buffers, sends them to adjacent processes, and red elements (circles) are updated using black elements (boxes). Next, each process sends updated red boundary elements to adjacent processes, and black elements are updated using red elements. These steps repeat until the matrix converges.

For convenience, a module is used in the parallel program.

```
      MODULE para
      INCLUDE 'mpif.h'
      PARAMETER (mmax = 6, nmax = 9)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      REAL x(0:mmax, 0:nmax)
      REAL bufs1(mmax), bufr1(mmax)
      REAL bufs2(mmax), bufr2(mmax)
      INTEGER istatus(MPI_STATUS_SIZE)
      INTEGER nprocs, myrank, jsta, jend, inext, iprev
      END
```

Arrays `bufs1()` and `bufr1()` are the send and receive buffers for the left neighbor process (rank=iprev), and arrays `bufs2()` and `bufr2()` are for the right neighbor process (rank=inext).



Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
i
1
is1
is2
m
bufs2
bufs1
bufr2
bufr1
ir1
ir2
● ■ : Updated in the current iteration
○ □ : Not updated yet in the current iteration

Figure 106. Red-Black SOR Method: Parallel Run

The following is the main program.

```
PROGRAM main
USE para
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, jsta, jend)
```

```
      inext = myrank + 1
      iprev = myrank - 1
      IF (inext == nprocs) inext = MPI_PROC_NULL
      IF (iprev == -1)     iprev = MPI_PROC_NULL
      ...
      DO k = 1, 300
        err1 = 0.0
        CALL shift(0)
        DO j = jsta, jend
          DO i = MOD(j+1,2) + 1, m, 2
            temp = 0.25 * (x(i,j-1) + x(i-1,j)
     &                  + x(i+1,j) + x(i,j+1)) - x(i,j)
            x(i,j) = x(i,j) + omega * temp
            IF (abs(temp) > err1) err1 = abs(temp)
          ENDDO
        ENDDO
        CALL shift(1)
        DO j = jsta, jend
          DO i = MOD(j,2) + 1, m, 2
            temp = 0.25 * (x(i,j-1) + x(i-1,j)
     &                  + x(i+1,j) + x(i,j+1)) - x(i,j)
            x(i,j) = x(i,j) + omega * temp
            IF (abs(temp) > err1) err1 = abs(temp)
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(err1, err2, 1, MPI_REAL, MPI_MAX,
     &                    MPI_COMM_WORLD, ierr)
        err1 = err2
        IF (err1 <= eps) exit
      ENDDO
      ...
      CALL MPI_FINALIZE(ierr)
      END
```

Subroutine `shift(iflg)` takes care of all the data transmissions: `iflg=0` is for black elements (boxes) and `iflg=1` is for red elements (circles).

```
      SUBROUTINE shift(iflg)
      USE para
      is1 = MOD(jsta + iflg, 2) + 1
      is2 = MOD(jend + iflg, 2) + 1
      ir1 = 3 - is1
      ir2 = 3 - is2
      IF (myrank /= 0) THEN
        icnt1 = 0
        DO i = is1, m, 2
          icnt1 = icnt1 + 1
          bufs1(icnt1) = x(i,jsta)
        ENDDO
      ENDIF
      IF (myrank /= nprocs - 1) THEN
        icnt2 = 0
        DO i = is2, m, 2
          icnt2 = icnt2 + 1
          bufs2(icnt2) = x(i,jend)
        ENDDO
      ENDIF
      CALL MPI_ISEND(bufs1,icnt1,MPI_REAL,iprev,1,MPI_COMM_WORLD,ireqs1,ierr)
      CALL MPI_ISEND(bufs2,icnt2,MPI_REAL,inext,1,MPI_COMM_WORLD,ireqs2,ierr)
      CALL MPI_IRECV(bufr1,mmax, MPI_REAL,iprev,1,MPI_COMM_WORLD,ireqr1,ierr)
      CALL MPI_IRECV(bufr2,mmax, MPI_REAL,inext,1,MPI_COMM_WORLD,ireqr2,ierr)
      CALL MPI_WAIT(ireqs1, istatus, ierr)
      CALL MPI_WAIT(ireqs2, istatus, ierr)
      CALL MPI_WAIT(ireqr1, istatus, ierr)
      CALL MPI_WAIT(ireqr2, istatus, ierr)
      IF (myrank /= 0) THEN
        icnt = 0
        DO i = ir1, m, 2
          icnt = icnt + 1
          x(i,jsta-1) = bufr1(icnt)
        ENDDO
      ENDIF
      IF (myrank /= nprocs - 1) THEN
        icnt = 0
        DO i = ir2, m, 2
          icnt = icnt + 1
          x(i,jend+1) = bufr2(icnt)
```

```
  ENDDO
ENDIF
END
```

4.4.2 Zebra SOR Method

If you don't plan to distribute the matrix in both dimensions, besides red-black SOR, consider the option of coloring columns alternately using two colors, for example, black and white. First, update all the white elements, and then update the black elements, which is called zebra SOR method in this publication.

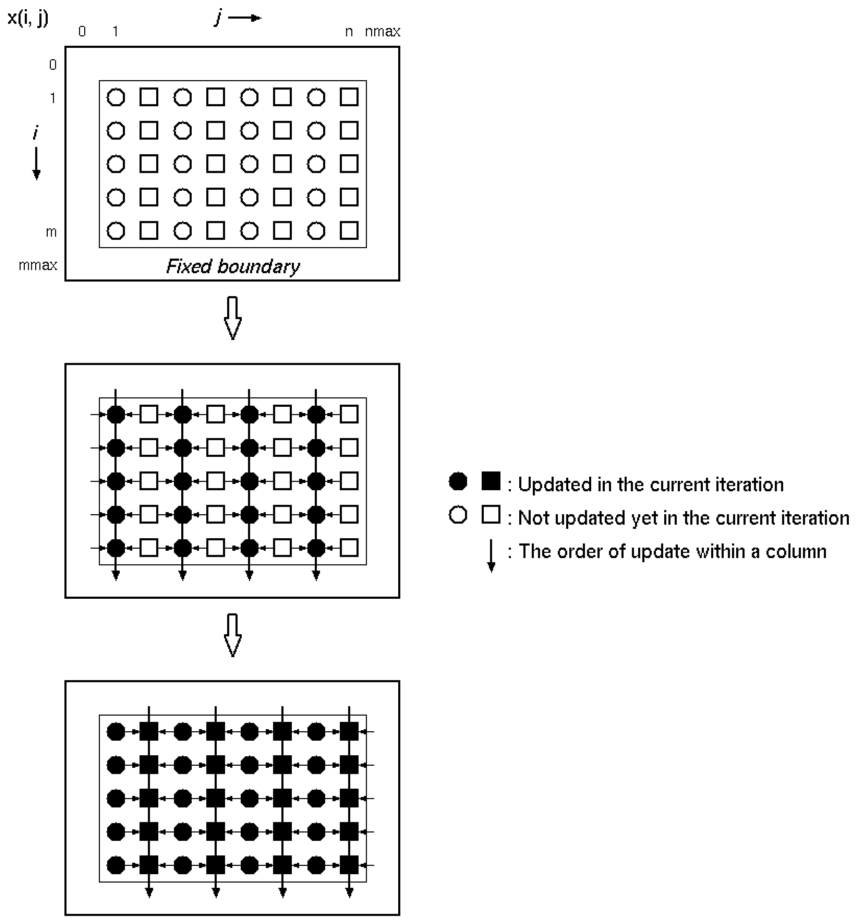

x(i, j)
j
0
1
n
nmax
i
m
mmax
Fixed boundary
● ■ : Updated in the current iteration
○ □ : Not updated yet in the current iteration
: The order of update within a column

Figure 107. Zebra SOR Method

Figure 107 shows how the computation proceeds in the zebra SOR method. The following is the serial program of zebra SOR.

```
PROGRAM zebra
PARAMETER (mmax = ..., nmax = ...)
PARAMETER (m = mmax - 1, n = nmax - 1)
DIMENSION x(0:mmax, 0:nmax)
...
DO k = 1, 300
  err1 = 0.0
  DO j = 1, n, 2
    DO i = 1, m
      Update x(i,j) and err1
    ENDDO
```

```
      ENDDO
      DO j = 2, n, 2
        DO i = 1, m
          Update x(i,j) and err1
        ENDDO
      ENDDO
      IF (err1 <= eps) EXIT
    ENDDO
    ...
    END
```

When updating elements in a column, you have to maintain the order of the elements. That is, you have to update elements from top to bottom in the figure. In other words, there is a flow dependence within a column. Remember that in red-black SOR, elements of the same color can be updated in any order. In that sense, the minimum block for parallelization is a matrix element in red-black SOR. In zebra SOR, the minimum block for parallelization is a column of elements: as long as the order of update within a column is kept unchanged, you can update the columns in any order. This property enables parallelization.

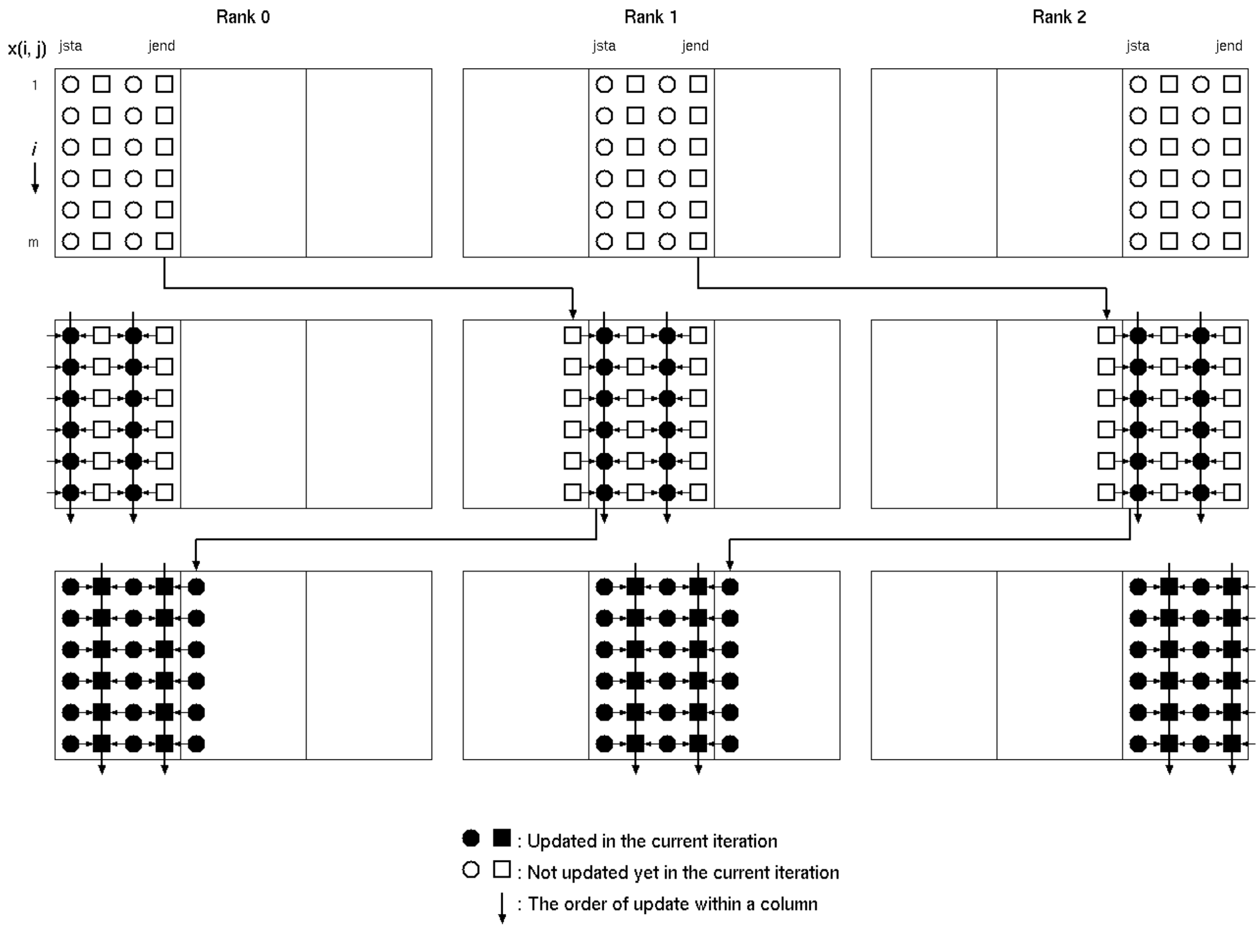

Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
1
i
m
● ■ : Updated in the current iteration
○ □ : Not updated yet in the current iteration
: The order of update within a column

Figure 108. Zebra SOR Method: Parallel Run

The Figure 108 shows how the matrix elements are updated in the parallelized zebra SOR method. The matrix elements are assigned to processes in column-wise block distribution. To make the data transmission uniform among processes, distribute columns so that the same pattern appears on the boundary between processes. Stated in another way, `jend-jsta+1` should be even in all the processes, except possibly for the last process.

```
      PROGRAM zebrap
      INCLUDE 'mpif.h'
      PARAMETER (mmax = ..., nmax = ...)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      DIMENSION x(0:mmax, 0:nmax)
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(1, (n + 1)/2, nprocs, myrank, jsta, jend)
      jsta = jsta * 2 - 1
      jend = MIN(n, jend * 2)
      inext = myrank + 1
      iprev = myrank - 1
      IF (inext == nprocs) inext = MPI_PROC_NULL
      IF (iprev == -1)     iprev = MPI_PROC_NULL
      ...
      DO k = 1, 300
        err1 = 0.0
        CALL MPI_ISEND(x(1,jend), m, MPI_REAL,
     &       inext, 1, MPI_COMM_WORLD, ireqs, ierr)
        CALL MPI_IRECV(x(1,jsta-1), m, MPI_REAL,
     &       iprev, 1, MPI_COMM_WORLD, ireqr, ierr)
        CALL MPI_WAIT(ireqs, istatus, ierr)
        CALL MPI_WAIT(ireqr, istatus, ierr)
        DO j = jsta, jend, 2
          DO i = 1, m
            Update x(i,j) and err1
          ENDDO
        ENDDO
        CALL MPI_ISEND(x(1,jsta), m, MPI_REAL,
     &       iprev, 1, MPI_COMM_WORLD, ireqs, ierr)
        CALL MPI_IRECV(x(1,jend+1), m, MPI_REAL,
     &       inext, 1, MPI_COMM_WORLD, ireqr, ierr)
        CALL MPI_WAIT(ireqs, istatus, ierr)
        CALL MPI_WAIT(ireqr, istatus, ierr)
        DO j = jsta + 1, jend, 2
          DO i = 1, m
            Update x(i,j) and err1
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(err1, err2, 1, MPI_REAL,
     &                     MPI_MAX, MPI_COMM_WORLD, ierr)
        IF (err2 <= eps) EXIT
      ENDDO
      ...
      CALL MPI_FINALIZE(ierr)
      END
```

Note that the data transmission is simple compared with red-black SOR. Since red-black SOR and zebra SOR use different orders in updating matrix elements, the number of time steps needed for convergence may be different. Choose the best algorithm in terms of the running time: there can be other variations of coloring, or ordering, of elements.

The zebra SOR can be regarded as having the same dependence of one-dimensional red-black SOR. The red-black coloring in a lower dimension can be applied to higher dimensions at the cost of the restriction as to how the arrays may be distributed. In the zebra SOR method presented in this section, it is not allowed to divide the matrix in row-wise distribution.

In solving three-dimensional SOR, there also can be various ways of coloring elements. Suppose that a matrix element $x(i, j, k)$ is updated using its six neighbors, $x(i \pm 1, j, k)$, $x(i, j \pm 1, k)$, and $x(i, j, k \pm 1)$ and that you use two colors. The following shows three examples of coloring.

If `i+j+k=1 (mod 2)`, color `x(i,j,k)` red; otherwise color it black

If `j+k=1 (mod 2)`, color `x(i,j,k)` red; otherwise color it black

If `k=1 (mod 2)`, color `x(i,j,k)` red; otherwise color it black

Again, you should decide how to color elements based on the performance of the parallel program and sometimes the amount of work for parallelization.

4.4.3 Four-Color SOR Method

In the following program, eight neighbors are involved in updating the value of `x(i,j)`. Red-black SOR does not work for this case.

```
      PROGRAM main
      PARAMETER (mmax = ..., nmax = ...)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      DIMENSION x(0:mmax, 0:nmax)
      ...
      DO k = 1, 300
        err1 = 0.0
        DO j = 1, n
          DO i = 1, m
            temp = 0.125 * (x(i,  j-1) + x(i-1,j)
     &                    + x(i+1,j)   + x(i,  j+1)
     &                    + x(i-1,j-1) + x(i+1,j-1)
     &                    + x(i-1,j+1) + x(i+1,j+1))
     &             - x(i,j)
            x(i,j) = x(i,j) + omega * temp
            IF (abs(temp) > err1) err1 = abs(temp)
          ENDDO
        ENDDO
      IF (err1 <= eps) EXIT
      ENDDO
      ...
      END
```

You can parallelize the above program using zebra SOR method, but another method, four-color SOR, is presented in this section.

Four-color SOR is a generalization of red-black SOR, which colors `x(i,j)` with one of the four colors depending on whether `i` is even/odd and `j` is even/odd, and updates `x(i,j)` one color at a time. In the following program, "*`Update x(i,j) and err1`*" represents the italic statements in the original program.

```
      PROGRAM fourcolor
      PARAMETER (mmax = ..., nmax = ...)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      DIMENSION x(0:mmax, 0:nmax)
      ...
      DO k = 1, 300
        err1 = 0.0
        DO j = 1, n, 2
          DO i = 1, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        DO j = 1, n, 2
          DO i = 2, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        DO j = 2, n, 2
          DO i = 1, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        DO j = 2, n, 2
          DO i = 2, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        IF (err1 <= eps) EXIT
      ENDDO
      ...
      END
```

Figure 109 shows one iteration of the four-color SOR method.

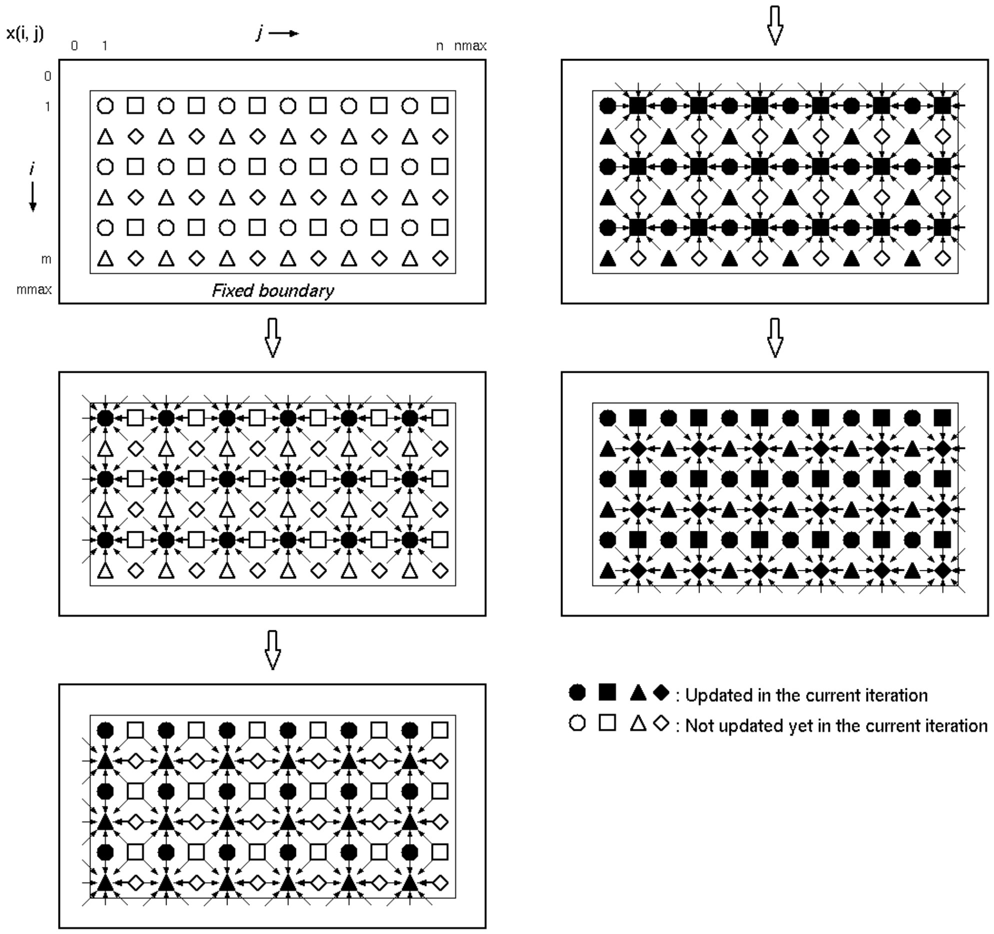
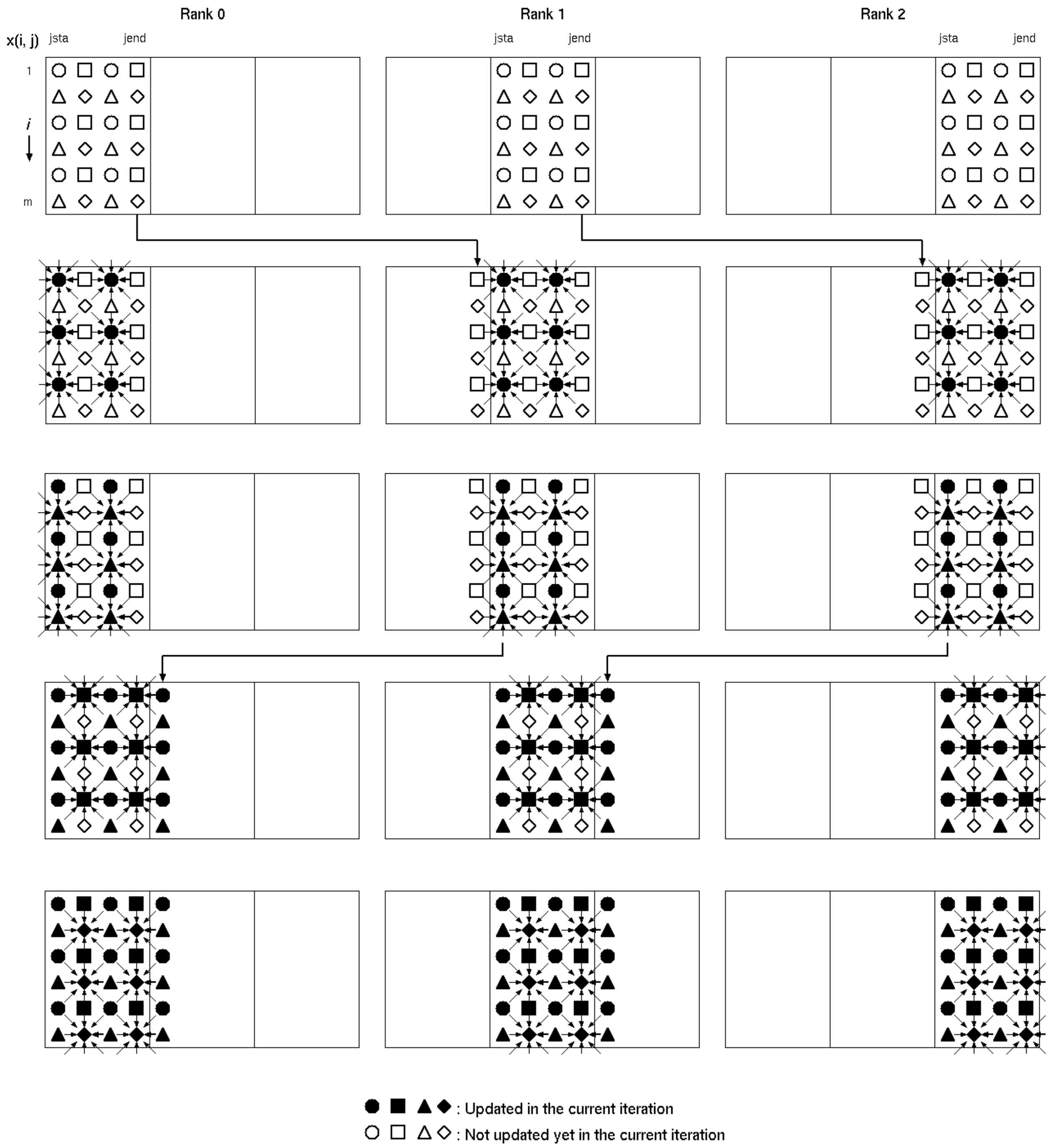

x(i, j)
j
0
1
n
nmax
i
m
mmax
Fixed boundary
● ■ ▲◆ : Updated in the current iteration
○ □ △◇ : Not updated yet in the current iteration

Figure 109. Four-Color SOR Method

Suppose you parallelize the four-color SOR program using column-wise block distribution. For the ease of programming, the number of columns (excluding the fixed boundary elements) assigned to processes should be an even integer except for process `nprocs-1`. The behavior of the parallelized program is shown in Figure 110 on page 130.



Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
1
i
m
● ■ ▲◆ : Updated in the current iteration
○ □ △◇ : Not updated yet in the current iteration

Figure 110. Four-Color SOR Method: Parallel Run

The difference from the red-black SOR method is that data transmissions take place in one direction at a time and that the whole column is transmitted instead of every other element packed into a working array. The following is the parallelized code.

```
      PROGRAM fourcolorp
      INCLUDE 'mpif.h'
      PARAMETER (mmax = ..., nmax = ...)
      PARAMETER (m = mmax - 1, n = nmax - 1)
      DIMENSION x(0:mmax, 0:nmax)
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(1, (n + 1)/2, nprocs, myrank, jsta, jend)
      jsta = jsta * 2 - 1
      jend = MIN(n, jend * 2)
      inext = myrank + 1
      iprev = myrank - 1
      IF (inext == nprocs) inext = MPI_PROC_NULL
      IF (iprev == -1)     iprev = MPI_PROC_NULL
      ...
      DO k = 1, 300
        err1 = 0.0
        CALL MPI_ISEND(x(1,jend), m, MPI_REAL,
     &       inext, 1, MPI_COMM_WORLD, ireqs, ierr)
        CALL MPI_IRECV(x(1,jsta-1), m, MPI_REAL,
     &       iprev, 1, MPI_COMM_WORLD, ireqr, ierr)
        CALL MPI_WAIT(ireqs, istatus, ierr)
        CALL MPI_WAIT(ireqr, istatus, ierr)
        DO j = jsta, jend, 2
          DO i = 1, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        DO j = jsta, jend, 2
          DO i = 2, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        CALL MPI_ISEND(x(1,jsta), m, MPI_REAL,
     &       iprev, 1, MPI_COMM_WORLD, ireqs, ierr)
        CALL MPI_IRECV(x(1,jend+1), m, MPI_REAL,
     &       inext, 1, MPI_COMM_WORLD, ireqr, ierr)
        CALL MPI_WAIT(ireqs, istatus, ierr)
        CALL MPI_WAIT(ireqr, istatus, ierr)
        DO j = jsta + 1, jend, 2
          DO i = 1, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        DO j = jsta + 1, jend, 2
          DO i = 2, m, 2
            Update x(i,j) and err1
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(err1, err2, 1, MPI_REAL,
     &                     MPI_MAX, MPI_COMM_WORLD, ierr)
        IF (err2 <= eps) EXIT
      ENDDO
      ...
      CALL MPI_FINALIZE(ierr)
      END
```

In spite of the more complex dependence compared to the original program, the parallelized four-color SOR is simpler than the parallelized red-black SOR.

4.5 Monte Carlo Method

As an example of the Monte Carlo method, a random walk in two-dimensions is considered. Issues about random number generation is the subject of this section. The following program simulates a random walk of 100,000 particles in 10 steps, and outputs the distribution of the distances that particles have traveled.

```
      PROGRAM main
      PARAMETER (n = 100000)
      INTEGER itotal(0:9)
```

```
REAL seed
pi = 3.1415926
DO i = 0, 9
  itotal(i) = 0
ENDDO
seed = 0.5
CALL srand(seed)
DO i = 1, n
  x = 0.0
  y = 0.0
  DO istep = 1, 10
    angle = 2.0 * pi * rand()
    x = x + cos(angle)
    y = y + sin(angle)
  ENDDO
  itemp = sqrt(x**2 + y**2)
  itotal(itemp) = itotal(itemp) + 1
ENDDO
PRINT *,'total =',itotal
END
```

A sample trajectory of a particle is illustrated in Figure 111.

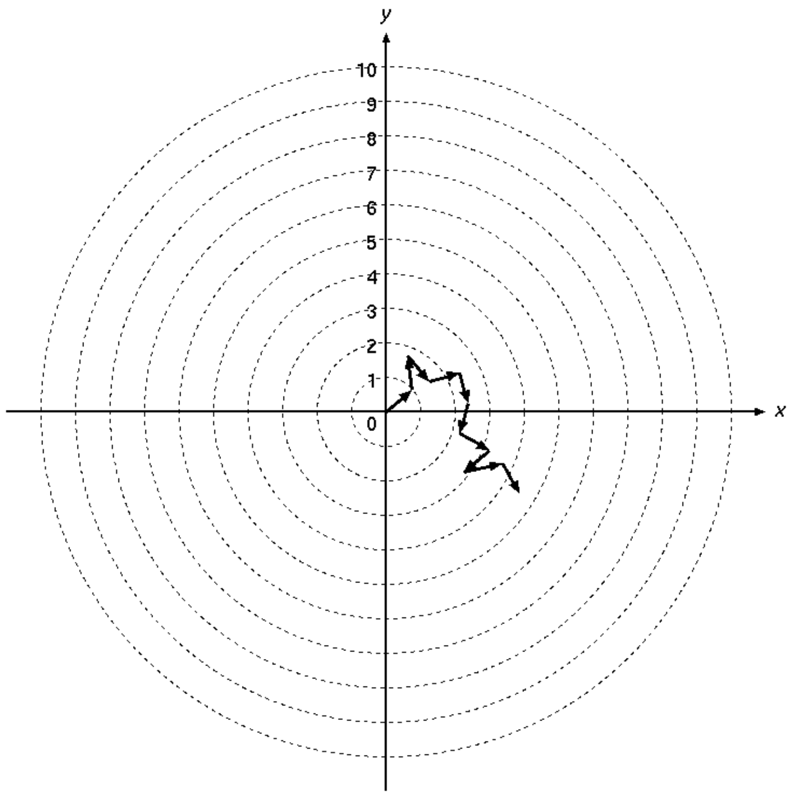


Figure 111. Random Walk in Two-Dimension

By distributing particles to processes, the program is easily parallelized. Make sure that processes use different seeds for random numbers.

```
PROGRAM main
INCLUDE 'mpif.h'
```

```
      PARAMETER (n = 100000)
      INTEGER itotal(0:9), iitotal(0:9)
      REAL seed
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL para_range(1, n, nprocs, myrank, ista, iend)
      pi = 3.1415926
      DO i = 0, 9
        itotal(i) = 0
      ENDDO
      seed = 0.5 + myrank
      CALL srand(seed)
      DO i = ista, iend
        x = 0.0
        y = 0.0
        DO istep = 1, 10
          angle = 2.0 * pi * rand()
          x = x + cos(angle)
          y = y + sin(angle)
        ENDDO
        itemp = sqrt(x**2 + y**2)
        itotal(itemp) = itotal(itemp) + 1
      ENDDO
      CALL MPI_REDUCE(itotal, iitotal, 10, MPI_INTEGER, MPI_SUM, 0,
     &                MPI_COMM_WORLD, ierr)
      PRINT *,'total =',iitotal
      CALL MPI_FINALIZE(ierr)
      END
```

Since the time spent for communication is much smaller than that of computation in the above program, it has ideal parallel speed-up. But, you have to keep in mind that the sequence of random numbers used in parallel execution is different from the one used in serial execution. Therefore, the result may be different. You might be able to use Parallel ESSL subroutine PDURNG to generate uniformly distributed random numbers for multiple processes, but if processes use different numbers of random numbers, subroutine PDURNG may not be applicable in terms of uniformity. This case happens when the underlying model allows the particles to be annihilated on the fly, for instance. One more tip for random numbers: You will get better performance if you generate a series of random numbers at one time rather than generate them one by one. There are several ESSL subroutines for this purpose.

SURAND, DURAND	Generate a vector of short-period uniformly distributed random numbers
SURXOR, DURXOR	Generate a vector of long-period uniformly distributed random numbers
SNRAND, DNRAND	Generate a vector of normally distributed random numbers

An S prefix is for single-precision and a D prefix is for double-precision.

4.6 Molecular Dynamics

In the program of molecular dynamics or the distinct element method, it is not unusual that among several loops within the outer-most time ticking loop, only a fraction of the loops account for most of the CPU time. The method described in “Pattern 1. Partial Parallelization of a DO Loop” on page 46 would be suitable for parallelizing these kind of programs.

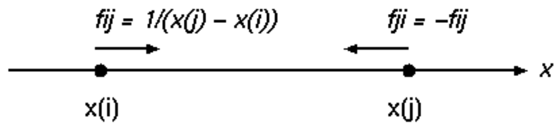


Figure 112. Interaction of Two Molecules

The following example model simulates interacting *n* particles in one dimension. The force on particle *i* from particle *j* is given by $f_{ij} = 1/(x_j - x_i)$ where x_i is the coordinate of particle *i*. The law of action and reaction applies: $f_{ij} = -f_{ji}$. Therefore, the total force acting on particle *i* is expressed as follows.

$$f_i = \sum_{j \neq i} f_{ij} = -\sum_{j<i} f_{ji} + \sum_{j>i} f_{ij}$$

The above formula is illustrated in Figure 113 for the seven particles.

f(1) =		+f12	+f13	+f14	+f15	+f16	+f17
f(2) =	–f12		+f23	+f24	+f25	+f26	+f27
f(3) =	–f13	–f23		+f34	+f35	+f36	+f37
f(4) =	–f14	–f24	–f34		+f45	+f46	+f47
f(5) =	–f15	–f25	–f35	–f45		+f56	+f57
f(6) =	–f16	–f26	–f36	–f46	–f56		+f67
f(7) =	–f17	–f27	–f37	–f47	–f57	–f67	

Figure 113. Forces That Act on Particles

The serial program calculates forces for particles and updates their coordinates in each time step. Using the antisymmetry, f_{ij} is only calculated for $i<j$.

```
...
PARAMETER (n = ...)
REAL f(n), x(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  DO i = 1, n-1
    DO j = i+1, n
      fij = 1.0 / (x(j)-x(i))
      f(i) = f(i) + fij
      f(j) = f(j) - fij
    ENDDO
```

```
      ENDDO
      DO i = 1, n
        x(i) = x(i) + f(i)
      ENDDO
    ENDDO
    ...
```

Note that within the time ticking loop, the second loop is the hot spot, which calculates the force. The second loop is a doubly-nested loop, which is not suitable for block distribution because the range of iteration variables `(i,j)` is limited to the “upper triangle” in Figure 113 on page 134. Two options of parallelizing the second loop are shown below: cyclic distribution regarding `i` and cyclic distribution regarding `j`. Which option performs better depends on various factors such as workload balance and cache misses, so it cannot be concluded here which is better. The following program parallelizes the second loop with respect to the outer loop by cyclic distribution.

```
    ...
    PARAMETER (n = ...)
    REAL f(n), x(n), ff(n)
    ...
    DO itime = 1, 100
      DO i = 1, n
        f(i) = 0.0
      ENDDO
      DO i = 1 + myrank, n-1, nprocs
        DO j = i+1, n
          fij = 1.0 / (x(j)-x(i))
          f(i) = f(i) + fij
          f(j) = f(j) - fij
        ENDDO
      ENDDO
      CALL MPI_ALLREDUCE(f, ff, n, MPI_REAL, MPI_SUM,
   &                    MPI_COMM_WORLD, ierr)
      DO i = 1, n
        x(i) = x(i) + ff(i)
      ENDDO
    ENDDO
    ...
```

Figure 114 illustrates three-process execution for seven particles.

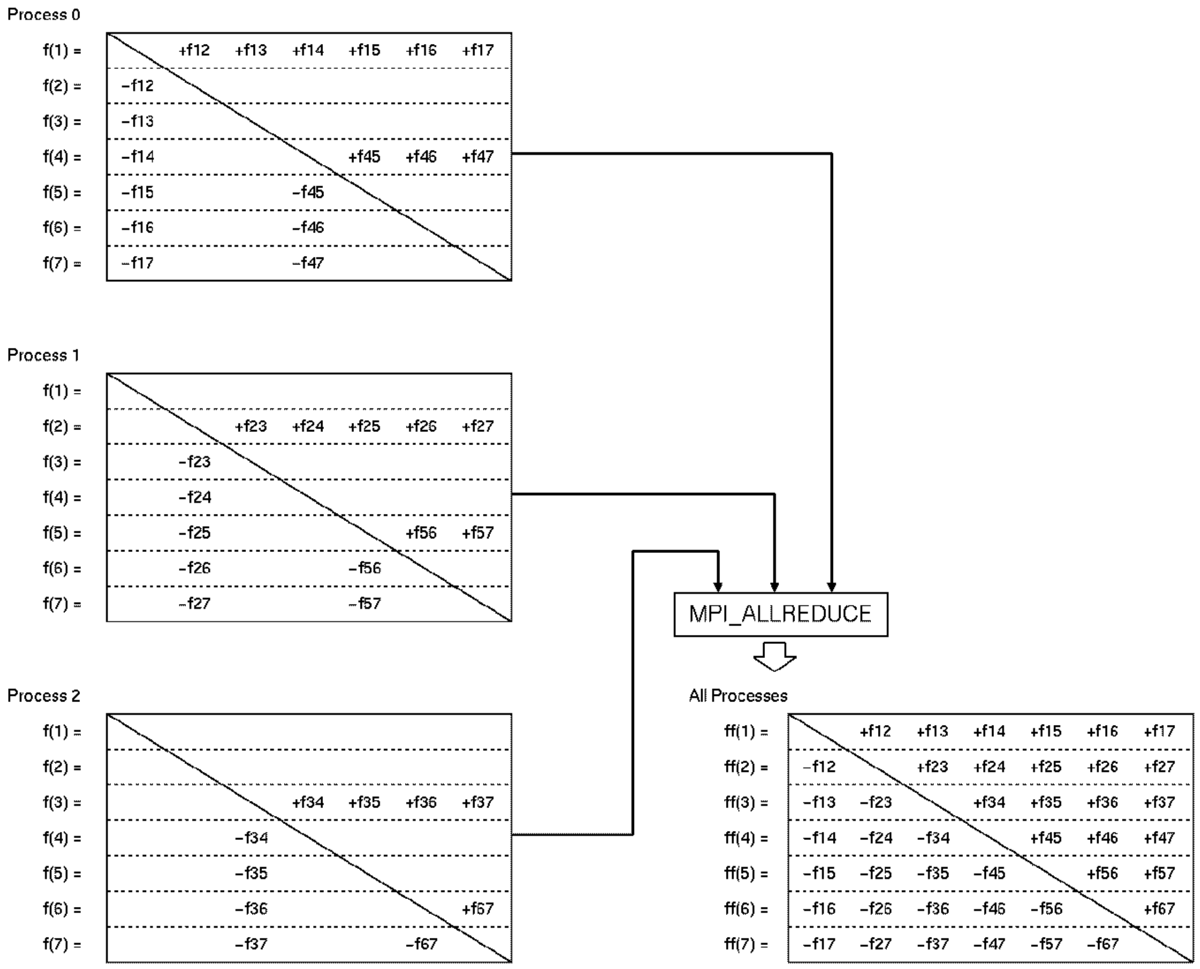

Process 0
f(1) = +f12 +f13 +f14 +f15 +f16 +f17
f(2) = -f12
f(3) = -f13
f(4) = -f14 +f45 +f46 +f47
f(5) = -f15 -f45
f(6) = -f16 -f46
f(7) = -f17 -f47
Process 1
f(2) = +f23 +f24 +f25 +f26 +f27
f(3) = -f23
f(4) = -f24
f(5) = -f25 +f56 +f57
f(6) = -f26 -f56
f(7) = -f27 -f57
Process 2
f(3) = +f34 +f35 +f36 +f37
f(4) = -f34
f(5) = -f35
f(6) = -f36 +f67
f(7) = -f37 -f67
MPI_ALLREDUCE
All Processes
ff(1) = +f12 +f13 +f14 +f15 +f16 +f17
ff(2) = -f12 +f23 +f24 +f25 +f26 +f27
ff(3) = -f13 -f23 +f34 +f35 +f36 +f37
ff(4) = -f14 -f24 -f34 +f45 +f46 +f47
ff(5) = -f15 -f25 -f35 -f45 +f56 +f57
ff(6) = -f16 -f26 -f36 -f46 -f56 +f67
ff(7) = -f17 -f27 -f37 -f47 -f57 -f67

Figure 114. Cyclic Distribution in the Outer Loop

The following program parallelizes the second loop regarding the inner loop by cyclic distribution.

```
...
PARAMETER (n = ...)
REAL f(n), x(n), ff(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  irank = -1
  DO i = 1, n-1
    DO j = i+1, n
      irank = irank + 1
      IF (irank == nprocs) irank = 0
      IF (myrank == irank) THEN
        fij = 1.0 / (x(j)-x(i))
        f(i) = f(i) + fij
        f(j) = f(j) - fij
      ENDIF
```

```
          ENDDO
        ENDDO
        CALL MPI_ALLREDUCE(f, ff, n, MPI_REAL, MPI_SUM,
     &                       MPI_COMM_WORLD, ierr)
        DO i = 1, n
          x(i) = x(i) + ff(i)
        ENDDO
      ENDDO
      ...
```

Figure 115 illustrates three-process execution for seven particles.

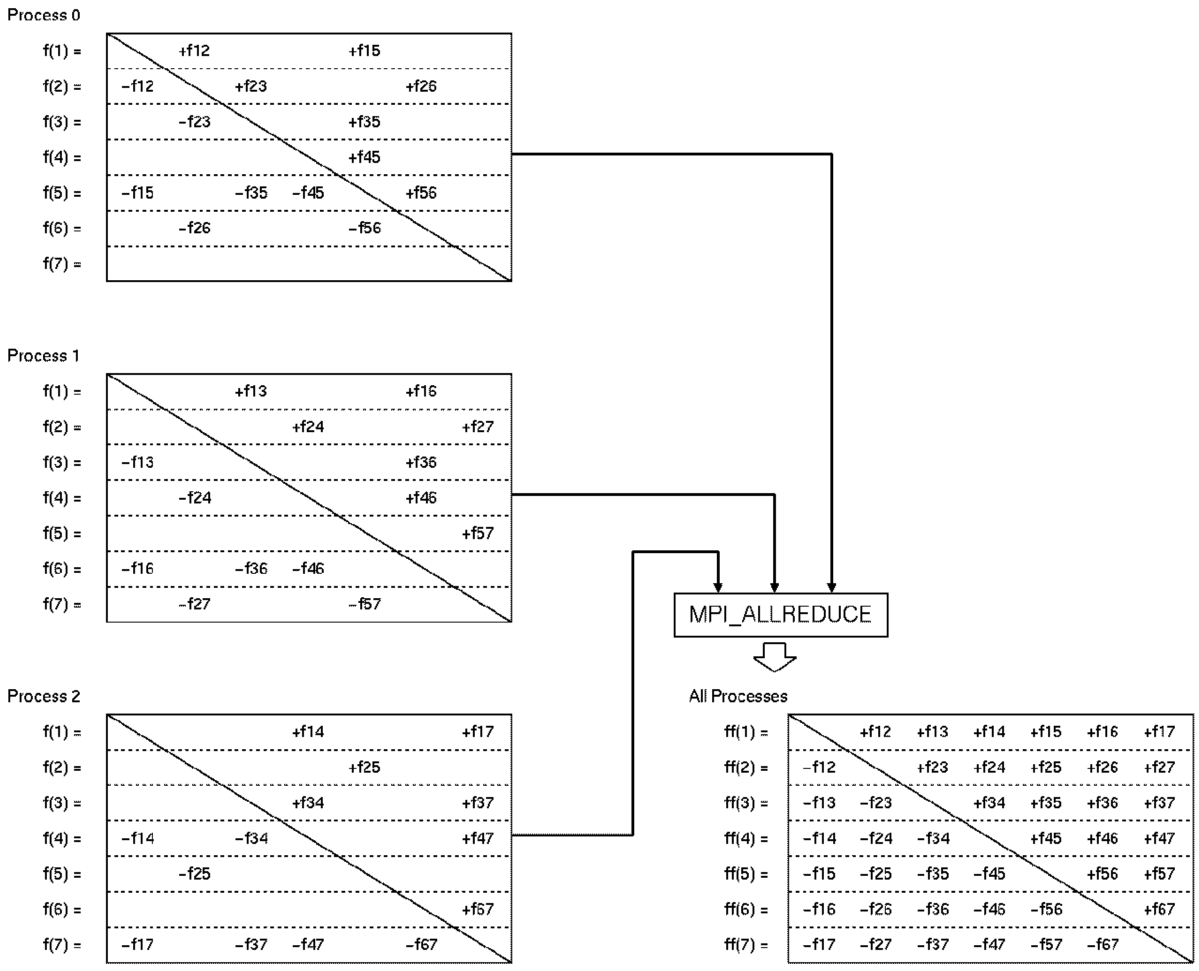


Figure 115. Cyclic Distribution of the Inner Loop

In the parallelized programs, one call of MPI_ALLREDUCE for *n* variables (f_i) per *n(n-1)/2* computations of f_{ij} would not affect the performance seriously.

4.7 MPMD Models

This section describes how to write and run programs in the MPMD (Multiple Programs Multiple Data) model. Unlike the SPMD (Single Program Multiple Data) model, different programs run in parallel and communicate with each other in the MPMD model. In Parallel Environment for AIX, set the value of environment variable MPI_PGMMODEL as “spmd” (default) or “mpmd” in order to select which model to use. Suppose you want to run a program for a coupled analysis, which consists of fluid dynamics and structural analysis. Prepare two programs, `fluid.f` and `struct.f`, and do as follows.

```
$ mpxlf fluid.f  -o fluid
$ mpxlf struct.f -o struct
(Copy the executables to remote nodes if necessary)
$ export MP_PGMMODEL=mpmd
$ export MP_CMDFILE=cmdfile
$ poe -procs 2
```

Environment variables which are irrelevant to MPMD execution are not shown. The contents of a sample `cmdfile` (file name is arbitrary) is shown below.

```
fluid
struct
```

In the command file, the executables are specified in the order of ranks. With this file, process 0 executes the `fluid` program and process 1 executes `struct` program. As described in Chapter 2.6, “Managing Groups” on page 36, you may have to create groups for fluid dynamics processes and structural analysis processes so that processes can call collective communication subroutines within each group. Figure 116 illustrates MPMD execution of `fluid` and `struct`.

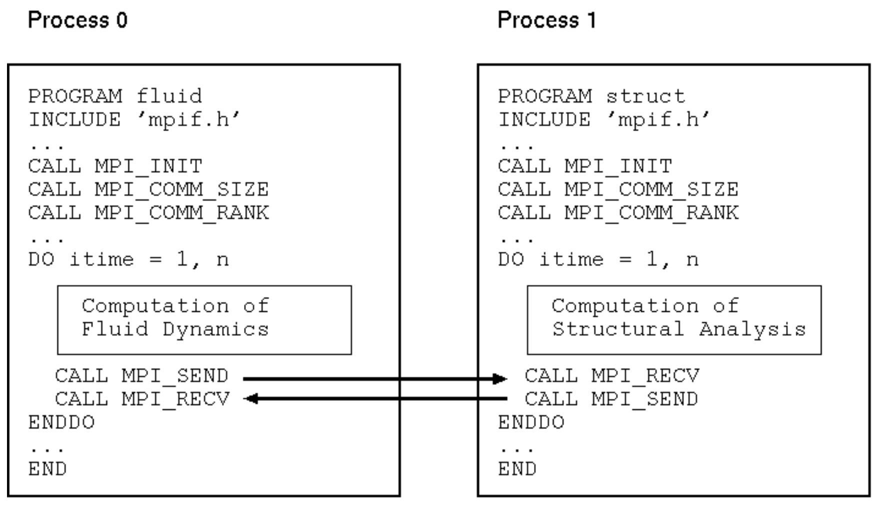
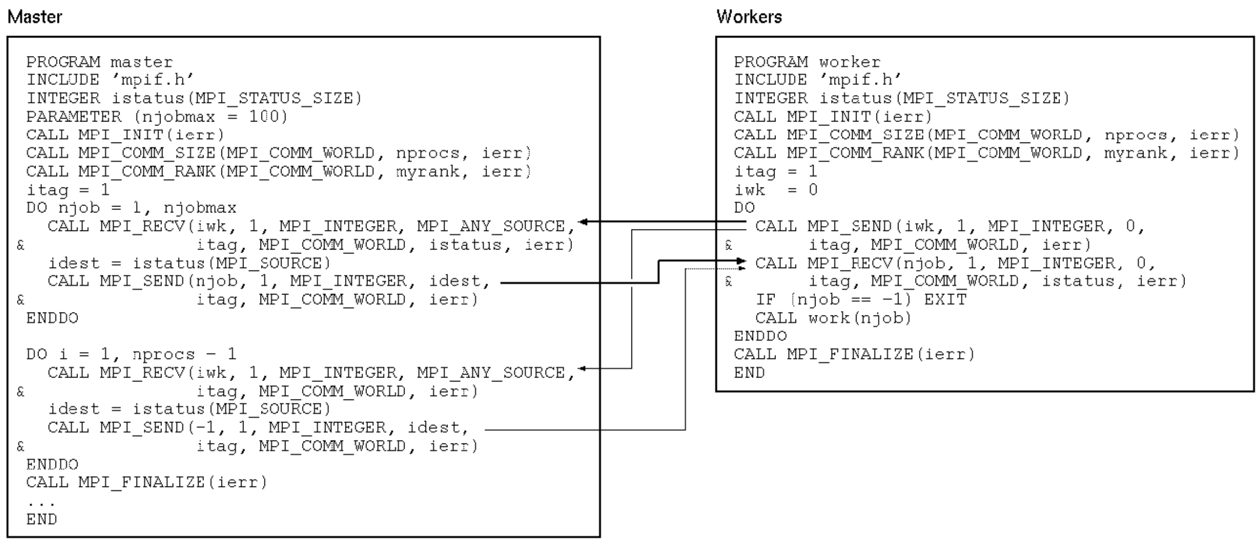


Figure 116. MPMD Model

The remainder of this section focuses on master/worker MPMD programs, where one process, the master, coordinates the execution of all the others, the workers. Consider the following program.

```
PROGRAM main
PARAMETER (njobmax = 100)
DO njob = 1, njobmax
  CALL work(njob)
ENDDO
...
END
```

There are 100 jobs to do and subroutine `work` does some computation according to the argument `njob`. Assume that the jobs are independent of each other so that they can be processed concurrently. Suppose that you want to parallelize this program. What if the processing time varies significantly from job to job, and neither block nor cyclic distribution does a good job in load balancing? Or what if you are using a heterogeneous environment where performance of machines is not uniform? Master/worker style of execution is suitable for such cases.



```
 PROGRAM master
 INCLUDE 'mpif.h'
 INTEGER istatus(MPI_STATUS_SIZE)
 PARAMETER (njobmax = 100)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 itag = 1
 DO njob = 1, njobmax
   CALL MPI_RECV(iwk, 1, MPI_INTEGER, MPI_ANY_SOURCE,
&               itag, MPI_COMM_WORLD, istatus, ierr)
   idest = istatus(MPI_SOURCE)
   CALL MPI_SEND(njob, 1, MPI_INTEGER, idest,
&               itag, MPI_COMM_WORLD, ierr)
 ENDDO

 DO i = 1, nprocs - 1
   CALL MPI_RECV(iwk, 1, MPI_INTEGER, MPI_ANY_SOURCE,
&               itag, MPI_COMM_WORLD, ierr)
   idest = istatus(MPI_SOURCE)
   CALL MPI_SEND(-1, 1, MPI_INTEGER, idest,
&               itag, MPI_COMM_WORLD, ierr)
 ENDDO
 CALL MPI_FINALIZE(ierr)
 ...
 END
```

```
 PROGRAM worker
 INCLUDE 'mpif.h'
 INTEGER istatus(MPI_STATUS_SIZE)
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
 itag = 1
 iwk  = 0
 DO
   CALL MPI_SEND(iwk, 1, MPI_INTEGER, 0,
&       itag, MPI_COMM_WORLD, ierr)
   CALL MPI_RECV(njob, 1, MPI_INTEGER, 0,
&       itag, MPI_COMM_WORLD, istatus, ierr)
   IF (njob == -1) EXIT
   CALL work(njob)
 ENDDO
 CALL MPI_FINALIZE(ierr)
 END
```

Figure 117. Master/Worker Model

Figure 117 shows a master/worker implementation of the original program. The master (process 0) dispatches jobs to workers in the first-come first-ordered basis. When a worker finishes a job, it sends a message to the master. By specifying MPI_ANY_SOURCE as the argument of MPI_RECV, the master process receives any message that comes first. After receiving the message, the sender is identified by looking into the status array. Messages sent from a worker to the master contains dummy data (`iwk`), because the message is only used to find out which worker gets free. In return, the master sends the worker the job number that the worker should do next. When all the jobs are completed, the master sends `-1` to workers, which lets workers exit from the infinite loop waiting for more jobs.

4.8 Using Parallel ESSL

Parallel ESSL is a scalable mathematical subroutine library that supports parallel processing applications on RS/6000 SP systems and clusters of RS/6000 workstations. It is highly tuned for POWER2 and PowerPC (including POWER3) processors and the RS/6000 SP architecture. It is recommended to use this library where applicable.

4.8.1 ESSL

Parallel ESSL is based on ESSL (Engineering and Scientific Subroutine Library). Indeed, some of the Parallel ESSL subroutines call ESSL subroutines internally. Before using Parallel ESSL, remember that it may happen that your program can be parallelized using ESSL subroutines. In such cases, you need to assess which option gives you a benefit. Given below are two examples that use ESSL subroutines for parallelization.

Matrix multiplication

Both ESSL and Parallel ESSL have subroutines for computing a product of two matrices, DGEMM and PDGEMM, for example. But parallel execution does not necessarily imply the use of Parallel ESSL. Figure 118 illustrates how a serial ESSL subroutine is used for parallel execution.

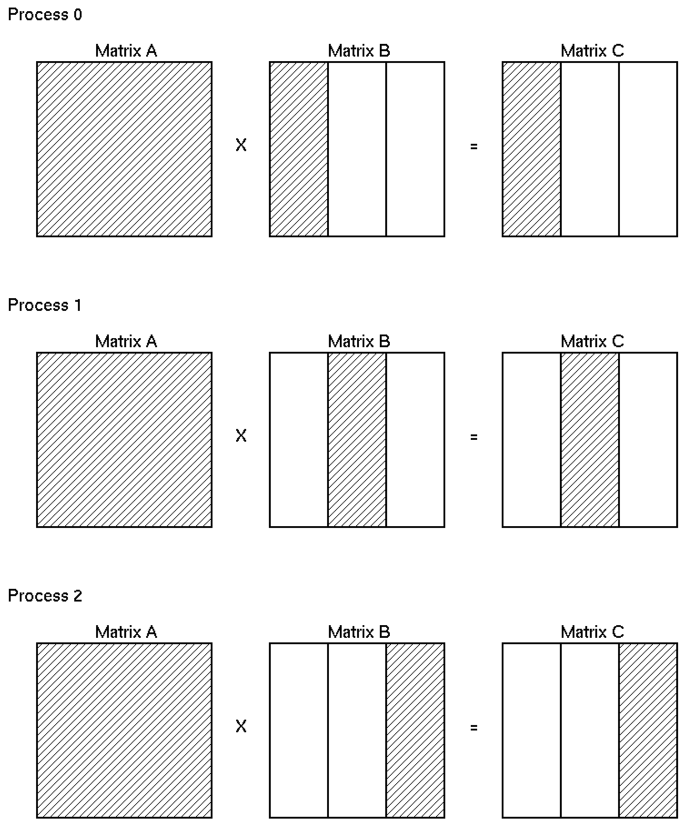


Figure 118. Using ESSL for Matrix Multiplication

In this figure, each process has all the elements of matrix `A` and one-third of `B` and `C`. Due to the property of matrix multiplication, each one-third piece of `C` can be calculated locally by using an ESSL subroutine.

Solving independent linear equations

Suppose that you have to solve six independent linear equations $A_i x_i = b_i$ ($i=1..6$) using three processes. For linear equations, there is an ESSL subroutine DGETRF and a Parallel ESSL subroutine PDGETRF. As Figure 119 shows, when using Parallel ESSL, communication overhead is inevitable, whereas using ESSL involves no communication at all. In this case, you should use ESSL (with the viewpoint of performance) unless the matrices are too large to fit in the real memory of one node.

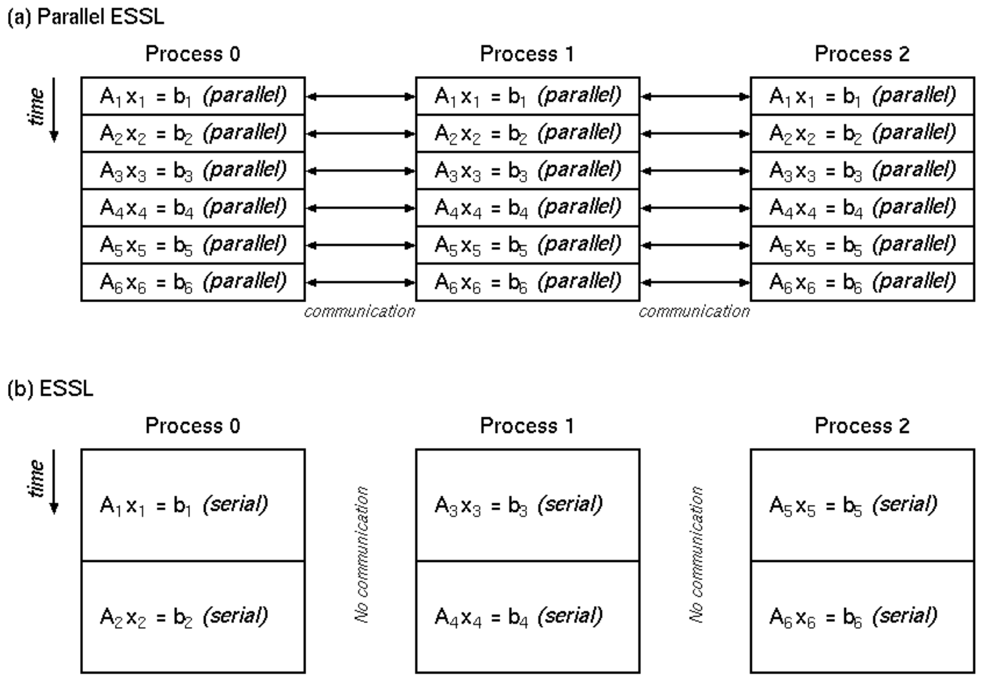


Figure 119. Using ESSL for Solving Independent Linear Equations

For additional information about ESSL, see *Engineering and Scientific Subroutine Library for AIX Guide and Reference*, SA22-7272.

4.8.2 An Overview of Parallel ESSL

Parallel ESSL provides subroutines in the following areas.

- Level 2 Parallel Basic Linear Algebra Subprograms (PBLAS)
- Level 3 PBLAS
- Linear Algebraic Equations
- Eigensystem Analysis and Singular Value Analysis
- Fourier Transforms
- Random Number Generation

The subroutines run under the AIX operating system and can be called from application programs written in Fortran, C, C++, and High Performance Fortran (HPF). For communication, Parallel ESSL includes the Basic Linear Algebra Communications Subprograms (BLACS), which use the Parallel Environment (PE) Message Passing Interface (MPI). For computations, Parallel ESSL uses the ESSL for AIX subroutines. Therefore in linking the program, you need to specify ESSL and BLACS libraries as well as the Parallel ESSL library. The following is a sample compilation and execution on POWER2 nodes.

```
$ mpxlf -O3 -qstrict -qarch=pwr2 -lesslp2 -lpesslp2 -lblacsp2
sample.f
$ a.out -procs 3
```

For POWER3 nodes, do the following.

```
$ mpxlf -O3 -qstrict -qarch=pwr3 -lessl -lpessl -lblacs sample.f
$ a.out -procs 3
```

If you have three POWER3 SMP nodes with two processors and your only concern is the subroutine call to Parallel ESSL, there are three options for compiling and executing the program.

The first option is the previous example where one User Space process runs on each node.

The second option is to use the same executable as the first option but to run two processes on each node, so there are six processes in total. As discussed in Chapter 1.2.3, “MPP Based on SMP Nodes (Hybrid MPP)” on page 4, you have to consider the performance of intranode communication and internode communication in this case; it is possible that the first option is faster than the second although the first one uses half the number of processes.

The third option is to use SMP-enabled Parallel ESSL subroutines, which is available by linking appropriate libraries.

```
$ mpxlf_r -O3 -qstrict -qarch=pwr3 -lesslsmp -lpesslsmp
-lblacssmp sample.f
$ a.out -procs 3
```

Use `mpxlf_r` in order to link to thread-safe libraries.

If you are using Parallel ESSL Version 1, you might get a performance improvement by specifying MP_CSS_INTERRUPT as “yes” (default is “no”). But the behavior of the code other than the Parallel ESSL subroutines is unknown. So, compare the two cases and take the faster one. If you want to turn on and off MP_CSS_INTERRUPT dynamically, you can use MP_ENABLEINTR and MP_DISABLEINTR subroutines in the program. See *MPI Programming and Subroutine Reference Version 2 Release 4*, GC23-3894 for details.

If you are using Parallel ESSL Version 2, MP_CSS_INERRUPT is set dynamically by Parallel ESSL subroutines. Therefore, you only need to set it if it improves the performance of the other part of the program.

In addition, if you are using the MPI threaded library and only a single message passing thread, specify MP_SINGLE_THREAD=yes to minimize thread overhead. More detailed information can be found in *Parallel Engineering and Scientific Subroutine Library for AIX Guide and Reference*, SA22-7273.

4.8.3 How to Specify Matrices in Parallel ESSL

Parallel ESSL requires shrunk arrays as input in a complicated manner, and you need to call several BLACS subroutines to prepare arrays and their data structures for Parallel ESSL. For a matrix *A*, each process holds a shrunk matrix *a*, where *A* and *a* are called the global matrix and local matrix, respectively.

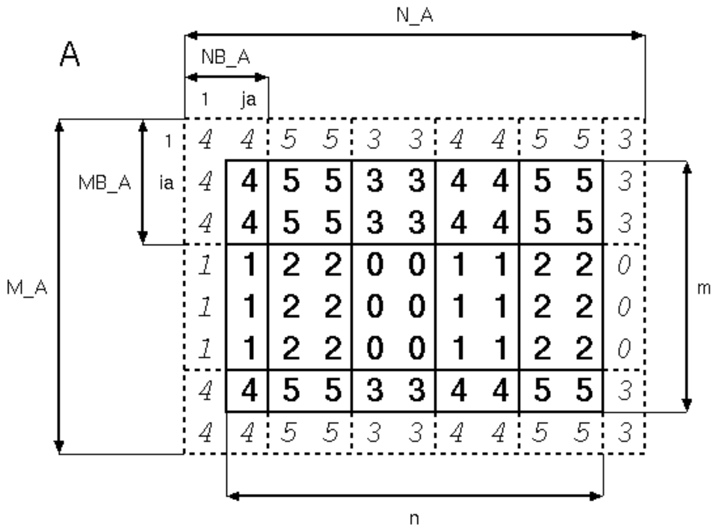


Figure 120. Global Matrix

Figure 120 illustrates a sample global matrix. The global matrix is an imaginary entity. In reality, it is stored as a combination of local matrices in processes, the process grid, and the array descriptor, which are described shortly. Note that the following setting is for Parallel ESSL subroutines related to dense matrices. For subroutines dealing with sparse matrices, another data structure is used.

The global matrix *A* with size `M_A x N_A` is divided into blocks by the block-cyclic distribution. Each block is an `MB_A x NB_A` matrix. The assignment of blocks to processes is determined by the process grid, which is identified by an integer called *context*. In Figure 121 on page 144 (a), the context is represented by a variable `CTXT_A`. The process grid is *patched* to the global matrix block by block. In doing so, process (`RSRC_A`, `CSRC_A`) in the process grid (process 4 in the case of Figure 121 (a)) is adjusted to the upper left block of the global matrix. In Figure 120, the number in the global matrix indicates which rank the element belongs to. Then the blocks belonging to the same process are packed together conserving their relative order in the global matrix. This packed matrix is nothing more than the local matrix. The leading dimension of the local matrix, that is, the number of rows of the local matrix, is specified by `LLD_A`. The value of `LLD_A` may be different from process to process. The eight values, `M_A`, `N_A`, `MB_A`, `NB_A`, `CTXT_A`, `RSRC_A`, `CSRC_A`, and `LLD_A`, combined with the descriptor type `DTYPE_A=1` constitute the array descriptor (Figure 121 (b)), which is an integer array with nine elements. A type 1 array descriptor is used in the Level 2 and 3 PBLAS, dense linear algebraic equations, and eigensystem analysis and singular value analysis subroutines. Other than type 1, there are type 501 and type 502, which are used for subroutine PDPBSV (positive definite symmetric band matrix factorization and solve) and so on.

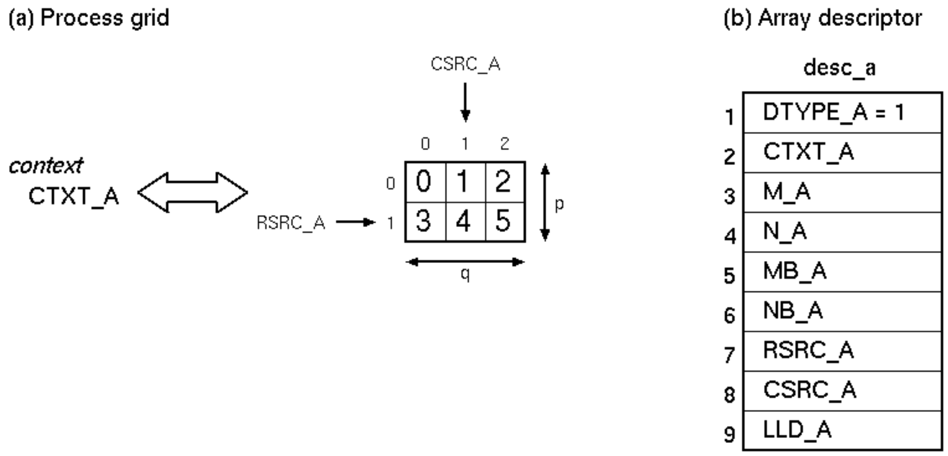


Figure 121. The Process Grid and the Array Descriptor

Important

The format of the descriptor array given here is for Parallel ESSL Version 2.1. If you are using a previous version of Parallel ESSL, consult the reference manual for the difference.

Given the process grid and the array descriptor shown in Figure 121, the local matrix representation of the global matrix in Figure 120 becomes as follows.

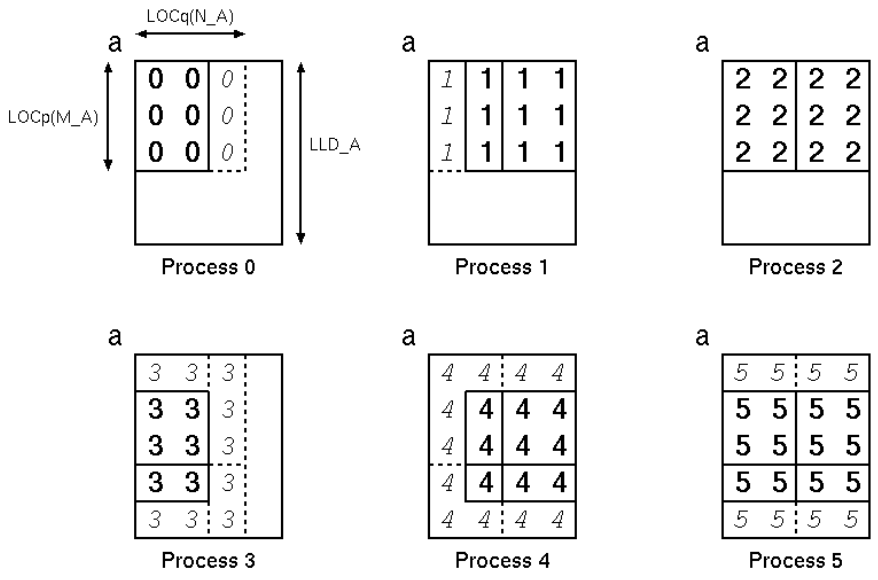


Figure 122. Local Matrices

Parallel ESSL subroutines can take either the entire global matrix or a part of it for computation. You have to specify the submatrix which a Parallel ESSL subroutine operates on. In Figure 120 on page 143 and Figure 122, the elements within this submatrix are printed in boldface, and the other elements are in italic.

When you call a Parallel ESSL subroutine, the following four integers are necessary.

`m`	The number of rows in the submatrix
`n`	The number of columns in the submatrix
`ia`	The row index of the global matrix, identifying the first row of the submatrix
`ja`	The column index of the global matrix, identifying the first column of the submatrix

There are two things missing in the above description. How do you specify the process grid? And do you need to calculate the effective size of the local matrix by yourself (see `LOCp(M_A)` and `LOCq(N_A)` in Figure 122). The answers to these questions are given in the next section.

4.8.4 Utility Subroutines for Parallel ESSL

The following is a reference of several utility subroutines for creating process grids and getting information from them.

4.8.4.1 BLACS_GET

Usage

```
CALL BLACS_GET(0, 0, icontxt)
```

Parameters

INTEGER icontxt The default system context. (OUT)

Description The most common use is shown in the above box. It retrieves a default system context for input into BLACS_GRIDINIT or BLACS_GRIDMAP. All processes that will be involved in `icontxt` must call this routine.

4.8.4.2 BLACS_GRIDINIT

Usage

```
CALL BLACS_GRIDINIT(icontxt, order, nprow, npcol)
```

Parameters

INTEGER icontxt The system context to be used in creating the BLACS context and a newly created context is returned. (INOUT)

CHARACTER(1) order Indicates how to map processes into the process grid. If `order='R'`, row-major natural ordering is used. This is the default. If `order='C'`, column-major natural ordering is used. (IN)

INTEGER nprow The number of rows in this process grid. (IN)

INTEGER npcol	The number of columns in this process grid. (IN)
Description	You call the BLACS_GRIDINIT routine when you want to map the processes sequentially in row-major order or column-major order into the process grid. You must specify the same input argument values in the calls to BLACS_GRIDINIT on every process.

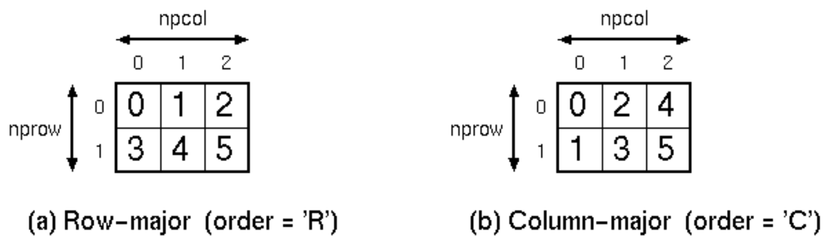


Figure 123. Row-Major and Column-Major Process Grids

Figure 123 shows row-major and column-major grids of six processes with the shape 2x3.

4.8.4.3 BLACS_GRIDINFO

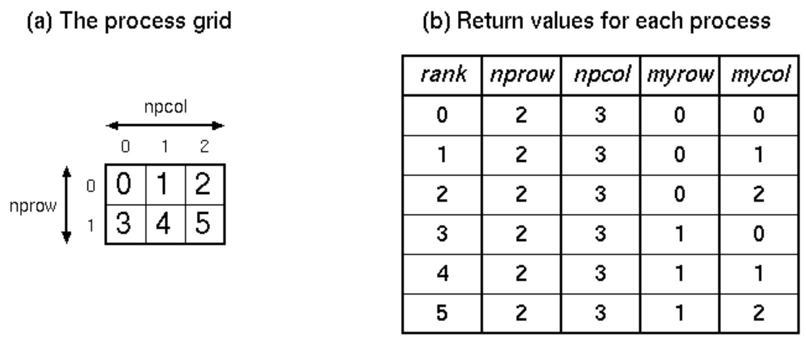
Usage

```
CALL BLACS_GRIDINFO(icontxt, nprow, npcol, myrow, mycol)
```

Parameters

INTEGER icontxt	The context that you want to get the information of. (IN)
INTEGER nprow	The number of rows in this process grid. (OUT)
INTEGER npcol	The number of columns in this process grid. (OUT)
INTEGER myrow	The process grid row index of the process which calls this subroutine. (OUT)
INTEGER mycol	The process grid column index of the process which calls this subroutine. (OUT)

Description Call this subroutine to obtain the process row and column index.



rank	*nprow*	*npcol*	*myrow*	*mycol*
0	2	3	0	0
1	2	3	0	1
2	2	3	0	2
3	2	3	1	0
4	2	3	1	1
5	2	3	1	2

Figure 124. BLACS_GRIDINFO

Figure 124 on page 147 shows a process grid and the return values of BLACS_GRIDINFO for processes in the grid.

4.8.4.4 NUMROC

Usage

```
num = NUMROC(n, nb, iproc, isrcproc, nprocs)
```

Parameters

INTEGER n	The number of rows (M_A) or columns (N_A) in a global matrix that has been block-cyclically distributed. (IN)
INTEGER nb	The row block size (MB_A) or the column block size (NB_A). (IN)
INTEGER iproc	The process row index (myrow) or the process column index (mycol). (IN)
INTEGER isrcproc	The process row (RSRC_A) or the process column (CSRC_A) over which the first row or column, respectively, of the global matrix is distributed. (IN)
INTEGER nprocs	The number of rows (nprow) or the number of columns (npcol) in the process grid. (IN)
INTEGER num	The local number of rows or columns of a block-cyclically distributed matrix contained in a process row or process column, respectively, indicated by the calling sequence argument iproc. (OUT)
Description	This function computes either the local number of rows, LOCp(M_A), or columns, LOCq(N_A). If you need both, you have to call this function twice. See also Figure 122 on page 144.

The program sequence of calling a Parallel ESSL subroutine is outlined as follows:

```
INTEGER desc_a(9)
...
DIMENSION a(LLD_A, ...)
CALL BLACS_GET(0, 0, CTXT_A)
CALL BLACS_GRIDINIT(CTXT_A, 'R', p, q)
CALL BLACS_GRIDINFO(CTXT_A, p, q, myrow, mycol)
LOCp      = NUMROC(M_A, MB_A, myrow, RSRC_A, p)
LOCq      = NUMROC(N_A, NB_A, mycol, CSRC_A, q)
desc_a(1) = 1
desc_a(2) = CTXT_A
desc_a(3) = M_A
desc_a(4) = N_A
desc_a(5) = MB_A
desc_a(6) = NB_A
desc_a(7) = RSRC_A
desc_a(8) = CSRC_A
desc_a(9) = LLD_A
Each process set values of its local matrix a()
CALL pessl_subroutine (..., m, n, a, ia, ja, desc_a,...)
...
```

4.8.5 LU Factorization by Parallel ESSL

This section takes an LU factorization as an example of how to use Parallel ESSL in real programs. There are two subroutines involved in solving a dense linear equation: PDGETRF factorizes the matrix and PDGETRS gets the solution based on the results of PDGETRF.

Usage

```
CALL PDGETRF(m, n, a, ia, ja, desc_a, ipvt, info)
```

Parameters

INTEGER m	The number of rows in submatrix A and the number of elements in vector IPVT used in the computation. (IN)
INTEGER n	The number of columns in submatrix A used in the computation. (IN)
REAL*8 a()	The local part of the global general matrix A, used in the system of equations. This identifies the first element of the local array a. On return from the subroutine call, a is replaced by the results of the factorization. (INOUT)
INTEGER ia	The row index of the global matrix A, identifying the first row of the submatrix A. (IN)
INTEGER ja	The column index of the global matrix A, identifying the first column of the submatrix A. (IN)
INTEGER desc_a()	The array descriptor for global matrix A. (IN)
INTEGER ipvt()	The local part of the global vector IPVT, containing the pivot information necessary to construct matrix L from the information contained in the (output) transformed matrix A. This identifies the first element of the local array ipvt. (OUT)

INTEGER info	If info=0, global submatrix A is not singular, and the factorization completed normally. If info>0, global submatrix A is singular. Even so, the factorization is completed. However, if you call PDGETRS with these factors, results are unpredictable. (OUT)

Description This subroutine factors double-precision general matrix *A* using Gaussian elimination with partial pivoting to compute the *LU* factorization of *A*. All processes involved in the computation need to call this subroutine. The global general matrix *A* must be distributed using a square block-cyclic distribution; that is, MB_A=NB_A. If you plan to call PDGETRS, the solver, after calling PDGETRF, `m=n` must hold.

Performance Consideration

If the size of the global matrix is large enough, the suggested block size is 70 for POWER2 nodes and 40 for POWER and POWER3 nodes. If you link the program to `-lpesslsmp -lesslsmp` and use POWER3 SMP nodes, the recommended block size is 100. The shape of the processor grid is suggested to be square or as close to square as possible for better performance. If the grid is not square, let the number of rows (*p*) be less than the number of columns (*q*) in the grid (meaning, *p*<*q*). See “Coding Tips for Optimizing Parallel Performance” in *Parallel Engineering and Scientific Subroutine Library for AIX Guide and Reference*, SA22-7273 for details.

Usage

```
CALL PDGETRS(trans, n, nrhs, a, ia, ja, desc_a, ipvt,
             b, ib, jb, desc_b, info)
```

Parameters

CHARACTER(1) trans	If `trans='N'`, *A* is used in the computation. If `trans='T'`, A^T is used in the computation. (IN)
INTEGER n	The order of the factored matrix *A* and the number of rows in submatrix *B*. (IN)
INTEGER nrhs	The number of right-hand sides, that is, the number of columns in submatrix *B* used in the computation. (IN)
REAL*8 a()	The local part of the global general matrix *A*, containing the factorization of matrix *A* produced by a preceding call to PDGETRF. This identifies the first element of the local array *a*. (IN)
INTEGER ia	The row index of the global matrix A, identifying the first row of the submatrix A. (IN)
INTEGER ja	The column index of the global matrix A, identifying the first column of the submatrix A. (IN)
INTEGER desc_a()	The array descriptor for global matrix A. (IN)
INTEGER ipvt()	The local part of the global vector IPVT, containing the pivoting indices produced on a preceding call to

	PDGETRF. This identifies the first element of the local array *ipvt*. (IN)
REAL*8 b()	The local part of the global general matrix *B*, containing the right-hand sides of the system. This identifies the first element of the local array *b*. On return from the subroutine call, it is replaced by the updated local part of the global matrix *B*, containing the solution vectors. (INOUT)
INTEGER ib	The row index of the global matrix *B*, identifying the first row of the submatrix *B*. (IN)
INTEGER jb	The column index of the global matrix *B*, identifying the first column of the submatrix *B*. (IN)
INTEGER desc_b()	The array descriptor for global matrix *B*. (IN)
INTEGER info	`info=0` is always returned. (OUT)
Description	PDGETRS solves one of the following systems of equations for multiple right-hand sides: $AX = B$ or $A^T X = B$. It uses the results of the factorization of matrix *A*, produced by a preceding call to PDGETRF. The following relations must hold: CTXT_A=CTXT_B and MB_A=NB_A=MB_B. In addition, in the process grid, the process row containing the first row of the submatrix *A* must also contain the first row of the submatrix *B*, which means RSRC_A=RSRC_B in the case of ia=ib=1.

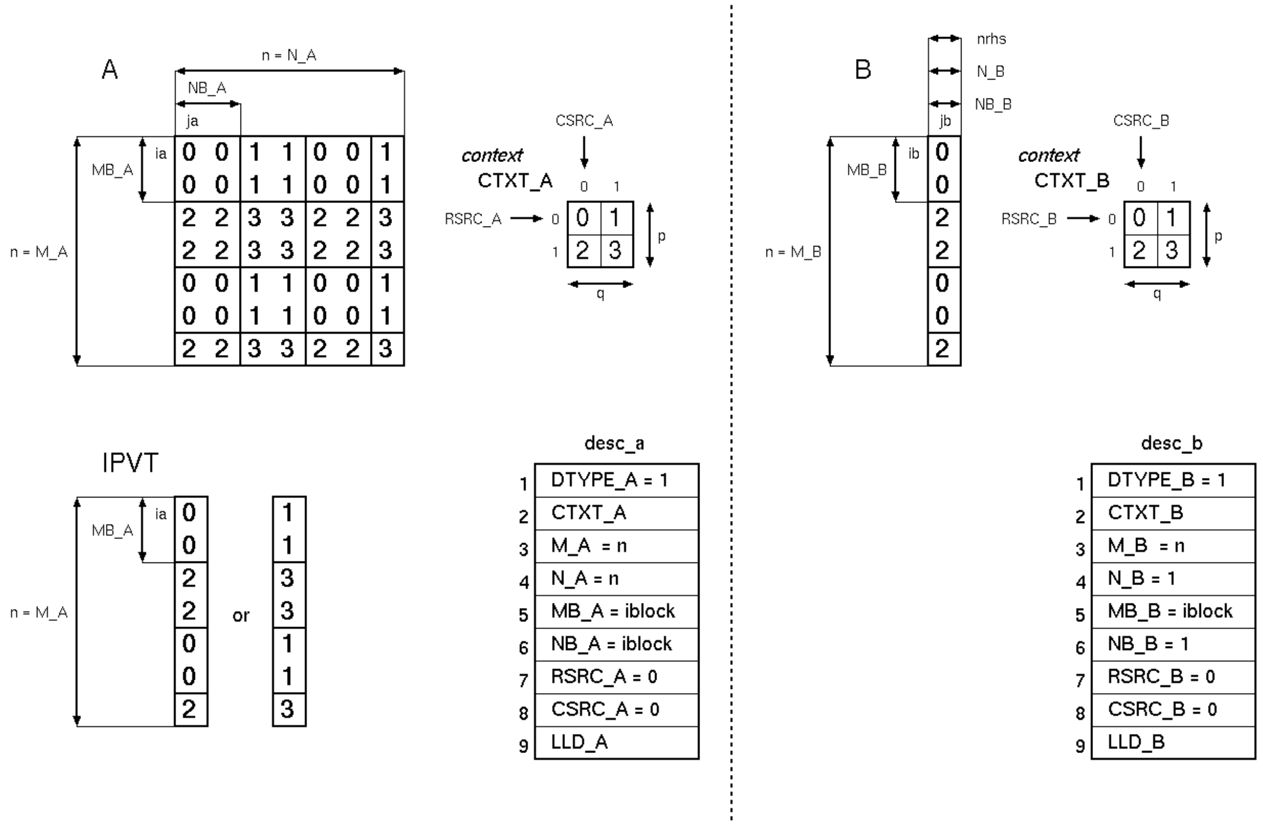


Figure 125. Global Matrices, Processor Grids, and Array Descriptors

Figure 125 shows how the global matrix, the pivot vector, and the right-hand-side vector are divided into blocks by the definition of processor grids and array descriptors.

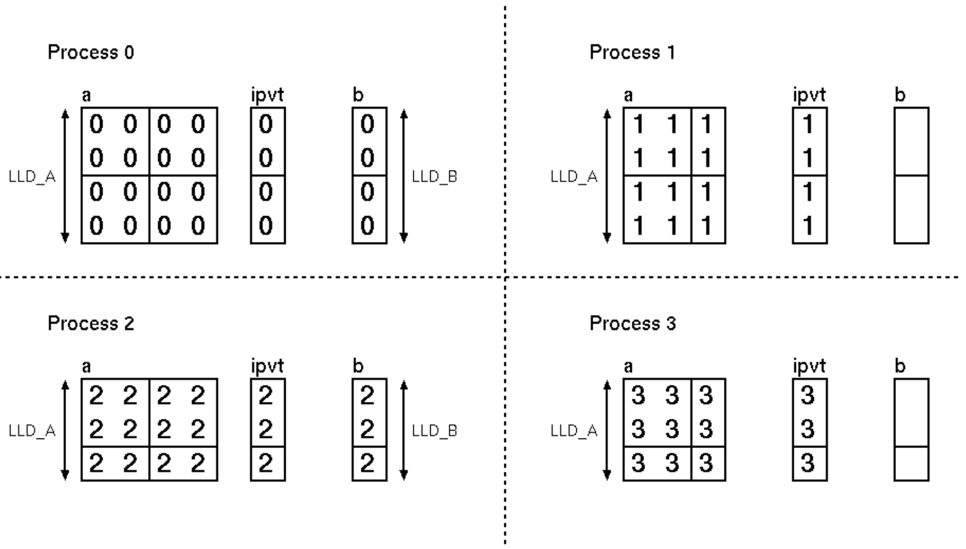


Figure 126. Local Matrices

Figure 126 shows the resulting local matrices. Note that processes 1 and 3 do not have valid elements for the local matrix *b*. Since global matrix *B* has only one column, only the processes in the first column of the process grid have some portion for *B*.

This sample program solves one system of linear equations (`nrhs=1`) with `ia`, `ja`, `ib`, and `jb` all set to `1` for simplicity.

```
PROGRAM main
IMPLICIT REAL*8 (a-h,o-z)
INCLUDE 'mpif.h'
PARAMETER (n = 2000, iblock = 80, nprow = 3, npcol = 4)
DIMENSION a(n,n), b(n), btemp(n)
INTEGER idesc_a(9), idesc_b(9)
REAL*8,  ALLOCATABLE :: aa(:,:), bb(:)
INTEGER, ALLOCATABLE :: iipvt(:)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (nprocs /= nprow * npcol) THEN
  PRINT *,'Incorrect number of processes'
  CALL MPI_FINALIZE(ierr)
  STOP
ENDIF
CALL BLACS_GET(0, 0, ictxt)
CALL BLACS_GRIDINIT(ictxt, 'R', nprow, npcol)
CALL BLACS_GRIDINFO(ictxt, idummy1, idummy2, myrow, mycol)
nrow = MAX(1, NUMROC(n, iblock, myrow, 0, nprow))
ncol = MAX(1, NUMROC(n, iblock, mycol, 0, npcol))
ALLOCATE (aa(nrow,ncol), bb(nrow), iipvt(nrow))
idesc_a(1) = 1
idesc_a(2) = ictxt
```

```
 idesc_a(3) = n
 idesc_a(4) = n
 idesc_a(5) = iblock
 idesc_a(6) = iblock
 idesc_a(7) = 0
 idesc_a(8) = 0
 idesc_a(9) = nrow
 idesc_b(1) = 1
 idesc_b(2) = ictxt
 idesc_b(3) = n
 idesc_b(4) = 1
 idesc_b(5) = iblock
 idesc_b(6) = 1
 idesc_b(7) = 0
 idesc_b(8) = 0
 idesc_b(9) = nrow
 Set values of global matrices A and B
 jcnt = 0
 DO jj = mycol * iblock + 1, n, npcol * iblock
   DO j = jj, MIN(jj + iblock - 1, n)
     jcnt = jcnt + 1
     icnt = 0
     DO ii = myrow * iblock + 1, n, nprow * iblock
       DO i = ii, MIN(ii + iblock - 1, n)
         icnt = icnt + 1
         aa(icnt,jcnt) = a(i,j)
       ENDDO
     ENDDO
   ENDDO
 ENDDO
 IF (mycol == 0) THEN
   icnt = 0
   DO ii = myrow * iblock + 1, n, nprow * iblock
     DO i = ii, MIN(ii + iblock - 1, n)
       icnt = icnt + 1
       bb(icnt) = b(i)
     ENDDO
   ENDDO
 ENDIF
 CALL PDGETRF(n, n, aa, 1, 1, idesc_a, iipvt, info)
 IF (info /= 0) PRINT *,'Error in PDGETRF'
 CALL PDGETRS('n', n, 1, aa, 1, 1, idesc_a, iipvt, bb, 1, 1,
&             idesc_b, info)
 IF (info /= 0) PRINT *,'Error in PDGETRS'
 DO i = 1, n
   btemp(i) = 0.0
 ENDDO
 IF (mycol == 0) THEN
   icnt = 0
   DO ii = myrow * iblock + 1, n, nprow * iblock
     DO i = ii, MIN(ii + iblock - 1, n)
       icnt = icnt + 1
       btemp(i) = bb(icnt)
     ENDDO
   ENDDO
 ENDIF
 CALL MPI_ALLREDUCE(btemp, b, n, MPI_DOUBLE_PRECISION, MPI_SUM,
&                    MPI_COMM_WORLD, ierr)
```

```
      ...
      CALL MPI_FINALIZE(ierr)
      END
```

For the optimal process grid and block size, you may have to experiment.

4.9 Multi-Frontal Method

The multi-frontal method is an algorithm for factoring sparse symmetric matrices. For sparse matrices with wide bands, the multi-frontal method is more efficient than the skyline method in terms of performance and memory consumption. The current version of ESSL (Version 3.1) and Parallel ESSL (Version 2.1) do not have subroutines using the multi-frontal method. As a complement to ESSL and Parallel ESSL, the Watson Symmetric Sparse Matrix Package (WSSMP) provides a high-performance, robust, and easy to use subroutines that exploit the modified multi-frontal algorithm. WSSMP can be used as a serial package, or in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. The details about WSSMP subroutines and how to get them can be found at
`http://www.research.ibm.com/mathsci/ams/ams_WSSMP.htm`

The following is a brief description of WSSMP and PWSSMP, the parallel version of WSSMP.

- WSSMP and PWSSMP solve linear equations in the following steps.
  1. Ordering
  2. Symbolic factorization
  3. Cholesky or LDL^T factorization
  4. Forward and backward elimination
  5. Iterative refinement
- PWSSMP has two modes: the 0-master mode and the peer mode. In the 0-master mode, process 0 has the whole matrix, whereas in the peer mode, all processes have a shrunk matrix.

As written in the user's guide you can retrieve from the URL mentioned above, WSSMP and PWSSMP libraries may be used solely for educational, research, and benchmarking purposes. Any other use of the libraries requires prior written permission from IBM Corporation.

Appendix A. How to Run Parallel Jobs on RS/6000 SP

This appendix gives a brief overview to Parallel Environment (PE) that you should understand when you run parallel jobs on RS/6000 SP. Detailed descriptions are found in the following documents:

- *IBM Parallel Environment for AIX Operation and Use, Volume 1: Using the Parallel Operating Environment Version 2 Release 4*, SC28-1979
- *IBM LoadLeveler for AIX Using and Administering Version 2 Release 1*, SA22-7311

This chapter assumes that you are using Parallel Environment 2.4.

A.1 AIX Parallel Environment

In Parallel Environment 2.4, the job control subsystem is unified in LoadLeveler. Prior to PE 2.4, parallel jobs are managed through either LoadLeveler or Resource Manager, which is a part of PE. The following are the highlight of PE 2.4.

- You can run up to four User Space (US) processes per node.
- The MPI library includes support for a subset of MPI I/O.
- With regard to MPI/LAPI jobs, PE supports a maximum of 2048 processes for IP and 1024 processes for US.

Note that PE compiles and runs all applications as 32 bit applications: 64-bit applications are not yet supported. Aside from the MPI library, PE includes tools for debugging, profiling, and visualizing parallel programs. For details, see *IBM Parallel Environment for AIX Operation and Use, Volume 2 Part 1: Debugging and Visualizing Version 2 Release 4*, SC28-1979 and *IBM Parallel Environment for AIX Operation and Use, Volume 2 Part 2: Profiling Version 2 Release 4*, SC28-1980.

A.2 Compiling Parallel Programs

To compile, use the commands `mpcc`, `mpCC`, or `mpxlf`. These commands not only compile your program, but also link in the Partition Manager and message passing interface libraries. To compile threaded C, C++, or Fortran programs, use the `mpcc_r`, `mpCC_r`, or `mpxlf_r` commands. These commands can also be used to compile non-threaded programs with the threaded libraries such as `libpesslsmp.a`. There are a lot of compiler options for optimization. To start with, use `-qarch=pwr2 -O3 -qstrict` on POWER2 nodes and `-qarch=pwr3 -O3 -qstrict` on POWER3 nodes. For more options, consult the User's Guide for XL Fortran.

A.3 Running Parallel Programs

First, make sure that each node can access the executable, especially, if you place the executable on a local file system. Copy the executable to remote nodes that will be involved in parallel execution. Setup the .rhosts file appropriately so that remote shell command (`rsh`) works for the nodes that you are going to use. Note that the root user cannot run parallel jobs in Parallel Environment for AIX.

A.3.1 Specifying Nodes

You can use a host list file to specify the nodes for parallel execution, or you can let LoadLeveler do the job. In the latter case, you generaly cannot tell which nodes will be used beforehand, but you can know them afterwards by use of the MP_SAVEHOSTFILE environment variable.

Host list Set the environment variable MP_HOSTFILE as the name of the text file that contains the list of nodes (one node for each line). If there is a file host.list in the current directory, it is automatically taken as MP_HOSTFILE. In this case, even if you set MP_RMPOOL, the entries in host.list are used for node allocation.

LoadLeveler Set the environment variable MP_RMPOOL as an appropriate integer. The command `/usr/lpp/LoadL/full/bin/llstatus -l` shows you a list of machines defined. Nodes which are in the class inter_class can be used for interactive execution of parallel jobs. Check the value of `Pool` for that node and use it for MP_RMPOOL.

The number of processes is specified by the MP_PROCS environment variable or by the `-procs` command-line flag. All the environment variables of PE have corresponding command-line flags. The command-line flags temporarily override their associated environment variable.

Since PE 2.4 allows you to run up to four User Space processes per node, you need to know how to specify node allocation. Environment variables MP_NODES and MP_TASKS_PER_NODE are used for this purpose: MP_NODES tells LoadLeveler how many nodes you use, and MP_TASKS_PER_NODE specifies how many processes per node you use. It is sufficient to specify two environment variables out of MP_PROCS, MP_NODES, and MP_TASKS_PER_NODE. If all of them are specified, MP_PROCS=MP_NODES x MP_TASKS_PER_NODE must hold. When you use IP instead of User Space protocol, there is no limitation on the number of processes per node as long as the total number of processes does not exceed 2048.

A.3.2 Specifying Protocol and Network Device

The protocol used by PE is either User Space protocol or IP. Set the environment variable MP_EUILIB as `us` or `ip`, respectively. The network device is specified by MP_EUIDEVICE. The command `netstat -i` shows you the names of the network interfaces available on the node. MP_EUIDEVICE can be css0, en0, tr0, fi0, and so on. Note that when you use User Space protocol (MP_EUILIB=us), MP_EUIDEVICE is assumed to be `css0`.

A.3.3 Submitting Parallel Jobs

The following shows how to submit a parallel job. The executable is `/u/nakano/work/a.out` and it is assumed to be accessible with the same path name by the nodes you are going to use. Suppose you run four User Space processes on two nodes, that is, two User Space processes per node.

From PE using host list

Change directory to /u/nakano/work, and create a file myhosts which contains the following four lines:

```
sps01e
sps01e
sps02e
sps02e
```

Then execute the following.

```
$ export MP_HOSTFILE=/u/nakano/work/myhosts
$ export MP_EUILIB=us
$ export MP_TASKS_PER_NODE=2
a.out -procs 4
```

If you are using a C shell, use the `setenv` command instead of `export`.

```
% setenv MP_HOSTFILE /u/nakano/work/myhosts
...
```

From PE using LoadLeveler

Change directory to /u/nakano/work, and execute the following.

```
$ export MP_RMPOOL=1
$ export MP_EUILIB=us
$ export MP_TASKS_PER_NODE=2
$ export MP_SAVEHOSTFILE=/u/nakano/work/usedhosts
a.out -procs 4
```

The value of MP_RMPOOL should be chosen appropriately. (See Appendix A.3.1, “Specifying Nodes” on page 156.) You can check which nodes were allocated by using the MP_SAVEHOSTFILE environment variable, if you like.

From LoadLeveler

Create a job command file test.job as follows.

```
# @ class          = A_Class
# @ job_type       = parallel
# @ network.MPI    = css0,,US
# @ node           = 2,2
# @ tasks_per_node = 2
# @ queue
export MP_SAVEHOSTFILE=/u/nakano/work/usedhosts
a.out
```

Specify the name of the class appropriately. The keyword *node* specifies the minimum and the maximum number of nodes required by the job. Now you can submit the job as follows.

```
$ /usr/lpp/LoadL/full/bin/llsubmit test.job
```

By default, you will receive mail when the job completes, which can be changed by the notification keyword.

A.4 Monitoring Parallel Jobs

Use `/usr/lpp/LoadL/full/bin/llq` for listing submitted jobs. The following shows the sample output.

```
$ /usr/lpp/LoadL/full/bin/llq
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sps01e.28.0              nakano      6/9  17:12 R  50  inter_class  sps01e

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held
```

By specifying `-l` option, you get detailed information.

```
$ /usr/lpp/LoadL/full/bin/llq -l
=============== Job Step sps01e.28.0 ===============
        Job Step Id: sps01e.28.0
           Job Name: sps01e.28
          Step Name: 0
  Structure Version: 9
              Owner: nakano
         Queue Date: Wed Jun  9 17:12:48 JST 1999
             Status: Running
      Dispatch Time: Wed Jun  9 17:12:48 JST 1999
...
...
Step Type: General Parallel (Interactive)
    Submitting Host: sps01e
        Notify User: nakano@sps01e
              Shell: /bin/ksh
  LoadLeveler Group: No_Group
              Class: inter_class
...
...
Adapter Requirement: (css0,MPI,not_shared,US)
-------------------------------------------------
Node
----

   Name            :
   Requirements    : (Pool == 1) && (Arch == "R6000") && (OpSys == "AIX43")
   Preferences     :
   Node minimum    : 2
   Node maximum    : 2
   Node actual     : 2
   Allocated Hosts : sps01e::css0(0,MPI,us),css0(1,MPI,us)
                   + sps02e::css0(0,MPI,us),css0(1,MPI,us)

   Task
   ----

      Num Task Inst:
      Task Instance:


llq: Specify -x option in addition to -l option to obtain Task Instance information.

1 job steps in queue, 0 waiting, 0 pending, 1 running, 0 held
```

The output section “Allocated Hosts”, shown in the preceding example, indicates on which nodes the parallel processes are running. In the above output, you see four User Space processes are running on nodes sps01e and sps02e, two processes per node.

A.5 Standard Output and Standard Error

The following three environment variables are often used to control standard output and standard error.

You can set the environment variable MP_LABELIO as yes, so that output from the parallel processes of your program are labeled by rank id. Default is no.

Using the environment variable MP_STDOUTMODE, you can specify that:

- All tasks should write output data to standard output asynchronously. This is unordered output mode. (MP_STDOUTMODE=unordered)
- Output data from each parallel process should be written to its own buffer, and later all buffers should be flushed, in rank order, to standard output. This is ordered output mode. (MP_STDOUTMODE=ordered)

- A single process should write to standard output. This is single output mode. (MP_STDOUTMODE=*rank_id*)

The default is unordered. The ordered and unordered modes are mainly used when you are developing or debugging a code. The following example shows how MP_STDOUTMODE and MP_LABELIO affect the output.

```
$ cat test.f
      PRINT *,'Hello, SP'
      END
$ mpxlf test.f
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file test.f.
$ export MP_LABELIO=yes
$ export MP_STDOUTMODE=unordered; a.out -procs 3
   1: Hello, SP
   0: Hello, SP
   2: Hello, SP
$ export MP_STDOUTMODE=ordered; a.out -procs 3
   0: Hello, SP
   1: Hello, SP
   2: Hello, SP
$ export MP_STDOUTMODE=0; a.out -procs 3
   0: Hello, SP
```

You can set the environment variable MP_INFOLEVEL to specify the level of messages you want from PE. The value of MP_INFOLEVEL should be one of 0..6. The integers 0, 1, and 2 give you different levels of informational, warning, and error messages. The integers 3 through 6 indicate debug levels that provide additional debugging and diagnostic information. Default is 1.

A.6 Environment Variable MP_EAGER_LIMIT

The environment variable MP_EAGER_LIMIT changes the threshold value for the message size, above which rendezvous protocol is used.

To ensure that at least 32 messages can be outstanding between any two processes, MP_EAGER_LIMIT will be adjusted based on the number of processes according to the following table (when MP_EAGER_LIMIT and MP_BUFFER_MEM have not been set by the user):

Table 10. Default Value of MP_EAGER_LIMIT

Number of processes	MP_EAGER_LIMIT (KB)
1 - 16	4096
17 - 32	2048
33 - 64	1024
65 - 128	512

The maximum value of MP_EAGER_LIMIT is 65536 KB. 65536 KB is also equal to the maximum value of MP_BUFFER_MEM, which is the default of MP_BUFFER_MEM.

MPI uses two message protocols, eager and rendezvous. With eager protocol the message is sent to the destination without knowing there is a matching receive. If there is not one, the message is held in an early arrival buffer (MP_BUFFER_MEM). By default, small messages use eager protocol and large ones use rendezvous. In rendezvous, the sending call does not return until the destination receive is found so the rate of receivers limits senders. With eager, the senders run wild. If you set MP_EAGER_LIMIT=0, you will make all messages use rendezvous protocol, but forcing rendezvous does increase latency and therefore affects performance in many cases.

Appendix B. Frequently Used MPI Subroutines Illustrated

Throughout this appendix, it is assumed that the environment variable MP_STDOUTMODE is set as ordered and MP_LABELIO is set to yes in running sample programs. In the parameters section, the term `CHOICE` indicates that any Fortran data type is valid.

B.1 Environmental Subroutines

In the sections that follow, several evironmental subroutines are introduced.

B.1.1 MPI_INIT

Purpose Initializes MPI.

Usage

```
CALL MPI_INIT(ierror)
```

Parameters

INTEGER ierror The Fortran return code

Description This routine initializes MPI. All MPI programs must call this routine once and only once before any other MPI routine (with the exception of MPI_INITIALIZED). Non-MPI statements can precede MPI_INIT.

Sample program

```
PROGRAM init
INCLUDE 'mpif.h'
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
PRINT *,'nprocs =',nprocs,'myrank =',myrank
CALL MPI_FINALIZE(ierr)
END
```

Sample execution

```
$ a.out -procs 3
   0: nprocs = 3 myrank = 0
   1: nprocs = 3 myrank = 1
   2: nprocs = 3 myrank = 2
```

B.1.2 MPI_COMM_SIZE

Purpose Returns the number of processes in the group associated with a communicator.

Usage

```
CALL MPI_COMM_SIZE(comm, size, ierror)
```

Parameters

INTEGER comm	The communicator (handle) (IN)
INTEGER size	An integer specifying the number of processes in the group comm (OUT)
INTEGER ierror	The Fortran return code

Description This routine returns the size of the group associated with a communicator.

Sample program and execution

See the sample given in B.1.1, “MPI_INIT” on page 161.

B.1.3 MPI_COMM_RANK

Purpose Returns the rank of the local process in the group associated with a communicator.

Usage

```
CALL MPI_COMM_RANK(comm, rank, ierror)
```

Parameters

INTEGER comm	The communicator (handle) (IN)
INTEGER rank	An integer specifying the rank of the calling process in group comm (OUT)
INTEGER ierror	The Fortran return code

Description This routine returns the rank of the local process in the group associated with a communicator. MPI_COMM_RANK indicates the rank of the process that calls it in the range from `0..size - 1`, where `size` is the return value of MPI_COMM_SIZE.

Sample program and execution

See the sample given in B.1.1, “MPI_INIT” on page 161.

B.1.4 MPI_FINALIZE

Purpose Terminates all MPI processing.

Usage

```
CALL MPI_FINALIZE(ierror)
```

Parameters

INTEGER ierror	The Fortran return code

Description Make sure this routine is the last MPI call. Any MPI calls made after MPI_FINALIZE raise an error. You must be sure that all pending communications involving a process have completed before the process calls MPI_FINALIZE.

You must also be sure that all files opened by the process have been closed before the process calls MPI_FINALIZE. Although MPI_FINALIZE terminates MPI processing, it does not terminate the process. It is possible to continue with non-MPI processing after calling MPI_FINALIZE, but no other MPI calls (including MPI_INIT) can be made.

Sample program and execution

See the sample given in B.1.1, “MPI_INIT” on page 161.

B.1.5 MPI_ABORT

Purpose Forces all processes of an MPI job to terminate.

Usage

```
CALL MPI_ABORT(comm, errorcode, ierror)
```

Parameters

INTEGER comm	The communicator of the processes to abort (IN)
INTEGER errorcode	The error code returned to the invoking environment (IN)
INTEGER ierror	The Fortran return code

Description If any process calls this routine, all processes in the job are forced to terminate. The argument comm currently is not used. The low order 8 bits of errorcode are returned as an AIX return code. This subroutine can be used when only one process reads data from a file and it finds an error while reading.

B.2 Collective Communication Subroutines

In the sections that follow, several communication subroutines are introduced.

B.2.1 MPI_BCAST

Purpose Broadcasts a message from root to all processes in comm.

Usage

```
CALL MPI_BCAST(buffer, count, datatype, root, comm, ierror)
```

Parameters

(CHOICE) buffer	The starting address of the buffer (INOUT)
INTEGER count	The number of elements in the buffer (IN)
INTEGER datatype	The data type of the buffer elements (handle) (IN)
INTEGER root	The rank of the root process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine broadcasts a message from root to all processes in comm. The contents of root's communication buffer is copied to all processes on return. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This means the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are allowed. All processes in comm need to call this routine.

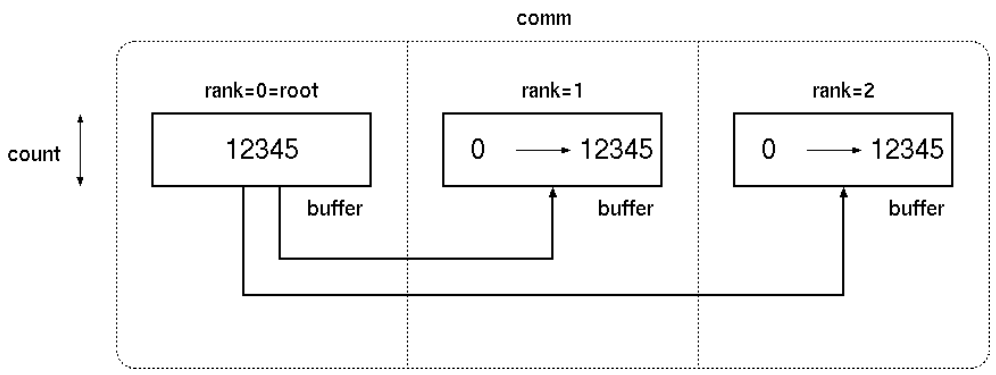


Figure 127. MPI_BCAST

Sample program

```
      PROGRAM bcast
      INCLUDE 'mpif.h'
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        ibuf=12345
      ELSE
        ibuf=0
      ENDIF
      CALL MPI_BCAST(ibuf, 1, MPI_INTEGER, 0,
     &               MPI_COMM_WORLD, ierr)
      PRINT *,'ibuf =',ibuf
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: ibuf = 12345
   1: ibuf = 12345
   2: ibuf = 12345
```

B.2.2 MPE_IBCAST (IBM Extension)

Purpose Performs a nonblocking broadcast operation.

Usage

```
CALL MPE_IBCAST(buffer, count, datatype, root, comm, request, ierror)
```

Parameters

(CHOICE) buffer	The starting address of the buffer (INOUT)
INTEGER count	The number of elements in the buffer (IN)
INTEGER datatype	The data type of the buffer elements (handle) (IN)
INTEGER root	The rank of the root process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER request	The communication request (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This routine is a nonblocking version of MPI_BCAST. It performs the same function as MPI_BCAST except that it returns a request handle that must be explicitly completed by using one of the MPI wait or test operations. All processes in comm need to call this routine. The MPE prefix used with this routine indicates that it is an IBM extension to the MPI standard and is not part of the standard itself. MPE routines are provided to enhance the function and the performance of user applications, but applications that use them will not be directly portable to other MPI implementations. Nonblocking collective communication routines allow for increased efficiency and flexibility in some applications. Because these routines do not synchronize the participating processes like blocking collective routines generally do, processes running at different speeds do not waste time waiting for each other. Applications using nonblocking collective calls often perform best when they run in interrupt mode.

Sample program

```
      PROGRAM ibcast
      INCLUDE 'mpif.h'
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        ibuf=12345
      ELSE
        ibuf=0
      ENDIF
      CALL MPE_IBCAST(ibuf, 1, MPI_INTEGER, 0,
     &                MPI_COMM_WORLD, ireq, ierr)
      CALL MPI_WAIT(ireq, istatus, ierr)
      PRINT *,'ibuf =',ibuf
      CALL MPI_FINALIZE(ierr)
      END
```

The above is a non-blocking version of the sample program of MPI_BCAST. Since MPE_IBCAST is non-blocking, don't forget to call MPI_WAIT to complete the transmission.

Sample execution

```
$ a.out -procs 3
   0: ibuf = 12345
   1: ibuf = 12345
   2: ibuf = 12345
```

B.2.3 MPI_SCATTER

Purpose Distributes individual messages from root to each process in comm.

Usage

```
CALL MPI_SCATTER(sendbuf, sendcount, sendtype,
                 recvbuf, recvcount, recvtype, root, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The address of the send buffer (significant only at root) (IN)
INTEGER sendcount	The number of elements to be sent to each process, not the number of total elements to be sent from root (significant only at root) (IN)
INTEGER sendtype	The data type of the send buffer elements (handle, significant only at root) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcount	The number of elements in the receive buffer (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER root	The rank of the sending process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine distributes individual messages from root to each process in comm. The number of elements sent to each process is the same (sendcount). The first sendcount elements are sent to process 0, the next sendcount elements are sent to process 1, and so on. This routine is the inverse operation to MPI_GATHER. The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes. (Type maps can be different.) This means the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are allowed. All processes in comm need to call this routine.

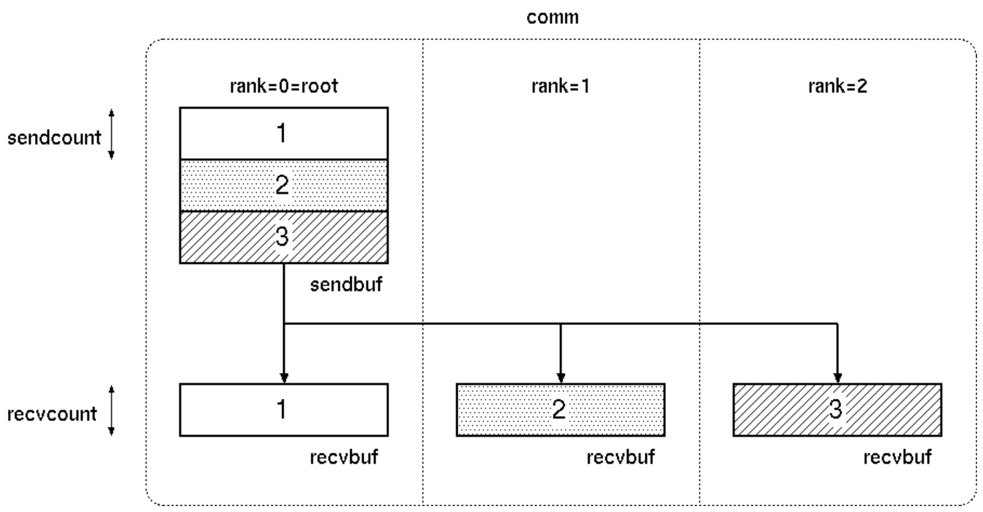


Figure 128. MPI_SCATTER

Sample program

```
      PROGRAM scatter
      INCLUDE 'mpif.h'
      INTEGER isend(3)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        DO i=1,nprocs
          isend(i)=i
        ENDDO
      ENDIF
      CALL MPI_SCATTER(isend, 1, MPI_INTEGER,
     &                 irecv, 1, MPI_INTEGER, 0,
     &                 MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1
   1: irecv = 2
   2: irecv = 3
```

B.2.4 MPI_SCATTERV

Purpose Distributes individual messages from `root` to each process in `comm`. Messages can have different sizes and displacements.

Usage

```
CALL MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype,
                  recvbuf, recvcount,         recvtype, root, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The address of the send buffer (significant only at root) (IN)
INTEGER sendcounts(*)	Integer array (of length group size) that contains the number of elements to send to each process (significant only at root) (IN)
INTEGER displs(*)	Integer array (of length group size). Entry i specifies the displacement relative to sendbuf from which to send the outgoing data to process i (significant only at root) (IN)
INTEGER sendtype	The data type of the send buffer elements (handle, significant only at root) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcount	The number of elements in the receive buffer (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER root	The rank of the sending process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine distributes individual messages from root to each process in comm. Messages can have different sizes and displacements. With sendcounts as an array, messages can have varying sizes of data that can be sent to each process. The array displs allows you the flexibility of where the data can be taken from the root. The type signature of sendcount(i), sendtype at the root must be equal to the type signature of recvcount, recvtype at process i. (The type maps can be different.) This means the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are allowed. All processes in comm need to call this routine.

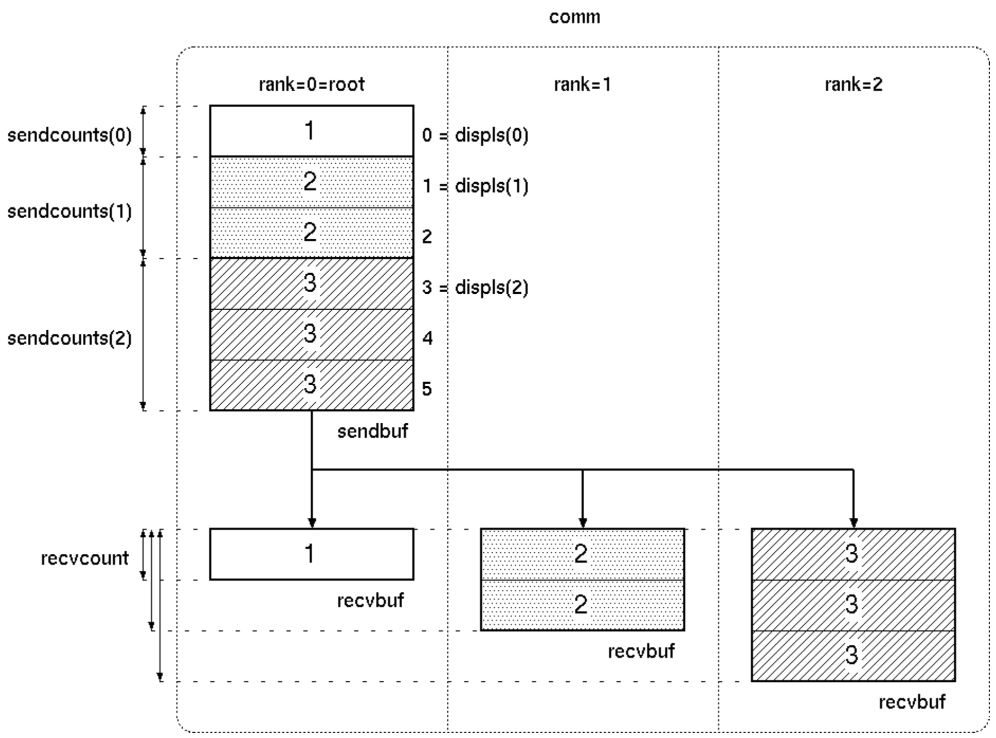


Figure 129. MPI_SCATTERV

Sample program

```
      PROGRAM scatterv
      INCLUDE 'mpif.h'
      INTEGER isend(6), irecv(3)
      INTEGER iscnt(0:2), idisp(0:2)
      DATA isend/1,2,2,3,3,3/
      DATA iscnt/1,2,3/ idisp/0,1,3/
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      ircnt=myrank+1
      CALL MPI_SCATTERV(isend, iscnt, idisp, MPI_INTEGER,
     &                  irecv, ircnt,        MPI_INTEGER,
     &                  0, MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1 0 0
   1: irecv = 2 2 0
   2: irecv = 3 3 3
```

B.2.5 MPI_GATHER

Purpose Collects individual messages from each process in `comm` at the root process.

Usage

```
CALL MPI_GATHER(sendbuf, sendcount, sendtype,
                recvbuf, recvcount, recvtype, root, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
INTEGER sendcount	The number of elements in the send buffer (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (significant only at root) (OUT)
INTEGER recvcount	The number of elements for any single receive (significant only at root) (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle, significant only at root) (IN)
INTEGER root	The rank of the receiving process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine collects individual messages from each process in comm to the root process and stores them in rank order. The amount of data gathered from each process is the same. The type signature of sendcount, sendtype on process i must be equal to the type signature of recvcount, recvtype at the root. This means the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are allowed. All processes in comm need to call this routine.

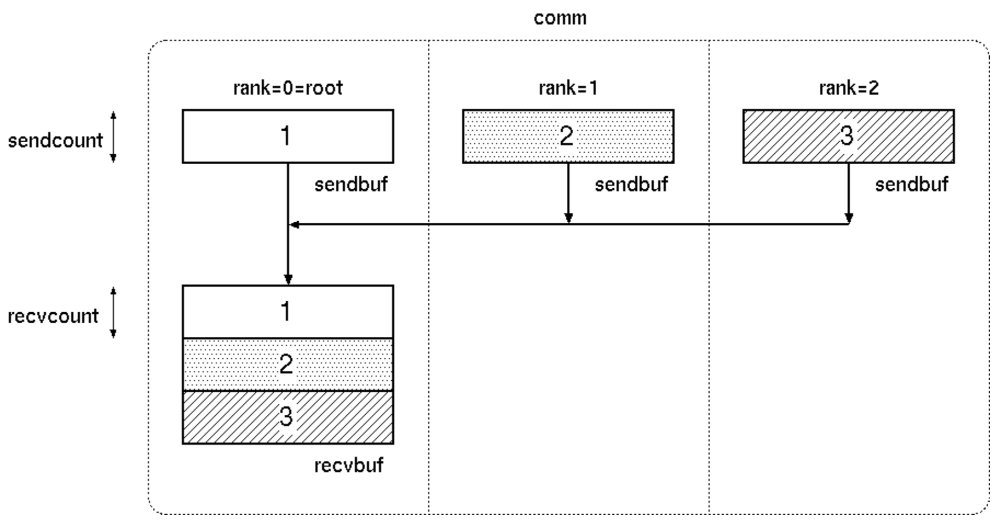


Figure 130. MPI_GATHER

Sample program

```
        PROGRAM gather
        INCLUDE 'mpif.h'
        INTEGER irecv(3)
        CALL MPI_INIT(ierr)
        CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
        CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
        isend = myrank + 1
        CALL MPI_GATHER(isend, 1, MPI_INTEGER,
       &                irecv, 1, MPI_INTEGER, 0,
       &                MPI_COMM_WORLD, ierr)
        IF (myrank==0) THEN
          PRINT *,'irecv =',irecv
        ENDIF
        CALL MPI_FINALIZE(ierr)
        END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1 2 3
```

B.2.6 MPI_GATHERV

Purpose Collects individual messages from each process in comm to the root process. Messages can have different sizes and displacements.

Usage

```
CALL MPI_GATHERV(sendbuf, sendcount,         sendtype,
                 recvbuf, recvcounts, displs, recvtype, root, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
INTEGER sendcount	The number of elements in the send buffer (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (significant only at root) (OUT)
INTEGER recvcounts(*)	Integer array (of length group size) that contains the number of elements received from each process (significant only at root) (IN)
INTEGER displs(*)	Integer array (of length group size), entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root) (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle, significant only at root) (IN)
INTEGER root	The rank of the receiving process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine collects individual messages from each process in comm at the root process and stores them in rank order. With recvcounts as an array, messages can have varying sizes, and displs allows you the flexibility of where the data is placed on the root. The type signature of sendcount, sendtype on process i must be equal to the type signature of recvcounts(i), recvtype at the root. This means the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are allowed. All processes in comm need to call this routine.

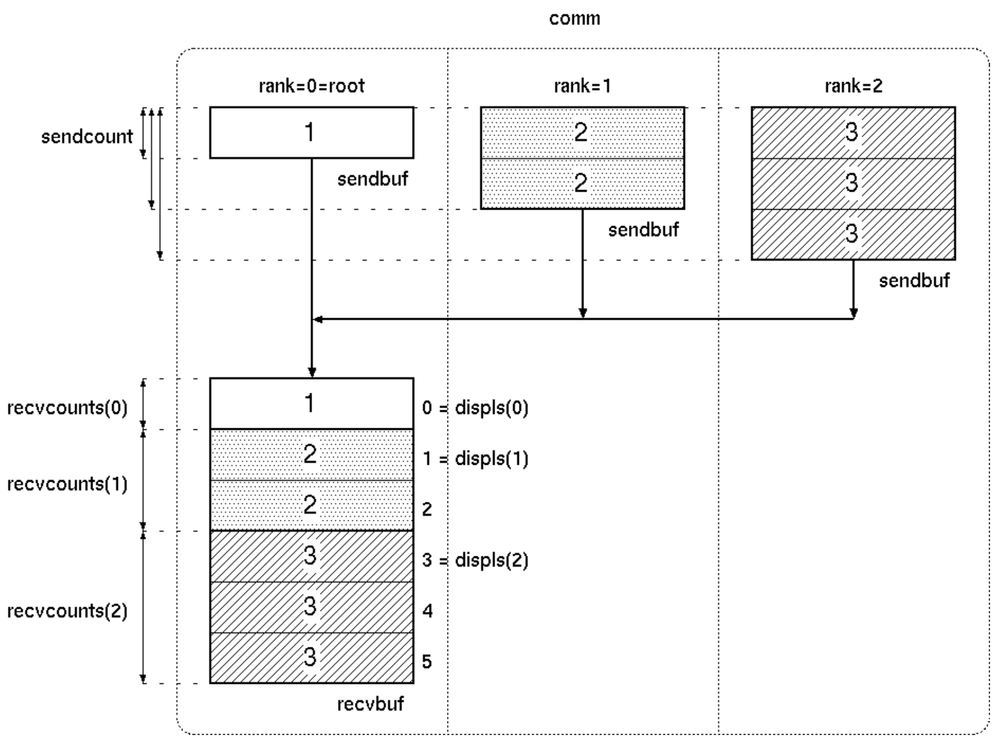


Figure 131. MPI_GATHERV

Sample program

```
        PROGRAM gatherv
        INCLUDE 'mpif.h'
        INTEGER isend(3), irecv(6)
        INTEGER ircnt(0:2), idisp(0:2)
        DATA ircnt/1,2,3/ idisp/0,1,3/
        CALL MPI_INIT(ierr)
        CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
        CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
        DO i=1,myrank+1
          isend(i) = myrank + 1
        ENDDO
        iscnt = myrank + 1
        CALL MPI_GATHERV(isend, iscnt,        MPI_INTEGER,
       &                 irecv, ircnt, idisp, MPI_INTEGER,
       &                 0, MPI_COMM_WORLD, ierr)
```

```
      IF (myrank==0) THEN
        PRINT *,'irecv =',irecv
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1 2 2 3 3 3
```

B.2.7 MPI_ALLGATHER

Purpose Gathers individual messages from each process in comm and distributes the resulting message to each process.

Usage

```
CALL MPI_ALLGATHER(sendbuf, sendcount, sendtype,
                   recvbuf, recvcount, recvtype, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
INTEGER sendcount	The number of elements in the send buffer (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcount	The number of elements received from any process (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description MPI_ALLGATHER is similar to MPI_GATHER except that all processes receive the result instead of just the root. The amount of data gathered from each process is the same. The block of data sent from process j is received by every process and placed in the j-th block of the buffer recvbuf. The type signature associated with sendcount, sendtype at a process must be equal to the type signature associated with recvcount, recvtype at any other process. All processes in comm need to call this routine.

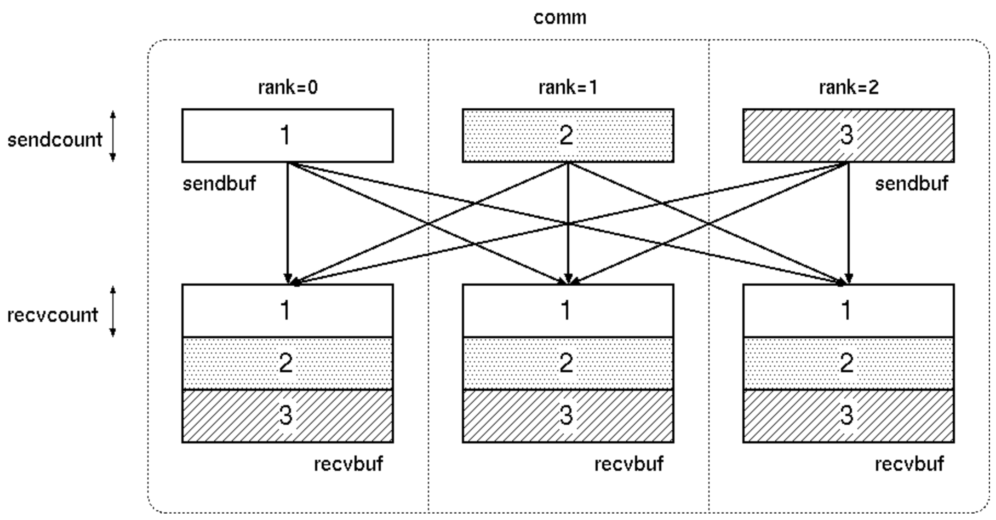


Figure 132. MPI_ALLGATHER

Sample program

```
      PROGRAM allgather
      INCLUDE 'mpif.h'
      INTEGER irecv(3)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      isend = myrank + 1
      CALL MPI_ALLGATHER(isend, 1, MPI_INTEGER,
     &                   irecv, 1, MPI_INTEGER,
     &                   MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(irerr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1 2 3
   1: irecv = 1 2 3
   2: irecv = 1 2 3
```

B.2.8 MPI_ALLGATHERV

Purpose Collects individual messages from each process in comm and distributes the resulting message to all processes. Messages can have different sizes and displacements.

Usage

```
CALL MPI_ALLGATHERV(sendbuf, sendcount,         sendtype,
                    recvbuf, recvcounts, displs, recvtype, comm, ierror)
```

Parameters

(CHOICE) sendbuf The starting address of the send buffer (IN)

INTEGER sendcount	The number of elements in the send buffer (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER recvcounts(*)	Integer array (of length group size) that contains the number of elements received from each process (IN)
INTEGER displs(*)	Integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code
Description	This routine collects individual messages from each process in comm and distributes the resulting message to all processes. Messages can have different sizes and displacements. The block of data sent from process j is recvcounts(j) elements long, and is received by every process and placed in recvbuf at offset displs(j). The type signature associated with sendcount, sendtype at process j must be equal to the type signature of recvcounts(j), recvtype at any other process. All processes in comm need to call this routine.

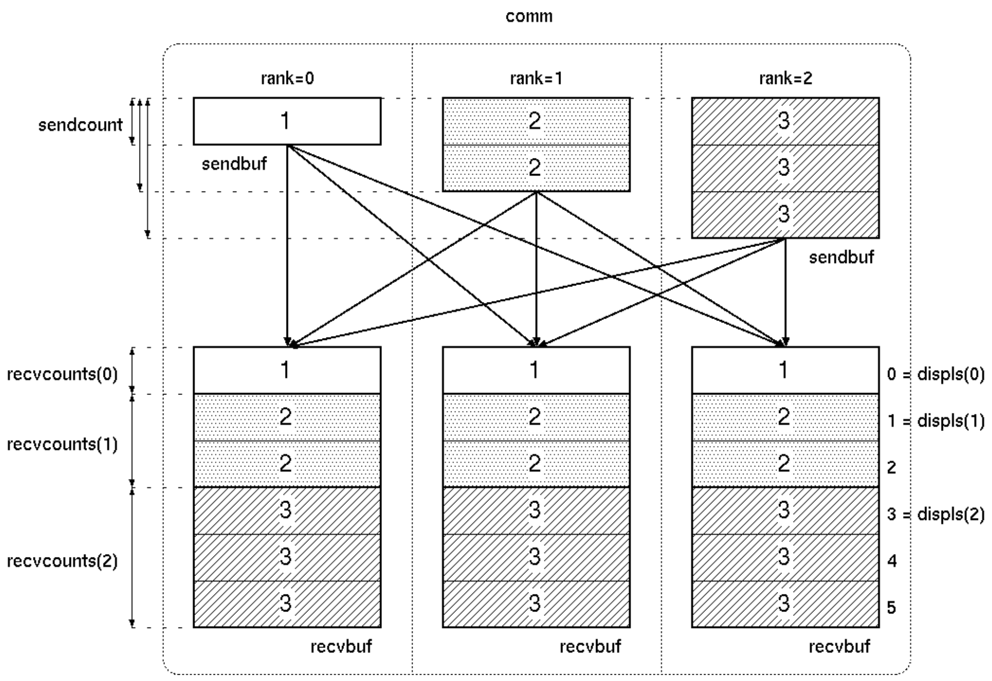


Figure 133. MPI_ALLGATHERV

Sample program

```
      PROGRAM allgatherv
      INCLUDE 'mpif.h'
      INTEGER isend(3), irecv(6)
      INTEGER ircnt(0:2), idisp(0:2)
      DATA ircnt/1,2,3/ idisp/0,1,3/
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i=1,myrank+1
        isend(i) = myrank + 1
      ENDDO
      iscnt = myrank + 1
      CALL MPI_ALLGATHERV(isend, iscnt,        MPI_INTEGER,
     &                    irecv, ircnt, idisp, MPI_INTEGER,
     &                    MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1 2 2 3 3 3
   1: irecv = 1 2 2 3 3 3
   2: irecv = 1 2 2 3 3 3
```

B.2.9 MPI_ALLTOALL

Purpose Sends a distinct message from each process to every other process.

Usage

```
CALL MPI_ALLTOALL(sendbuf, sendcount, sendtype,
                  recvbuf, recvcount, recvtype, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
INTEGER sendcount	The number of elements sent to each process (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcount	The number of elements received from any process (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine sends a distinct message from each process to every process. The j-th block of data sent from process i is received by process j and placed in the i-th block of the buffer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This means the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. The type maps can be different. All arguments on all processes are significant. All processes in comm need to call this routine.

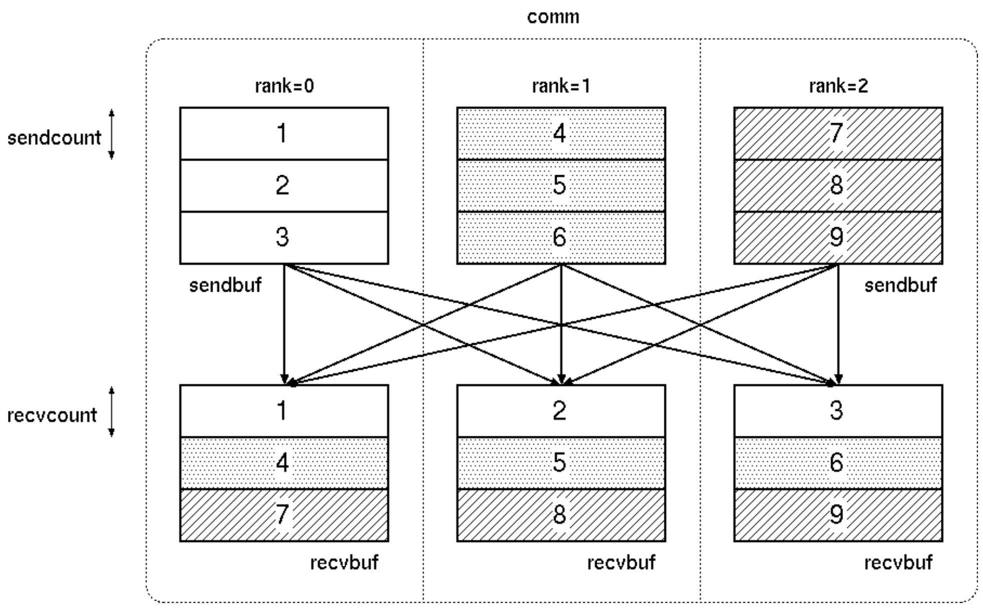


Figure 134. MPI_ALLTOALL

Sample program

```
      PROGRAM alltoall
      INCLUDE 'mpif.h'
      INTEGER isend(3), irecv(3)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i=1,nprocs
        isend(i) = i + nprocs * myrank
      ENDDO
      PRINT *,'isend =',isend
      CALL MP_FLUSH(1)                    ! for flushing stdout
      CALL MPI_ALLTOALL(isend, 1, MPI_INTEGER,
     &                  irecv, 1, MPI_INTEGER,
     &                  MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: isend = 1 2 3
   1: isend = 4 5 6
   2: isend = 7 8 9
   0: irecv = 1 4 7
   1: irecv = 2 5 8
   2: irecv = 3 6 9
```

B.2.10 MPI_ALLTOALLV

Purpose	Sends a distinct message from each process to every process. Messages can have different sizes and displacements.

Usage

```
CALL MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype,
                   recvbuf, recvcounts, rdispls, recvtype, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
INTEGER sendcounts(*)	Integer array (of length group size) specifying the number of elements to send to each process (IN)
INTEGER sdispls(*)	Integer array (of length group size). Entry j specifies the displacement relative to sendbuf from which to take the outgoing data destined for process j (IN)
INTEGER sendtype	The data type of the send buffer elements (handle) (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER recvcounts(*)	Integer array (of length group size) specifying the number of elements to be received from each process (IN)
INTEGER rdispls(*)	Integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i. (IN)
INTEGER recvtype	The data type of the receive buffer elements (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description

This routine sends a distinct message from each process to every process. Messages can have different sizes and displacements. This routine is similar to MPI_ALLTOALL with the following differences. MPI_ALLTOALLV allows you the flexibility to specify the location of the data for the send with sdispls and the location of where the data will be placed on the receive with rdispls. The block of data sent from process i is sendcounts(j) elements long, and is received by process j and placed in recvbuf at offset rdispls(i). These blocks do not have to be the same size. The type signature associated with sendcount(j), sendtype at process i must be equal to the type signature associated with recvcounts(i), recvtype at process j. This means the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are allowed. All arguments on all processes are significant. All processes in comm need to call this routine.

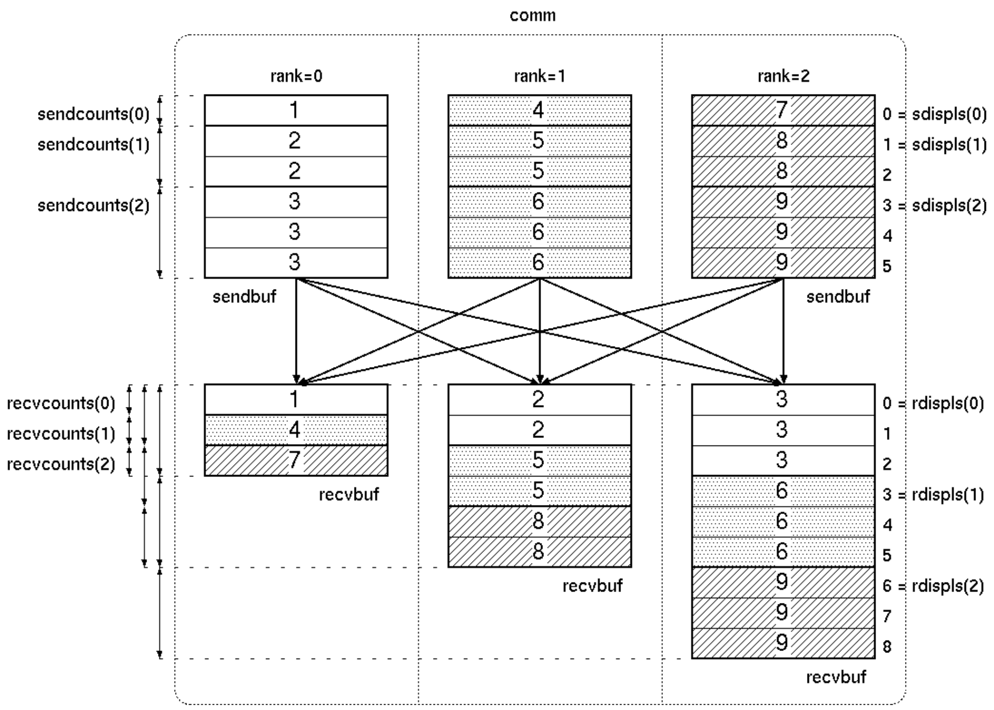


Figure 135. MPI_ALLTOALLV

Sample program

```
      PROGRAM alltoallv
      INCLUDE 'mpif.h'
      INTEGER isend(6), irecv(9)
      INTEGER iscnt(0:2), isdsp(0:2), ircnt(0:2), irdsp(0:2)
      DATA isend/1,2,2,3,3,3/
      DATA iscnt/1,2,3/ isdsp/0,1,3/
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i=1,6
        isend(i) = isend(i) + nprocs * myrank
      ENDDO
      DO i=0,nprocs-1
        ircnt(i) = myrank + 1
        irdsp(i) = i * (myrank + 1)
      ENDDO
      PRINT *,'isend =',isend
      CALL MP_FLUSH(1)                    ! for flushing stdout
      CALL MPI_ALLTOALLV(isend, iscnt, isdsp, MPI_INTEGER,
     &                   irecv, ircnt, irdsp, MPI_INTEGER,
     &                   MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: isend = 1 2 2 3 3 3
   1: isend = 4 5 5 6 6 6
```

```
2: isend = 7 8 8 9 9 9
0: irecv = 1 4 7 0 0 0 0 0 0
1: irecv = 2 2 5 5 8 8 0 0 0
2: irecv = 3 3 3 6 6 6 9 9 9
```

B.2.11 MPI_REDUCE

Purpose	Applies a reduction operation to the vector sendbuf over the set of processes specified by comm and places the result in recvbuf on root.

Usage

```
CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The address of the send buffer (IN)
(CHOICE) recvbuf	The address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (significant only at root) (OUT)
INTEGER count	The number of elements in the send buffer (IN)
INTEGER datatype	The data type of elements of the send buffer (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)
INTEGER root	The rank of the root process (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description

This routine applies a reduction operation to the vector sendbuf over the set of processes specified by comm and places the result in recvbuf on root. Both the input and output buffers have the same number of elements with the same type. The arguments sendbuf, count, and datatype define the send or input buffer and recvbuf, count and datatype define the output buffer. MPI_REDUCE is called by all group members using the same arguments for count, datatype, op, and root. If a sequence of elements is provided to a process, then the reduce operation is executed element-wise on each entry of the sequence. Here's an example. If the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)). Users may define their own operations or use the predefined operations provided by MPI. User-defined operations can be overloaded to operate on several data types, either basic or derived. All processes in comm need to call this routine.

Table 11. Predefined Combinations of Operations and Data Types

Operation	Data Type
MPI_SUM, MPI_PROD	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX
MPI_MAX, MPI_MIN	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
MPI_MAXLOC, MPI_MINLOC	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_LAND, MPI_LOR, MPI_LXOR	MPI_LOGICAL
MPI_BAND, MPI_BOR, MPI_BXOR	MPI_INTEGER, MPI_BYTE

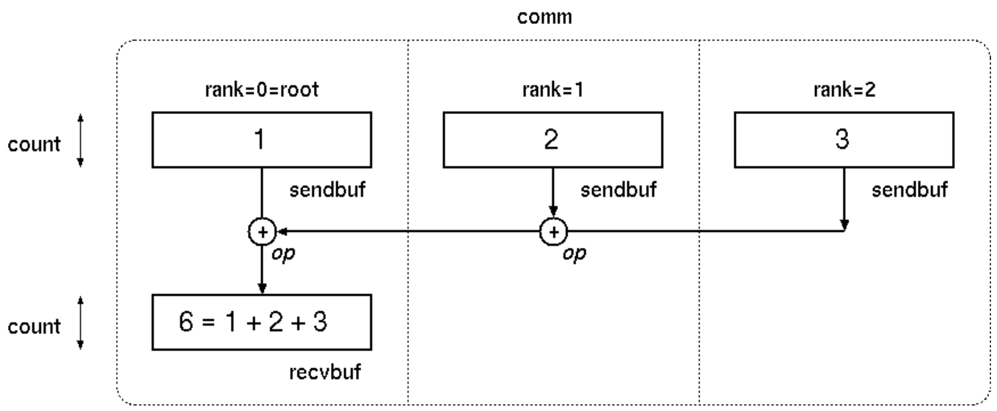


Figure 136. MPI_REDUCE for Scalar Variables

Sample program

```
        PROGRAM reduce
        INCLUDE 'mpif.h'
        CALL MPI_INIT(ierr)
        CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
        CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
        isend = myrank + 1
        CALL MPI_REDUCE(isend, irecv, 1, MPI_INTEGER,
       &                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
        IF (myrank==0) THEN
          PRINT *,'irecv =',irecv
        ENDIF
        CALL MPI_FINALIZE(ierr)
        END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 6
```

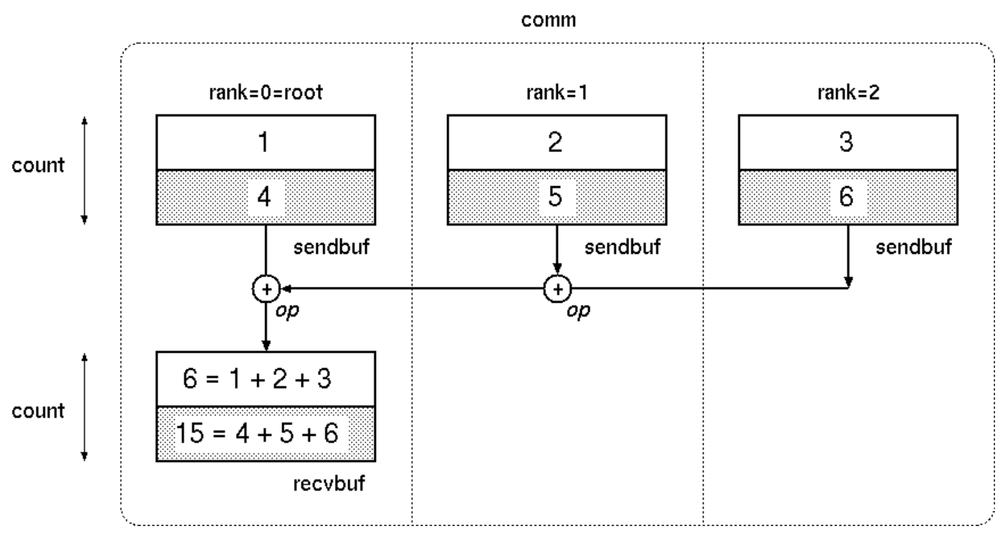


Figure 137. MPI_REDUCE for Arrays

Figure 137 shows how MPI_REDUCE acts on arrays.

B.2.12 MPI_ALLREDUCE

Purpose	Applies a reduction operation to the vector sendbuf over the set of processes specified by comm and places the result in recvbuf on all of the processes in comm.

Usage

```
CALL MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer. sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER count	The number of elements in the send buffer (IN)
INTEGER datatype	The data type of elements of the send buffer (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code
Description	This routine applies a reduction operation to the vector sendbuf over the set of processes specified by comm and places the result in recvbuf on all of the processes. This routine is similar to MPI_REDUCE, except the result is returned to the receive buffer of all the group members. All processes in comm need to call this routine.

For predefined combinations of operations and data types, see Table 11 on page 181.

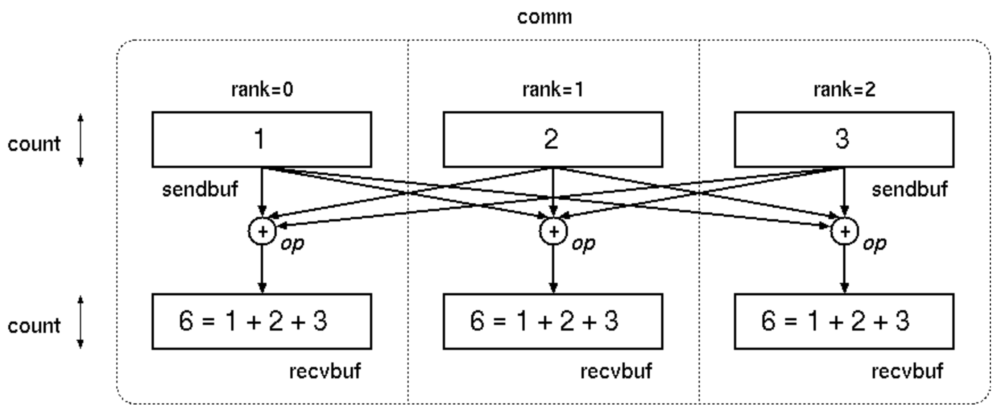


Figure 138. MPI_ALLREDUCE

Sample program

```
      PROGRAM allreduce
      INCLUDE 'mpif.h'
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      isend = myrank + 1
      CALL MPI_ALLREDUCE(isend, irecv, 1, MPI_INTEGER,
     &                   MPI_SUM, MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 6
   1: irecv = 6
   2: irecv = 6
```

B.2.13 MPI_SCAN

Purpose Performs a parallel prefix reduction on data distributed across a group.

Usage

```
CALL MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer. sendbuf and recvbuf cannot overlap in memory (OUT)
INTEGER count	The number of elements in sendbuf (IN)
INTEGER datatype	The data type of elements of sendbuf (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)

INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0..i (inclusive). The type of operations supported, their semantics, and the restrictions on send and receive buffers are the same as for MPI_REDUCE. All processes in comm need to call this routine.

For predefined combinations of operations and data types, see Table 11 on page 181.

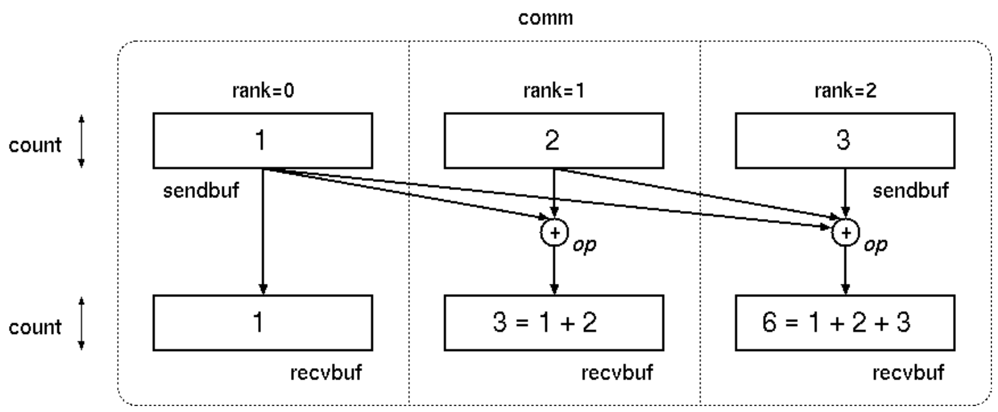


Figure 139. MPI_SCAN

Sample program

```
      PROGRAM scan
      INCLUDE 'mpif.h'
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      isend = myrank + 1
      CALL MPI_SCAN(isend, irecv, 1, MPI_INTEGER,
     &              MPI_SUM, MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 1
   1: irecv = 3
   2: irecv = 6
```

B.2.14 MPI_REDUCE_SCATTER

Purpose Applies a reduction operation to the vector sendbuf over the set of processes specified by comm and scatters the result according to the values in recvcounts.

Usage

```
CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                        ierror)
```

Parameters

(CHOICE) sendbuf	The starting address of the send buffer (IN)
(CHOICE) recvbuf	The starting address of the receive buffer. sendbuf and recvbuf cannot overlap in memory. (OUT)
INTEGER recvcounts(*)	Integer array specifying the number of elements in result distributed to each process. Must be identical on all calling processes. (IN)
INTEGER datatype	The data type of elements of the input buffer (handle) (IN)
INTEGER op	The reduction operation (handle) (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description MPI_REDUCE_SCATTER first performs an element-wise reduction on vector of count = Σ_i recvcounts(i) elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcounts(i) elements. The ith segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts(i) and datatype. MPI_REDUCE_SCATTER is functionally equivalent to MPI_REDUCE with count equal to the sum of recvcounts(i) followed by MPI_SCATTERV with sendcounts equal to recvcounts. All processes in comm need to call this routine.

For predefined combinations of operations and data types, see Table 11 on page 181.

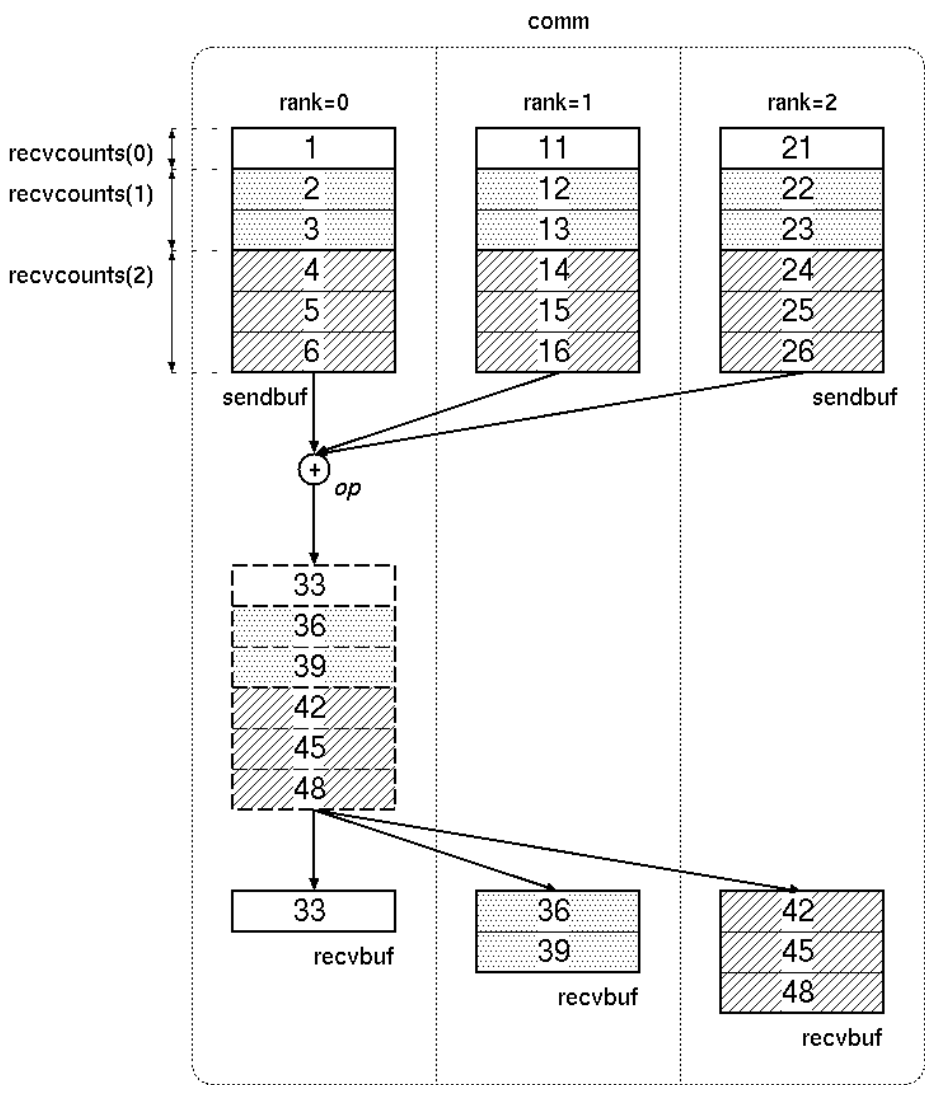


Figure 140. MPI_REDUCE_SCATTER

Sample program

```
      PROGRAM reduce_scatter
      INCLUDE 'mpif.h'
      INTEGER isend(6), irecv(3)
      INTEGER ircnt(0:2)
      DATA ircnt/1,2,3/
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      DO i=1,6
        isend(i) = i + myrank * 10
      ENDDO
      CALL MPI_REDUCE_SCATTER(isend, irecv, ircnt, MPI_INTEGER,
     &                        MPI_SUM, MPI_COMM_WORLD, ierr)
      PRINT *,'irecv =',irecv
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 3
   0: irecv = 33  0  0
   1: irecv = 36  39  0
   2: irecv = 42  45  48
```

B.2.15 MPI_OP_CREATE

Purpose Binds a user-defined reduction operation to an op handle.

Usage

```
CALL MPI_OP_CREATE(func, commute, op, ierror)
```

Parameters

EXTERNAL func	The user-defined reduction function (IN)
INTEGER commute	.TRUE. if commutative; otherwise it's .FALSE. (IN)
INTEGER op	The reduction operation (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This routine binds a user-defined reduction operation to an op handle which you can then use in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER and MPI_SCAN and their nonblocking equivalents. The user-defined operation is assumed to be associative. If commute = .TRUE., then the operation must be both commutative and associative. If commute = .FALSE., then the order of the operation is fixed. The order is defined in ascending process rank order and begins with process zero. func is a user-defined function. It must have the following four arguments: invec, inoutvec, len, and datatype.

```
SUBROUTINE func(invec(*), inoutvec(*), len, type)
<type> invec(len), inoutvec(len)
INTEGER len, type
```

Sample programs 1 and 2 create operations *my_sum* and *my_maxloc*, respectively. The operation *my_sum* is for adding double compex numbers and the operation *my_maxloc* is for finding and locating the maximum element of a two-dimensional integer array. Both operations are not defined by default. (See Table 11 on page 181.)

Table 12. Adding User-Defined Operations

Operation	Data Type
my_sum	*MPI_DOUBLE_COMPLEX*
my_maxloc	*MPI_INTEGER x 3*

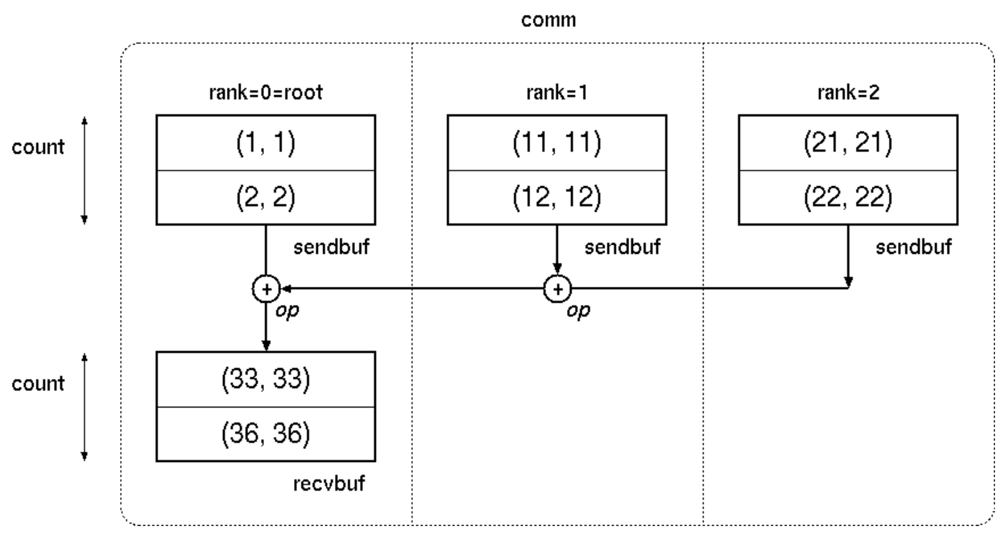


Figure 141. MPI_OP_CREATE

Sample program 1

This program defines an operation *my_sum* that adds double-precision complex numbers, which is used by MPI_REDUCE as shown in Figure 141.

```
      PROGRAM op_create
      INCLUDE 'mpif.h'
      EXTERNAL my_sum
      COMPLEX*16 c(2), csum(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL MPI_OP_CREATE(my_sum, .TRUE., isum, ierr)
      c(1) = (myrank * 10 + 1, myrank * 10 + 1)
      c(2) = (myrank * 10 + 2, myrank * 10 + 2)
      CALL MPI_REDUCE(c, csum, 2, MPI_DOUBLE_COMPLEX, isum,
     &                0, MPI_COMM_WORLD, ierr)
      IF (myrank==0) THEN
        PRINT *,'csum =',csum
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END

      SUBROUTINE my_sum(cin, cinout, len, itype)
      COMPLEX*16 cin(*), cinout(*)
      DO i=1,len
        cinout(i) = cinout(i) + cin(i)
      ENDDO
      END
```

Sample execution 1

```
$ a.out -procs 3
   0: csum = (33.0000000000000000,33.0000000000000000)
(36.0000000000000000,36.0000000000000000)
```

Sample program 2

This program defines an operation *my_maxloc*, which is a generalization of MPI_MAXLOC to two-dimensional arrays.

```
      PROGRAM op_create2
      INCLUDE 'mpif.h'
      EXTERNAL my_maxloc
      INTEGER m(100,100), n(3), nn(3)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      CALL MPI_OP_CREATE(my_maxloc, .TRUE., imaxloc, ierr)
      Each process finds local maxima, imax,
      and its location, (iloc,jloc).
      n(1) = imax
      n(2) = iloc
      n(3) = jloc
      CALL MPI_REDUCE(n, nn, 3, MPI_INTEGER, imaxloc,
     &                0, MPI_COMM_WORLD, ierr)
      IF (myrank==0) THEN
        PRINT *, 'Max =', nn(1), 'Location =', nn(2), nn(3)
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END

      SUBROUTINE my_maxloc(in, inout, len, itype)
      INTEGER in(*), inout(*)
      IF (in(1) > inout(1)) THEN
        inout(1) = in(1)
        inout(2) = in(2)
        inout(3) = in(3)
      ENDIF
      END
```

B.2.16 MPI_BARRIER

Purpose Blocks each process in comm until all processes have called it.

Usage

```
CALL MPI_BARRIER(comm, ierror)
```

Parameters

INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine blocks until all processes have called it. Processes cannot exit the operation until all group members have entered. All processes in comm need to call this routine.

B.3 Point-to-Point Communication Subroutines

The following sections introduce several point-to-point communication subroutines.

B.3.1 MPI_SEND

Purpose Performs a blocking standard mode send operation.

Usage

```
CALL MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
```

Parameters

(CHOICE) buf	The initial address of the send buffer (IN)
INTEGER count	The number of elements in the send buffer (IN)
INTEGER datatype	The data type of each send buffer element (handle) (IN)
INTEGER dest	The rank of the destination process in comm or MPI_PROC_NULL (IN)
INTEGER tag	The message tag. You can choose any integer in the range 0..2^{32}-1 (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER ierror	The Fortran return code

Description This routine is a blocking standard mode send. MPI_SEND causes count elements of type datatype to be sent from buf to the process specified by dest. dest is a process rank which can be any value from 0 to n-1, where n is the number of processes in comm. The message sent by MPI_SEND can be received by either MPI_RECV or MPI_IRECV.

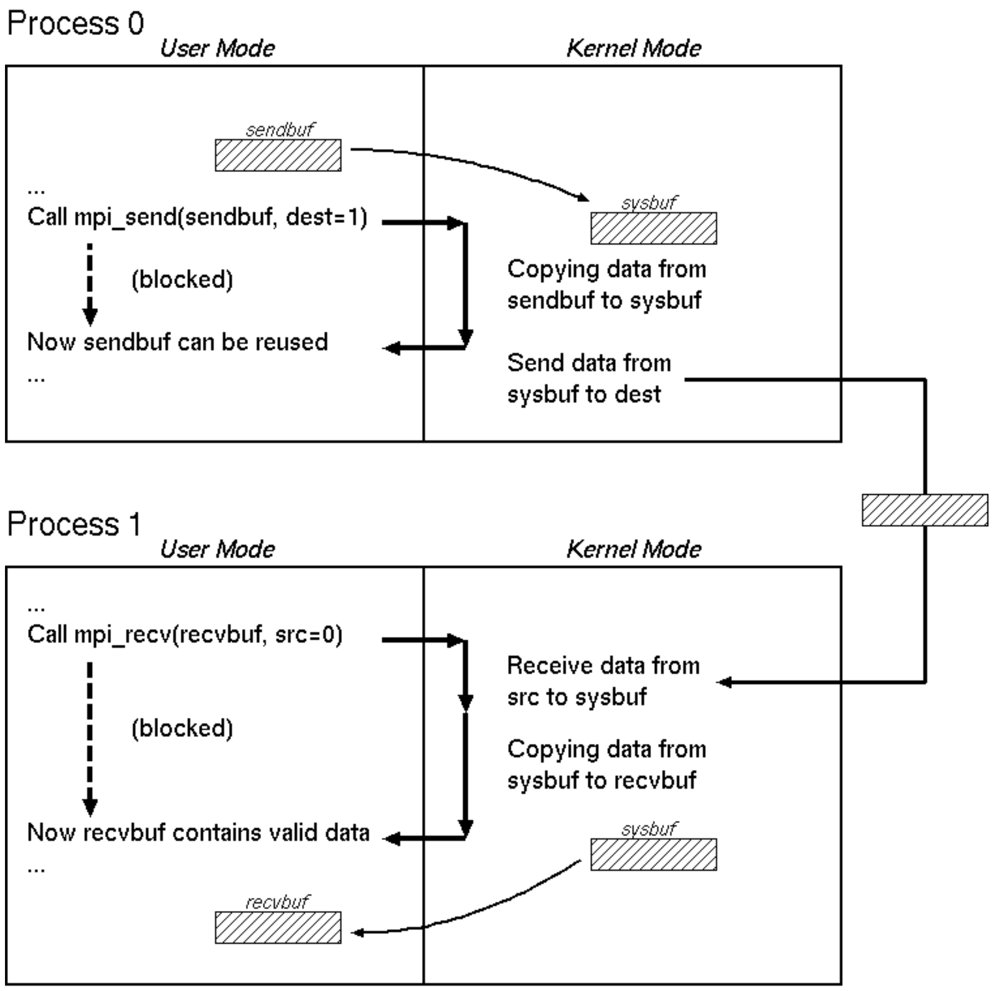

Process 0
User Mode
Kernel Mode
sendbuf
...
Call mpi_send(sendbuf, dest=1)
(blocked)
Now sendbuf can be reused
...
sysbuf
Copying data from sendbuf to sysbuf
Send data from sysbuf to dest
Process 1
User Mode
Kernel Mode
...
Call mpi_recv(recvbuf, src=0)
(blocked)
Now recvbuf contains valid data
...
recvbuf
Receive data from src to sysbuf
Copying data from sysbuf to recvbuf
sysbuf

Figure 142. MPI_SEND and MPI_RECV

Sample program

This program sends `isbuf` of process 0 to `irbuf` of process 1.

```
      PROGRAM send
      INCLUDE 'mpif.h'
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      itag = 1
      IF (myrank==0) THEN
        isbuf = 9
        CALL MPI_SEND(isbuf, 1, MPI_INTEGER, 1, itag,
     &                MPI_COMM_WORLD, ierr)
      ELSEIF (myrank==1) THEN
        CALL MPI_RECV(irbuf, 1, MPI_INTEGER, 0, itag,
     &                MPI_COMM_WORLD, istatus, ierr)
        PRINT *,'irbuf =',irbuf
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   1: irbuf = 9
```

B.3.2 MPI_RECV

Purpose Performs a blocking receive operation.

Usage

```
CALL MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
```

Parameters

(CHOICE) buf	The initial address of the receive buffer (OUT)
INTEGER count	The number of elements to be received (IN)
INTEGER datatype	The data type of each receive buffer element (handle) (IN)
INTEGER source	The rank of the source process in comm, MPI_ANY_SOURCE, or MPI_PROC_NULL (IN)
INTEGER tag	The message tag or MPI_ANY_TAG (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER status(MPI_STATUS_SIZE)	The status object (OUT)
INTEGER ierror	The Fortran return code

Description MPI_RECV is a blocking receive. The receive buffer is storage containing room for count consecutive elements of the type specified by datatype, starting at address buf. The message received must be less than or equal to the length of the receive buffer. If all incoming messages do not fit without truncation, an overflow error occurs. If a message arrives that is shorter than the receive buffer, then only those locations corresponding to the actual message are changed. MPI_RECV can receive a message sent by either MPI_SEND or MPI_ISEND.

Sample program and execution

See the program in B.3.1, “MPI_SEND” on page 190.

B.3.3 MPI_ISEND

Purpose Performs a nonblocking standard mode send operation.

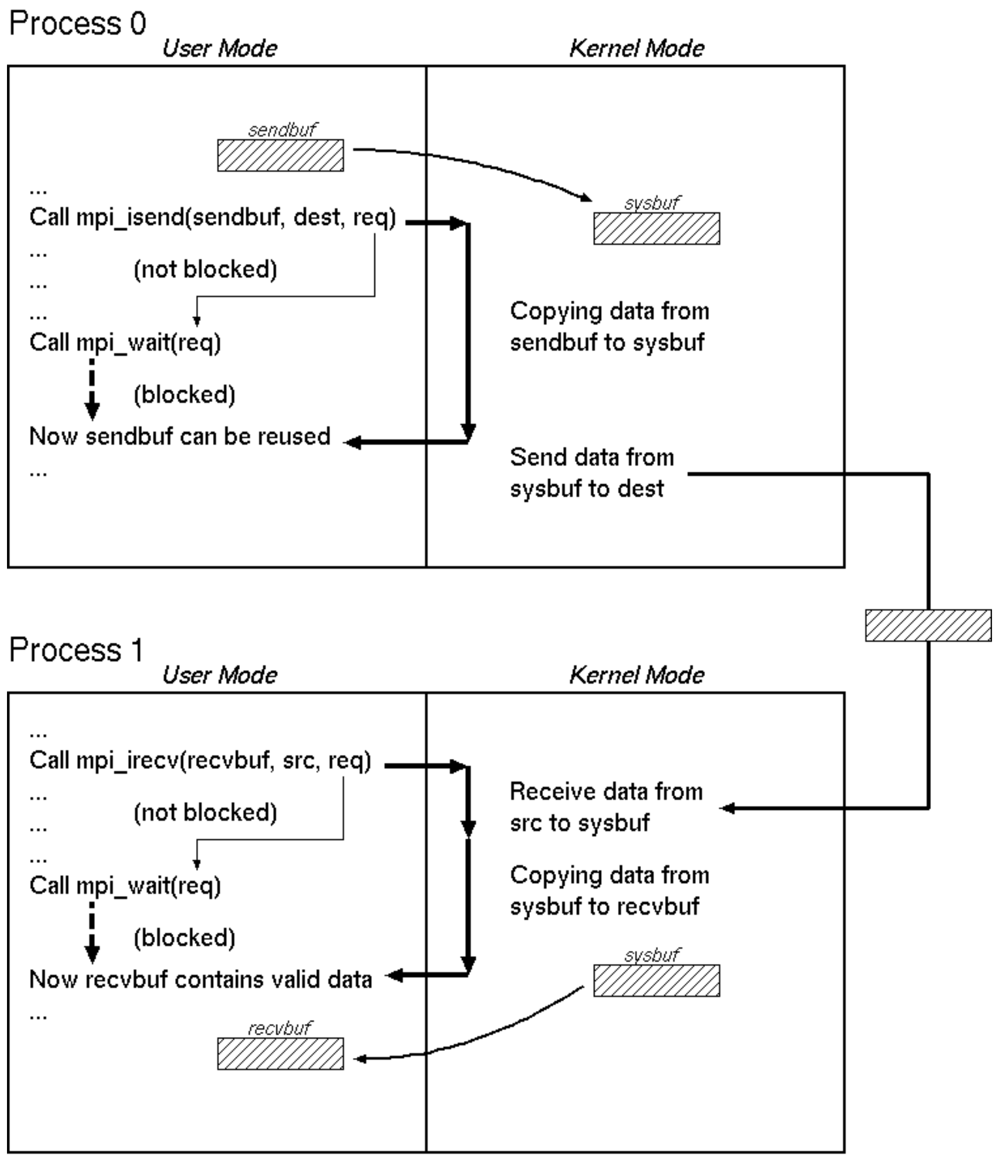
Usage

```
CALL MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)
```

Parameters

(CHOICE) buf	The initial address of the send buffer (IN)

INTEGER count	The number of elements in the send buffer (IN)
INTEGER datatype	The data type of each send buffer element (handle) (IN)
INTEGER dest	The rank of the destination process in comm or MPI_PROC_NULL (IN)
INTEGER tag	The message tag. You can choose any integer in the range 0..2^{32}-1. (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER request	The communication request (handle) (OUT)
INTEGER ierror	The Fortran return code
Description	This routine starts a nonblocking standard mode send. The send buffer may not be modified until the request has been completed by MPI_WAIT, MPI_TEST, or one of the other MPI wait or test functions. The message sent by MPI_ISEND can be received by either MPI_RECV or MPI_IRECV. See B.3.1, “MPI_SEND” on page 190 for additional information.



Process 0
User Mode
Kernel Mode
sendbuf
sysbuf
...
Call mpi_isend(sendbuf, dest, req)
...
(not blocked)
...
...
Call mpi_wait(req)
(blocked)
Now sendbuf can be reused
...
Copying data from sendbuf to sysbuf
Send data from sysbuf to dest
Process 1
User Mode
Kernel Mode
...
Call mpi_irecv(recvbuf, src, req)
...
(not blocked)
...
...
Call mpi_wait(req)
(blocked)
Now recvbuf contains valid data
...
recvbuf
Receive data from src to sysbuf
Copying data from sysbuf to recvbuf
sysbuf

Figure 143. MPI_ISEND and MPI_IRECV

Sample program

This program sends `isbuf` of process 0 to `irbuf` of process 1 in non-blocking manner.

```
      PROGRAM isend
      INCLUDE 'mpif.h'
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      itag = 1
      IF (myrank==0) THEN
        isbuf = 9
        CALL MPI_ISEND(isbuf, 1, MPI_INTEGER, 1, itag,
     &                 MPI_COMM_WORLD, ireq, ierr)
        CALL MPI_WAIT(ireq, istatus, ierr)
      ELSEIF (myrank==1) THEN
        CALL MPI_IRECV(irbuf, 1, MPI_INTEGER, 0, itag,
     &                 MPI_COMM_WORLD, ireq, ierr)
        CALL MPI_WAIT(ireq, istatus, ierr)
        PRINT *,'irbuf =',irbuf
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   1: irbuf = 9
```

B.3.4 MPI_IRECV

Purpose Performs a nonblocking receive operation.

Usage

```
CALL MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierror)
```

Parameters

(CHOICE) buf	The initial address of the receive buffer (OUT)
INTEGER count	The number of elements in the receive buffer (IN)
INTEGER datatype	The data type of each receive buffer element (handle) (IN)
INTEGER source	The rank of source, MPI_ANY_SOURCE, or MPI_PROC_NULL (IN)
INTEGER tag	The message tag or MPI_ANY_TAG (IN)
INTEGER comm	The communicator (handle) (IN)
INTEGER request	The communication request (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This routine starts a nonblocking receive and returns a handle to a request object. You can later use the request to query the status of the communication or wait for it to complete. A nonblocking receive call means the system may start writing data into the receive buffer. Once the nonblocking receive operation is called, do not access any part of the receive buffer until the receive is complete. The message received must be less than or equal to the length of the receive buffer. If all incoming messages do not fit without truncation, an overflow error occurs. If a message arrives that is shorter than the receive buffer, then only those locations corresponding to the actual message are changed. If an overflow occurs, it is flagged at the MPI_WAIT or MPI_TEST. MPI_IRECV can receive a message sent by either MPI_SEND or MPI_ISEND. See B.3.2, “MPI_RECV” on page 192 for additional information.

Sample program and execution

See the program in B.3.3, “MPI_ISEND” on page 192.

B.3.5 MPI_WAIT

Purpose Waits for a nonblocking operation to complete.

Usage

```
CALL MPI_WAIT(request, status, ierror)
```

Parameters

INTEGER request The request to wait for (handle) (INOUT)

INTEGER status(MPI_STATUS_SIZE)
The status object (OUT)

INTEGER ierror The Fortran return code

Description MPI_WAIT returns after the operation identified by request completes. If the object associated with request was created by a nonblocking operation, the object is deallocated and request is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation. You can call MPI_WAIT with a null or inactive request argument. The operation returns immediately. The status argument returns tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE. The status argument is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0. (This is called an empty status.) Information on the completed operation is found in status. You can query the status object for a send or receive operation with a call to MPI_TEST_CANCELLED. For receive operations, you can also retrieve information from status with MPI_GET_COUNT and MPI_GET_ELEMENTS. If wildcards were used by the receive for either the source or tag, the actual source and tag can be retrieved by:

```
source = status(MPI_SOURCE)
tag = status(MPI_TAG)
```

Sample program and execution

See the program in B.3.6, “MPI_GET_COUNT” on page 196.

B.3.6 MPI_GET_COUNT

Purpose Returns the number of elements in a message.

Usage

```
CALL MPI_GET_COUNT(status, datatype, count, ierror)
```

Parameters

INTEGER status(MPI_STATUS_SIZE)
The status object (IN)

INTEGER datatype	The data type of each message element (handle) (IN)
INTEGER count	The number of elements (OUT)
INTEGER ierror	The Fortran return code
Description	This subroutine returns the number of elements in a message. The datatype argument and the argument provided by the call that set the status variable should match. When one of the MPI wait or test calls returns status for a non-blocking operation request and the corresponding blocking operation does not provide a status argument, the status from this wait/test does not contain meaningful source, tag or message size information.

Sample program

Point-to-point communication between processes 0 and 1. After the receiving process, source, tag, and number of elements sent are examined.

```
      PROGRAM get_count
      INCLUDE 'mpif.h'
      INTEGER istatus(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      itag = 1
      IF (myrank==0) THEN
        isbuf = 9
        CALL MPI_ISEND(isbuf, 1, MPI_INTEGER, 1, itag,
     &                 MPI_COMM_WORLD, ireq, ierr)
      ELSEIF (myrank==1) THEN
        CALL MPI_IRECV(irbuf, 1, MPI_INTEGER, 0, itag,
     &                 MPI_COMM_WORLD, ireq, ierr)
        CALL MPI_WAIT(ireq, istatus, ierr)
        CALL MPI_GET_COUNT(istatus, MPI_INTEGER, icount, ierr)
        PRINT *,'irbuf  =',irbuf
        PRINT *,'source =',istatus(MPI_SOURCE)
        PRINT *,'tag    =',istatus(MPI_TAG)
        PRINT *,'count  =',icount
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   1: irbuf  = 9
   1: source = 0
   1: tag    = 1
   1: count  = 1
```

B.4 Derived Data Types

The following sections introduce several derived data types.

B.4.1 MPI_TYPE_CONTIGUOUS

Purpose Returns a new data type that represents the concatenation of count instances of oldtype.

Usage

```
CALL MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)
```

Parameters

INTEGER count	The replication count (IN)
INTEGER oldtype	The old data type (handle) (IN)
INTEGER newtype	The new data type (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This routine returns a new data type that represents the concatenation of count instances of oldtype. MPI_TYPE_CONTIGUOUS allows replication of a data type into contiguous locations. newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

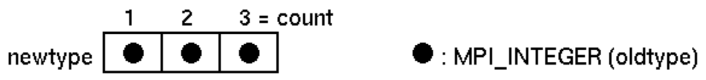


Figure 144. MPI_TYPE_CONTIGUOUS

Sample program

This program defines a data type representing three contiguous integers.

```
PROGRAM type_contiguous
INCLUDE 'mpif.h'
INTEGER ibuf(20)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
  DO i=1,20
    ibuf(i) = i
  ENDDO
ENDIF
CALL MPI_TYPE_CONTIGUOUS(3, MPI_INTEGER, inewtype, ierr)
CALL MPI_TYPE_COMMIT(inewtype, ierr)
CALL MPI_BCAST(ibuf, 3, inewtype, 0, MPI_COMM_WORLD, ierr)
PRINT *,'ibuf =',ibuf
CALL MPI_FINALIZE(ierr)
END
```

Sample execution

```
$ a.out -procs 2
   0: ibuf = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
   1: ibuf = 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0 0
```

B.4.2 MPI_TYPE_VECTOR

Purpose Returns a new data type that represents equally spaced blocks. The spacing between the start of each block is given in units of extent (`oldtype`).

Usage

```
CALL MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
```

Parameters

INTEGER count	The number of blocks (IN)
INTEGER blocklength	The number of oldtype instances in each block (IN)
INTEGER stride	The number of units between the start of each block (IN)
INTEGER oldtype	The old data type (handle) (IN)
INTEGER newtype	The new data type (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This function returns a new data type that represents count equally spaced blocks. Each block is a a concatenation of blocklength instances of oldtype. The origins of the blocks are spaced stride units apart where the counting unit is extent(oldtype). That is, from one origin to the next in bytes = stride * extent (oldtype). newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

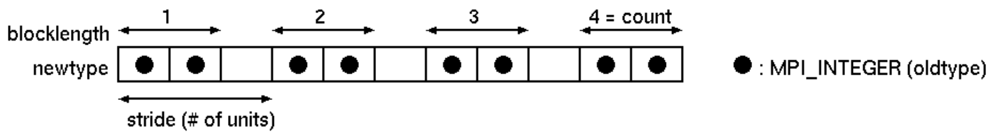


Figure 145. MPI_TYPE_VECTOR

Sample program

This program defines a data type representing eight integers with gaps (see Figure 145).

```
PROGRAM type_vector
INCLUDE 'mpif.h'
INTEGER ibuf(20)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
  DO i=1,20
    ibuf(i) = i
  ENDDO
ENDIF
CALL MPI_TYPE_VECTOR(4, 2, 3, MPI_INTEGER, inewtype, ierr)
CALL MPI_TYPE_COMMIT(inewtype, ierr)
```

```
      CALL MPI_BCAST(ibuf, 1, inewtype, 0, MPI_COMM_WORLD, ierr)
      PRINT *,'ibuf =',ibuf
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   0: ibuf = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
   1: ibuf = 1 2 0 4 5 0 7 8 0 10 11 0 0 0 0 0 0 0 0 0
```

B.4.3 MPI_TYPE_HVECTOR

Purpose Returns a new data type that represents equally-spaced blocks. The spacing between the start of each block is given in bytes.

Usage

```
CALL MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype, ierror)
```

Parameters

INTEGER count	The number of blocks (IN)
INTEGER blocklength	The number of oldtype instances in each block (IN)
INTEGER stride	An integer specifying the number of bytes between start of each block. (IN)
INTEGER oldtype	The old data type (handle) (IN)
INTEGER newtype	The new data type (handle) (OUT)
INTEGER ierror	The Fortran return code

Description This routine returns a new data type that represents count equally spaced blocks. Each block is a concatenation of blocklength instances of oldtype. The origins of the blocks are spaced stride units apart where the counting unit is one byte. newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

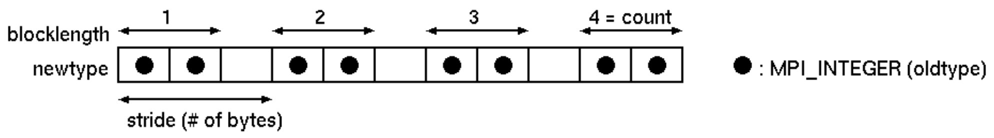


Figure 146. MPI_TYPE_HVECTOR

Sample program

This program defines a data type representing eight integers with gaps (see Figure 146).

```
      PROGRAM type_hvector
      INCLUDE 'mpif.h'
      INTEGER ibuf(20)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        DO i=1,20
          ibuf(i) = i
        ENDDO
      ENDIF
      CALL MPI_TYPE_HVECTOR(4, 2, 3*4, MPI_INTEGER,
     &                      inewtype, ierr)
      CALL MPI_TYPE_COMMIT(inewtype, ierr)
      CALL MPI_BCAST(ibuf, 1, inewtype, 0, MPI_COMM_WORLD, ierr)
      PRINT *,'ibuf =',ibuf
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   0: ibuf = 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
   1: ibuf = 1  2  0  4  5  0  7  8  0  10  11  0  0  0  0  0  0  0  0  0
```

B.4.4 MPI_TYPE_STRUCT

Purpose Returns a new data type that represents `count` blocks. Each is defined by an entry in `array_of_blocklengths`, `array_of_displacements` and `array_of_types`. Displacements are expressed in bytes.

Usage

```
CALL MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements,
                     array_of_types, newtype, ierror)
```

Parameters

INTEGER count An integer specifying the number of blocks. It is also the number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths. (IN)

INTEGER array_of_blocklengths(*)
The number of elements in each block (array of integer). That is, array_of_blocklengths(i) specifies the number of instances of type array_of_types(i) in block(i). (IN)

INTEGER array_of_displacements(*)
The byte displacement of each block (array of integer) (IN)

INTEGER array_of_types(*)
The data type comprising each block. That is, block(i) is made of a concatenation of type array_of_types(i). (array of handles to data type objects) (IN)

INTEGER newtype The new data type (handle) (OUT)

INTEGER ierror The Fortran return code

Description This routine returns a new data type that represents count blocks. Each is defined by an entry in array_of_blocklengths, array_of_displacements and array_of_types. Displacements are expressed in bytes. MPI_TYPE_STRUCT is the most general type constructor. It allows each block to consist of replications of different data types. This is the only constructor which allows MPI pseudo types MPI_LB and MPI_UB. Without these pseudo types, the extent of a data type is the range from the first byte to the last byte rounded up as needed to meet boundary requirements. For example, if a type is made of an integer followed by 2 characters, it will still have an extent of 8 because it is padded to meet the boundary constraints of an int. This is intended to match the behavior of a compiler defining an array of such structures. Because there may be cases in which this default behavior is not correct, MPI provides a means to set explicit upper and lower bounds which may not be directly related to the lowest and highest displacement data type. When the pseudo type MPI_UB is used, the upper bound will be the value specified as the displacement of the MPI_UB block. No rounding for alignment is done. MPI_LB can be used to set an explicit lower bound but its use does not suppress rounding. When MPI_UB is not used, the upper bound of the data type is adjusted to make the extent a multiple of the type's most boundary constrained component. In order to define a data type ending with one or more empty slots, add a trailing block of length=1 and type=MPI_UB. For a data type beginning with one or more empty slots, add an opening block of length=1 and type=MPI_LB. newtype must be committed using MPI_TYPE_COMMIT before being used for communication.

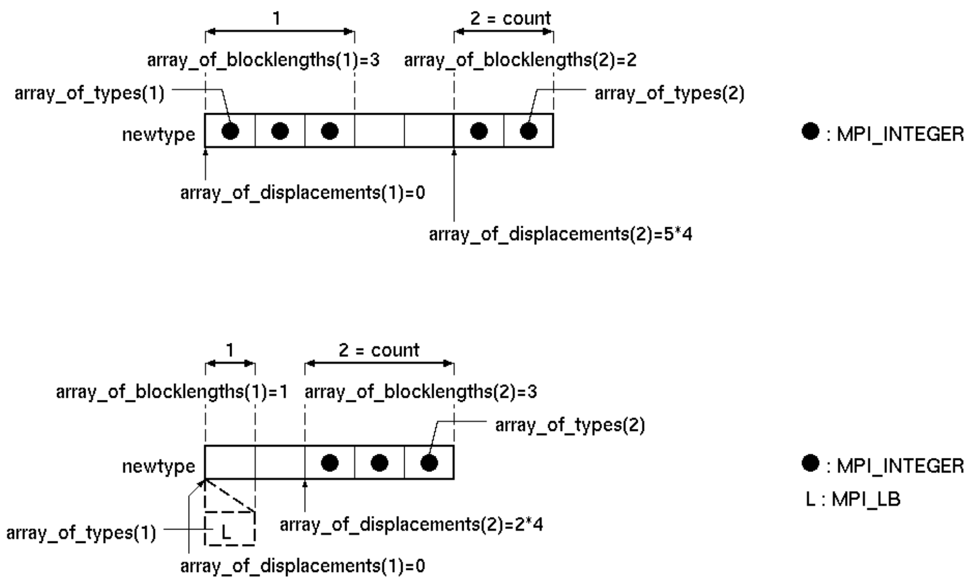

1
2 = count
array_of_blocklengths(1)=3
array_of_blocklengths(2)=2
array_of_types(1)
array_of_types(2)
newtype
array_of_displacements(1)=0
array_of_displacements(2)=5*4
●: MPI_INTEGER
1
2 = count
array_of_blocklengths(1)=1
array_of_blocklengths(2)=3
array_of_types(2)
newtype
array_of_types(1)
L
array_of_displacements(2)=2*4
array_of_displacements(1)=0
●: MPI_INTEGER
L : MPI_LB

Figure 147. MPI_TYPE_STRUCT

Sample program

This program defines two data types shown in Figure 147.

```
PROGRAM type_struct
INCLUDE 'mpif.h'
```

```
      INTEGER ibuf1(20), ibuf2(20)
      INTEGER iblock(2), idisp(2), itype(2)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        DO i=1,20
          ibuf1(i) = i
          ibuf2(i) = i
        ENDDO
      ENDIF
      iblock(1) = 3
      iblock(2) = 2
      idisp(1)  = 0
      idisp(2)  = 5 * 4
      itype(1)  = MPI_INTEGER
      itype(2)  = MPI_INTEGER
      CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype,
     &                     inewtype1, ierr)
      CALL MPI_TYPE_COMMIT(inewtype1, ierr)
      CALL MPI_BCAST(ibuf1, 2, inewtype1, 0, MPI_COMM_WORLD, ierr)
      PRINT *,'Ex. 1:',ibuf1
      iblock(1) = 1
      iblock(2) = 3
      idisp(1)  = 0
      idisp(2)  = 2 * 4
      itype(1)  = MPI_LB
      itype(2)  = MPI_INTEGER
      CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype,
     &                     inewtype2, ierr)
      CALL MPI_TYPE_COMMIT(inewtype2, ierr)
      CALL MPI_BCAST(ibuf2, 2, inewtype2, 0, MPI_COMM_WORLD, ierr)
      PRINT *,'Ex. 2:',ibuf2
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 2
   0: Ex. 1: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
   0: Ex. 2: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
   1: Ex. 1: 1 2 3 0 0 6 7 8 9 10 0 0 13 14 0 0 0 0 0 0
   1: Ex. 2: 0 0 3 4 5 0 0 8 9 10 0 0 0 0 0 0 0 0 0 0
```

B.4.5 MPI_TYPE_COMMIT

Purpose Makes a data type ready for use in communication.

Usage

```
CALL MPI_TYPE_COMMIT(datatype, ierror)
```

Parameters

INTEGER datatype	The data type that is to be committed (handle) (INOUT)
INTEGER ierror	The Fortran return code

Description A data type object must be committed before you can use it in communication. You can use an uncommitted data type as an argument in data type constructors. This routine makes a data type ready for use in communication. The data type is the formal description of a communication buffer. It is not the content of the buffer. Once the data type is committed it can be repeatedly reused to communicate the changing contents of a buffer or buffers with different starting addresses.

Sample program and execution

See the program in B.4.1, “MPI_TYPE_CONTIGUOUS” on page 198.

B.4.6 MPI_TYPE_EXTENT

Purpose Returns the extent of any defined data type.

Usage

```
CALL MPI_TYPE_EXTENT(datatype, extent, ierror)
```

Parameters

INTEGER datatype	The data type (handle) (IN)
INTEGER extent	The data type extent (integer) (OUT)
INTEGER ierror	The Fortran return code

Description This routine returns the extent of a data type. The extent of a data type is the span from the first byte to the last byte occupied by entries in this data type and rounded up to satisfy alignment requirements. Rounding for alignment is not done when MPI_UB is used to define the data type. Types defined with MPI_LB, MP_UB or with any type that contains MPI_LB or MPI_UB may return an extent which is not directly related to the layout of data in memory. Refer to MPI_TYPE_STRUCT for more information on MPI_LB and MPI_UB.

Sample program

```
PROGRAM type_extent
INCLUDE ’mpif.h’
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL MPI_TYPE_EXTENT(MPI_INTEGER, iextent, ierr)
PRINT *,’iextent =’,iextent
CALL MPI_FINALIZE(ierr)
END
```

Sample execution

```
$ a.out -procs 1
   0: iextent = 4
```

B.5 Managing Groups

The following sections introduce several managing groups.

B.5.1 MPI_COMM_SPLIT

Purpose Splits a communicator into multiple communicators based on `color` and `key`.

Usage

```
CALL MPI_COMM_SPLIT(comm, color, key, newcomm, ierror)
```

Parameters

INTEGER comm	The communicator (handle) (IN)
INTEGER color	An integer specifying control of subset assignment (IN)
INTEGER key	An integer specifying control of rank assignment (IN)
INTEGER newcomm	The new communicator (handle) (OUT)
INTEGER ierror	The Fortran return code

Description MPI_COMM_SPLIT is a collective function that partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key. Ties are broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. If a process supplies the color value MPI_UNDEFINED, newcomm returns MPI_COMM_NULL. Even though this is a collective call, each process is allowed to provide different values for color and key. This call applies only to intracommunicators. The value of color must be greater than or equal to zero. All processes in comm need to call this routine.

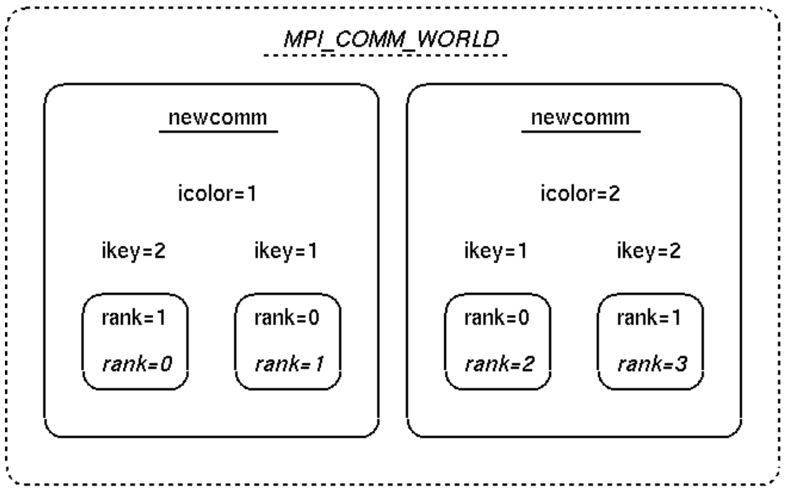


Figure 148. MPI_COMM_SPLIT

Sample program

```
      PROGRAM comm_split
      INCLUDE 'mpif.h'
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
      IF (myrank==0) THEN
        icolor = 1
        ikey   = 2
      ELSEIF (myrank==1) THEN
        icolor = 1
        ikey   = 1
      ELSEIF (myrank==2) THEN
        icolor = 2
        ikey   = 1
      ELSEIF (myrank==3) THEN
        icolor = 2
        ikey   = 2
      ENDIF
      CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, icolor, ikey,
     &                    newcomm, ierr)
      CALL MPI_COMM_SIZE(newcomm, newprocs, ierr)
      CALL MPI_COMM_RANK(newcomm, newrank,  ierr)
      PRINT *,'newcomm =', newcomm,
     &        'newprocs =',newprocs,
     &        'newrank =', newrank
      CALL MPI_FINALIZE(ierr)
      END
```

Sample execution

```
$ a.out -procs 4
   0: newcomm = 3 newprocs = 2 newrank = 1
   1: newcomm = 3 newprocs = 2 newrank = 0
   2: newcomm = 3 newprocs = 2 newrank = 0
   3: newcomm = 3 newprocs = 2 newrank = 1
```

In Figure 148 on page 205, ranks in MPI_COMM_WORLD are printed in italic, and ranks in `newcomm` are in boldface.

Appendix C. Special Notices

This publication is intended to help application designers and developers exploit the parallel architecture of the RS/6000 SP. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM Parallel Environment for AIX (PE) and Parallel System Support Programs (PSSP). See the PUBLICATIONS section of the IBM Programming Announcement for PE and PSSP for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Appendix D. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

D.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 211.

- *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, SG24-5155
- *RS/6000 Systems Handbook*, SG24-5120

D.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037

D.3 Other Publications

These publications are also relevant as further information sources:

- *Parallel Environment for AIX: MPI Programming and Subroutine Reference Version 2 Release 4*, GC23-3894
- *AIX Version 4 Optimization and Tuning Guide for FORTRAN, C, and C++*, SC09-1705
- *Engineering and Scientific Subroutine Library for AIX Guide and Reference*, SA22-7272 (latest version available at `http://www.rs6000.ibm.com/resource/aix_resource/sp_books`)
- *Parallel Engineering and Scientific Subroutine Library for AIX Guide and Reference*, SA22-7273 (latest version available at `http://www.rs6000.ibm.com/resource/aix_resource/sp_books`)
- *Parallel Environment for AIX: Operation and Use, Volume 1: Using the Parallel Operating Environment Version 2 Release 4*, SC28-1979
- *Parallel Environment for AIX: Operation and Use, Volume 2*, SC28-1980
- *IBM LoadLeveler for AIX Using and Administering Version 2 Release 1*, SA22-7311

- A. Tanenbaum, *Structured Computer Organization (4th Edition)*, Prentice Hall (1999)
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach (2nd Edition)*, Morgan Kaufmann (1996)
- D. Culler, J.Singh, and A. Gupta, *Parallel Computer Architecture A Hardware/Software Approach,* Morgan Kaufmann (1999)
- K. Dowd and C. Severance, *High Performance Computing (2nd Edition),* O'Reilly (1998)
- W. Gropp, E. Lusk, and A. Skjellum, *Using MPI,* MIT Press (1994)
- M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI The Complete Reference,* MIT Press (1996)
- P. Pacheco, *Parallel Programming with MPI,* Morgan Kaufmann (1997)
- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in Fortran 77 (2nd Edition),* Cambridge University Press (1996)

This redbook is based on an unpublished document written in Japanese. Contact nakanoj@jp.ibm.com for details.

D.4 Information Available on the Internet

The following information is available on-line.

- `http://www.mpi-forum.org/`
  Message Passing Interface Forum
- `http://www.research.ibm.com/mathsci/ams/ams_WSSMP.htm`
  WSSMP: Watson Symmetric Sparse Matrix Package
- `http://www.pallas.de/pages/pmb.htm`
  Pallas MPI Benchmarks

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** `http://www.redbooks.ibm.com/`

  Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at `http://w3.itso.ibm.com/` and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access `MyNews` at `http://w3.ibm.com/` for redbook, residency, and workshop announcements.

IBM Redbook Fax Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

ADI	Alternating Direction Implicit
BLACS	Basic Linear Algebra Communication Subroutines
ESSL	Engineering and Scientific Subroutine Library
FDM	Finite Difference Model
GPFS	General Parallel File System
HPF	High Performance Fortran
ICCG	Incomplete Cholesky Conjugate Gradient
ITSO	International Technical Support Organization
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
MPMD	Multiple Programs Multiple Data
MPP	Massively Parallel Processors
MUSPPA	Multiple User Space Processes Per Adapter
NUMA	Non-Uniform memory Access
PDPBSV	Positive Definite Symmetric Band Matrix Factorization and Solve
PE	Parallel Environment
PESSL	Parallel ESSL
SMP	Symmetric Multiprocessor
SOR	Successive Over-Relaxation
SPMD	Single Program Multiple Data
US	User Space
WSSMP	Watson Symmetric Sparse Matrix Package

Index

ITSO Redbook Evaluation

RS/6000 SP: Practical MPI Programming
SG24-5380-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to `redbook@us.ibm.com`

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-5380-00

Printed in the U.S.A.

RS/6000 SP: Practical MPI Programming

SG24-5380-00