MPI

- Introduction to MPI
  - What it is
  - Where it came from
- MPI overview
- Point-to-point communication

Large-Scale Scientific Computing

- Goal: delivering computing performance to applications
- Deliverable computing power (in flops)
  - Current leader is the Roadrunner
  - 13000 computing processors, over 1 petaflop
- Parallelism taken for granted
  - Fortunately, physics appears to be parallel

Parallel Programming Research

- Independent research projects contribute new ideas to programming models, languages, and libraries
  - Most make a prototype available and encourage use by others
  - Users require commitment, support, portability
  - Not all research groups can provide this
- Failure to achieve critical mass of users can limit impact of research
  - PVM (and few others) succeeded

Standardization

- Parallel computing community has resorted to “community-based” standards
  - HPF
  - MPI
  - OpenMP
- Some commercial products are becoming “de facto” standards, but *only because they are portable*
  - TotalView parallel debugger, PBS batch scheduler

Standardization Benefits

- Multiple implementations promote competition
- Vendors get clear direction on where to devote effort
- Users get portability for applications
- Wide use consolidates the research that is incorporated into the standard
- Prepares community for next round of research
- Rediscovery is discouraged

Risks of Standardization

- Failure to involve all stakeholders can result in standard being ignored
  - Application programmers
  - Researchers
  - Vendors
- Premature standardization can limit production of new ideas by shutting off support for further research projects in the area

Models for Parallel Computation

- Shared memory (load, store, lock, unlock)
- Message Passing (send, receive, broadcast, ...)
- Transparent (compiler works magic)
- Directive-based (compiler needs help)

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space
- Message passing is for communication among processes, which have separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's

What is MPI?

- A message-passing library specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers

Where Did MPI Come From?

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - Portability library writers: PVM, p4
  - Users: application scientists and library writers
  - MPI-1 standard finished in 18 months
  - MPI-2 standard finished in 1996

Novel Features of MPI

- *Communicators* encapsulate communication spaces for library safety
- *Datatypes* reduce copying costs and permit heterogeneity
- Multiple *communication modes* allow precise buffer management
- Extensive *collective operations* for scalable global communication
- *Process topologies* permit efficient process placement, user views of process layout
- *Profiling interface* encourages portable tools

Problem Decomposition

- Domain decomposition
  - Also known as data parallelism
  - Data are divided into pieces that are approximately the same size and then mapped to different processors.
  - Each processor works only on the portion of the data that is assigned to it.
  - The processes may need to communicate periodically in order to exchange data.
  - Domain decomposition provides the advantage of maintaining a single flow of control. A data-parallel algorithm consists of a sequence of elementary instructions applied to the data: an instruction is initiated only if the previous instruction is ended.

Problem Decomposition

- Functional Decomposition
  - Useful when, for example, the individual subsets of data assigned to the different processes require greatly different lengths of time to process.
  - Client-server paradigm.
  - The tasks are allocated to a group of slave processes by a master process that may also perform some of the tasks.

Load Balancing

- Load balancing divides the required work equally among all of the available processes.
- This ensures that one or more processes do not remain idle while the other processes are actively working on their assigned subproblems so that valuable computational resources are not wasted.
- Load balancing can be easy when the same operations are being performed by all the processes on different pieces of data.
- It is not trivial when the processing time depends upon the data values.

Execution Time

- A primary concern in parallel programming because it is an essential component for comparing and improving all programs. Three components make up execution time:
  - Computation time
  - Idle time
  - Communication time
    - Latency
    - Bandwidth

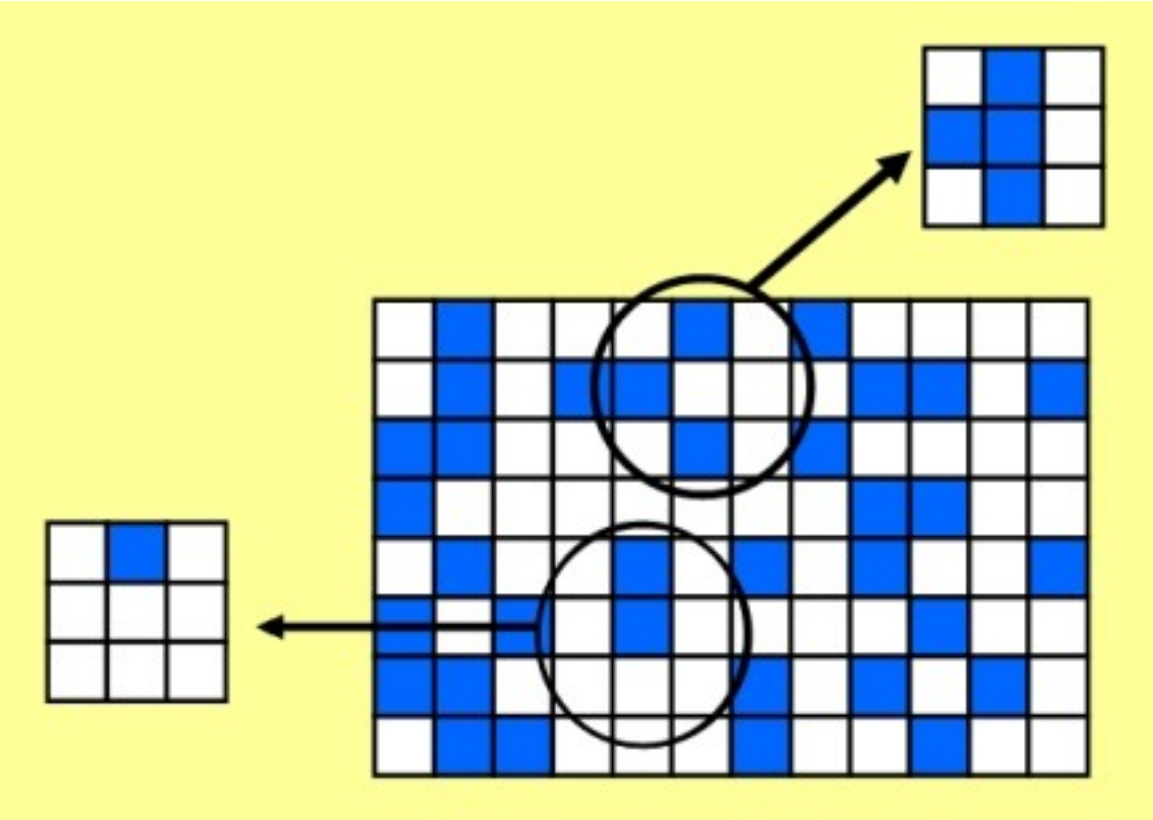
Process Idle Time

- It is important to minimize the time that processes remain idle to reduce the impact on execution time.
  - One strategy is to use overlapping communication and computation, which is termed latency hiding.
  - This method involves occupying a process with one or more new tasks while it waits for a communication event to complete so it can proceed to another task.
    - Non-blocking communication
    - Data-unspecific computation

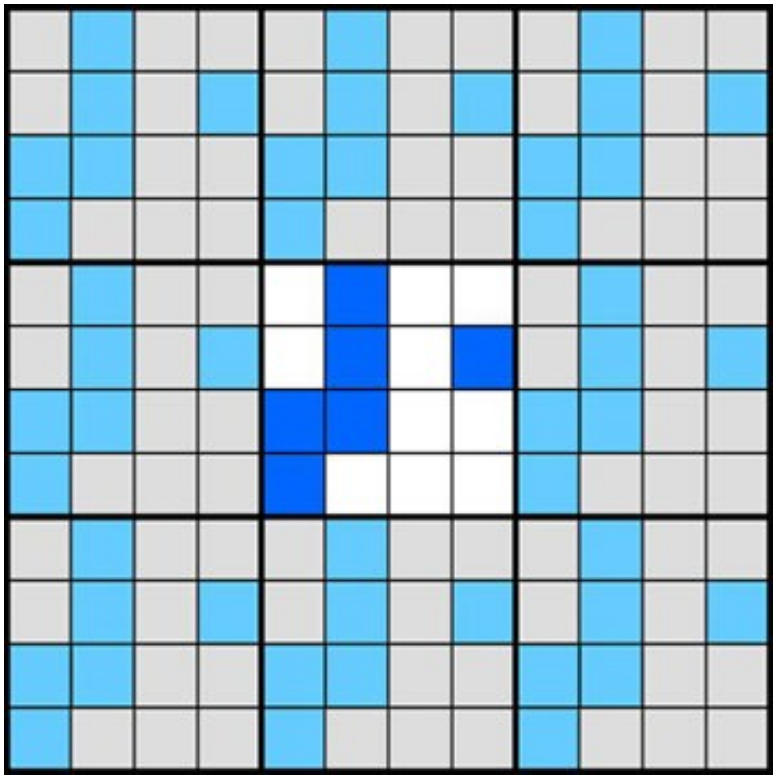
Game of Life

- A simple simulation developed by John Conway
- 2-dimensional array of cells
- Each cell can have one of two possible states, usually referred to as "alive" or "dead"
- At each time step, each cell may or may not change its state, based on the number of adjacent alive cells, including diagonals
  - If a cell has three neighbors that are alive, the cell will be alive. If it was already alive, it will remain so, and if it was dead, it will become alive.
  - If a cell has two neighbors that are alive, there is no change to the cell. If it was dead, it will remain dead, and if it was alive, it will remain alive.
  - In all other cases — the cell will be dead.

Game of Life



Cell birth and death examples



Periodic boundary conditions

MPI Overview

- A MPI program consists of two or more autonomous processes
  - Each executing their own codes
  - Code may or may not be identical on a given pair of processes.
- These processes communicate via calls to MPI communication routines
- They are identified according to their relative rank within a group (0, 1, . . . , groupsize-1).
- MPI does not allow for dynamic allocation of processes during the execution of a parallel program.
- You specify the number of processes at the start of your program and that number remains fixed throughout the entire program.

Point-to-Point Communications

- Direct communication between two processors, one of which sends data and the other receives this same data.
- In a generic send or receive, a message consisting of some block of data is transferred between processors.
  - An envelope indicates the source and destination processors
  - A body that contains the actual data to be sent
- MPI uses the following three pieces of information to characterize the message body in a flexible way:
  - Buffer - the starting location in memory where outgoing data is to be found or incoming data is to be stored
  - Datatype - the type of data to be sent.
    - Elementary type such as float (REAL), int (INTEGER)
    - A user-defined datatype built from the basic types.
  - Count - the number of items of type datatype to be sent.
- MPI deals with big/little endian issues etc.

Communication Modes and Completion Criteria

- A variety of communication modes define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event is complete.
  - A synchronous send is defined to be complete when receipt of the message at its destination has been acknowledged.
  - A buffered send, is complete when the outgoing data has been copied to a local buffer.
    - Nothing at all is implied about the arrival of the message at its destination.
  - In all cases, completion of a send implies that it is safe to overwrite the buffer where the data were originally stored.
- There are four communication modes available for sends:
  - Standard
  - Synchronous
  - Buffered
  - Ready

Blocking and Nonblocking Communication

- A blocking send or receive does not return from the subroutine call until the operation has actually completed. Thus it ensures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed.
- A nonblocking send or receive returns immediately with no information about whether the completion criteria have been satisfied.
  - This approach has the advantage that the processor is free to do something else while the communication proceeds in the background.
  - You can test later to see whether the communication has actually completed.
  - A nonblocking synchronous send returns immediately, although the send will not be complete until receipt of the message has been acknowledged.

Collective Communications

- A communicator is an MPI object that defines a group of processes that are permitted to communicate with one another.
  - Every MPI message must specify a communicator via a “name” that is included as an explicit parameter within the argument list of the MPI call.
  - By default, all processes are defined as being members of the communicator MPI_COMM_WORLD.
- Collective communication routines, also called collective operations, transmit data among all processes in a group.
  - These routines allow larger groups of processors to communicate in various ways, for example, one-to-several or several-to-one.
  - All collective communication events are blocking.

Collective Communications

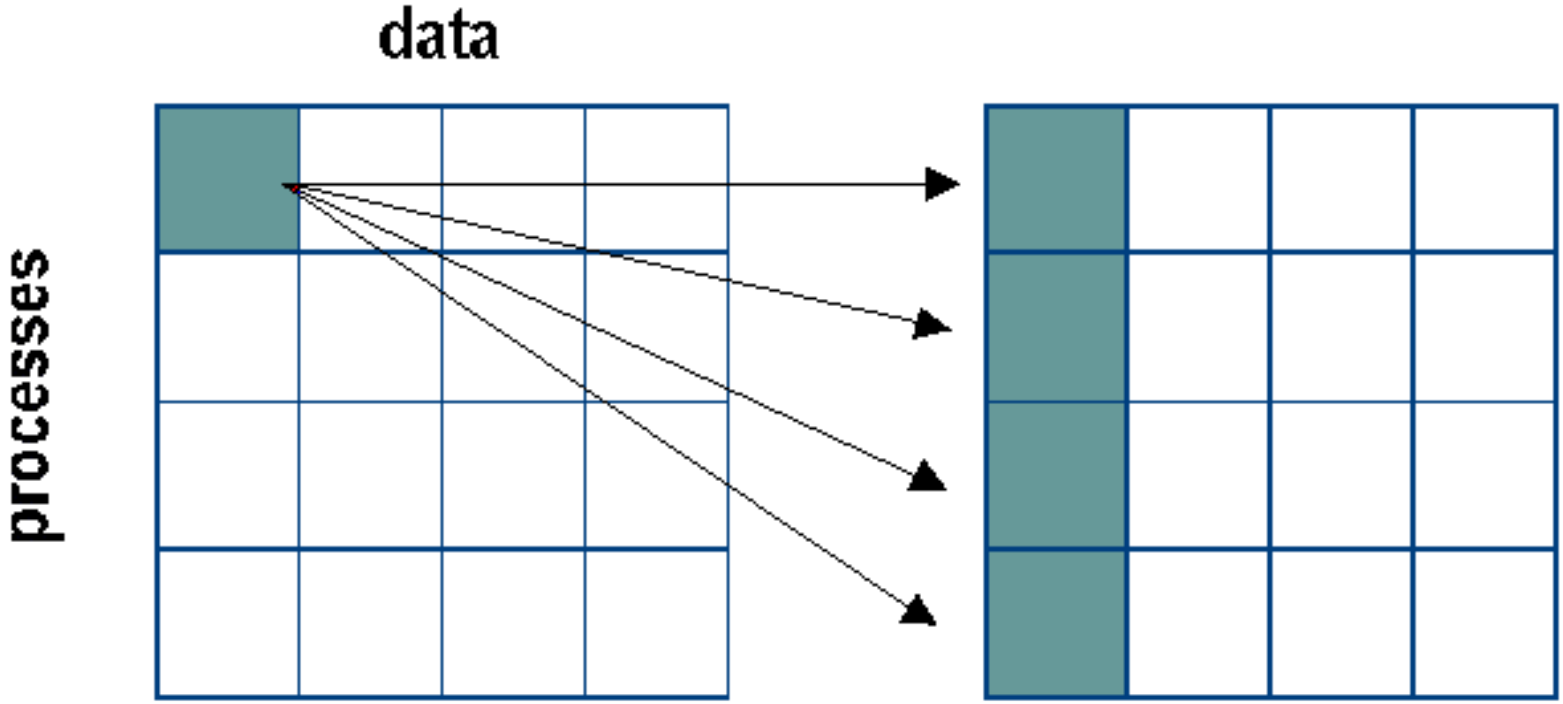
- There are three basic types of collective communication events in MPI.
  - Synchronization - each process waits until all processes included within its group have reached the specified synchronization point.
  - Data movement - data is transferred to all processes included within its group.
  - Collective computation — one process within a group collects data from other processes within that group and performs an operation (addition, multiplication, etc.) on that data.

Collective Communications

- The main advantages of using the collective communication routines over building the equivalent operation out of point-to-point communications are:
  - The possibility of error is significantly reduced. A single line of code - the call to the collective routine - typically replaces several point-to-point calls.
  - The source code is much more readable, thus simplifying code debugging and maintenance.
  - Optimized forms of the collective routines are often faster than the equivalent operation expressed in terms of point-to-point routines.

Broadcast Operations

- A single process sends a copy of some data to all the other processes in a group.
- Each row in the figure represents a different process. Each colored block in a column represents the location of a piece of the data. Blocks with the same color that are located on multiple processes contain copies of the same data.

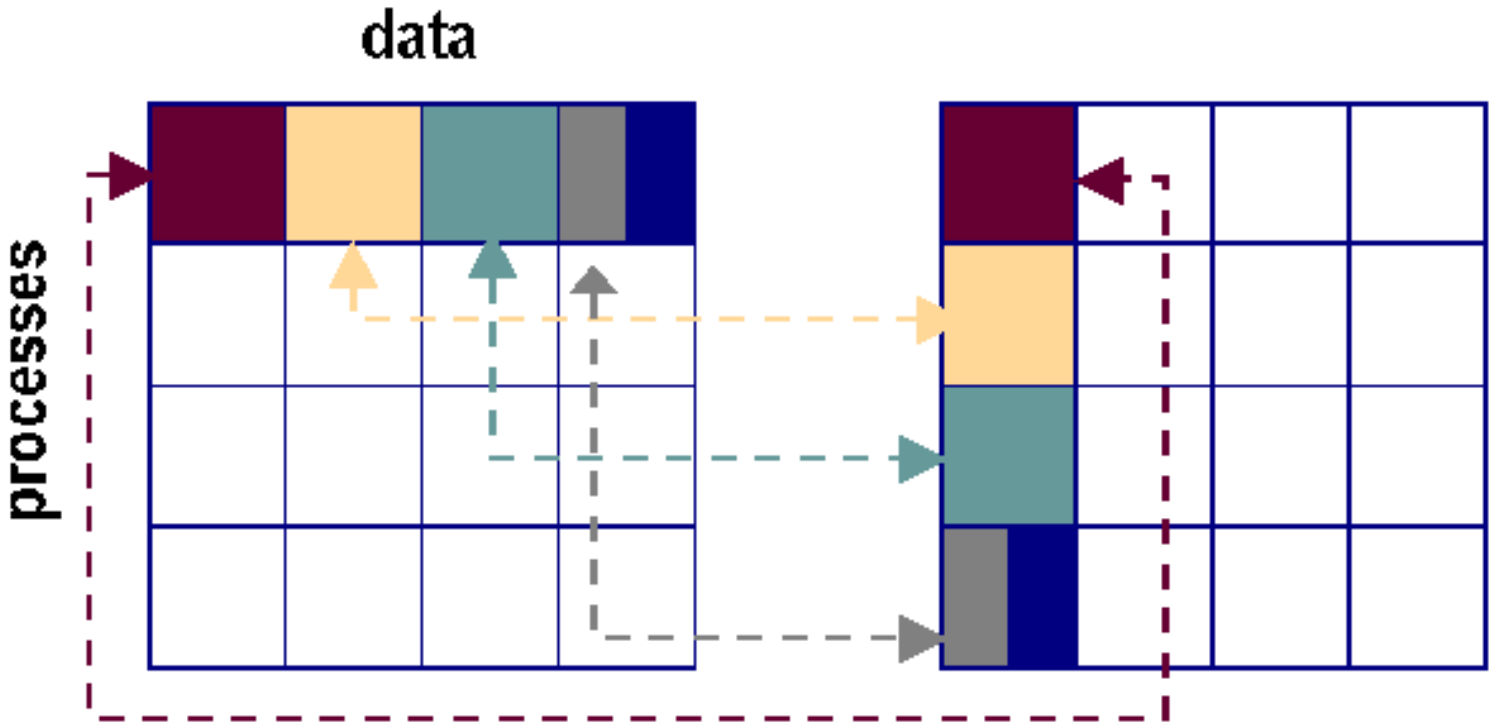


Scatter and Gather Operations

- MPI provides two kinds of scatter and gather operations, depending upon whether the data can be evenly distributed across processors.
  - In a scatter operation, all of the data (an array of some type) are initially collected on a single processor. After the scatter operation, pieces of the data are distributed on different processors
  - The gather operation is the inverse operation to scatter: it collects pieces of the data that are distributed across a group of processors and reassembles them in the proper order on a single processor.

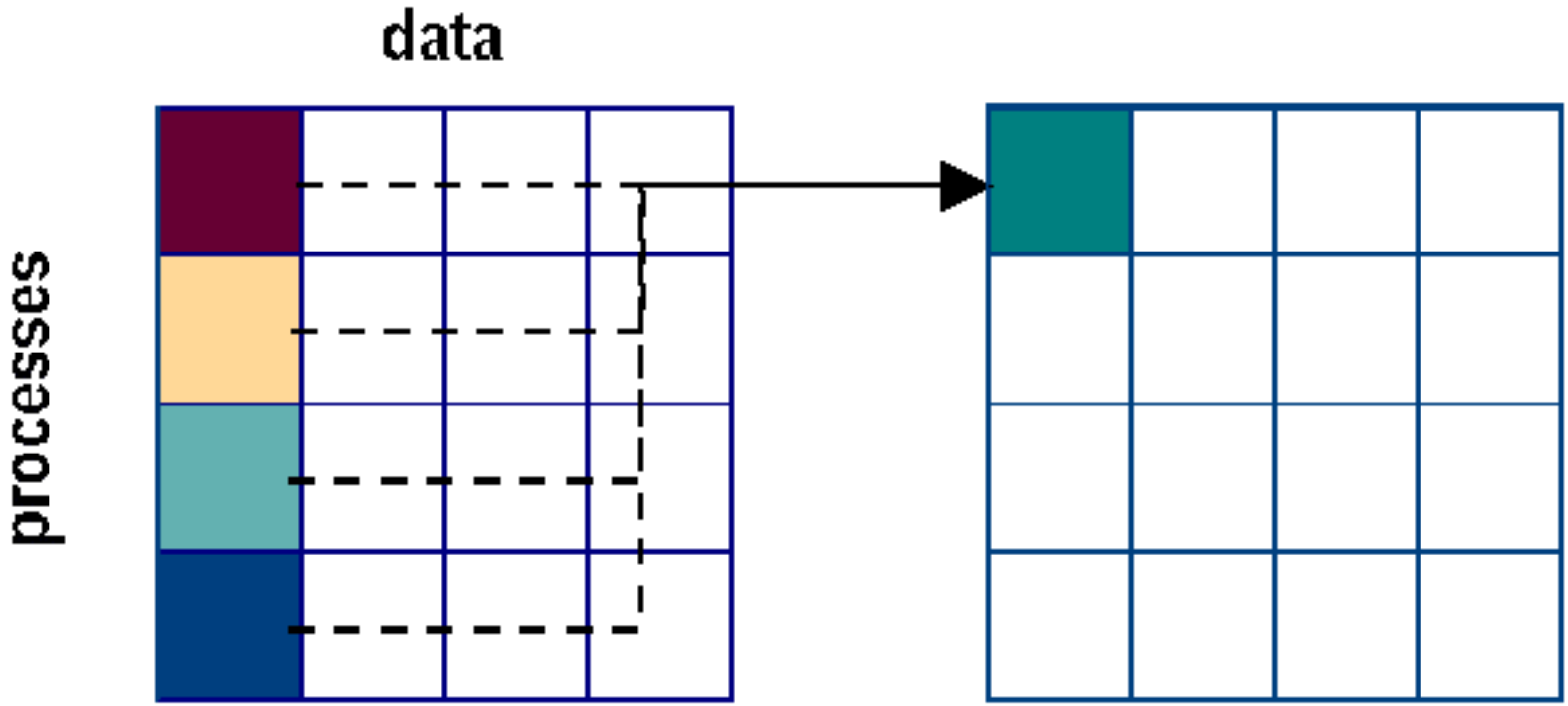
Scatter and Gather Operations

- The multicolored box reflects the possibility that the data may not be evenly divisible across the processors.



Reduction Operations

- Collective operations in which a single process (the root process) collects data from the other processes in a group and performs an operation on that data, which produces a single value.
  - You might use a reduction to compute the sum of the elements of an array that is distributed across several processors.



Process Groups

- A process group is simply an ordered set of processes where each process in a group is associated with a unique integer value referred to as the rank of the process.
  - A process rank is also referred to as the process "ID."
  - Rank values in MPI always start at zero and run sequentially through N-1, where N is the number of processes in the group.
- Although the number of processes specified in an MPI program remains fixed throughout the program, both groups and communicators can be dynamically created and eliminated during the execution of the program.
- A given process can be a member of more than one group or communicator and will have a unique rank within each group or communicator.

Process Topologies

- In MPI, a topology is a mechanism for associating different identification schemes with the processes belonging to a particular group.
  - A topology describes a mapping or ordering of MPI processes into some geometric shape.
  - MPI supports two main types of topologies
    - A Cartesian, or grid, topology
    - A graph topology.

Process Topologies

- All MPI topologies are virtual
  - There may be no simple relation between the process structure implicit in the MPI topology and the actual underlying physical arrangement of the processors within the computer itself.
- Virtual topologies are used in MPI to provide communication efficiency and for programming convenience.
  - A Cartesian or grid topology is likely to be convenient in an application involving nearest-neighbor communication between points on a rectangular grid.

Environment Management and Inquiry

- A number of MPI routines are available for managing and inquiring about the state of the environment.
- They are used for a number of purposes, such as:
  - Initializing and terminating the MPI execution environment
  - Terminating all processes belonging to a given MPI communicator
  - Determining the number of processes belonging to a given communicator
  - Determining the rank of the calling process within a given communicator

MPI Program Structure

- Header file
  - `#include <mpi.h>`
- Naming convention
  - `MPI_Xxxxx(parameter, ... )`
- Return values

```
int err;
err = MPI_Init(&argc, &argv);
if (err == MPI_SUCCESS) {
    ...routine ran correctly...
    }
```

- MPI Handles
  - MPI defines and maintains its own internal data structures.
  - These data structures are referenced through handles, which are returned by various MPI calls and may be used as arguments in other MPI calls.
  - In C, handles are pointers to specially defined datatypes that are created via the C typedef mechanism

MPI Datatypes

- MPI allows automatic translation between representations in a heterogeneous environment by providing its own reference datatypes corresponding to the various elementary datatypes in C and Fortran.
- Variables are normally declared as C/Fortran types and MPI type names are used as arguments in MPI routines when a type is needed.
- As a general rule, the MPI datatype given in a receive must match the MPI datatype specified in the send. MPI hides the details of the floating-point representation, which is an issue for the implementor so you will need to see the vendor's documentation for more detail.
- MPI allows for the definition of arbitrary datatypes, called derived datatypes, that are built from the basic types.

Basic MPI Datatypes

MPI Datatype	C Type
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	none
MPI_PACKED	none

Initializing MPI

- The initialization routine MPI_INIT must be the first MPI routine called in any MPI program.
- This routine establishes the MPI environment and will return an error code if there is a problem.
- MPI_INIT may be called only once in any program.

```
int err;
...
err = MPI_Init(&argc, &argv);
```

- The arguments to MPI_Init are the addresses of argc and argv, the variables that contain the command-line arguments for the program.

Communicators

- A communicator is a MPI handle that defines a group of processes that are permitted to communicate with one another.
  - Every MPI message must specify a communicator via a name that is included as an explicit parameter within the argument list of the MPI call.
  - The communicator specified in the send and receive calls must agree for communication to take place.

Communicators

- There can be many communicators, and a given processor can be a member of a number of different communicators.
  - Within each communicator, processors are numbered consecutively (starting at 0).
    - This identifying number is known as the rank of the processor in that communicator
    - The rank is also used to specify the source and destination in send and receive calls.
    - If a processor belongs to more than one communicator, its rank in each can (and usually will) be different.
- MPI automatically provides a basic communicator called MPI_COMM_WORLD, consisting of all processors.
  - Using MPI_COMM_WORLD, every processor can communicate with every other processor.
  - You can define additional communicators consisting of subsets of the available processors.

Getting Communicator Information: Rank

- A processor can determine its rank in a communicator with a call to MPI_COMM_RANK.
  - Ranks are consecutive and start with 0.
  - A given processor may have different ranks in the various communicators to which it belongs.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

  - the argument comm is a variable of type MPI_COMM, a communicator.
  - You could use MPI_COMM_WORLD here or alternatively, you could pass the name of another communicator you have defined elsewhere.
  - Such a variable would be declared as:

```
MPI_Comm some_comm;
```

Getting Communicator Information: Size

- A processor can also determine the size, i.e., number of processors, of any communicator to which it belongs with a call to MPI_COMM_SIZE.

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- The argument comm is of type MPI_COMM, a communicator. The second argument is the address of the integer variable size.
- If the communicator is MPI_COMM_WORLD, the number of processors returned from MPI_COMM_SIZE equals the number defined by the command-line input to MPIRUN

```
% mpirun -np 4 a.out
```

Terminating MPI

- MPI_FINALIZE is the last MPI routine called in a program.
  - It terminates the program by cleaning up all MPI data structures, canceling operations that never completed, and so on.
- MPI_FINALIZE must be called by all processes; if any one process does call it, the program will appear to hang.
- Once MPI_FINALIZE has been called, no other MPI routines (including MPI_INIT) may be called....

```
err =  MPI_Finalize();
```

Simple Example

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char *argv[];
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

```
mpicc hello.c -o hello
mpirun -np 2 ./hello
```

Point-to-Point Communication

- The point-to-point communication facilities are two-sided and require active participation from the processes on both sides.
  - One process (the source) sends, and another process (the destination) receives.
- In general, the source and destination processes operate asynchronously.
  - Even the sending and receiving of a single message is typically not synchronized.
  - The source process may complete sending a message long before the destination process gets around to receiving it, and the destination process may initiate receiving a message that has not yet been sent.

Messages

- Messages consist of two parts:
  - the envelope
  - the message body.
- The envelope of an MPI message has four parts:
  - Source - the sending process
  - Destination - the receiving process
  - Communicator - specifies a group of processes to which both source and destination belong
  - Tag - used to classify messages
- The tag field is required, but its use is left up to the program.
- A pair of communicating processes can use tag values to distinguish classes of messages.
- One tag value can be used for messages containing data and another tag value for messages containing status information.

Messages

- The message body has three parts:
  - Buffer - the message data
  - Datatype - the type of the message data
  - Count - the number of items of type datatype in buffer
- Think of the buffer as an array; the dimension is given by count, and the type of the array elements is given by datatype.
- Using datatypes and counts, rather than bytes and bytecounts, allows structured data and noncontiguous data to be handled smoothly.
- It also allows transparent support of communication between heterogeneous hosts.

Sending and Receiving Messages

- The source (the identity of the sender) is determined implicitly, but the rest of the message (envelope and body) is given explicitly by the sending process.
- Sending and receiving are typically not synchronized.
  - Processes often have one or more messages that have been sent but not yet received.
  - These messages that have not yet been received are called pending messages.
  - Pending messages are not maintained in a simple FIFO queue.
    - Each pending message has several attributes and the destination process (the receiving process) can use the attributes to determine which message to receive.

Sending and Receiving Messages

- To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages.
  - If there is a match, a message is received.
  - Otherwise, the receive operation cannot be completed until a matching message is sent.
  - The process receiving a message must provide storage into which the body of the message can be copied.
  - The receiving process must be careful to provide enough storage for the entire message.

Blocking Send and Receive

- The basic point-to-point communication routines in MPI are MPI_SEND and MPI_RECV.
- Both routines block the calling process until the communication operation is completed.

Sending a Message: MPI_SEND

- MPI_SEND takes the following arguments:
  - Message body:
    - buffer
    - count
    - datatype
  - Message envelope:
    - destination
    - tag
    - communicator
- The message body contains the data to be sent: count items of type datatype ,the message envelope tells where to send it.
- In addition, an error code is returned.

```
int MPI_Send(void *buf, int count,
MPI_Datatype dtype, int dest, int tag,
MPI_Comm comm);
```

Receiving a Message: MPI_RECV

- MPI_RECV takes a set of arguments similar to MPI_SEND, but several of the arguments are used in a different way.
  - Message body:
    - buffer
    - count
    - datatype
  - Message envelope:
    - source
    - tag
    - communicator
- The arguments in the message envelope determine what messages can be received by the call.
  - The source, tag, and communicator arguments must match those of a pending message in order for the message to be received.

Receiving a Message: MPI_RECV

- Wildcard values may be used for the source (accept a message from any process) and the tag (accept a message with any tag value).
- If wildcards are not used, the call can accept messages only from the specified sending process and with only the specified tag value.
- Communicator wildcards are not available.
- The message body arguments specify where the arriving data are to be stored, what type it is assumed to be, and how much of it the receiving process is prepared to accept.
  - If the received message has more data than the receiving process is prepared to accept, it is an error and the program will abort.

Receiving a Message: MPI_RECV

- In general, the sender and receiver must agree about the message datatype, and it is your responsibility to guarantee that agreement.
  - If the sender and receiver use incompatible message datatypes, the results are undefined.
- The status argument returns information about the message that was received. The source and tag of the received message are available this way, which is needed if wildcards were used, and also available is the actual count of data received. In addition, an error code is returned.

```
int MPI_Recv(void *buf, int count,
MPI_Datatype dtype, int source, int tag,
MPI_Comm comm, MPI_Status *status);
```

- buf and status are output arguments; the rest are inputs.
- An error code is returned by the function.

Receiving a Message: MPI_RECV

- Some issues on receiving messages to remember are:
  - A maximum of COUNT items of type DTYPE are accepted; if the message contains more, it is an error.
  - The sending and receiving processes must agree on the datatype; if they disagree, results are undefined (MPI does not check).
  - When this routine returns, the received message data have been copied into the buffer; and the tag, source, and actual count of data received are available via the status argument

Message Passing Example

```
/* simple send and receive */
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char **argv) {

  int myrank,i;
  MPI_Status status;
  double a[100],b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 )          /* Send a message */
  {
    for (i=0;i<100;++i)
       a[i]=sqrt(i);
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 )   /* Receive a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );...

  MPI_Finalize();          /* Terminate MPI */
  return 0;
}
```

Wildcard Reception

- The MPI_RECV call made by Processor 1 in the previous example program is a completely specified reception.
  - Processor 1 will only receive a message from Processor 0 with tag 17.
- It is possible to configure the reception so it is open to messages from any processor or any message tag.
  - This is called wildcard reception.
- For source wildcard reception, use MPI_ANY_SOURCE for the source argument in MPI_RECV.
- For tag wildcard reception use MPI_ANY_TAG for the tag argument in MPI_RECV.

Wildcard Reception

- Once the MPI_RECV function is completed with some message, the destination process has little information about it.
  - If source and tag wildcard reception were both used, the destination will not know where the message came from or its tag.
- This "envelope" information is stored in the status variable and can be retrieved from it.
- The status variable is a structure containing the message information as members.
- The following expression is used to get the source of the wildcarded reception:

  `status.MPI_SOURCE`

- To get the tag information, use the expression:

  `status.MPI_TAG`

Message Size

- The message-receive statement from our example program was:

```
MPI_Recv(b,100,MPI_DOUBLE,0,17,MPI_COMM_WORLD,
&status);
```

- Remember the meaning of the second argument: the maximum number of elements that the array b could hold, not necessarily the number of elements actually received.
- The sending process could have transmitted a smaller number.
- The message count is stored in the status variable and can be extracted from it.
- The auxiliary MPI function MPI_GET_COUNT is used for this purpose.

```
int MPI_Get_count(MPI_Status *status,
MPI_Datatype dtype, int *count);
```

Example Wildcard Program

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

int main (int argc, char **argv) {

  int myrank,i,count;
  MPI_Status status;
  double a[100],b[300];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {          /* Send a message */
    for (i=0;i<100;++i)
         a[i]=sqrt(i);
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }else if( myrank == 1 ){   /* Receive a message */
    MPI_Recv( b, 300, MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,.
                                        MPI_COMM_WORLD, &status );...
    MPI_Get_count(&status,MPI_DOUBLE,&count);
    printf("P:%d message came from rank %d\n",myrank,status.MPI_SOURCE);
    printf("P:%d message had tag %d\n",myrank,status.MPI_TAG);
    printf("P:%d message size was %d\n",myrank,count);
  }
 MPI_Finalize();           /* Terminate MPI */
 return(0);
}
```

Runtime Behaviour

- When a message is sent using MPI_SEND one of two things may happen:
  - The message may be copied into an MPI internal buffer and transferred to its destination later, in the background, or
  - The message may be left where it is, in the program's variables, until the destination process is ready to receive it. At that time, the message is transferred to its destination.
- In the first case, the sending process is allowed to move on to other things after the copy is completed. The second case minimizes copying and memory use, but may result in extra delay to the sending process and the delay can be significant.

Runtime Behaviour

- In the first case, a call to MPI_SEND may return before any non-local action has been taken or even begun, i.e., before anything has happened that might naively be associated with sending a message.
- In the second case, a synchronization between sender and receiver is implied.

Blocking and Completion

- Both MPI_SEND and MPI_RECV block the calling processes.
  - Neither returns until the communication operation it invoked is completed.
- The meaning of completion for a call to MPI_RECV is simple and intuitive - a matching message has arrived, and the message's data have been copied into the output arguments of the call.
  - In other words, the variables passed to MPI_RECV contain a message and are ready to be used.

Blocking and Completion

- For MPI_SEND, the meaning of completion is simple but not as intuitive.
  - A call to MPI_SEND is completed when the message specified in the call has been handed off to MPI.
  - In other words, the variables passed to MPI_SEND can now be overwritten and reused.
- If MPI copied the message into an internal buffer, then the call to MPI_SEND may be officially completed, even though the message has not yet left the sending process.
- If a message passed from MPI_SEND is larger than MPI's available internal buffer, then simple, one-time buffering cannot be used.
  - The sending process must block until the destination process begins to receive the message, or until more buffer is available.

Deadlock

- When two (or more) processes are blocked and each is waiting for the other to make progress, deadlock occurs.
- Neither process makes progress because each depends on the other to make progress first.
- The program shown on the next slide is an example - it fails to run to completion because processes 0 and 1 deadlock.

Deadlock Example

```
/* simple deadlock */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {

  int myrank;
  MPI_Status status;
  double a[100], b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );...
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  }
  MPI_Finalize();           /* Terminate MPI */
  return 0;
}
```

Avoiding Deadlock

```
/* safe exchange */
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv) {

  int myrank;
  MPI_Status status;
  double a[100], b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Receive a message, then send one */
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 ) {
    /* Send a message, then receive one */
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );...
  }

  MPI_Finalize();           /* Terminate MPI */
  return 0;
}
```

Avoiding Deadlock (Sometimes)

```
/* depends on buffering */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {

  int myrank;
  MPI_Status status;
  double a[100], b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Send a message, then receive one */
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
  }
  else if( myrank == 1 ) {
    /* Send a message, then receive one */
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );...
  }
  MPI_Finalize();           /* Terminate MPI */
  return 0;
}
```

Probable Deadlock

```
/* probable deadlock */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {
  int myrank;
  MPI_Status status;
#define N 100000
  double a[N], b[N];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Send a message, then receive one */
    MPI_Send( a, N, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    MPI_Recv( b, N, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
  }
  else if( myrank == 1 ) {
    /* Send a message, then receive one */
    MPI_Send( a, N, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Recv( b, N, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );...
  }
  MPI_Finalize();           /* Terminate MPI */
  return 0;
}
```

Nonblocking Sends and Receives

- MPI provides a way to invoke send and receive operations that does not block the calling process.
  - It is possible to separate the initiation of a send or receive operation from its completion by making two separate calls to MPI.
  - The first call initiates the operation, and the second call completes it.
  - If such a separation is used, it is referred to as a non-blocking communication.
  - Between the two calls, the program is free to perform other operations.
- The underlying communication operations are the same whether they are invoked by a single call or by two separate calls - one to initiate the operation and another to complete it. The communication operations are the same, but the interface to the library is different.

Nonblocking Sends and Receives

- Blocking and nonblocking communication can be mixed together for the same data transfer.
  - The source processor might use a blocking send and the destination process could use a nonblocking receive process, or vice-versa.
- Initiating a send operation is called posting a send.
- Initiating a receive operation is called posting a receive.
- Once a send or receive operation has been posted, MPI provides two distinct ways of completing it.
  - A process can test to see if the operation has completed without blocking on the completion.
  - Alternately, a process can wait for the operation to complete.

Nonblocking Sends and Receives

- After posting a send or receive with a call to a nonblocking routine, the posting process needs some way to refer to the posted operation.
- MPI uses request handles for this purpose.
  - Nonblocking send and receive routines all return request handles, which are used to identify the operation posted by the call.

Posting Sends without Blocking

- A process calls the routine MPI_ISEND to post (Initiate) a send without blocking on completion of the send operation.
- The calling sequence is similar to the calling sequence for the blocking routine MPI_SEND but includes an additional output argument, a request handle.

```
int MPI_Isend(void *buf, int count,
MPI_Datatype dtype, int dest, int tag,
MPI_Comm comm, MPI_Request *request);
```

- None of the arguments passed to MPI_ISEND should be read or written until the send operation is completed.

Posting Receives without Blocking

- A process calls the routine MPI_IRECV to post (Initiate) a receive without blocking on its completion.
- The calling sequence is similar to the calling sequence for the blocking routine MPI_RECV, but the status argument is replaced by a request handle.

```
int MPI_Irecv(void *buf, int count,
MPI_Datatype dtype, int source, int tag,
MPI_Comm comm, MPI_Request *request);
```

- None of the arguments passed to MPI_IRECV should be read or written until the receive operation is completed.

Completion: Waiting and Testing

- Posted sends and receives must be completed.
- If a send or receive is posted by a nonblocking routine, then its completion status can be checked by calling one of a family of completion routines.
- MPI provides both blocking and nonblocking completion routines.
  - The blocking routines are MPI_WAIT and its variants.
  - The nonblocking routines are MPI_TEST and its variants.

Completion: Waiting

- A process that has posted a send or receive by calling a nonblocking routine can subsequently wait for the posted operation to complete by calling MPI_WAIT.
- The posted send or receive is identified by passing a request handle.
- The arguments for the MPI_WAIT routine are:
  - request - a request handle (returned when the send or receive was posted
  - status - for receive, information on the message received; for send, may contain an error code
  - In addition, an error code is returned.

```
int MPI_Wait( MPI_Request *request, MPI_Status
*status );
```

Completion: Waiting

- If the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the posted operation was a send, the status argument may contain an error code for the send operation (different from the error code for the call to MPI_WAIT).

Completion: Testing

- A process that has posted a send or receive by calling a nonblocking routine can subsequently test for the posted operation's completion by calling MPI_TEST.
- The posted send or receive is identified by passing a request handle.
- The arguments for the MPI_TEST routine are:
  - request - a request handle (returned when the send or receive was posted)
  - flag - "true" if the send or receive has completed
  - status - undefined if flag equals "false". Otherwise, like MPI_WAIT

```
int MPI_Test( MPI_Request *request, int *flag,
MPI_Status *status );
```

Completion: Testing

- The request argument is expected to identify a previously posted send or receive.
- MPI_TEST returns immediately.
- If the flag argument is "true," then the posted operation is complete.
- If the flag argument is "true" and the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the flag argument is "true" and the posted operation was a send, then the status argument may contain an error code for the send operation (not for MPI_TEST).

Advantages and Disadvantages

- Selective use of nonblocking routines makes it much easier to write deadlock-free code. This is a big advantage because it is easy to unintentionally write deadlock into programs.
- On systems where latencies are large, posting receives early is often an effective, simple strategy for masking communication overhead.
  - Latencies tend to be large on physically distributed collections of hosts (for example, clusters of workstations) and relatively small on shared memory multiprocessors.
  - Masking communication overhead requires careful attention to algorithms and code structure.
- On the downside, using nonblocking send and receive routines may increase code complexity, which can make code harder to debug and harder to maintain.

Code Structure for Latency Hiding

```
MPI_IRECV(...,request)

...

arrived=FALSE

while (arrived == FALSE) {

   "work planned for processor to do while waiting for message data"

   MPI_TEST(request,arrived,status)

}

   "work planned for processor to do with the message data"
```

Non-blocking Send/Receive Example

```
/* deadlock avoided */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {
  int myrank;
  MPI_Request request;
  MPI_Status status;
  double a[100], b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Post a receive, send a message, then wait */
    MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &request );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    MPI_Wait( &request, &status );
  }
  else if( myrank == 1 ) {
    /* Post a receive, send a message, then wait */
    MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &request );...
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Wait( &request, &status );
  }

  MPI_Finalize();           /* Terminate MPI */
  return 0;
}
```

Send and Receive Modes

- In MPI, there are four send modes but only one receive mode. The four send modes are:
  - Standard Mode Send
  - Synchronous Mode Send
  - Ready Mode Send
  - Buffered Mode Send
- A receiving process can use the same call to MPI_RECV or MPI_IRECV, regardless of the send mode used to send the message.
- The standard mode send is the most widely used.
- Both blocking and nonblocking calls are available for each of the four send modes.

Naming Conventions and Calling Sequences

- The blocking send functions take the same arguments (in the same order) as MPI_SEND. The nonblocking send functions take the same arguments (in the same order) as MPI_ISEND.

Send Mode	Blocking Function	Nonblocking Function
Standard	MPI_SEND	MPI_ISEND
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Standard Mode Send

- When MPI executes a standard mode send, one of two things happens.
  - The message is copied into an MPI internal buffer and is transferred asynchronously to the destination process
  - The source and destination processes synchronize on the message.
- The MPI implementation is free to choose (on a case-by-case basis) between buffering and synchronizing, depending on message size, resource availability, and so on.
  - If the message is copied into an MPI internal buffer, then the send operation is formally completed as soon as the copy is done.
  - If the two processes synchronize, then the send operation is formally completed only when the receiving process has posted a matching receive and actually begun to receive the message.

Synchronous Mode Send

- Synchronous mode send requires MPI to synchronize the sending and receiving processes.
- When a synchronous mode send operation is completed, the sending process may assume the destination process has begun receiving the message.
  - The destination process need not be done receiving the message.
- The nonblocking call has the same advantages the nonblocking standard mode send has: the sending process can avoid blocking on a potentially lengthy operation.

Ready Mode Send

- Ready mode send requires that a matching receive has already been posted at the destination process before ready mode send is called.
  - If a matching receive has not been posted at the destination, the result is undefined.
  - It is your responsibility to make sure the requirement is met.
- In some cases, knowledge of the state of the destination process is available without doing extra work.

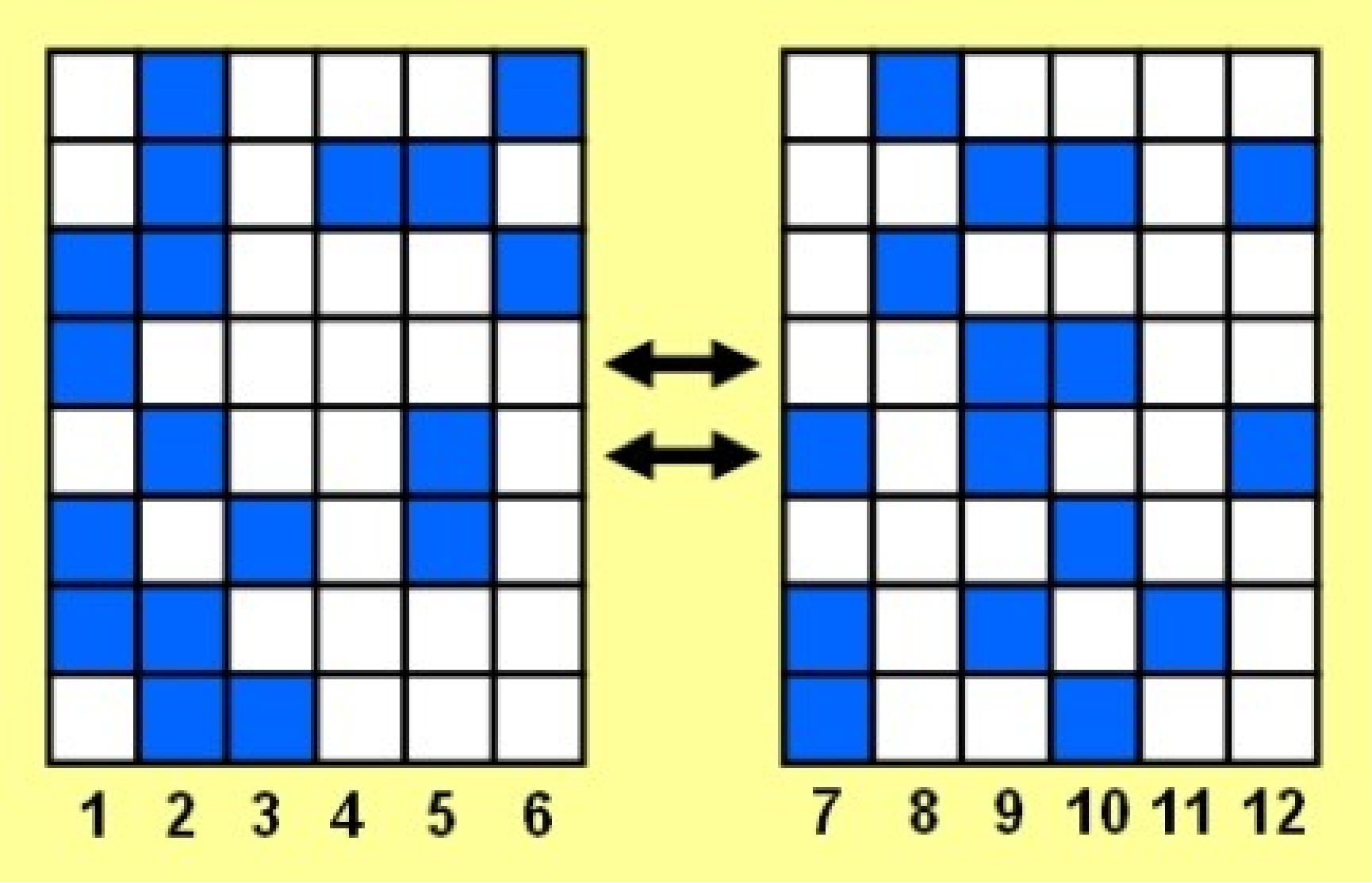
Buffered Mode Send

- Buffered mode send requires MPI to use buffering. The downside is that you must assume responsibility for managing the buffer.
- If at any point, insufficient buffer is available to complete a call, the results are undefined.
- The functions MPI_BUFFER_ATTACH and MPI_BUFFER_DETACH allow a program to make buffer available to MPI.

Game of Life

- In order to truly run the "Game of Life" program in parallel, we must set up our domain decomposition, i.e., divide the domain into chunks and send one chunk to each processor.
- In the current exercise, we will limit ourselves to two processors.
  - If you are writing your code in C, divide the domain with a horizontal line, so the upper half will be processed on one processor and the lower half on a different processor.
  - If you are using Fortran, divide the domain with a vertical line, so the left half goes to one processor and the right half to another.

Domain Decomposition



Ghost Cells

- One issue that you need to consider is that of internal domain boundaries.
- Each cell needs information from all adjacent cells to determine its new state.
- With domain decomposition, some of the required cells no longer are available on the local processor.
- A common way to tackle this problem is through the use of ghost cells.
- In the current example, a column of ghost cells is added to the right side of the left domain, and a column is also added to the left side of the right domain.
- After each time step, the ghost cells are filled by passing the appropriate data from the other processor.

Ghost Cells