DISTRIBUTED-MEMORY PARALLEL PROGRAMMING WITH MPI

Noor Mahammad Sk

Advantages of Parallel Programming

Need to solve larger problems

- more memory intensive
- more computation
- more data intensive

Parallel programming provides

- more CPU resources
- more memory resources
- solve problems that were not possible with serial program
- solve problems more quickly

Two Basic Architectures

Distributed Memory (ex. Compute cluster)

- collection of nodes which have multiple cores
- each node uses its own local memory
- work together to solve a problem
- communicate between nodes and cores via messages
- nodes are networked together

Shared Memory Computer

- multiple cores
- share a global memory space
- cores can efficiently exchange/share data

Parallel Programming Models

Directives-based parallel programming language

- OpenMP (most widely used)
- High Performance Fortran (HPF)
- Directives tell processor how to distribute data and work across the processors
- Directives appear as comments in the serial code
- Implemented on shared memory architectures

Message Passing (MPI)

- pass messages to send/receive data between processes
- each process has its own local variables
- can be used on either shared or distributed memory architectures
- outgrowth of PVM software

Pros and Cons of OpenMP/MPI

Pros of OpenMP

- easier to program and debug than MPI
- directives can be added incrementally - gradual parallelization
- can still run the program as a serial code
- serial code statements usually don't need modification
- code is easier to understand and maybe more easily maintained

Cons of OpenMP

- can only be run in shared memory computers
- requires a compiler that supports OpenMP
- mostly used for loop parallelization

Pros and Cons of OpenMP/MPI

Pros of MPI

- Runs on either shared or distributed memory architectures
- Can be used on a wider range of problems than OpenMP
- Each process has its own local variables
- Distributed memory computers are less expensive than large shared memory computers

Cons of MPI

- Requires more programming changes to go from serial to parallel version
- Can be harder to debug
- Performance is limited by the communcation network between the nodes

Parallel Programming Issues

Goal is to reduce execution time

- computation time
- idle time - waiting for data from other processors
- communication time - time the processors take to send and receive messages

Load Balancing

- divide the work equally among the available processors

Minimizing Communication

- reduce the number of messages passed
- reduce amount of data passed in messages

- Where possible - overlap communication and computation
- Many problems scale well to only a limited number of processors

MPI

- MPI, the Message Passing Interface.
- The use of explicit *message passing (MP), i.e., communication between processes*
- *MPI is the* most tedious and complicated but also the most flexible parallelization method.
- MPI is a *programming model and used* on shared-memory or hybrid systems

Message passing

- Message passing is required if a parallel computer is of the distributed-memory type,
- In Distributed Memory there is no way for one processor to directly access the address space of another processor.

MPI Working Approch

- The same program runs on all processes (Single Program Multiple Data, or *SPMD).*
- *This is no restriction compared to the more general MPMD (Multiple* Program Multiple Data) model as all processes taking part in a parallel calculation can be distinguished by a unique identifier called *rank (see below).*
- The program is written in a sequential language like Fortran, C or C++.
- Data exchange, i.e., sending and receiving of messages, is done via calls to an appropriate library.
- All variables in a process are local to this process.
- There is no concept of shared memory.

Message Passing Program

- Messages carry data between processes.
- Those processes could be running on separate compute nodes, or different cores inside a node, or
- Even on the same processor core, time-sharing its resources.
- A message can be as simple as a single item (like a DP word) or
- Even a complicated structure, perhaps scattered all over the address space.
- For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance.

MPI Parameters that need to be fixed are

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be left on the receiving process?
- What amount of data is the receiving process prepared to accept?
- **Note**:
- The above parameters strictly relate to point-to-point communication, where there is always exactly one sender and one receiver

MPI

- MPI supports much more than just sending a single message between two processes;
- There is a similar set of parameters for those more complex cases as well.
- MPI is a very broad standard with a huge number of library routines.
- Fortunately, most applications merely require less than a dozen of those.

“Hello World” MPI program in C

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv) {
5    int rank, size;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   printf("Hello World, I am %d of %d\n", rank, size);
12
13   MPI_Finalize();
14   return 0;
15 }
```

MPI

- **`MPI_Init()`** - This initializes the parallel environment
- **`MPI_COMM_WORLD`** – MPI World Communicator
- A communicator defines a group of MPI processes that can be referred to by a communicator *handle.*
- *The* **`MPI_COMM_WORLD`** *handle describes* all processes that have been started as part of the parallel program.
- If required, other communicators can be defined as subsets of MPI_COMM_WORLD.
- Nearly all MPI calls require a communicator as an argument.

MPI

- The calls to **`MPI_Comm_size()`** and **`MPI_Comm_rank()`** is to determine the number of processes (size) in the parallel program and the unique identifier (rank) of the calling process, respectively.
- Note that the C bindings require output arguments (like rank and size above) to be specified as pointers.
- The ranks in a communicator, in this case MPI_COMM_WORLD, are consecutive, starting from zero.
- The parallel program is shut down by a call to **`MPI_Finalize()`**.
- Note that no MPI process except rank 0 is guaranteed to execute any code beyond **`MPI_Finalize()`**.

MPI Compilation

- mpirun -np 4 ./hello.exe
- This would compile the code and start it with four processes.
- Be aware that processors may have to be allocated from some resource management (batch) system before parallel programs can be launched.
- How exactly MPI processes are started is entirely up to the implementation.
- Ideally, the start mechanism uses the resource manager's infrastructure (e.g., daemons running on all nodes) to launch processes.
- The same is true for process-to-core affinity;
- if the MPI implementation provides no direct facilities for affinity control.

MPI Program Output

```
1 Hello World, I am 3 of 4
2 Hello World, I am 0 of 4
3 Hello World, I am 2 of 4
4 Hello World, I am 1 of 4
```

Messages and point-to-point communication

- The “Hello World” example did not contain any real communication apart from starting and stopping processes.
- An MPI message is defined as an array of elements of a particular MPI data type.
- Data types can either be basic types or *derived types, which* must be defined by appropriate MPI calls.
- The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions.
- For any message transfer to proceed, the data types on sender and receiver sides must match.

point-to-point communication

- If there is exactly one sender and one receiver we speak of *point-to-point communication.*
- Both ends are identified uniquely by their ranks.
- Each point-to-point message can carry an additional integer label, the so-called *tag, which may be used to* identify the type of a message, and which must match on both ends.
- It may carry any accompanying information, or just be set to some constant value if it is not needed.

Details of Message Passing

- For a Communication to Succeed
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver's buffer must be large enough

MPI C Data Types

MPI Data type	C Data Type
MPI_CHAR	Signed Char
MPI_SHORT	Signed Short Int
MPI_INT	Signed Int
MPI_LONG	Signed Long Int
MPI_UNSIGNED_CHAR	Unsigned Char
MPI_UNSIGNED_SHORT	Unsigned Short Int
MPI_UNSIGNED_INT	Unsigned Int
MPI_UNSIGNED_LONG	Unsigned Long Int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long Double
MPI_BYTE	
MPI_PACKED	

MPI Point-to-Point Communication Modes

Send Modes	MPI function	Completion Condition
Synchronous send	MPI_Ssend	only completes when the receive has completed
Buffered send	MPI_Bsend	always completes (unless an error occurs) irrespective of the receiver
**Standard send	MPI_Send	message sent (receive state unknown)
Ready send	MPI_Rsend	may be used only when the a matching receive has already been posted

Receive Mode	MPI function	Completion Condition
Receive	MPI_Recv	Complete when a message has arrived

MPI_Send()

- The basic MPI function to send a message from one process to another is **`MPI_Send()`**

```
1 <type> buf(*)
2 integer :: count, datatype, dest, tag, comm, ierror
3 call MPI_Send(buf,      ! message buffer
4               count,    ! # of items
5               datatype, ! MPI data type
6               dest,     ! destination rank
7               tag,      ! message tag (additional label)
8               comm,     ! communicator
9               ierror)   ! return value
```

MPI_Recv()

- A message may be received with the **`MPI_Recv()`** function:

```
1  <type> buf(*)
2  integer :: count, datatype, source, tag, comm,
3  integer :: status(MPI_STATUS_SIZE), ierror
4  call MPI_Recv(buf,      ! message buffer
5                count,    ! maximum # of items
6                datatype, ! MPI data type
7                source,   ! source rank
8                tag,      ! message tag (additional label)
9                comm,     ! communicator
10               status,   ! status object (MPI_Status* in C)
11               ierror)   ! return value
```

- Compared with MPI_Send(), this function has an additional output argument, the status object.
- After MPI_Recv() has returned, the status object can be used to determine parameters that have not been fixed by the call's arguments.
- Primarily, this pertains to the length of the message, because the count parameter is only a maximum value at the receiver side;
- The message may be shorter than count elements.

Status

- The MPI_Get_count() function can retrieve the real number:

```
1 integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
2 call MPI_Get_count(status,     ! status object from MPI_Recv()
3                    datatype,   ! MPI data type received
4                    count,      ! count (output argument)
5                    ierror)     ! return value
```

- However, the status object also serves another purpose.
- The source and tag arguments of MPI_Recv() may be equipped with the special constants (“wildcards”) MPI_ANY_SOURCE and MPI_ANY_TAG, respectively.
- MPI_ANY_SOURCE specifies that the message may be sent by anyone
- MPI_ANY_TAG determines that the message tag should not matter.
- After MPI_Recv() has returned, status(MPI_SOURCE) and status (MPI_TAG) contain the sender’s rank and the message tag, respectively.
- In C, the status object is of type struct MPI_Status, and access to source and tag information works via the “.” operator.

Simple program can be improved in several ways:

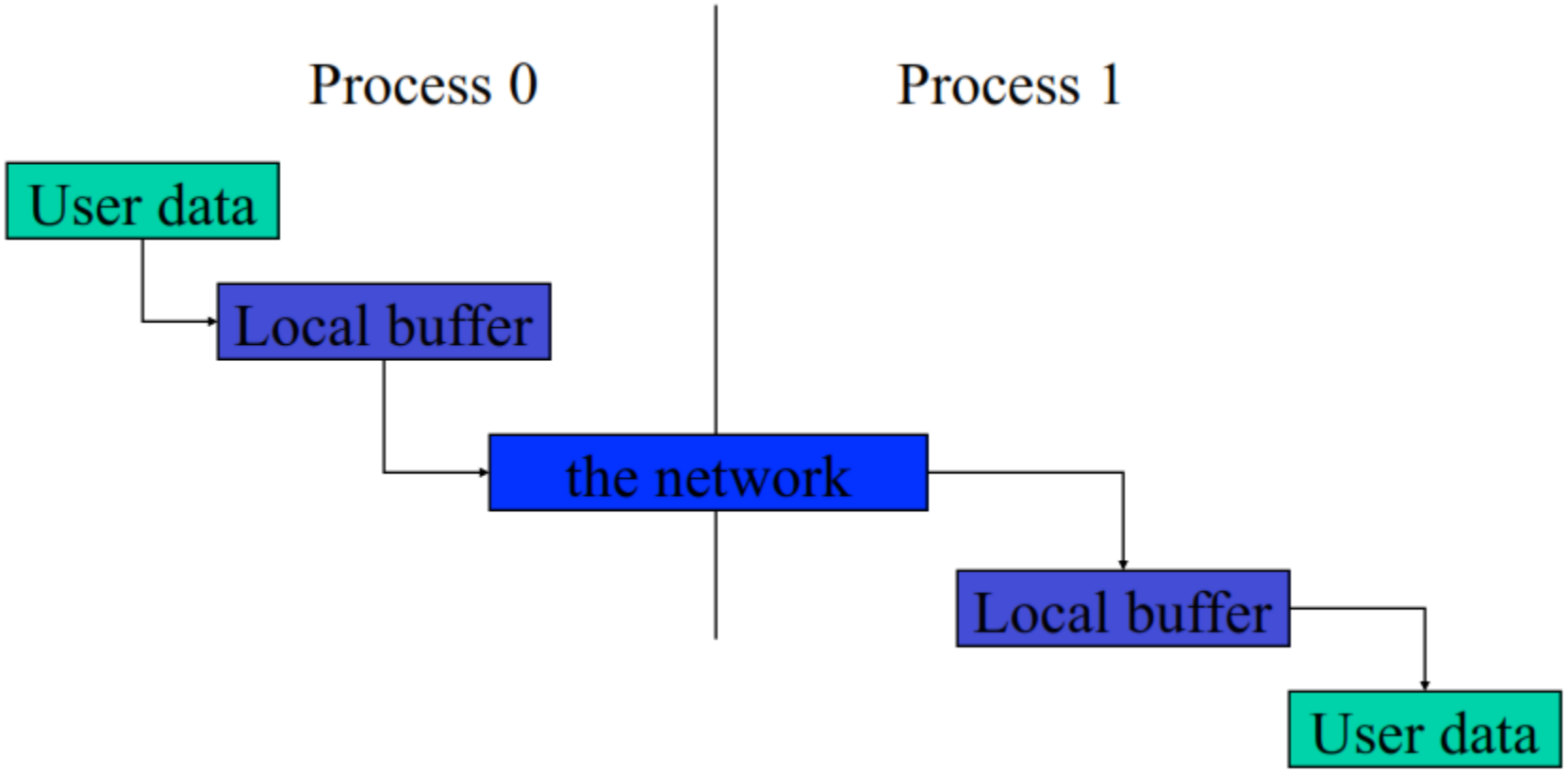
- MPI does not preserve the temporal order of messages unless they are transmitted between the same sender/receiver pair (and with the same tag).
- To allow the reception of partial results at rank 0 without delay due to different execution times of the integrate() function
- It may be better to use the MPI_ANY_SOURCE wildcard instead of a definite source rank in line 23.
- Rank 0 does not call MPI_Recv() before returning from its own execution of integrate().
- If other processes finish their tasks earlier, communication cannot proceed, and it cannot be overlapped with computation.
- The MPI standard provides *nonblocking point-to-point communication facilities that allow* multiple outstanding receives (and sends), and even let implementations support asynchronous messages.

Simple program can be improved in several ways:

- Since the final result is needed at rank 0, this process is necessarily a communication bottleneck if the number of messages gets large.
- We will demonstrate optimizations that can significantly reduce communication overhead in those situations.
- Fortunately, nobody is required to write explicit code for this.
- In fact, the global sum is an example for a *reduction operation* and is well supported within MPI.
- Vendor implementations are assumed to provide optimized versions of such global operations.

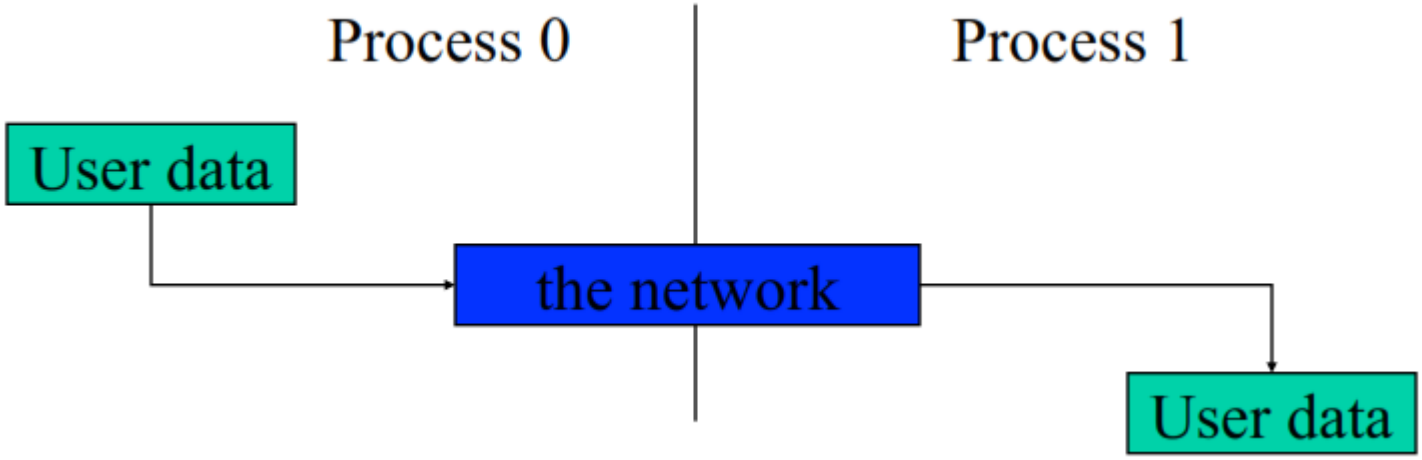
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



- This requires that **MPI_Send** wait on delivery, or that **MPI_Send** return before transfer is complete, and we wait later.

Sources of Deadlocks

- Send a large message from process 0 to process 1
- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Communication Modes

- MPI provides multiple *modes for sending messages:*
- Synchronous mode (MPI_Ssend): the send does not complete until amatching receive has begun. (Unsafe programs deadlock.)
- Buffered mode (MPI_Bsend): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
- Ready mode (MPI_Rsend): user guarantees that a matching receive has been posted.
  - Allows access to fast protocols
  - undefined behavior if matching receive not posted
- Non-blocking versions (MPI_Issend, etc.)
- MPI_Recv receives messages sent in any mode.

Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.
- int bufsize;
- char *buf = malloc(bufsize);
- MPI_Buffer_attach(buf, bufsize);
- ...
- MPI_Bsend(... same as MPI_Send ...)
- ...
- MPI_Buffer_detach(&buf, &bufsize);
- MPI_Buffer_detach waits for completion.
- Performance depends on MPI implementation and size of message.

Other Point-to Point Features

- **MPI_Sendrecv**
- **MPI_Sendrecv_replace**
- **MPI_Cancel(request)**
  - Cancel posted Isend or Irecv
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance
  - MPI_Send_init(…., &request)
  - MPI_Start(request)

MPI_Sendrecv

- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than “send left”

Process 0	Process 1
`SendRecv(1)`	`SendRecv(0)`

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has down nbr)
        Call MPI_Send( … down … )
if (has up nbr)
        Call MPI_Recv( … up … )
…
```

- sequentializes (all except the bottom process blocks)

Sequentialization

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Deadlock or Race Conditions

- Deadlock or race conditions occur when the message passing cannot be completed.
- Consider the following and assume that the MPI_Send does not complete until the corresponding MPI_Recv is posted and visa versa.
- The MPI_Send commands will never be completed and the program will deadlock.

```
if (rank == 0) {
     MPI_Send(..., 1, tag, MPI_COMM_WORLD);
     MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
     MPI_Send(..., 0, tag, MPI_COMM_WORLD);
     MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
}
```

Avoid Deadlock

- There are a couple of ways to fix this problem.
- One way is to reverse the order of one of the send/receive pairs:

```
if (rank == 0) {
            MPI_Send(..., 1, tag, MPI_COMM_WORLD);
            MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
            MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

Avoiding MPI Deadlock or Race Conditions

- Another way is to make the send be a non-blocking one (MPI_Isend)

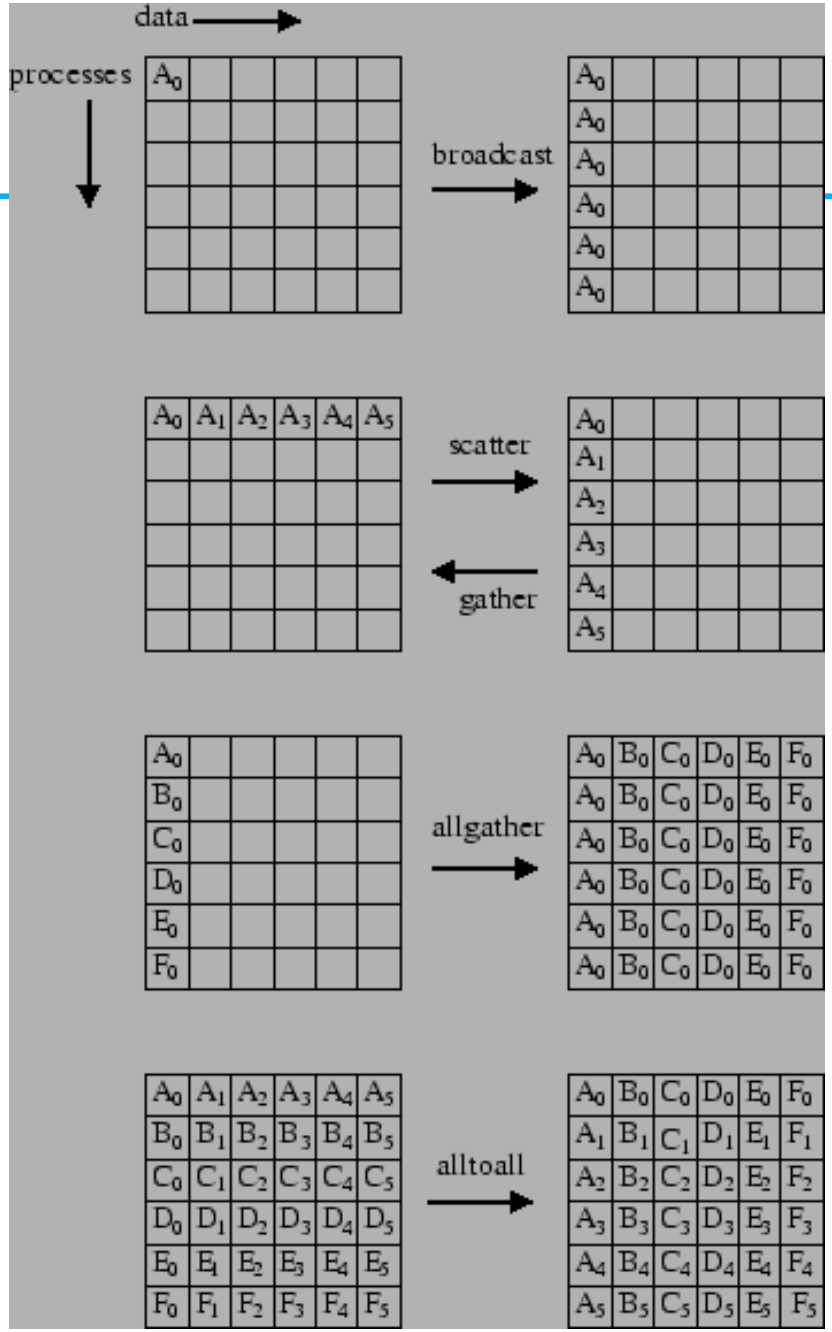
```
if (rank == 0) {
            MPI_Isend(..., 1, tag, MPI_COMM_WORLD, &req);
            MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
            MPI_Wait(&req, &status);
 } else if (rank == 1) {
            MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

MPI Error Handling

- MPI provides two predefined error handlers
  - **MPI_ERRORS_ARE_FATAL** (the default): causes MPI to abort
  - **MPI_ERRORS_RETURN**: causes MPI to return an error values instead of aborting
- MPI Error Handling Functions
  - **MPI_Errhandler_set**: set the error handler
  - **MPI_Error_class**: convert an error code into an error class
  - **MPI_Error_string**: returns a string for a given error code

MPI Collective Communication Routines

MPI_Function	Function Description
MPI_Bcast	Broadcast a message from on process to all others
MPI_Barrier	blocks until all processes have reached this routine
MPI_Reduce	Reduce values from all processes to a single value (add,mult, min, max, etc.)
MPI_Gather	Gathers together values from a group of processes
MPI_Scatter	Sends data from process to the other processes in a group
MPI_Allgather	Gathers data from all tasks and distributes it to all
MPI_Allreduce	Reduces values from all processes and distributes the result back to all processes

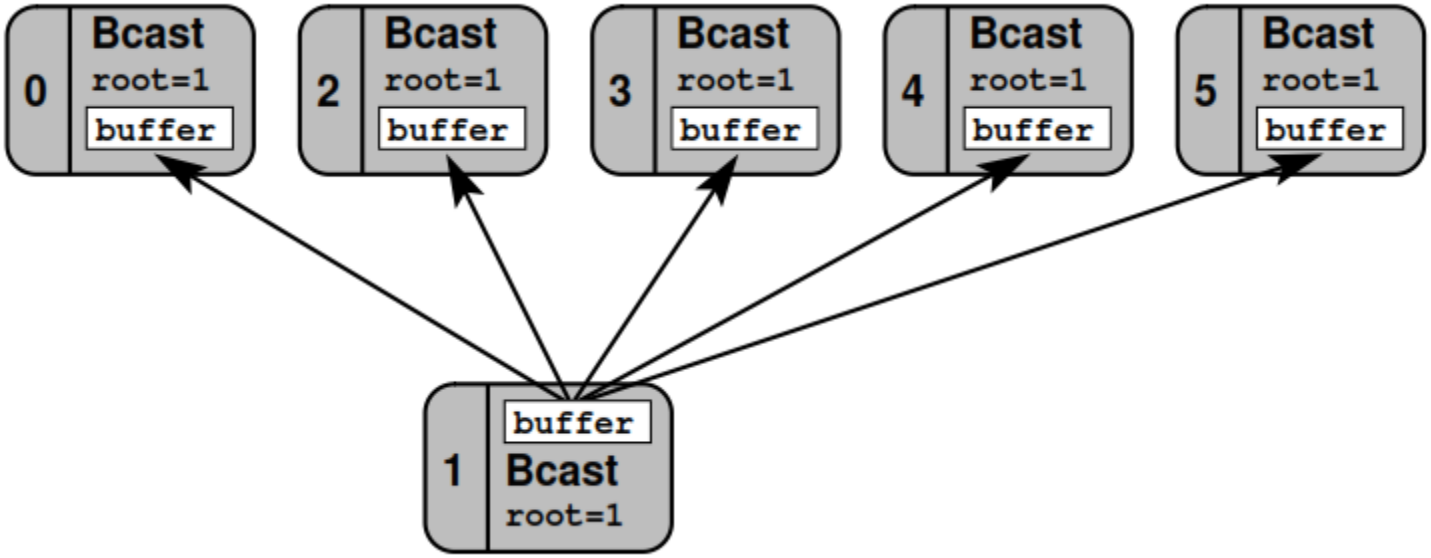

data
processes
broadcast
scatter
gather
allgather
alltoall

Broadcast

```
1 <type> buf(*)
2 integer :: count, datatype, root, comm, ierror
3 call MPI_Bcast(buffer,     ! send/receive buffer
4                count,      ! message length
5                datatype,   ! MPI data type
6                root,       ! rank of root process
7                comm,       ! communicator
8                ierror)     ! return value
```

MPI Broadcast

- The “root” process (rank 1 in this example) sends the same message to all others.
- Every rank in the communicator must call MPI_Bcast() with the same root argument.



Broadcast

- The buffer argument to MPI_Bcast() is a send buffer on the root and a receive buffer on any other process.
- Every process in the communicator must call the routine.
- A broadcast is needed whenever one rank has information that it must share with all others;
- E.g., there may be one process that performs some initialization phase after the program has started, like reading parameter files or command line options.
- This data can then be communicated to everyone else via MPI_Bcast().

Advanced Collective Calls

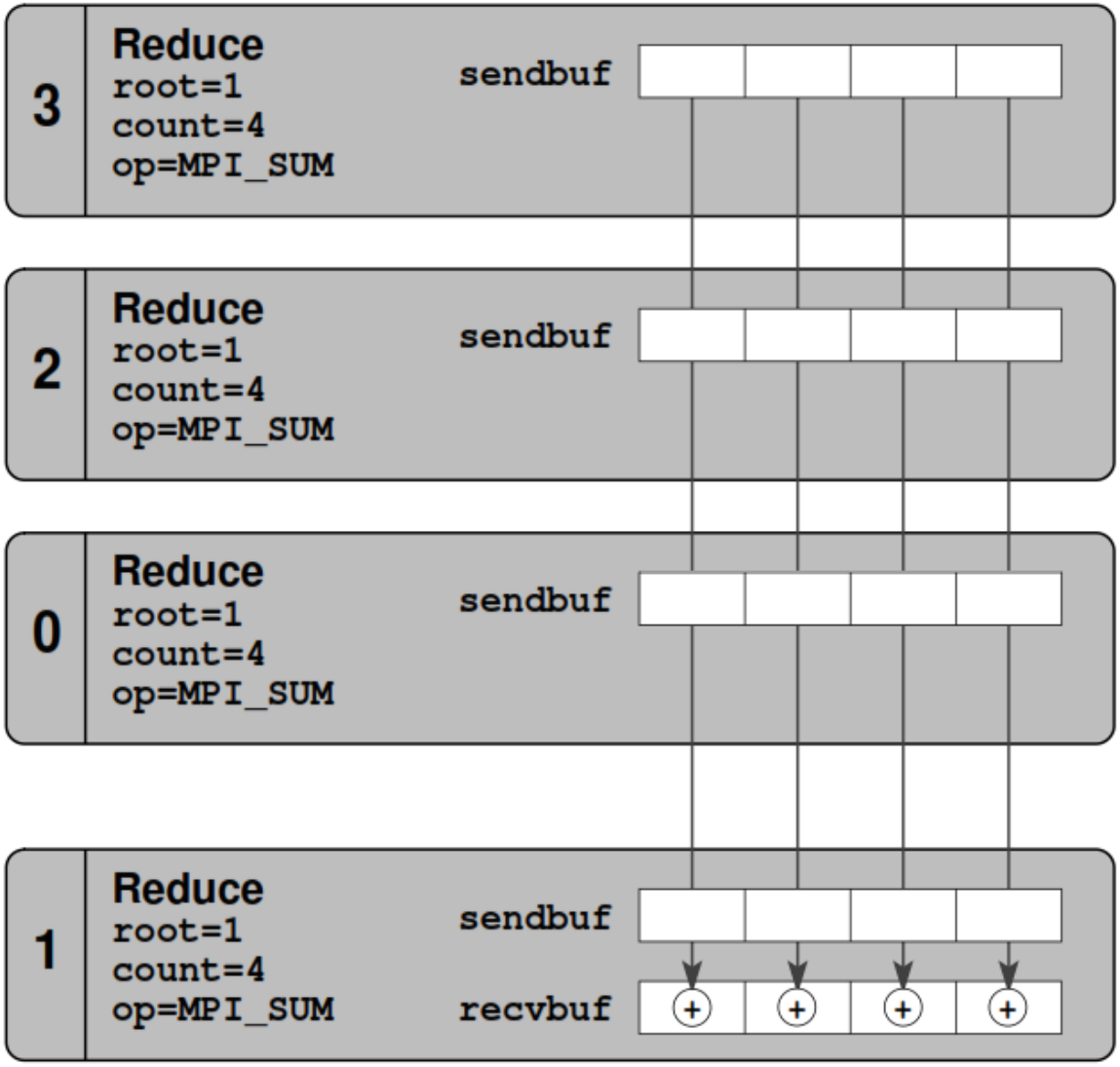
- Advanced Collective Calls are used for global data distribution
- MPI_Gather() collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root (Rank 0) process.
- MPI_Scatter() does the reverse, distributing equal-sized chunks of the root's send buffer.
- Both exist in variants (with a "v" appended to their names) that support arbitrary per-rank chunk sizes.
- MPI_Allgather() is a combination of MPI_Gather() and MPI_Bcast().

MPI_Reduce()

- MPI_Reduce() combines the contents of the sendbuf array on all processes, element-wise, using an operator encoded by the op argument, and stores the result in recvbuf on root.
- There are twelve predefined operators, the most important being MPI_MAX, MPI_MIN, MPI_SUM and MPI_PROD, which implement the global maximum, minimum, sum, and product, respectively.
- User-defined operators are also supported.

```
1  <type> sendbuf(*), recvbuf(*)
2  integer :: count, datatype, op, root, comm, ierror
3  call MPI_Reduce(sendbuf,     ! send buffer
4                  recvbuf,     ! receive buffer
5                  count,       ! number of elements
6                  datatype,    ! MPI data type
7                  op,          ! MPI reduction operator
8                  root,        ! root rank
9                  comm,        ! communicator
10                 ierror)      ! return value
```

MPI Reduce



- A reduction on an array of length count (a sum in this example) is performed by MPI_Reduce().
- Every process must provide a send buffer.
- The receive buffer argument is only used on the root process.
- The local copy on root can be prevented by specifying MPI_IN_PLACE instead of a send buffer address.

MPI_Allreduce()

- MPI_Allreduce() is a fusion of a reduction with a broadcast, and MPI_Reduce_scatter()
- MPI_Reduce_scatter() combines MPI_Reduce() with MPI_Scatter()
- Collective communication are prone to deadlock hazards as blocking point-to-point communication.
- This means, e.g., that collectives must be executed by all processes in the same order.

Note Key Point

- It is a good idea to prefer collectives over point-to-point constructs or
- Combinations of simpler collectives that “emulate” the same semantics.
- Good MPI implementations are optimized for data flow on collective communication and
- Should also have some knowledge about network topology built in.

MPI Blocking Communication

- Received data has arrived completely and sent data has left the buffer so that it can be safely modified without inadvertently changing the message.
- Collective communication in MPI is always blocking in the current MPI standard
- Point-to-point communication can be performed with *nonblocking semantics as well.*

MPI nonblocking Communication

- A nonblocking point-to-point call merely initiates a message transmission and returns very quickly to the user code.
- In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation. Synchronization is ruled out.
- Nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently.
- The message buffer must not be used as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls).
- Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

nonblocking send is MPI_Isend()

- As opposed to the blocking send.
- MPI_Isend() has an additional output argument, the *request handle.*
- *It serves as an identifier by which the program* can later refer to the "pending" communication request (in C, it is of type struct MPI_Request).

```
1  <type> buf(*)
2  integer :: count, datatype, dest, tag, comm, request, ierror
3  call MPI_Isend(buf,          ! message buffer
4                 count,        ! # of items
5                 datatype,     ! MPI data type
6                 dest,         ! destination rank
7                 tag,          ! message tag
8                 comm,         ! communicator
9                 request,      ! request handle (MPI_Request* in C)
10                ierror)       ! return value
```

MPI_Irecv() initiates a nonblocking Receive

```
1  <type> buf(*)
2  integer :: count, datatype, source, tag, comm, request, ierror
3  call MPI_Irecv(buf,          ! message buffer
4                 count,        ! # of items
5                 datatype,     ! MPI data type
6                 source,       ! source rank
7                 tag,          ! message tag
8                 comm,         ! communicator
9                 request,      ! request handle
10                ierror)       ! return value
```

- The status object known from MPI_Recv() is missing here, because it is not needed
- No actual communication has taken place when the call returns to the user code.

Checking Pending Communication

- Checking a pending communication for completion can be done via the MPI_Test() and MPI_Wait() functions.
- MPI_Test() only tests for completion and returns a flag.
- MPI_Wait() blocks until the buffer can be used:

```
1 logical :: flag
2 integer :: request, status(MPI_STATUS_SIZE), ierror
3 call MPI_Test(request,      ! pending request handle
4               flag,         ! true if request complete (int* in C)
5               status,       ! status object
6               ierror)       ! return value
7 call MPI_Wait(request,      ! pending request handle
8               status,       ! status object
9               ierror)       ! return value
```

The status object contains useful information only if the pending communication is a completed receive (i.e., in the case of MPI_Test() the value of flag must be true).

MPI's communication modes

	Point-to-point	Collective

Blocking	`MPI_Send()`
	`MPI_Ssend()`
	`MPI_Bsend()`
	`MPI_Recv()`
		`MPI_Barrier()`
		`MPI_Bcast()`
		`MPI_Scatter()/`
		`MPI_Gather()`
		`MPI_Reduce()`
		`MPI_Reduce_scatter()`
		`MPI_Allreduce()`

Nonblocking	`MPI_Isend()`
	`MPI_Irecv()`
	`MPI_Wait()/MPI_Test()`
	`MPI_Waitany()/`
	`MPI_Testany()`
	`MPI_Waitsome()/`
	`MPI_Testsome()`
	`MPI_Waitall()/`
	`MPI_Testall()`
		N/A