CSE5351: Parallel Processing

Part IV

Message-Passing Libraries

Name	Original Creator	Distinct Features
CMMD	Thinking Machines	Use Active Messages for low latency
Express	Parasoft	Collective communication and I/O
Fortran-M	Argonne National Lab	Modularity and determinacy
MPI	MPI Forum	A widely adopted standard
Nx	Intel	Originated from the Intel hypercube MPPs
P4	Argonne National Lab	Integrate shared memory and message passing
PARMACS	ANL/GMD	Mainly used in Europe
PVM	Oak Ridge National Lab	A widely-used, stand-alone system
UNIFY	Mississippi State	A system allowing both MPI and PVM calls
Zipcode	Livermore National Lab	Contributed to the context concept

Communication using message-Passing

There are three aspects of a communication mode that a user should understand:

How many processes are involved?

How are the processes synchronized?

How are communication buffers managed?

In the following code, process *P* sends a message contained in variable *M* to process *Q*, which receives the message into its variable *S*.

```
Process P:          Process Q:

    M = 10;         S = –100
L1:send M to Q;     L1:         receive S from P;
L2:M = 20;          L2:         X = S + 1;
    goto L1;
```

The variable *M* is often called the *send message buffer* (or *send buffer*), and *S* is called the *receive message buffer* (or *receive buffer*).

Methods of Communication

Three communication modes are used in today's message-passing systems.

- Synchronous
- Blocking
- Non-Blocking

Synchronous Message Passing

- When process *P* executes a *synchronous send M* to *Q*, it has to wait until process *Q* executes a corresponding *synchronous receive S* from *P*.
- Both processes will not return from the send or the receive until the message *M* is both sent and received.
- When the *send* and *receive* return, *M* can be immediately overwritten by *P* and *S* can be immediately read by *Q*, in the subsequent statements L2.
- No additional buffer needs to be provided in synchronous message passing.
- The receive message buffer *S* is available to hold the arriving message.

Blocking Send/Receive

- A *blocking send* is executed when a process reaches it, without waiting for a corresponding receive.
- A *blocking send* does not return until the message is sent, meaning the message variable *M* can be safely rewritten.
- Note that when the send returns, a corresponding receive is not necessarily finished, or even started. It may be temporarily buffered in the sending node, somewhere in the network, or it may have arrived at a buffer in the receiving node.
- A *blocking receive* is executed when a process reaches it, without waiting for a corresponding send. However, it cannot return until the message is received. The system may have to provide a temporary buffer for blocking-mode message passing.

NonBlocking Send/Receive

- A *nonblocking send* is executed when a process reaches it, without waiting for a corresponding receive.
- A *nonblocking send* can return immediately after it notifies the system that the message *M* is to be sent. The message data are not necessarily out of *M*. Therefore, it is unsafe to overwrite *M*.
- A *nonblocking receive* is executed when a process reaches it, without waiting for a corresponding send. It can return immediately after it notifies the system that there is a message to be received. The message may have already arrived, may be still in transit, or may have not even been sent yet.
- The system may have to provide a temporary buffer for nonblocking message passing.

Comparison of Three Modes

- For synchronous, when a process returns from a send subroutine, it knows for certain that the message is sent and received.
- For synchronous, a separate data buffer is not required to be set up by the user or by the system. The message *M* can be directly copied into *S* in the receiver process' address space.
- A disadvantage is that the sender must wait for the receiver, which leads to some wasted cycles.
- In real parallel systems, there are variants on this definition of synchronous (or the other two) mode.

Comparison of Three Modes

Communication Event	Synchronous	Blocking	Nonblocking
Send Start Condition	**Both send and** receive reached	Send reached	Send reached
Return of send indicates	Message received	Message sent	Message send initiated
Semantics	Clean	In-Between	Error-Prone
Buffering Message	Not needed	Needed	Needed
Status Checking	Not needed	Not needed	Needed
Wait Overhead	Highest	In-Between	Lowest
Overlapping in Communications and Computations	No	Yes	Yes

Non-Blocking Example

Process P:

```
M = 10;
L: send M to Q;
   do some computation
   which does not change M
   wait for M to be sent
   M = 20;
```

Process Q:

```
S = – 100
receive S from P;
  do some computation
  which does not use S
  wait for S to be received
  X = S + 1;
```

Non-Blocking Send and Receive

- The main motivation for using the nonblocking mode is to overlap communication and computation.
- The nonblocking mode reduces the wait time to a minimum. However, it causes an additional problem.
- Nonblocking introduces its own inefficiencies, such as extra memory space for the temporary buffers, allocation of the buffer, copying message into and out of the buffer, and the execution of an extra wait-for function.
- These overheads may significantly offset any gains from overlapping communication with computation.
- If sufficient buffer space is not available, the system may fail, or the parallel program may hang (deadlock).

Introduction to Message-Passing Interface (MPI)

- **A message-passing library specification**

- message-passing model
- not a compiler specification
- not a specific product

- **For parallel computers, clusters and heterogeneous networks**

- **Full-featured**

- **Designed to permit the development of parallel software libraries**

- **Designed to provide access to advanced parallel hardware for**

- end users
- library writers
- tool developers

The MPI Process

- Began at Williamsburg Workshop in April, 1992
- Organized at Supercomputing ‘92 (November)
- Extensive, open email discussions
- Drafts, readings, votes
- Pre-final draft distributed at Supercomputing ‘93
- Two month public comment period
- Final version of draft in May, 1994
- Public implementations available
- Vendor implementations coming soon

Portability of MPI

- **The most important feature of MPI is that it is portable. This leads to three advantages.**
- It can be implemented on a variety of machines such as Stampede, Titan and Tianhe.
- It can be implemented on various types of workstation networks.
- Portability: Programs written under MPI for one machine can be run on another machine and the user does not have to worry about the hardware.

Interprocessor communication in MPI

- Blocking communication among nodes (read and write a message, read and write a vector.
- Non-blocking communication.
- Global communication (broadcast, exchange, global concatenation, global reduction operations, etc.).
- Synchronization.

Execution Model

- The languages supported by MPI are C and Fortran.
- Both languages are equally well supported.
- Programming in MPI is done by including calls to MPI libraries in regular C or Fortran programs.
- Both SPMD and MPMD models can be used.

Installing MPI

- MPICH is a freely available, portable implementation of MPI
- Implemented by Argonne National Laboratory, Mathematics and Computer Science Division (ANL/MCS)
- Download from:
- http://www.mcs.anl.gov/mpi/mpich/download.html

Installing MPI (cont'd)

- Obtaining the distribution mpich.tar.gz or mpich.tar.Z
- Unpacking the distribution with gzip, uncompress and tar utilities
- Run 'configure' script in mpich directory and it will attempt to choose an appropriate default architecture and device for you
- Run make with the generated Makefile and the executables will be compiled
- A complete installation guide is available from MPICH download page
- Installed package can be found in CS UG domain under /usr/local/ packages/mpich/

Basic Skeleton of MPI Programs

```
#include “mpi.h”
#include <stdio.h>
int main (argc, argv)
int argc;
char **argv;
{
   /* No MPI functions called before this */
   MPI_Init(&argc, &argv);
   /* Normal C code and calls to MPI libraries */
   printf(“Hello World\n”);
   MPI_Finalize();
   /* No MPI functions called after this */
return 0;
}
```

Commentary

`#include ”mpi.h”`

provides basic MPI definitions and types

`MPI_Init`

starts MPI

`MPI_Finalize`

exits MPI

Note that all non-MPI routines are local; thus the `printf` run on each process

Compiling and Linking

- For simple programs, special compiler commands can be used. For large projects, it is best to use a standard Makefile.
- The MPICH implementations provides the commands `mpicc` and `mpif77` as well as Makefile examples in `/usr/local/packages/mpich/examples/Makefile.in'`
- `The file 'Makefile.in'` is a template Makefile. The script `'mpireconfig'` translates this to a Makefile for a particular system.
- This allows you to use the same Makefile for a network of workstations and a massively parallel computer, even when they use different compilers, libraries and linker options.

Special Compilation Commands

- **The commands**
  `mpicc -o first first.c`
  `mpif77 -o first first.f`
  may be used to build simple programs when using MPICH
- **Special Options that exploit the profiling features of MPI**

`mpilog` Generate log files of MPI calls
`mpitrace` Trace execution of MPI calls
`mpianim` Real-time animation of MPI (not available on all systems)

Running MPI Programs

% mpirun -np 2 first

'mpirun' is not part of the standard, but some version of it is common with MPI implementations

The version shown here is for the MPICH implementation of MPI

-np Specify the number of processors to run on

On a network (e.g., a lab) machines can be chosen from default machine-file '/usr/local/packages/mpich/util/machines/machines.solaris' with a list of all available machines

-machinefile Use user-specified machine-file with self-defined list of all available machines

-help Show all options to mpirun

Finding Out About the Environment

- **Two of the first questions asked in a parallel program are:**
  - How many processes are there?
  - Who am I?
- **How many is answered with `MPI_Comm_size`**
- **Who am I is answered with `MPI_Comm_rank`**
- **The rank is a number between `zero and size-1`**

A Simple Program

```
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
int argc;
char **argv;
{
int rank, size;
MPI_Init(&argc, &argv);                    /* Starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Hello World! I'm %d of %d\n",rank, size);          /* printf() is local, */
                                                          /* run on each process */
   MPI_Finalize();                         /* Exits MPI */
   return 0;
}
```

A Simple Program (cont'd)

- **Compilation and linking**
  - **mpicc -o hello hello.c**
- **Running the executable**
  - **mpirun -np 4 -machinefile mfile hello**
- **Execution output**
  - **Hello World! I'm 1 of 4**
  - **Hello World! I'm 2 of 4**
  - **Hello World! I'm 0 of 4**
  - **Hello World! I'm 3 of 4**

Example

```
#include "mpi.h"
int foo(i) int i ; { ... }
main(argc, argv)
int argc;
char * argv[] ;
{  int i, tmp, sum = 0, group_size, my_rank, N, K;
   MPI_Init (&argc, &argv) ;
   MPI_Comm_size (MPI_COMM_WORLD, &group_size) ;
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank) ;
   if ( my_rank == 0 ) {
            printf("Enter N: "); scanf("%d", &N);
            for (i=1; i<group_size; i++)
S1:                 MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD) ;
            for ( i = my_rank ; i<N ; i = i + group_size )
                    sum = sum + foo(i) ;
            for (i=1; i<group_size; i++) {
S2:                 MPI_Recv(&tmp, 1, MPI_INT, i, i, MPI_COMM_WORLD, &status) ;
                    sum = sum + tmp ;
            }
            printf ("\n The result = %d", sum) ;
   }
```

```
else {                          /* if my_rank != 0 */
S3:                 MPI_Recv(&N, 1, MPI_INT, 0, my_rank, MPI_COMM_WORLD,
&status) ;
            for ( i = my_rank ; i<N ; i = i + group_size )
                    sum = sum + foo(i) ;
S4:                 MPI_Send(&sum, 1, MPI_INT, 0, my_rank,
MPI_COMM_WORLD) ;
   }
   MPI_Finalize() ;
}
```

MPI Messages

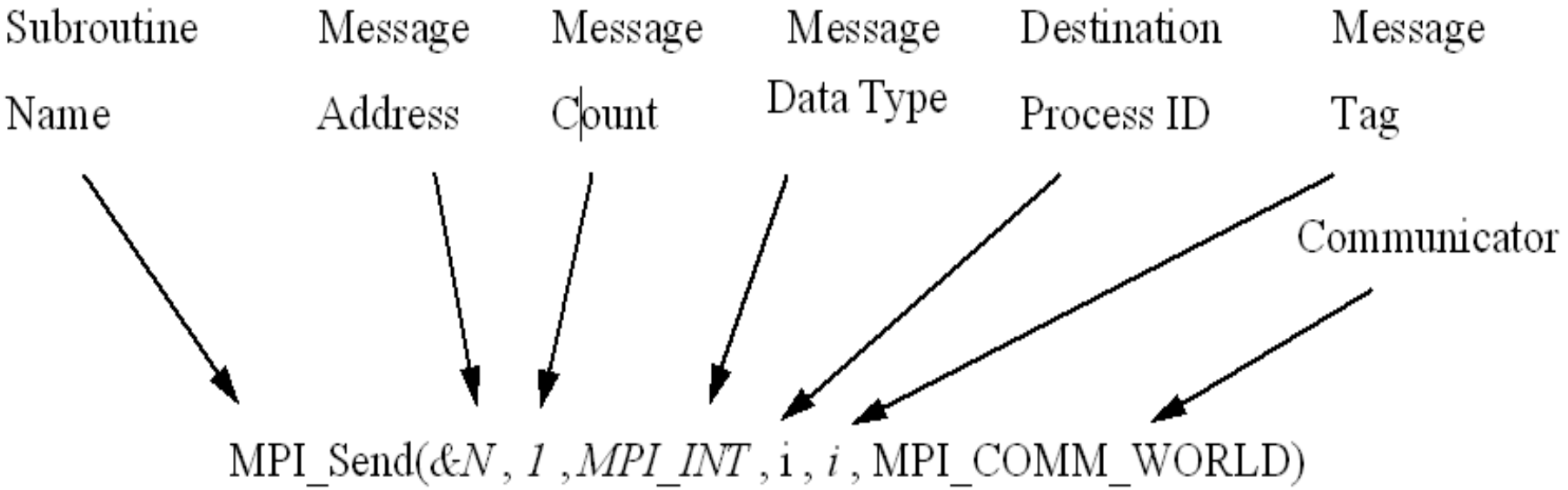
- **The subroutines MPI_Send and MPI_Recv are point-to-point communication operations, which pass a message between a pair of processes. These subroutines**

```
MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD) ;
MPI_Recv(&n, 1, MPI_INT, 0, my_rank, MPI_COMM_WORLD, &status) ;
```

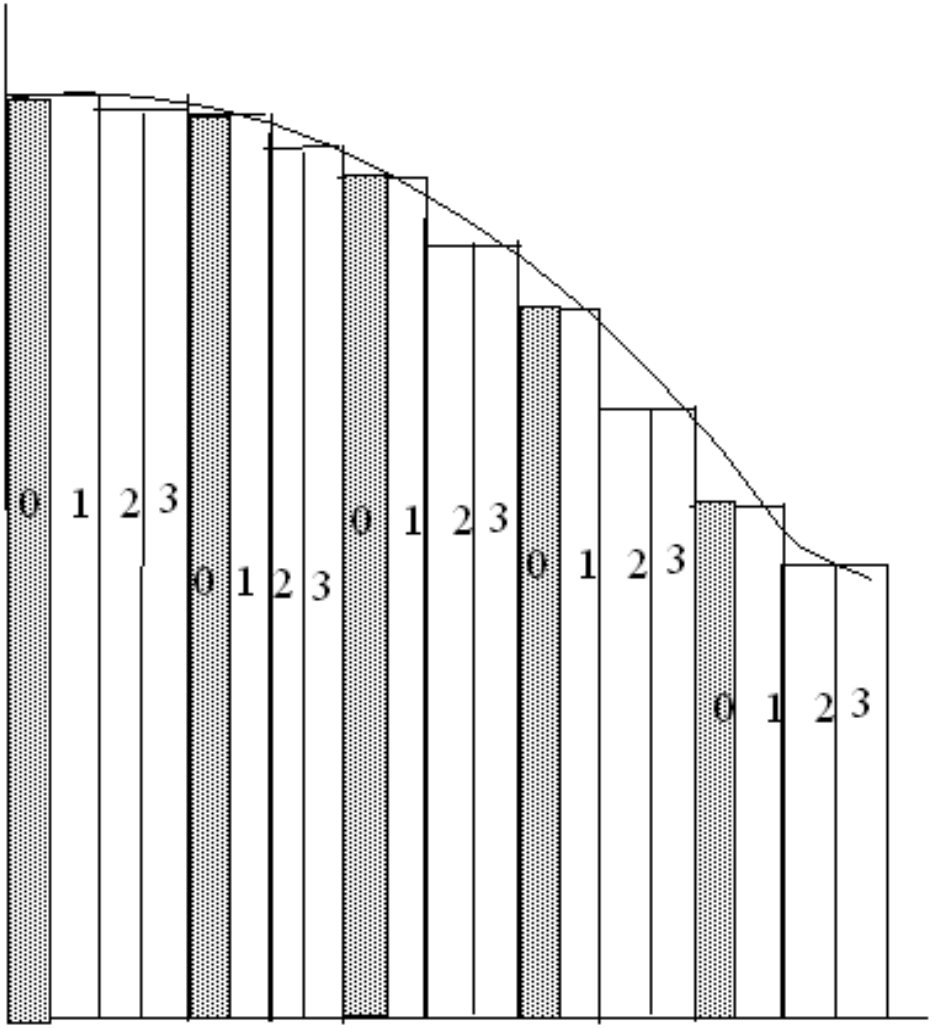
- **As an analogy, a message is like a letter. We need to specify the contents of the message (the letter itself), and the intended recipient of the message (what is on the envelope). The former is often called a *message buffer* and the latter a *message envelope*.**

Send in MPI

- **The address field does not have to be the starting address of a data structure.**
- **It could be any memory address in the application's address space.**



PI Calculation in Parallel



- **Divide intervals in a cyclic style to all the processors (4 in this case).**

PI Calculation with MPI

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
int done = 0, n, myid, numprocs, i;
double PI25DT = 3.14159265358979323846264 3;
double mypi, pi, width, sum, x;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

while (!done)
{
   if (myid == 0)
   {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
   }

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;

width = 1.0 / (double) n;
sum = 0.0;
```

```
for (i = myid + 1; i <= n; i += numprocs)
{
x = width * ((double)i - 0.5);
sum += 4.0 / (1.0 + x*x);
}
mypi = width* sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);

if (myid == 0)
printf("pi is approximately %.16f, Error is %.16f\n",pi, fabs(pi - PI25DT));
}

MPI_Finalize();
return 0;
}
```

Predefined Data Types in MPI

MPI (C Binding)	C	MPI (Fortran Binding)	Fortran
MPI_BYTE		MPI_BYTE	
MPI_CHAR	Signed char	MPI_CHARACTER	CHARACTER(1)
		MPI_COMPLEX	COMPLEX
MPI_DOUBLE	Double	MPI_DOUBLE_ PRECISION	DOUBLE_ PRECISION
MPI_FLOAT	Float	MPI_REAL	REAL
MPI_INT	Int	MPI_INTEGER	INTEGER
		MPI_LOGICAL	LOGICAL
MPI_LONG	Long		
MPI_LONG_DOUBLE	Long double		
MPI_PACKED		MPI_PACKED	
MPI_SHORT	Short		
MPI_UNSIGNED _CHAR	Unsigned char		
MPI_UNSIGNED	Unsignedint		
MPI_UNSIGNED _LONG	Unsigned long		
MPI_UNSIGNED _SHORT	Unsigned short		

Send non-contiguous data items

```
double A[100] ;

MPI_Pack_size( 50, MPI_DOUBLE, comm, &BufferSize ) ;

TempBuffer = malloc( BufferSize ) ;

j = sizeof( MPI_DOUBLE ) ;

Position = 0 ;

for ( i = 0 ; i <50 ; i++ )
   MPI_Pack (A+i*j, 1, MPI_DOUBLE, TempBuffer, BufferSize,
   &Position, comm) ;

MPI_Send ( TempBuffer, Position, MPI_PACKED, destination, tag, comm );
```

Sending mixed data types in message passing

- **Assume each double-precision number is 8 bytes long, a character is 1 byte, and an integer is 4 bytes.**

➔ All elements of a double-precision, 100-element array A. This message consists of 100 items. Each item has a double data type, which also determines the size of each item (8 bytes). The (starting) address of the *i*th item is A+8(*i*–1).

➔ The third and the fourth elements of array A. This message consists of two items A[2] and A[3]. Each item has a double data type. The first item starts at A+16, and the second at A+24.

➔ The even-indexed elements of array A. This message consists of 50 items A[0], A[2], A[4], ..., A[98]. Each item has a double data type. The address of the *i*th item is A+16(*i*–1).

➔ The third element of array A, followed by a character *c*, followed by an integer *k*. This message consists of three items with different data types.

Suppose they are part of a structure:

struct { double A[100]; char b, c; int j, k; } S ;

Then the address is S+16 for the first item (A[2]), S+801 for the second (*c*), and S+806 for the third element (*k*).

- **Messages in (1) and (2) have two properties: The data items are stored consecutively, and all data items have the same data types. The message in case (1) are specified by (A, 100, MPI_DOUBLE), and that in (2) by (A+16, 2, MPI_DOUBLE).**
- **The data items in (3) do not reside at contiguous locations.**
- **The data items in (4) are noncontiguous and have mixed data types.**

Message Buffer

The term *message buffer* (or simply, *buffer*) has been used with different meanings by the message passing community.

- **A buffer refers to an application memory area specified by the programmer, where the data values of a message are stored. For instance, in send(A, 16, Q, tag), the buffer A is a variable declared in the user application. The starting address of the buffer is used in a message-passing routine.**

- **A buffer could also mean some memory area created and managed by the message-passing system (not the user), which temporarily stores a message while it is being sent. Such a buffer does not appear in the user's application program and is sometimes called a *system message buffer* (or system buffer).**

- **MPI allows a third possibility. The user may set aside a memory area of a certain size, to be used as an intermediate buffer to hold arbitrary messages that could appear in her application.**

Sending messages between a pair of processes

Consider the following code to pass a message M stored in array A of process P to array B of process Q:

Process P:

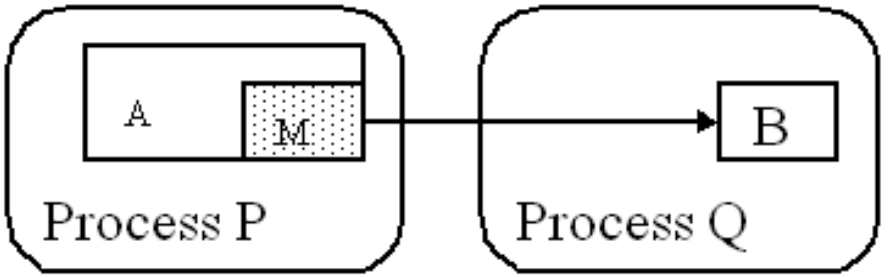
```
double A[2000000] ;
send (A, 32, Q, tag);
```

Process Q:

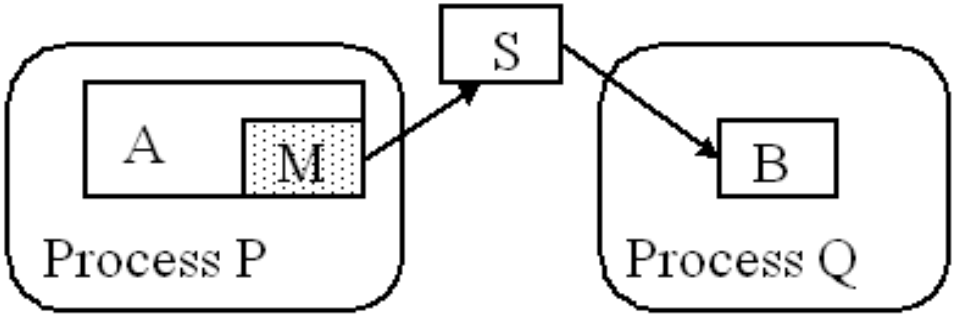
```
double B[32] ;
recv(B, 32, P, tag);
```

Three types of buffers

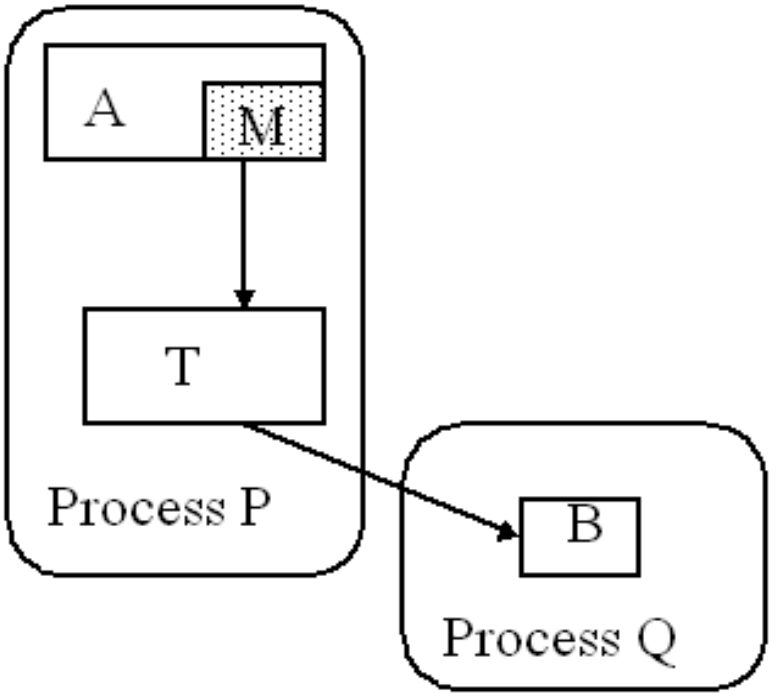
M indicates the 32-byte message, consisting of the first four elements of array A. In case (a), the message is directly transferred between user-level buffers A and B, both of which are declared in the user application. The receive buffer must be large enough.



(a) User buffers only



(b) A system buffer S is used



(c) A user-level temporary buffer T is used

- **An error would occur if send (A, 64, Q, tag) were executed by P.**
- **In case (b), the message is first temporarily copied into a dynamically created system buffer S, before being deposited in the receive user buffer B. There are two problems.**
  - An extra copying introduces additional overhead. A
  - system buffer with sufficient size is not guaranteed. The buffer needed may be too big to fit in the physical memory, resulting in excessive paging. The buffer may even be too big to fit into the virtual memory, causing the program to abort or hang.
- **Case (c) is another alternative used in MPI. The user first declares a temporary buffer T which is large enough to hold any message that needs to be buffered. During a message-passing operation, the message is first copied into the buffer T, before being deposited in the receive user buffer B. If the system cannot accommodate buffer T, it will generate an error message and terminate. Otherwise, the application is guaranteed to have enough buffer space.**

Message Envelope in MPI

How does one specify the intended recipient of a message? Listed below is an MPI send routine where the message envelope consists of three entities, shown in italic. The *destination* field is an integer identifying the process to which the message is sent.

MPI_Send(address, count, datatype, ***destination, tag, communicator*****)**

A *message tag*, also known as a *message type*, is an integer used by the programmer to label different types of message and to restrict message reception.

Example

- Consider the following code where tags are absent:

Process P:
send(A,32,Q)
send(B,16,Q)

Process Q:
recv(X, P, 32)
recv(Y, P, 16)

- What if message B, although sent later, arrives at process Q earlier, thus being received into X by the first **recv**()? This error can be avoided by using tags:

Process P:
send(A,32,Q, tag1)
send(B,16,Q, tag2)

Process Q:
recv(X, P, tag1, 32)
recv(Y, P, tag2, 16)

- Now message A (with tag1) is guaranteed to be received first. If message B arrives at Q first, it will be queued (buffered) until **recv**(Y, P, tag2, 16) is executed.

Tags used in message passing

Another reason for tags is exemplified by the following scenario. Suppose two client processes P and R each send a service request message to a server process Q:

Process P:
send(request1,32,Q, tag1)

Process R:
send(request2,32,Q, tag2)

It is unknown which send will be executed first. The server process Q is required to process the clients' requests in a first-come first-served order:

Process Q:
while (true) {
recv(received_request, Any_Proccess, 32) ;
process received_request ;
}

This code is not very flexible, since all requests are processed the same way. By using tags, different messages can be processed differently:

Process P:
send(request1,32,Q, tag1)

send(request3,32,Q, tag3)

Process R:
send(request2,32,Q, tag2)

Process Q:
while (true) {
recv(received_request, 32, Any_Proccess, Any_Tag, Status)
if (Status.Tag==tag1) ***process received_request in one way;***
if (Status.Tag==tag2) ***process received_request in another way;***
}

The recv() statement says that it will receive a message of 32 bytes with any tag (Any_Tag is known as a *wild-card tag*) from any process (Any_Process is known as a *wild-card process ID*) into received_request.

The actual tag of the message received is stored in the Tag field of Status to guide the branching in program.

What Is a Communicator?

- A *communicator* is a *process group* plus a *context*.
- A process group is a *finite* and *ordered* set of processes.
- The finiteness implies that a group has a finite number *n* of processes, where *n* is called the group size.
- The ordering means that the *n* processes are ranked by integers 0, 1, ..., *n*–1.
- A process is identified by its rank in a communicator (group). The group size and the rank of a process are obtained by calling the two MPI routines:

MPI_Comm_size(communicator, &group_size)

MPI_Comm_rank (communicator, &my_rank)

- All MPI communication routines have a communicator argument.
- *Contexts* in MPI are like system-designated supertags that safely separates different communications from adversely interfering with one another.
- Each communicator has a distinct context.
- A message sent in one context cannot be received in another context.
- To see why the concept of context is needed, let us look at the following example.

The use of communicator

- Consider the following code fragment:

Process 0:
MPI_Send (msg1, count1, MPI_INT, 1, tag1, comm1) ;
parallel_fft (...) ;

Process 1:
MPI_Recv (msg1, count1, MPI_INT, 0, tag1, comm1) ;
parallel_fft (...) ;

- The intent is that process 0 will send msg1 to process 1, and then both execute a subroutine parallel_fft().

- Now suppose parallel_fft() contains another send routine:

if (my_rank==0) MPI_Send (msg2, count1, MPI_INT,1,tag2, comm2);

If there were no communicators, the MPI_Recv in process 1 could mistakenly receive the msg2 sent by the MPI_Send in parallel_fft(), when tag2 happens to have the same integer value as tag1.

So why cannot we use different tag values? It is impossible to guarantee that tag1 and tag2 are distinct for the following three reasons:

- **Tags are integer values specified by users, and users make mistakes.**
- **Even if a user does not make mistakes, it is difficult or impossible to ensure that the value of tag1 is different from that of tag2. The function parallel_fft() could be written by another user, or it could be a library routine. So the user cannot know the value of tag2.**
- **Even if the user could always know the value of tag2, errors could still occur, because the MPI_Recv routine may decide to use a wildcard tag MPI_Any_tag.**

MPI solves these problems by using communicators.

- Communications in parallel_fft use different communicators, which may contain the same group of processes (e.g., processes 0 and 1), but each communicator will have a distinct, system-assigned context that is different from that of comm1. Therefore, there is no danger that MPI_Recv accidentally receives msg2 from the MPI_Send in parallel_fft.

- MPI is designed so that communications within different communicators are separated and any collective communication is separate from any point-to-point communication, even if they are within the same communicator. This communicator concept facilitates the development of parallel libraries, among other things.

Nonblocking Communication

There are four nonblocking send routines and one nonblocking receive routine in MPI. They are used to *initiate* (start) a send or receive (hence the letter I in their names). MPI provides other routines to check the completion of a send or receive. We discuss only two basic ones through examples.

Nonblocking Communication

- The basis of the nonblocking communication system are routines `MPI_Isend` and `MPI_Irecv`

MPI_Isend(buffer, count, datatype, destination, tag, comm, request);

MPI_Irecv(buffer, count, datatype, source, tag, comm, request);

- The "I" stands for "immediate"
- `MPI_Wait` blocks until the operation identified by `request` completes
- `MPI_Test` checks for completion of the communication operation associated with `request`

An Example on Nonblocking Communication

```
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
int rank, size;
char msg[20];
int i;
MPI_Request request;
int flag;
MPI_Status status;
MPI_Init(&argc, &argv);                    /* Starts MPI */
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Processor 0 send msg to all processor */
```

```
/* with nonblocking send; note that      */
/* there are 2-second pauses between     */
/* 2 consecutive send operations         */
if (rank == 0) {
strcpy(msg, “Hello World!”);
for (i=0; i<size; i++) {
sleep(2);
MPI_Isend(msg, 20, MPI_CHAR, i, 0, MPI_COMM_WORLD, &request);
}
}
```

An Example on Nonblocking Communication (cont'd)

```
/* Receive the msg with nonblocking receive */
MPI_Irecv(msg, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD,  &request);
/* Loop to check whether receive is completed */
for (i=0; ;i++) {
MPI_Test(&request, &flag, &status);
if (flag != 0)      /* receive completed */
break;
}
/* Show the msg and number of loops */
MPI_Barrier();
printf("[Proc %d] msg=\"%s\", loop=%d\n", rank, msg, i);
MPI_Finalize();                  /* Exits MPI */
return 0;
}
```

Output

[Proc 1] msg=”Hello World!”, loop=11742

[Proc 2] msg=”Hello World!”, loop=40678

[Proc 3] msg=”Hello World!”, loop=47603

[Proc 0] msg=”Hello World!”, loop=0

[Proc 4] msg=”Hello World!”, loop=62152

Collective MPI Communications

Type	Routine	Functionality
Data movement	MPI_Bcast	One-to-all, identical message
	MPI_Gather	All-to-one, personalized messages
	MPI_Gatherv	A generalization of MPI_Gather
	MPI_Allgather	A generalization of MPI_Gather
	MPI_Allgatherv	A generalization of MPI_Allgather
	MPI_Scatter	One-to-all, personalized massages
	MPI_Scatterv	A generalization of MPI_Scatter
	MPI_Alltoall	All-to-all, personalized messages
	MPI_Alltoallv	A generalization of MPI_Alltoall
Aggregation	MPI_Reduce	All-to-one reduction
	MPI_Allreduce	A generalization of MPI_Reduce
	MPI_Reduce_scatter	A generalization of MPI_Reduce
	MPI_Scan	All-to-all parallel prefix
Synchroni-zation	MPI_Barrier	Barrier synchronization

Broadcast

In the following *broadcast* operation, process ranked Root sends the same message to all processes (including itself) in the communicator labeled Comm:

MPI_Bcast (Address, Count, Datatype, Root, Comm)

The content of the message is identified by the triple (Address, Count, Datatype), just as in a point-to-point communication.

For the Root process, this triple specifies both the send buffer and the receive buffer.

For other processes, this triple specifies the receive buffer.

Gather

**MPI_Gather (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Root, Comm)**

The Root process receives a personalized message from each of the *n* processes (including itself).

These *n* messages are concatenated in rank order, and stored in the receive buffer of the Root process.

Each send buffer is identified by the triple (SendAddress, SendCount, SendDatatype).

The receive buffer is ignored for all non-Root processes. For the Root process, it is identified by the triple (RecvAddress, RecvCount, RecvDatatype).

Scatter

**MPI_Scatter (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Root, Comm)**

A scatter performs just the opposite operation of a gather.

The Root process sends out a personalized message to each of the *n* processes, itself included.

These *n* messages are originally stored in rank order in the send buffer of the Root process.

Each receive buffer is identified by the triple (RecvAddress, RecvCount, RecvDatatype).

The send buffer is ignored for all non-Root processes. For the Root process, it is identified by the triple (SendAddress, SendCount, SendDatatype).

Total Exchange

In an all-to-all, total exchange routine:

MPI_Alltoall (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Comm)

Every process sends a personalized message to each of the *n* processes, including itself.

These *n* messages are originally stored in rank order in its send buffer. Looking at the communication from another way, every process receives a message from each of the *n* processes.

These *n* messages are concatenated in rank order, and stored in the receive buffer.

A total exchange is equivalent to *n* gathers, each by a different process. Therefore, the Root argument is not needed any more. All in all, n^2 messages are communicated in a total exchange.

Aggregation (Reduction)

An MPI reduction routine has the following syntax:

MPI_Reduce(SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)

Here each process holds a partial value stored in SendAddress.

All processes reduce these partial values into a final result and store it in RecvAddress of the Root process.

The reduction operator is specified by the Op field.

Aggregation (Scan)

The scan operation has very similar syntax to reduction:

MPI_Scan (SendAddress, RecvAddress, Count, Datatype, Op, Comm)

The Root field is absent, since a scan combines partial values into *n* final results and stores them in RecvAddress of the *n* processes.

The scan operator is specified by the Op field.

The MPI reduction and scan routines allow each process to contribute a vector, not just a scalar value.

The length of the vector is specified in Count.

MPI supports user-defined reduction or scan operations.

Barrier

In a *barrier operation* MPI_Barrier(), all processes in the communicator Comm synchronize with one another; i.e., they wait until all processes execute their respective MPI_Barrier function.

Topologies

MPI provides routines to provide structure to collections of processes

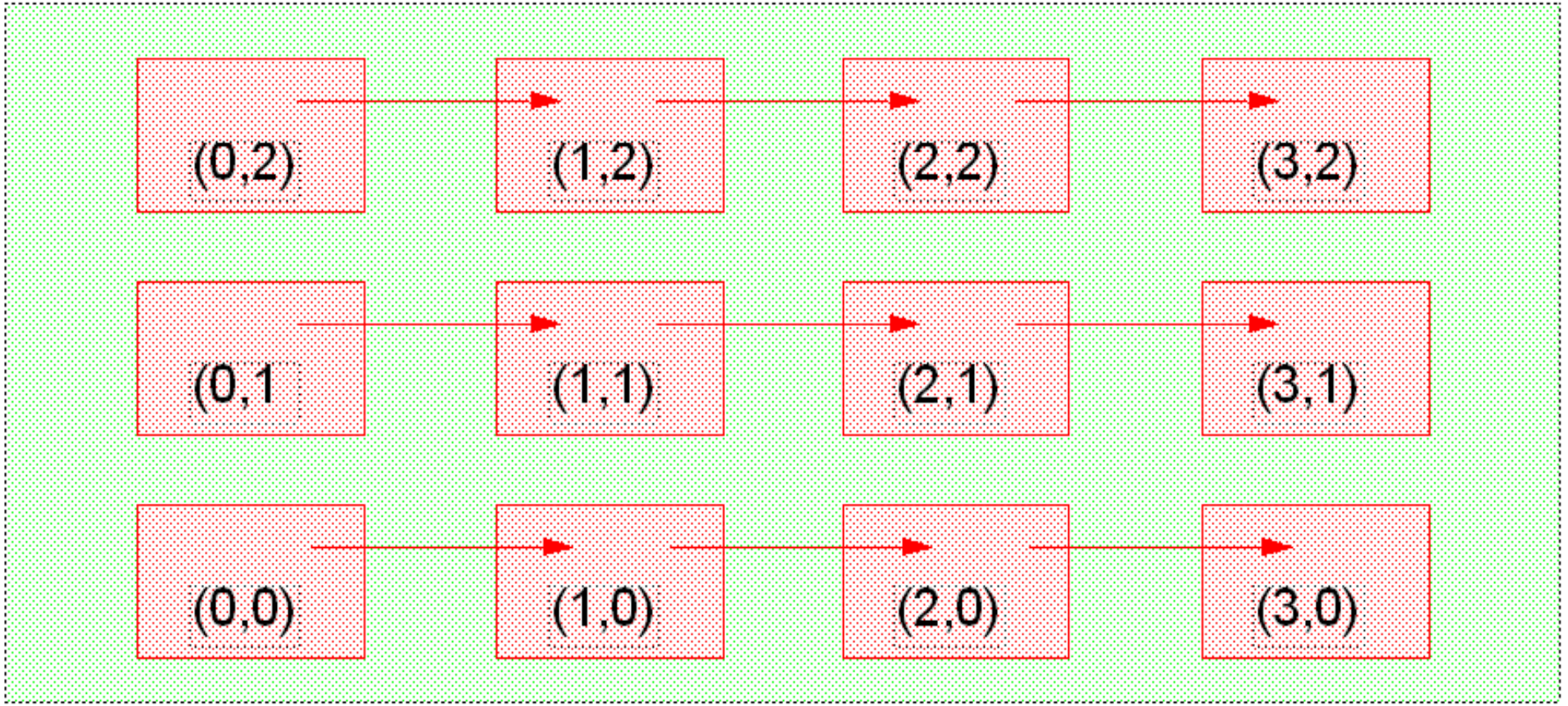
This helps to answer the question:

Who are the neighbours?

Cartesian Topologies

A Cartesian topology is a mesh

Example of 3 x 4 Cartesian mesh with arrows pointing at the *right neighbours*:



Defining a Cartesian Topology

The routine MPI_Cart_create creates a Cartesian decomposition of the processes, with the number of dimensions given by the ndim argument.

dims (1) = 4

dims (2) = 3

periods (1) = .false.

periods (2) = .false.

reorder = .true.

ndim = 2

call MPI_CART_CREATE(MPI_COMM_WORLD, ndim, dims, periods, reorder, comm2d, ierr)

Finding Neighbours

`MPI_Cart_create` creates a new communicator with the same process as the input communicator but with the specified topology

The question, Who are my neighbours, can now be answered with `MPI_Cart_shift`:

call MPI_Cart_shift (comm2d, 0, 1, nbrleft, nbrright, ierr)

call MPI_Cart_shift (comm2d, 1, 1, nbrbottom, nbrtop, ierr)

The values returned are ranks, in the communicator `comm2d`, of the neighbours shifted by +_1 in the two dimensions.

Who am I?

Can be answered with

```
integer coords (2)
call MPI_COMM_RANK( comm1d, myrank, ierr )
call MPI_CART_COORDS( comm1d, myrank, 2, coords, ierr )
```

Returns the Cartesian coordinates of the calling process in coords.

Partitioning

When creating a Cartesian topology, one question is “what is a good choice for the decomposition of the processors?”

This question can be answered with `MPI_Dims_create:`

```
integer dims(2)
dims (1) = 0
dims (2) = 0
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr )
call MPI_DIMS_CREATE ( size, 2, dims, ierr )
```

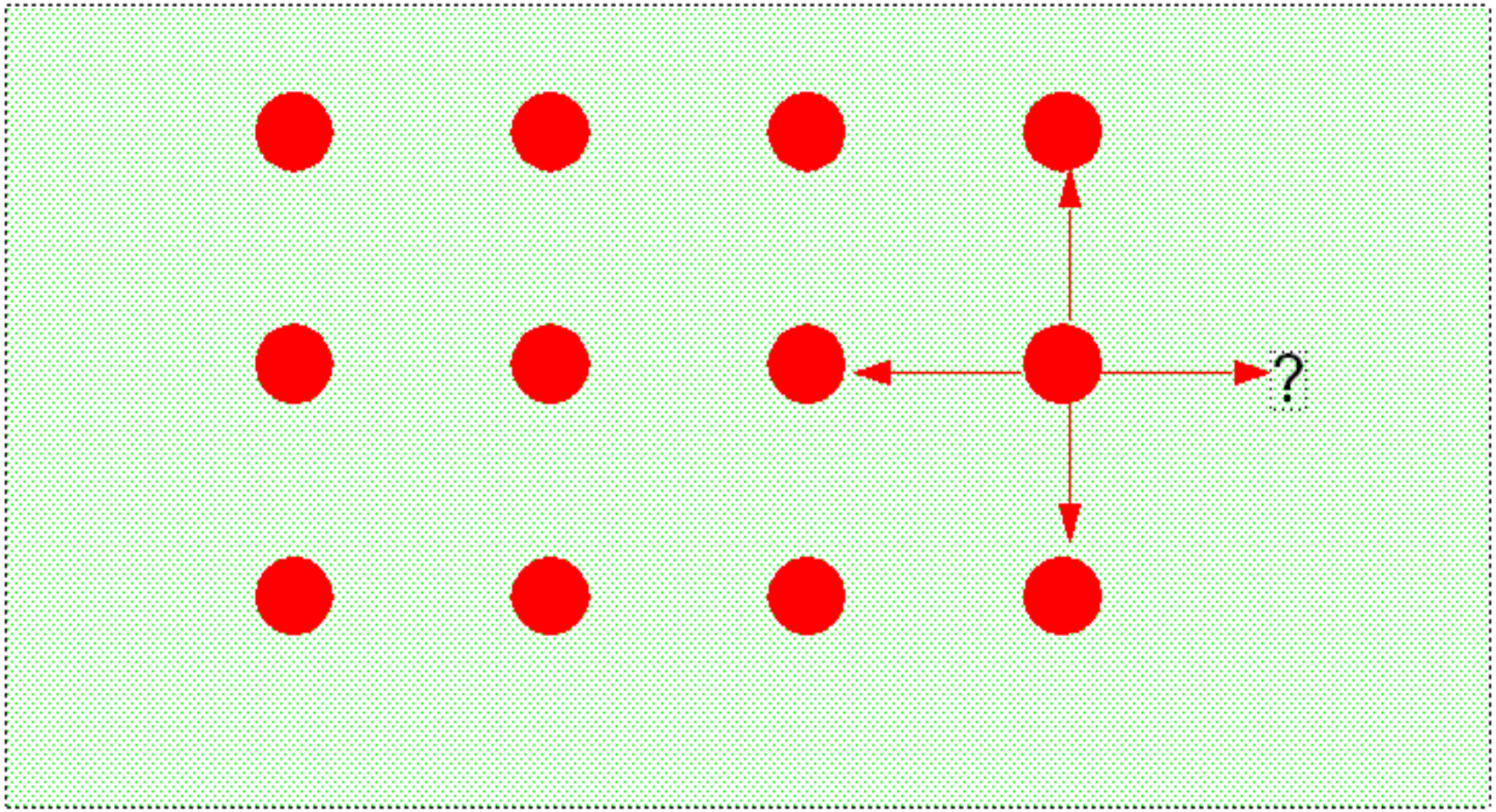
Other Topology Routines

MPI_contains routines to translate between Cartesian coordinates and ranks in a communicator, and to access the properties of a Cartesian topology.

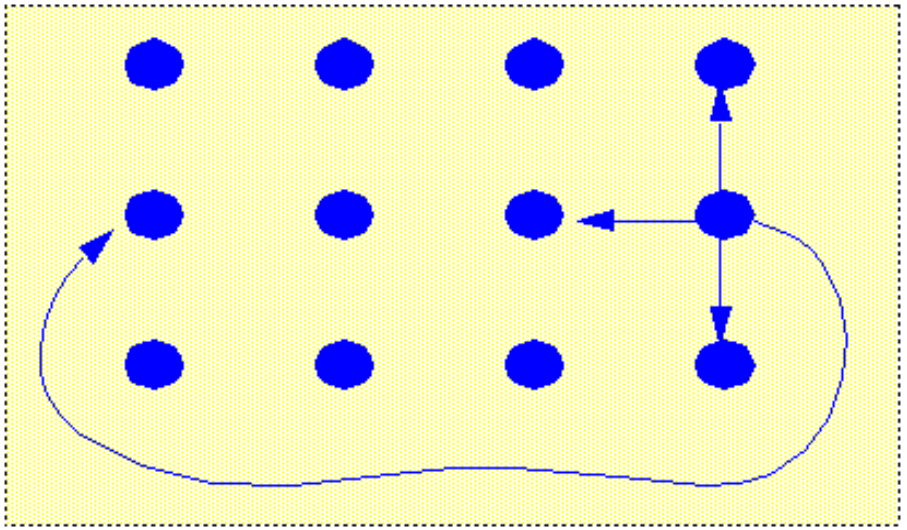
The routine MPI_Graph_create allows the creation of a general graph topology.

The Periods Argument

Who are my neighbors if I am at the edge of Cartesian Mesh?



Periodic Grids



Specify this in MPI_Cart_create with

dims (1) = 4

dims (2) = 3

periods (1) = .false.

periods (2) = .false.

reorder = .true.

ndim = 2

call MPI_CART_CREATE(MPI_COMM_WORLD, ndim, dims, periods, reorder, comm2d, ierr)

Nonperiodic Grids

In the nonperiodic case, a neighbor may not exist. This is indicated by a rank of `MPI_PROC_NULL`.

This rank may be used in send and receive calls in MPI. The action in both cases is as if the call was not made.

Built-in Collective Computation Operations

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical of
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Maximum value and location

Defining Your Own Collective Operations

MPI_Op_create (user_function, commute, op)
MPI_Op_free(op)

user_function(invec, inoutvec, len, datatype)

The user function should perform

inoutvec [i] = invec [i] op inoutvec [i];
for `i` from 0 to `len-1`.

`user_function can be non-communicative (e.g. matrix multiply)`

Sample User Function

For example to create an operation that has the same effect as `MPI_SUM` of Fortran double precision values, use

```
subroutine my func( invec, inoutvec, len, datatype)
integer len, datatype,
      double precision invec(len), inoutvec(len)
         integer i
         do 10 i = 1, len
               10 inoutvec(i) = invec(i) + inoutvec(i)
         return
end
```

To use just

integer myop

call MPI_Op_create(myfunc, .true., myop, ierr)

call MPI_Reduce(a, b, 1, MPI_DOUBLE_PRECISON, MYOP,.....

The routine `MPI_Op_free` destroys user-functions when they are no longer needed.