SF2568: Parallel Computations for Large-Scale Problems

Lecture 4: Introduction to MPI

January 23, 2024

Acknowledgements

These slides are an extension of slides by Michael Hanke and Niclas Jansson.

Programming Options Distributed Memory

Programming a message passing computer can be achieved by

1. Designing a **special** parallel programming language
2. Extending the **syntax/reserved words** of an existing high-level language to handle message-passing
3. Using an **existing** sequential high-level language and provide a library of external procedures for message-passing

What is MPI?

- **M**essage **P**assing **I**nterface
- The first standard and portable message passing library with **good performance**
- "Standard" by consensus of MPI Forum participants from over 40 organisations
- Finished and published in May 1994, updated in June 1995 (C and Fortran77)
- MPI 2 complete as of July 1997. Extends MPI (Fortran90 and C++ bindings)
- Documents that define MPI are on online from the MPI Forum
- MPI 2.1 released in 2008
- MPI 2.2 released in 2009
- MPI 3.0 released in 2012 (Fortran 2008 bindings, **C++ removed!**)
- MPI 3.1 released in 2015
- MPI 4.0 released in 2021!

Warning

Warning

In many respects, using MPI is low-level and thus is often called *"the assembly language of parallel programming"*

Format of MPI Routines

- C Bindings
  - `rc = MPI_Xxxxx(parameter, ...)`
  - `rc` is error code, = `MPI_SUCCESS` if successful
- Fortran 77 bindings
  - `call MPI_XXXXX(parameter,...,ierror)`
  - case insensitive
  - `ierror` is error code, = `MPI_SUCCESS` if successful
- Exception: timing functions return double precision floating points
- Header file required
  - `#include <mpi.h>` for C/C++ programs
  - `include 'mpif.h'` for Fortran77 programs
  - `use mpi` and `use mpi_f08` for Fortran90 and Fortran 2008 respectively

`MPI_ERRORS_ARE_FATAL` (default), `MPI_ERRORS_RETURN` user-defined error handling

MPI Routines

Initialization

`MPI_Init` initializes the MPI environment

`MPI_Comm_size` returns the number of processes

`MPI_Comm_rank` returns this process' number (its rank)

Communication to share data between PEs

`MPI_Send` sends a message

`MPI_Recv` receives a message

Finalization

`MPI_Finalize` exits in a "clean" fashion

An MPI Sample Program (Fortran Version)

```
program hello
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT (ierror)
call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierror)
tag = 42

if (rank .eq. 0) then
  message = 'Hello, world'
  do i=1, size-1
    call MPI_SEND(message, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, ierror)
  enddo
else
  call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, ierror)
endif

print*, 'node', rank, ':', message
call MPI_FINALIZE (ierror)
end
```

An MPI Sample Program (C Version)

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 42;

  if (rank == 0) {
    strcpy(message, "Hello world!");
    for (i = 1; i < size; i++) {
      rc = MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
  }
  else {
    rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  }
  printf("rank %d : %.13s\n", rank, message);
  rc = MPI_Finalize();
}
```

MPI Example

Let rank 0 send "Hello world!" to all other ranks in an SPMD program

Observations

- The program logic depends on the process' rank
- All ranks executes `print*` or `printf` statement
- What are these strange MPI types
  - `MPI_COMM_WORLD`
  - `MPI_CHAR` and `MPI_CHARACTER`
  - `MPI_Status`
- What is a tag?

What is a Message?

Communication to share data between PEs

An MPI message = Data + Envelope

```
call MPI_SEND(startbuf, count, datatype, dest, tag,
comm, ierror)
```

The data part of the message

```
call MPI_SEND(startbuf, count, datatype, dest, tag,
comm, ierror)
```

- Arguments
  - `startbuf` beginning of the buffer containing the data to be sent (starting location of data)
  - `count` number of elements to be sent (not bytes)
    - **Note: receive elements $\geq$ send elements**
  - `datatype` type of data, e.g. `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`
    - **Note: receive datatype $=$ send datatype**
- MPI Datatypes
  - Basic (mapped to C or Fortran datatypes)
  - Derived (mixed, non-contiguous data, user-defined, etc.)

MPI Basic Datatypes in Fortran

MPI Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes in C

MPI Datatype	C datatype
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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	

The envelop part of the message

```
call MPI_SEND(startbuf, count, datatype, dest, tag,
comm, ierror)
```

- Arguments
  - `dest` rank of the process, which is the destination (or source) for the message
    - **Note: receiver = sender or `MPI_ANY_SOURCE`**
  - `tag` number, which can be used to distinguish among messages
    - **Note: receiver = sender or `MPI_ANY_TAG`**
  - `comm` communicator: a collection of processes that can send messages to each other (defines a communication "space"), e.g. `MPI_COMM_WORLD`
    - **Note: receiver = sender**
    - Predefined communicator: `MPI_COMM_WORLD` (all the processes running when execution begins)

Point-to-Point Communication

- When two processes exchange a message between each other it is called point-to-point communication
- It only involves the sender and the receiver
- There can be many millions of concurrent point-to-point communication operation at a given time on a supercomputer like Dardel

Communication Modes

Communication Mode	Blocking Routines	Non-blocking Routines
synchronous	MPI_SSEND	MPI_ISSEND
ready	MPI_RSEND	MPI_IRSEND
buffered	MPI_BSEND	MPI_IBSEND
standard	MPI_SEND	MPI_ISEND
	MPI_RECV	MPI_IRECV
	MPI_SENDRECV	
	MPI_SENDRECV_REPLACE	

Standard Mode

- The operation is implementation dependent, not prescribed by the MPI standard
- Should be the best compromise amoung different scenarious on a given hardware
- Usually good, not necessarily optimal

Standard Blocking Send and Receive

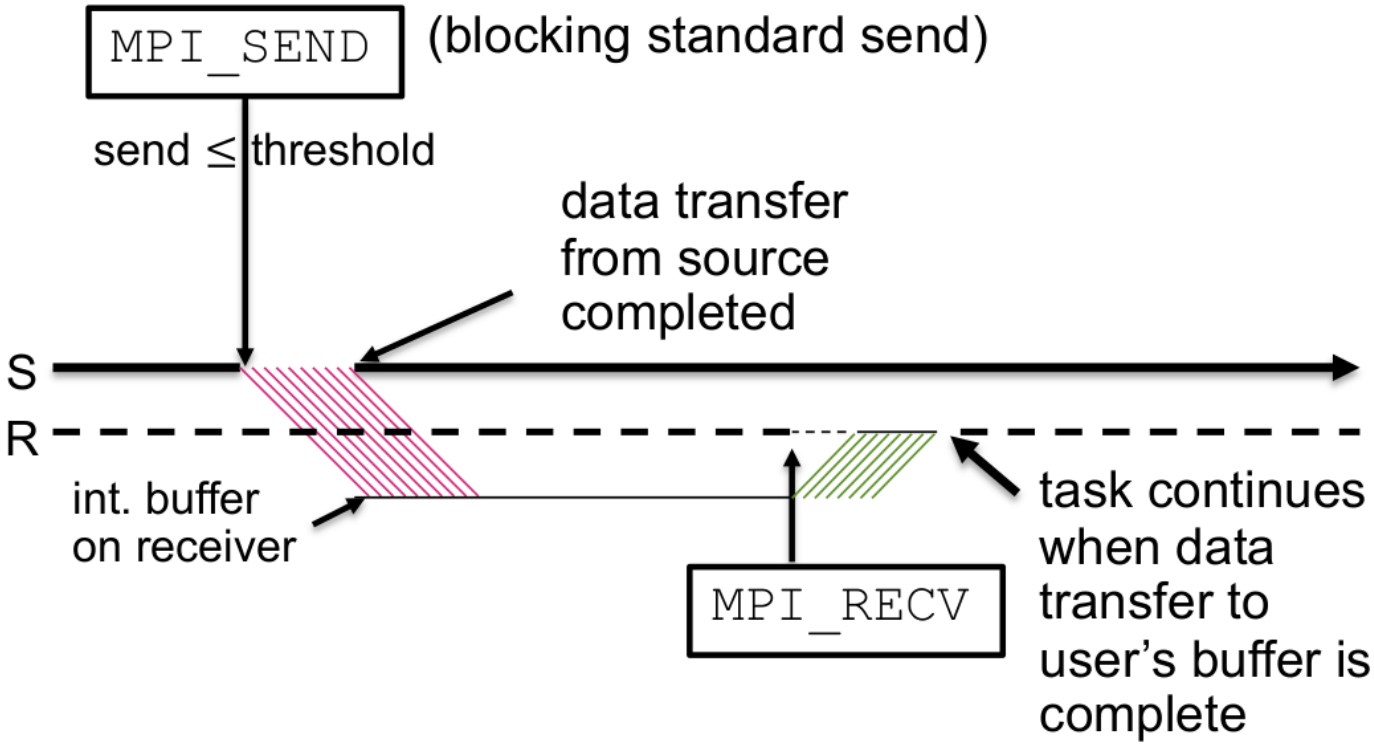
Fortran

```
MPI_Send(buf, count, datatype, dest, tag, comm,
ierror)

MPI_Recv(buf, count, datatype, source, tag, comm,
status,ierror)
```

C

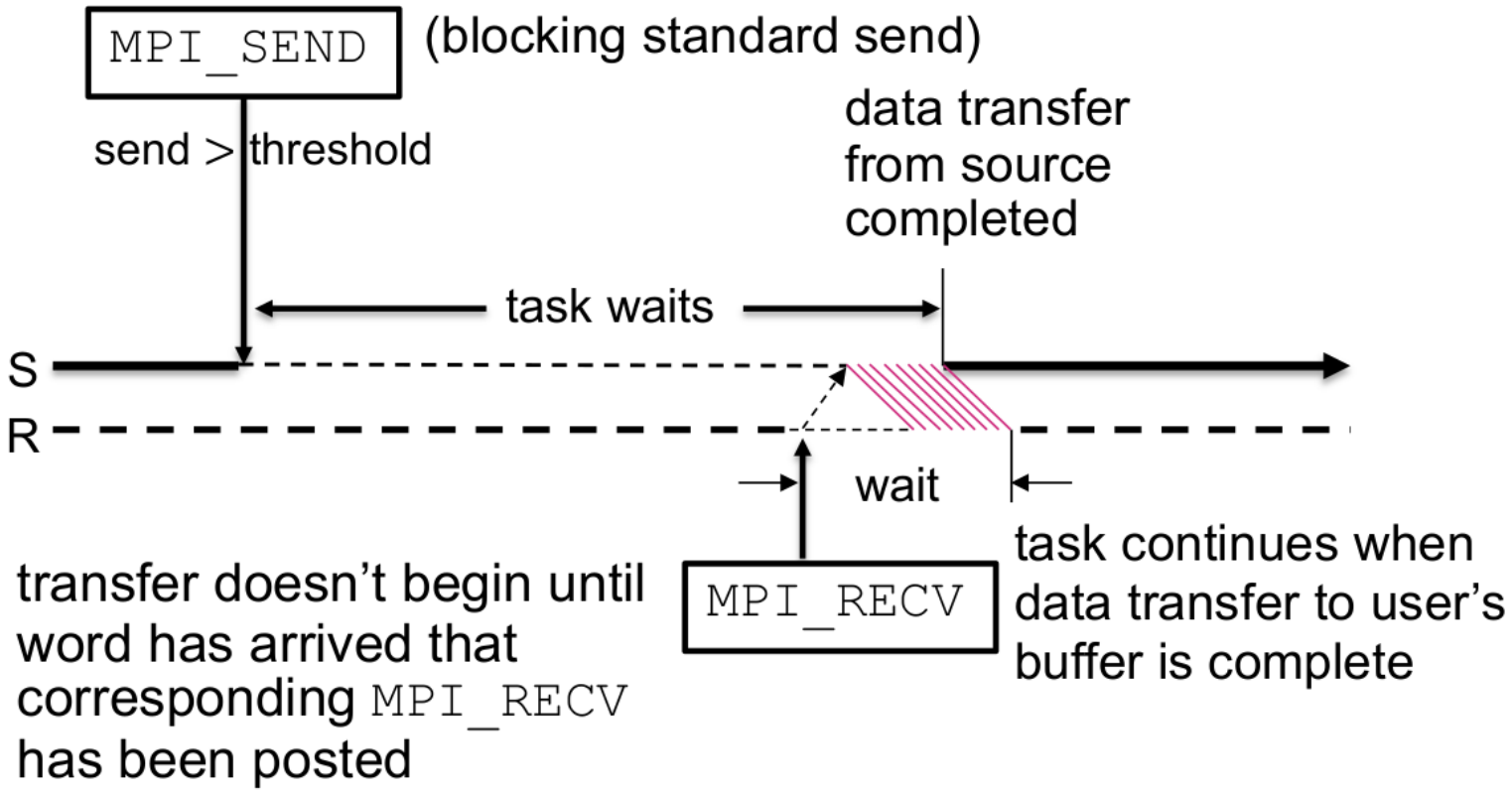
```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)

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

Standard Blocking Send (Short Messages)

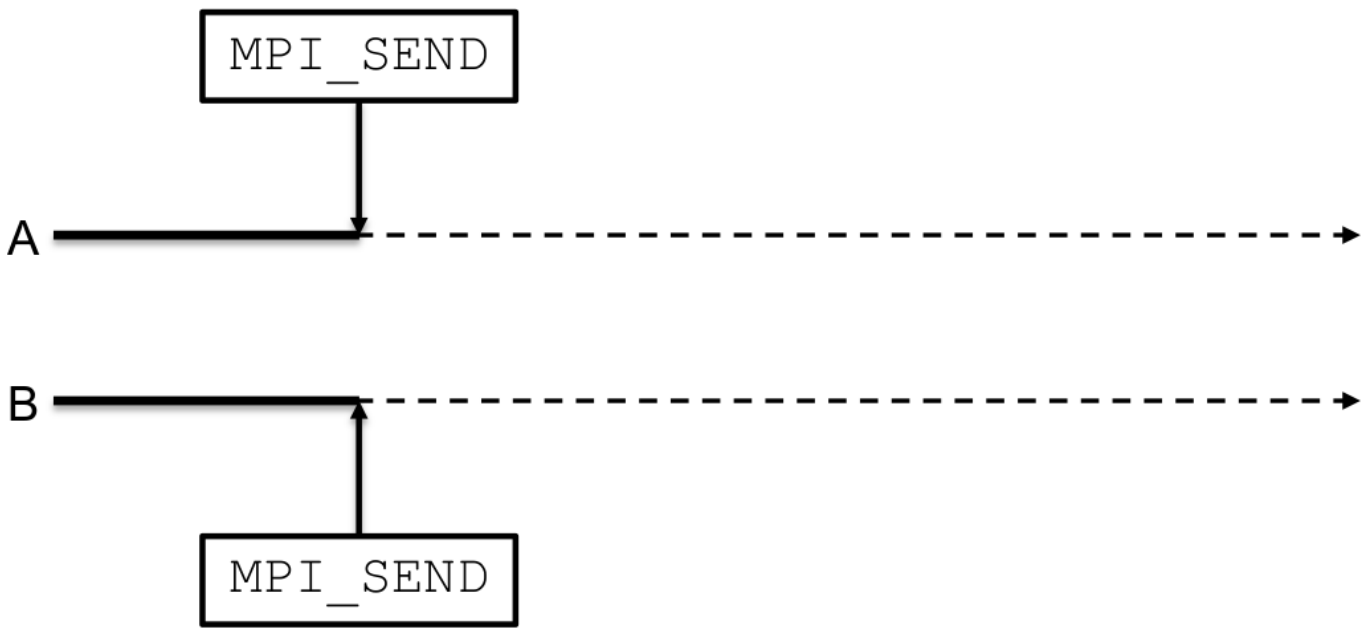


Exceeding system buffer space causes communication to stall until space is freed (or something worse. . .)

Standard Blocking Send (Long Messages)



What is a Deadlock?



Buffering

Suppose we have

```
if (rank==0)
MPI_Send(sendbuf,...,1,...)
if (rank==1)
MPI_Recv(recvbuf,...,0,...)
```

These are blocking communications, which means they will not return until the arguments to the functions can be safely modified by subsequent statements in the program.

Assume that process 1 is not ready to receive There are 3 possibilities for process 0:

- Stops and waits until process 1 is ready to receive
- Copies the message at sendbuf into a system buffer (can be on process 0, process 1 or somewhere else) and returns from `MPI_Send`
- Fails

Buffering

As long as buffer space is available, the second option is a reasonable alternative

An MPI implementation is permitted to copy the message to be sent into internal storage, but it is not required to do so

What if not enough space is available?

- In applications communicating large amounts of data, there may not be enough memory (left) in buffers
- Until receive starts, no place to store the send message
- Practically, first results in a serial execution

A programmer should not assume that the system provides adequate buffering

Example

Consider a program executing

Process 0	**Process 1**
`MPI_Send` to Process 1	`MPI_Send` to Process 0
`MPI_Recv` from Process 1	`MPI_Recv` frcom Process 0

This program may work in many cases, but is certain to fail for messages of some size that is large enough

Possible Solutions

- **Ordered send and receive** - make sure each receive is matched with send in execution order across processes

  This matched pairing can be difficult in complex applications. An alternative is to use `MPI_Sendrecv`. It performs both send and receive such that if no buffering is available, no deadlock will occur
- **Buffered sends** - MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in buffer) via `MPI_Bsend`
- **Non-blocking communication** - With buffering, a send may return before a matching receives is posted. With no buffering, communication is deferred until a place for receiving is provided. **Important:** in this case you must make certain that you do not modify (or use) the data until you are certain communication has completed.

Non-Blocking Communications

- Non-blocking communications are useful for overlapping communication with computation, and ensuring safe programs
- That is, **compute while communicating data**
- A non-blocking operation requests the MPI library to perform an operation (when it can)
- Non-blocking operations do not wait for any communication events to complete
- Non-blocking send and receive: return almost immediately
- The user can modify a send (receive) buffer only after send (receive) is completed
- There are "wait" routines to figure out when a non-blocking operation is done

Non-Blocking Standard Send and Receive

Fortran

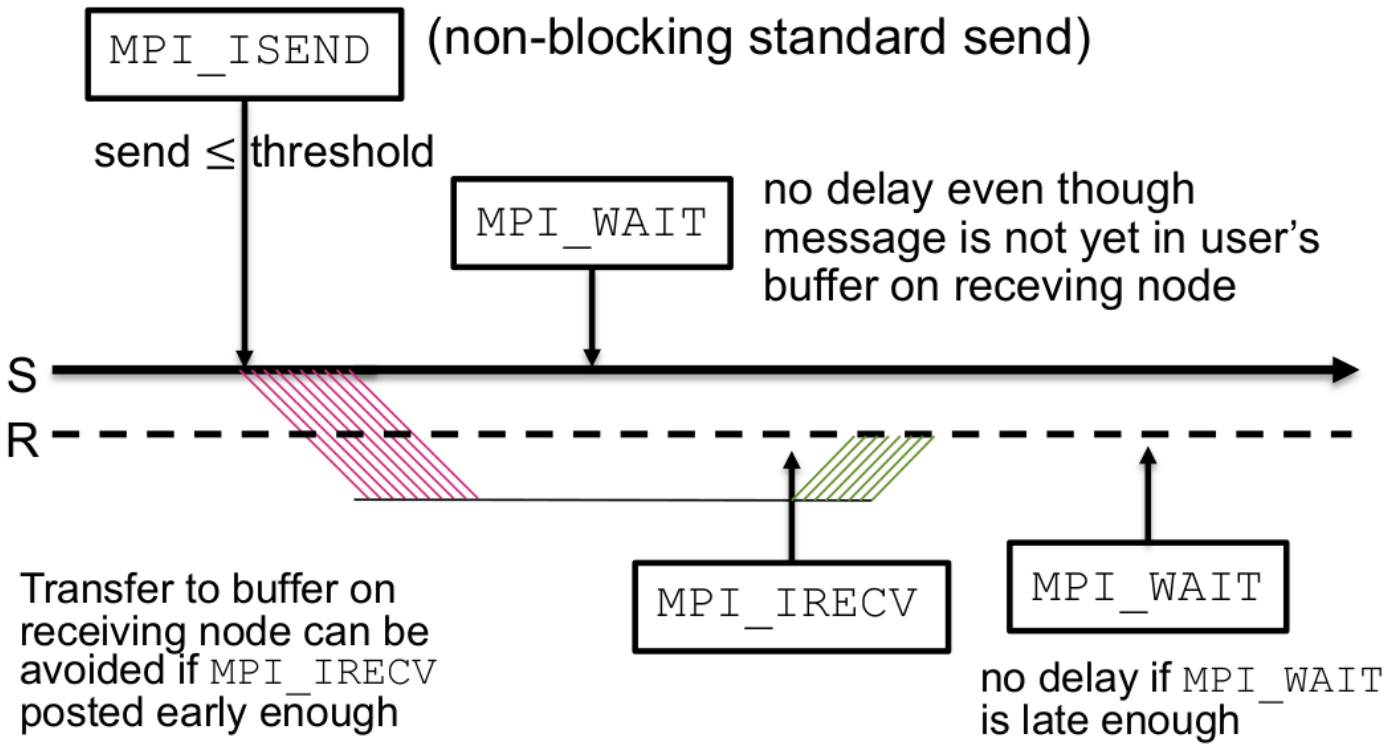
```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)

MPI_Irecv(buf, count, datatype, source, tag, comm, request,
ierror)
```

C

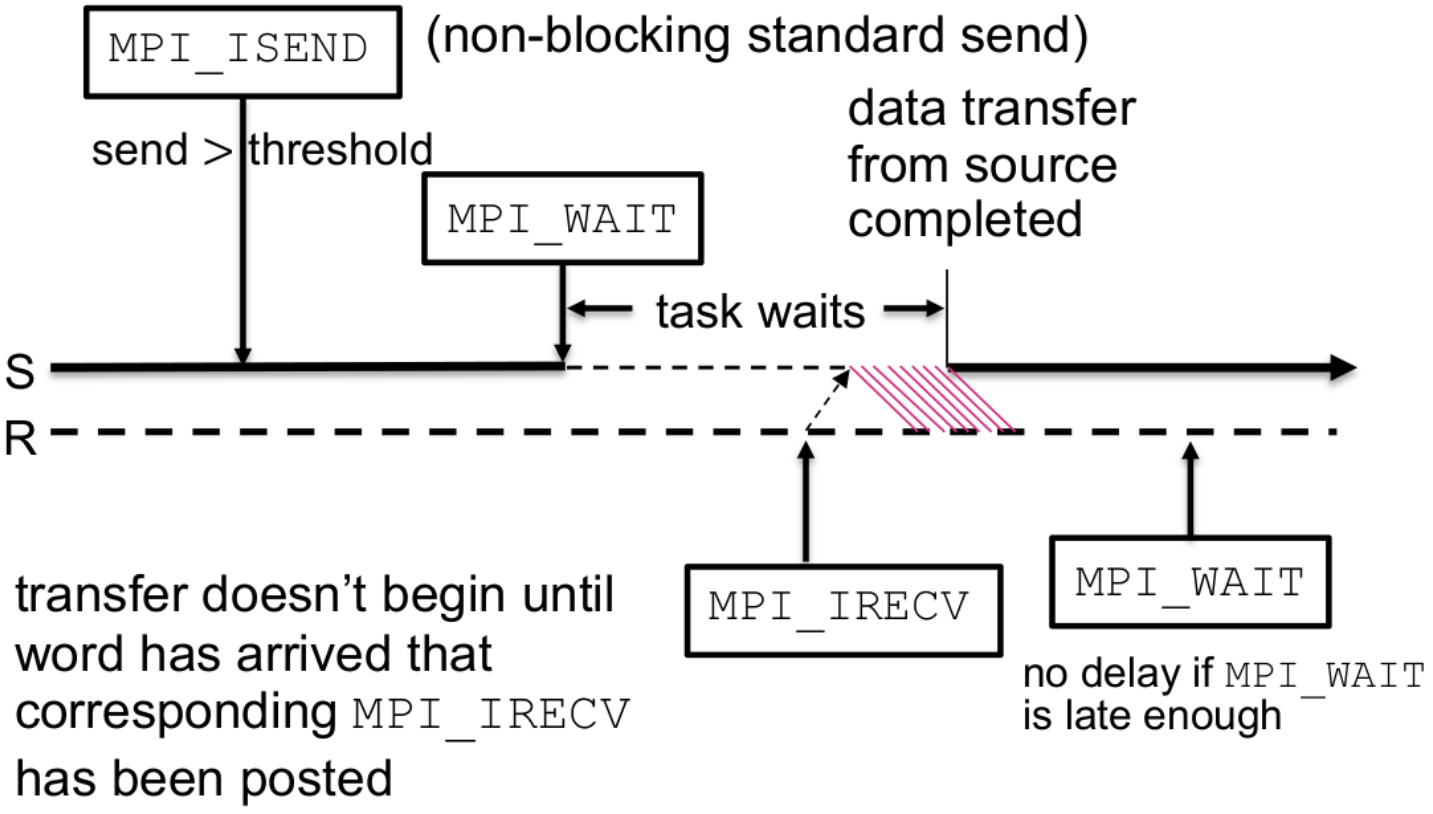
```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)

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

All calls must be followed by a `MPI_Wait(request, status)` sometime in the future!

Non-Blocking Standard Send



Non-Blocking Standard Send



Non-Blocking Calls

- Gains
  - Avoid deadlock (potenially. . .)
  - Decrease synchronisaton overhead
  - Reduce system's overhead (on some machines)
- Best to post non-blocking sends and receives as early as possible, and to do waits as late as possible
- Must avoid writing to send buffers between `MPI_Isend` and `MPI_Wait`
- Must avoid reading and writing in receive buffers between `MPI_Irecv` and `MPI_Wait`

Non-Blocking Calls

- Gains
  - Avoid deadlock (potenially. . .)
  - Decrease synchronisaton overhead
  - Reduce system's overhead (on some machines)
- Best to post non-blocking sends and receives as early as possible, and to do waits as late as possible
- Must avoid writing to send buffers between `MPI_Isend` and `MPI_Wait`
- Must avoid reading and writing in receive buffers between `MPI_Irecv` and `MPI_Wait`

Non-Blocking Calls

- Different ordering of MPI calls between tasks
- Use non-blocking calls
  - Note: They can still deadlock on `MPI_Wait`
- Use combined send+recv calls
- Can use buffered mode...

Safe programs

- A program is safe if it will produce correct results even if the system provides no buffering
- Need safe programs for portability
- Most programmers expect the system to provide some buffering, so unsafe MPI programs are around
- Write safe programs using matching send with receive, `MPI_Sendrecv`, allocating own buffers, nonblocking operations

Collective Communication

- Some communication pattern requires all processes to participate
  - Often when there is a global **dependency** of data
  - Or the operation needs to be synchronised between all processes
- A collective operation is when an **identical** MPI call is made **by all processes**
  - Limited to all processes in the same MPI communicator
- Supports easy manipulation of a "common" piece or set of information

MPI Collective Communication Routines

- Barrier synchronisation
- Broadcast from one member to all other members
- Gather data from an array spread across processors into one array
- Scatter data from one member to all members
- All-to-all exchange of data
- Global reductions (e.g., sum, min of "common" data elements)
- Scan across all members of a communicator

Characteristic

- Performed on a group of processes, identified by a communicator
- Substitute for a sequence of point-to-point calls
  - Often *highly optimized* by the MPI implementation
- Communications are locally blocking
- Synchronisation is not guaranteed (implementation dependent)
- Some routines use a root process to originate or receive all data
- Data amounts must exactly match
- Many variations to the basic categories presented
- No message tags are needed

Barrier Synchronisation

Blocks the calling process until all group members have called the routine

Fortran

```
MPI_Barrier(comm, ierr)
```

C

```
MPI_Barrier(MPI_comm comm)
```

Question: How will a barrier affect the scalability of a parallel program?

Data Movement Routines

- Broadcast
- Gather
- Scatter
- Allgather
- Alltoall

Barrier Synchronisation

Send a message from the root process (not necessarily zero!) to all processes in the group, including the root process

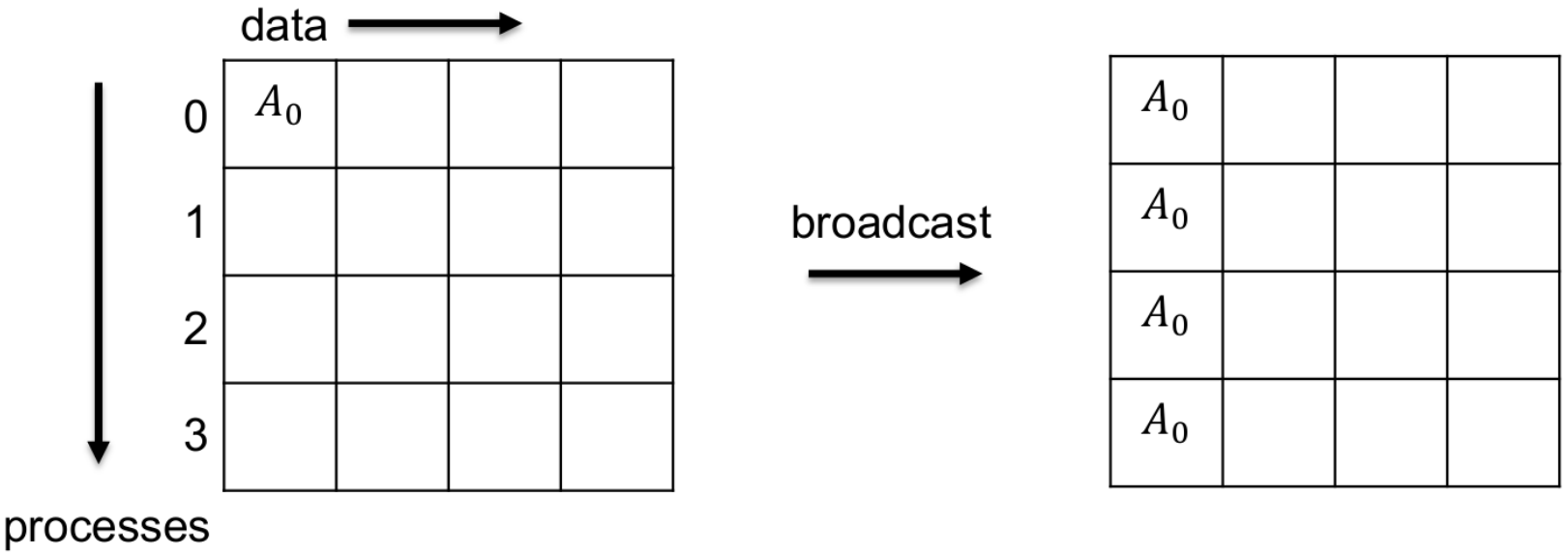
Fortran

```
MPI_Bcast(buffer, count, datatype, root, comm, ierr)
```

C

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm)
```

Broadcast – Effect



Example – Hello World (Fortran Version)

```
program hello
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT (ierror)
call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierror)
tag = 42

if (rank .eq. 0) then
  message = 'Hello, world'
  do i=1, size-1
    call MPI_SEND(message, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, ierror)
  enddo
else
  call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, ierror)
endif

print*, 'node', rank, ':', message
call MPI_FINALIZE (ierror)
end
```

Example – Hello World (Fortran Version Bcast)

```
program hello
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT (ierror)
call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierror)
tag = 42

if (rank .eq. 0) then
  message = 'Hello, world'
endif

call MPI_BCAST(message, 12, MPI_CHARACTER, root, MPI_COMM_WORLD, ierror)


print*, 'node', rank, ':', message
call MPI_FINALIZE (ierror)
end
```

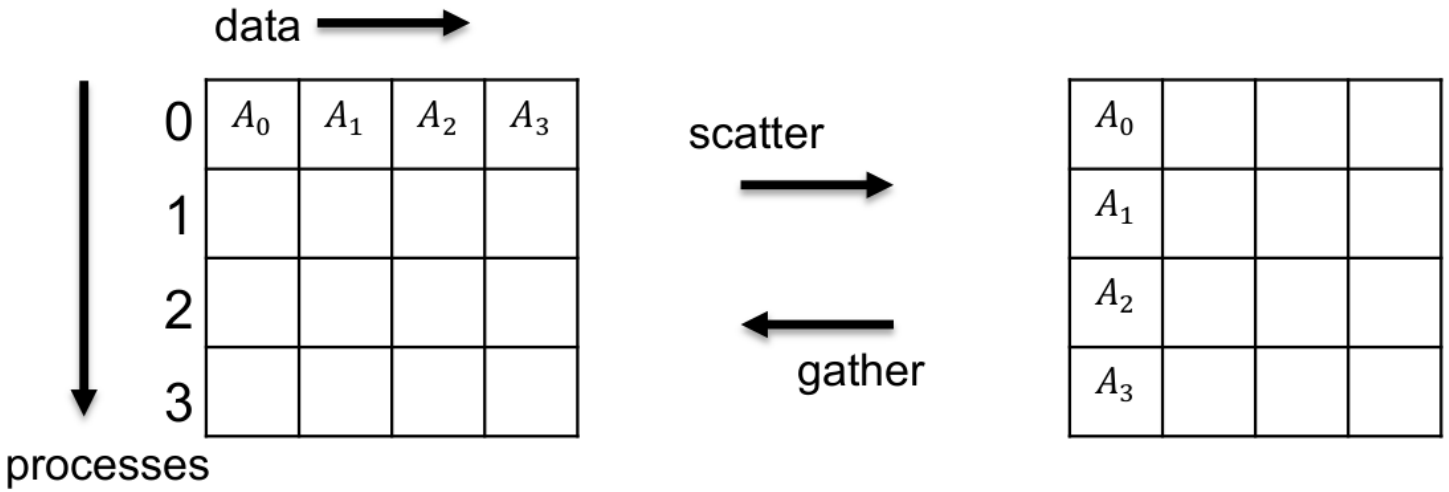
Example – Hello World (C Version)

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 42;

  if (rank == 0) {
    strcpy(message, "Hello world!");
    for (i = 1; i < size; i++) {
      rc = MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
  }
  else {
    rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  }
  printf("rank %d : %.13s\n", rank, message);
  rc = MPI_Finalize();
}
```

Example – Hello World (C Version Bcast)

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 42;

  if (rank == 0) {
    strcpy(message, "Hello world!");
    }

    rc = MPI_Bcast(message, 13, MPI_CHAR, 0, MPI_COMM_WORLD);

  printf("rank %d : %.13s\n", rank, message);
  rc = MPI_Finalize();
}
```

Gather and Scatter



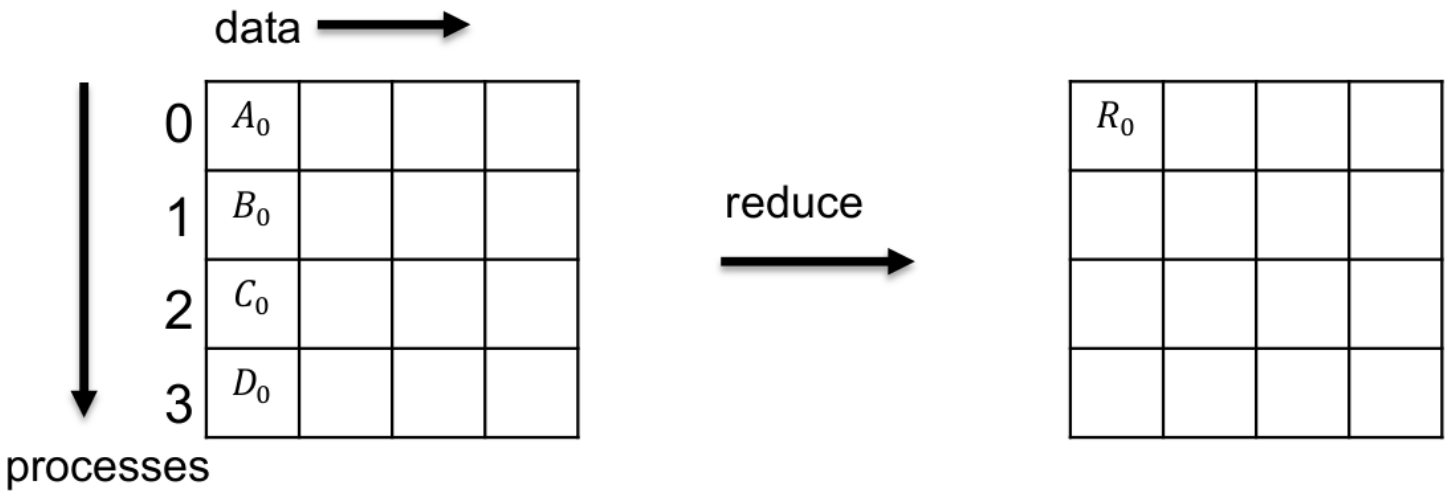
- Purpose of Gather
  - Each process sends the content of its send buffer to the root process
  - The root process receives the messages and stores them in rank order
  - Our Mandelbrot program is a typical instance of a gather operation
- Purpose of Scatter
  - The root process splits a buffer of data into chunks, then sends one chunk to each process in the group
  - Reverse of the gather operation

Global Computation Routines

- Global communication routines that include a computation
- The computation function can be either
  - An MPI predefined routine or
  - A user-supplied function
- Two types of global computation routines
  - Reduce
  - Scan

Reduce

- Combine elements of the input buffer of each process using a specified operation
- Return result to root process



Reduce

Fortran

```
MPI_Reduce(sbuf, rbuf, count, datatype, op, root,
comm, ierr)
```

C

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
comm)
```

Question: Result will be available on root, how can we distribute it to all processes?

- A very common operation in scientific computing e.g. dot products
  - Combined reduction + broadcast (optimised) `MPI_Allreduce`

Predefined Operators

Name	Meaning
MPI_MAX	Maximum value
MPI_MIN	Minimum value
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor
MPI_BXOR	Bit-wise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Min value and location

Performance Considerations

- A great deal of hidden communication takes place within collective communication
- Performance depends greatly on the particular implementation of MPI
- Due to forced synchronisation, not always the best idea to use collective communication