Message Passing Interface

MPhil in Scientific Computing, 2022-2023

Maria Nikodemou

Centre for Scientific Computing
University of Cambridge

Outline

Suggested Reading

Books

- W. Gropp et al. *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1 (MIT Press, 1999) - thorough coverage of MPI
- Peter Pacheco, *An introduction to parallel programming* (Elsevier, 2011) - introductory
- R. Robey and Y. Zamora, *Parallel and High Performance Computing* (Simon and Schuster, 2021) - introductory

MPI Standard

- http://www.mpi-forum.org/docs/ - MPI Standard

About this course

Prerequisites

- Knowledge of C/C++ or Fortran
- Access to a multi-core computer

This course

- aims to show you how to safely implement MPI in your code
- covers a small part of MPI, sufficient for most applications
- provides MPI functions and variables in both C and Fortran, but most examples are given in C++ only
- is based upon material provided by Dr. Rutter, who previously lectured this course

Parallel programming for scientific computing

What is parallel computing

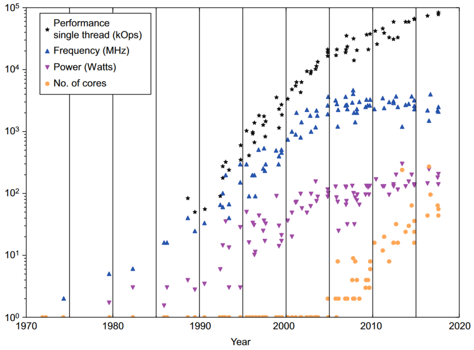
Parallel computing

The simultaneous execution of multiple operations

Can be achieved through

- process-based parallelisation (e.g. MPI)
- thread-based parallelisation (e.g. OpenMP)
- stream processing (e.g. CUDA)
- vectorisation

Why parallel?



Microprocessor trend data (Robey and Zamora, *Parallel and High Performance Computing*)

Before 2005,

- only single core CPUs existed
- clock frequency (instruction execution speed) kept increasing

From 2005 onward,

- clock frequency flattens out
- number of cores keep increasing

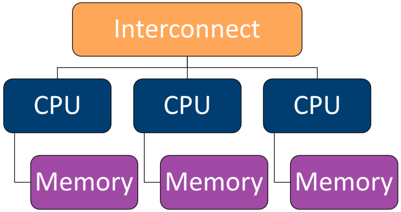
Optimal CPU performance is only achievable through parallel computing!

Benefits of parallel computing

- **Speedup**: reduction of run time by orders of magnitude
- **Bigger scales**: ability to reach problem dimensions not possible with serial codes
- **Energy efficiency**: reduction of power consumption (and costs) and extension of battery life

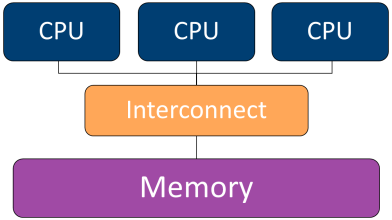
Distributed vs. Shared Memory Architecture

Distributed memory architecture



- each processor has its own local memory
- processors exchange messages with each other through a communication network

Shared memory architecture



- processors share the same address space
- processors have access to the pool of shared memory

Terminology

Thread

A separate instruction pathway through a process

Process

An independent unit of computation that has ownership of a portion of memory and control over resources in user space

Rank

A unique identifier (usually an integer) to distinguish the individual process within the set of processes

Core

The basic element of the system that does the mathematical and logical operations

Computer processing unit (CPU), or processor

The discrete processing device composed of one or more computational cores that is placed on the socket of a circuit board to provide the main computational operations

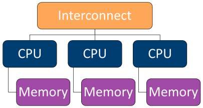
Node

A basic building block of a compute cluster with its own memory (accessible by all its processors) and a network to communicate with other compute nodes

Definitions from Robey and Zamora, *Parallel and High Performance Computing*

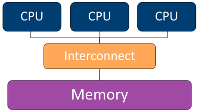
Distributed vs. Shared Memory Architecture

Distributed memory architecture



+ Scalable: limitless ability to incorporate more nodes
– Needs more thought:
  - data distribution among the different memory regions
  - keeping process communication to a minimum

Shared memory architecture



+ Simplified programming
– Need for memory access synchronisation to avoid potential memory conflicts
– Limited scalability:
  - adding more CPUs does not increase the amount of memory available
  - large core-count machines are rare and expensive
– Required process communication is hidden at the hardware level (so less obvious to tune)

Process vs. Thread based parallelisation

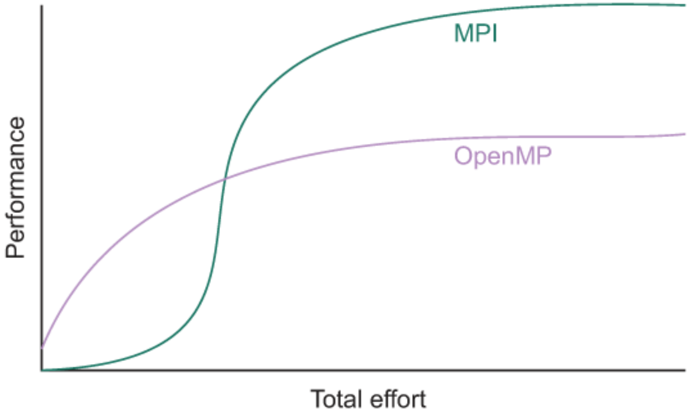
Process-based parallelisation

- Developed for distributed memory architectures
- The program creates separate processes, or *ranks*, with their own memory space
- Messages are used to move data between processes explicitly
- The OS distributes the processes on the processors and can move these during run time
- The **Message Passing Interface** (MPI) dominates process-based parallel programming

Thread-based parallelisation

- Developed for shared memory architectures
- The program creates separate instruction paths, or *threads*, within the same process
- The process memory can be easily shared between threads
- The OS decides where to place the threads
- The **OpenMP** standard dominates thread-based parallel programming

Process vs. Thread based parallelisation



Conceptual visualization of the programming effort required to improve performance using either MPI or OpenMP (Robey and Zamora, *Parallel and High Performance Computing*)

Message Passing Interface

What is MPI?

Message Passing Interface (MPI)

A **message-passing library interface specification**, designed for running jobs of multiple processes on multi-core computers and on multiple computers.

- MPI is a library, not a language
- It addresses the message-passing parallel programming model
- It provides the appropriate functions, subroutines or methods to move data from the address space of one process to that of another process
- As such, MPI can be used to parallelise existing, or new code

History of MPI

May 1994	**Version 1.0**: the first MPI standard, with participation from over 40 vendor-neutral organisations internationally. The goal was to develop a widely used standard for writing message-passing programs
Jul 1997	**Version 2.0**: corrections and extensions to MPI-1.0
Sep 2012	**Version 3.0**: significant extensions to MPI functionality
Jun 2021	**Version 4.0**: major update with addition of new features

Language Bindings

MPI is **not** a language. MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings:

- MPI was originally developed for Fortran and C languages
- Fortran:
  - originally, MPI-1.1 provided bindings for F77
  - MPI can still be used with most Fortran 77 compilers
  - now, MPI bindings are for Fortran 90 or later
- C++ programmers can use MPI through C bindings
  - C++ bindings were introduced in MPI-2.0 but deprecated in MPI-2.2 and removed in MPI-3.0 - these are not recommended

MPI implementations

- MPI is a specification, not an implementation
- Various implementations of the MPI standard exist
  - offered by parallel computer vendors for their machines, or
  - free, publicly available on the Internet
- The two major open source implementations of MPI are
  - **OpenMPI**: www.open-mpi.org
  - **MPICH**: www.mpich.org
- OpenMPI is the more widely used free implementation
- Many commercial MPIs are based on MPICH
- These will conform to the MPI standard, but may differ in matters not specified by the standard, like
  - the initial loading of the executables onto the parallel machine
  - which processes are allowed to execute input/output tasks
- It is very important to **use the same MPI implementation** at compile time, link time and run time!

MPI: The basics

Compiling an MPI program

- MPI is a library, not a language
- No special compiler or accommodations for the OS are required
- You can compile code as usual, with the use of flags to link in the MPI library
- Or use a **compiler wrapper** which ensures that all required libraries and options are properly applied:

Language	Wrapper compiler name
C	`mpicc`
C++	`mpicxx`, `mpiCC`, `mpic++`
Fortran	`mpifort`, `mpif90`, `mpif77`

- For example,

```
$ mpicxx my_program.cpp -o my_program
```

- Other/more compiler wrappers are defined by different MPI vendors, so one should check the corresponding MPI library manual (e.g. `mpiicc`, `mpiicpc` and `mpiifort` for Intel MPI)

Running an MPI program

- To run an MPI program, use
  ```
  $ mpiexec -n <number of processes> ./my_program
  ```
- The startup command for MPI programs is not part of the MPI standard - it may vary between implementations
- The MPI Forum recommends the use of `mpiexec`
- Most implementations also support the syntax
  ```
  $ mpirun -np <number of processes> ./my_program
  ```
- You should generally use no more processes than the number of CPU cores in your machine.
- Note: if you use more processes than CPU cores, `mpiexec` may produce an error, depending on the MPI implementation. For example, in OpenMPI you can use the `–oversubscribe` option to get around this.

Code structure - Hello parallel world

helloWorld.cpp

```cpp
#include <iostream>
#include <mpi.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  std::cout << "Hello parallel world" << std::endl;

  MPI_Finalize();
  return 0;
}
```

helloWorld.f08

```fortran
program hello

  use mpi_f08

  call mpi_init()

  write(*,*) 'Hello parallel world'

  call mpi_finalize()

end program hello
```

- These are simply C++/Fortran codes, with the addition of a few lines of MPI-related code:
  - The header file `mpi.h` or module `mpi_f08` provides definitions of named constants and types, and function prototypes. Fortran: You may also see `use mpi` or `include 'mpif.h'` in place of `use mpi_f08`, but these are now discouraged by the standard
  - The call to `MPI_Init` for MPI initialisation
  - The call to `MPI_Finalize` for MPI completion

Hello parallel world

```
$ mpicxx helloWorld.cpp
$ mpiexec -n 4 ./a.out
Hello parallel world
Hello parallel world
Hello parallel world
Hello parallel world
```

```
$ mpifort helloWorld.f08
$ mpiexec -n 3 ./a.out
Hello parallel world
Hello parallel world
Hello parallel world
```

MPI syntax

- All identifiers defined by MPI start with the string `MPI_`.
- C/C++:
  - MPI function names and MPI-defined types: The first letter after `MPI_` is capitalised
  - MPI-defined macros and constants: All letters are capitalised
- To avoid confusion, do not use the prefix `MPI_` (or even `PMPI_`) in your variable and function names!
- In general, MPI functions in C/C++ return an `int`, and Fortran MPI routines have an `ierror` argument, which contains the error code. When the call is successful, this gives `MPI_SUCCESS` (which is equal to `0`)
  - Fortran: with the `mpi_f08` module, this argument is declared as optional
- The most basic MPI functions are `MPI_Init` and `MPI_Finalize`
- These should be called exactly once in any MPI program

MPI_Init(): Initialising MPI

```c
int MPI_Init(int* argc_p, char*** argv_p);
```

```fortran
MPI_Init(ierror)
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- This sets up the MPI environment, e.g. decides which process gets which rank, allocates storage for message buffers
- The call to `MPI_Init` should precede all other MPI function calls [1]
- This **does not** mark the point at which the program moves from begin serial to parallel!
- `mpiexec` launches the specified number of processes from the beginning, and this number is fixed for the duration of the MPI program's execution
- C/C++: `MPI_Init` accepts pointers to the `argc` and `argv` arguments of the `main` function, but also accepts `NULL`

[1] With a few exceptions, like `MPI_Get_version` and `MPI_Initialized`

MPI_Finalize(): Finalising MPI

```c
int MPI_Finalize(void);
```

```fortran
MPI_Finalize(ierror)
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- This terminates the MPI environment: it cleans up all MPI data-structures
- The call to MPI_Finalize should follow all other MPI function calls [2]
- Unless the program is aborted, all MPI processes should call MPI_Finalize
- MPI_Finalize must be called when all communications have completed - this is the programmer's responsibility

[2]With a few exceptions, like MPI_Get_version and MPI_Finalized

MPI_Get_version(): MPI standard version

```c
int MPI_Get_version(int *version, int *subversion);
```

```fortran
MPI_Get_version(version, subversion, ierror)
  INTEGER, INTENT(OUT) :: version, subversion
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- This is used to determine which version of the MPI Standard is in use
- The output is stored in the variables `version` and `subversion`
- So, for example `version` = 4 and `subversion` = 0 corresponds to MPI 4.0
- This is one of the few functions that can be called at any time in an MPI program - even before `MPI_Init` or after `MPI_Finalize`

Hello parallel world: size and rank

helloWorld_ranks.cpp

```cpp
#include <iostream>
#include <mpi.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  std::cout << "Hello parallel world, I am process "\
    << rank << " out of " << size << "\n";

  MPI_Finalize();
  return 0;
}
```

helloWorld_ranks.f08

```fortran
program hello

  use mpi_f08

  integer :: size, rank

  call mpi_init()
  call mpi_comm_size(mpi_comm_world, size)
  call mpi_comm_rank(mpi_comm_world, rank)

  write(*, 101) rank, size
101 format('Hello parallel world, I am process ', &
    I2, ' out of ', I2)

  call mpi_finalize()

end
```

Hello parallel world: size and rank

```
$ mpicxx helloWorld_ranks.cpp
$ mpiexec -n 8 ./a.out
Hello parallel world, I am process 2 out of 8
Hello parallel world, I am process 3 out of 8
Hello parallel world, I am process 4 out of 8
Hello parallel world, I am process 6 out of 8
Hello parallel world, I am process 7 out of 8
Hello parallel world, I am process 0 out of 8
Hello parallel world, I am process 1 out of 8
Hello parallel world, I am process 5 out of 8
```

- **Non-deterministic output**: processes compete for access to the shared output device, resulting in unpredictable output order
- This order might change each time you run the program
- We could make all other processes send their message to process 0, and process 0 print the output in process rank order

Communicators and `MPI_COMM_WORLD`

- MPI has the concept of **communicators**, to provide the appropriate scope for all communication operations
- Communicators can be thought as collections of processes that can send messages to each other
- All MPI communication function calls require a communicator argument
- `MPI_COMM_WORLD` is the **default communicator**, which consists of all the processes in the program - this is defined in `mpi.h` and `mpi_f08`
- It is possible to create user-defined communicators - this is more advanced, but useful for multiple levels of parallelism

Communicator size and rank

- **Communicator size:** the total number of processes in the communicator. This is stored in the variable `size` after calling:

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

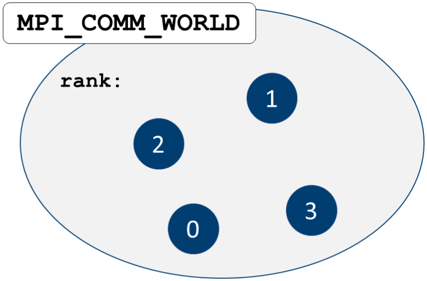
```
MPI_Comm_size(comm, size, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- **Process rank:** the unique identifier assigned to the individual process within the communicator. This is an integer between 0 and `size`−1, obtained in variable rank after calling:

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

```
MPI_Comm_rank(comm, rank, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: rank
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Communicator size and rank



size = 4

MPI_Abort(): Aborting MPI programs

```c
int MPI_Abort(MPI_Comm comm, int errorcode);
```

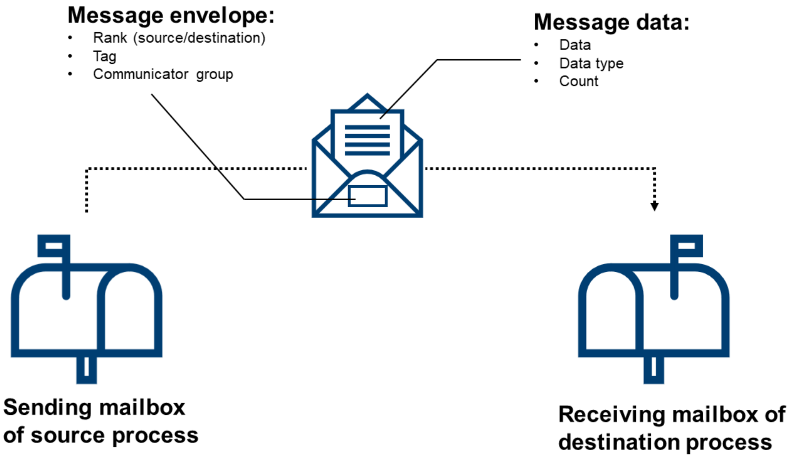
```fortran
MPI_Abort(comm, errorcode, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: errorcode
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- To abort all processes in an MPI program, run this with MPI_COMM_WORLD as the comm argument
- the errorcode is returned from the main program to the invoking environment
- For example, the following will abort because of too few processes:

```cpp
if ((rank == 0) && (nproc < 2)){
  std::cerr << "Too few processes\n" << std::endl;
  MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Point-to-Point Communication

Send/Receive message structure


Message envelope:
- Rank (source/destination)
- Tag
- Communicator group
Message data:
- Data
- Data type
- Count
Sending mailbox of source process
Receiving mailbox of destination process

Send/Receive message structure

- Passing a message from one process to another involves two operations:
  - **Sending** the item from the source process
  - **Receiving** the item from the destination process
- At each operation, the following is specified
  - the **pointer to memory** where the message can be found, or will be stored
  - the message **size**
  - the message **data type**
  - the **rank** of the source/destination process
  - the **tag** (a non-negative integer used to distinguish messages)
  - the **communicator** within which the message is sent

MPI_Send(): Send operation

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

```fortran
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- buf: initial address of send buffer
- count: number of elements in send buffer
- datatype: datatype of each send buffer element
- dest: rank of destination
- tag: message tag
- comm: communicator

MPI_Recv(): Receive operation

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

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
  TYPE(*), DIMENSION(..) :: buf
  INTEGER, INTENT(IN) :: count, source, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- buf: initial address of receive buffer
- count: number of elements in receive buffer
- datatype: datatype of each receive buffer element
- dest: rank of source
- tag: message tag
- comm: communicator
- status: status object

Example: One-way communication

oneWay_comm.cpp

```cpp
#include <iostream>
#include <mpi.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Status status;

  double x;
  if (rank == 0)
  {
    x = 10.0;
    MPI_Send(&x, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1)
  {
    MPI_Recv(&x, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,\
    status);
    std::cout << "Rank 1 has received x = " << x << \
    " from rank 0" << std::endl;
  }

  MPI_Finalize();
  return 0;
}
```

oneWay_comm.f08

```fortran
program hello

  use mpi_f08

  integer :: rank
  real :: x
  type(MPI_Status) :: status

  call mpi_init()
  call mpi_comm_rank(mpi_comm_world, rank)

  if (rank .eq. 0) then
    x = 10.0
    call MPI_Send(x, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD)
  else if (rank .eq. 1) then
    call MPI_Recv(x, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD,&
        status)
  write(*, 101) x
101 format('Rank 1 has received x = ', F4.1, &
    ' from rank 0')
  end if

  call mpi_finalize()

end
```

Send/Receive rules

- For the message to be successfully received, we need
  - the same communicator in both calls
  - the sending/receiving processes to match those specified in the receive/send calls, respectively
  - the same tag in both calls
  - the same data types in both calls
  - the receiving buffer size to be at least as big as the message size
- The tag should be between 0 and 32767, but need not be unique for each message

Send/Receive rules

- The wildcard `MPI_ANY_TAG` can be used in the receive call if an explicit number is not desired
- The wildcard `MPI_ANY_SOURCE` can be specified in the receive call to match data coming from any source within the specified communicator group
- No wildcards are available for `MPI_Send` - the sender must specify both a destination rank and a tag

Status

- A process can receive a message without knowing the message's sender, tag or size
- The above details are returned in the `status` argument of `MPI_Recv` and obtained using:
  - source process: `status.MPI_SOURCE` or `status%MPI_SOURCE`
  - message tag: `status.MPI_TAG` or `status%MPI_TAG`
  - the number of entries received (count of datatype, not bytes!): `MPI_Get_count(status, datatype, count)`
- If these details are not required, one may pass the predefined constant `MPI_STATUS_IGNORE` as an argument of the receive function

MPI data types

- It is important to always specify the correct datatypes for messages
- MPI provides a large set of predefined datatypes, including all basic datatypes in C and Fortran

MPI datatype	C datatype
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
MPI_C_BOOL	_Bool
MPI_LONG	signed long int

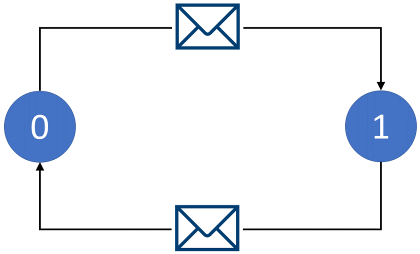
MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_CHARACTER	CHARACTER(1)
MPI_LOGICAL	LOGICAL

- If using C++, the MPI datatype for a bool is `MPI_CXX_BOOL`

Blocking Communication

- The MPI_Send and MPI_Recv are **blocking** calls
- MPI_Send only returns when the message data and envelope are safely stored away and the contents of the send buffer can be safely modified without affecting the data being sent
- The message might be copied directly into the matching receive buffer, or temporarily stored into a system buffer
- It is up to MPI to decide whether sent messages will be buffered or not (based on available buffer space and performance)
- Therefore, MPI_Send may return before or after the call to a matching receive
- MPI_Recv only returns when the message is received

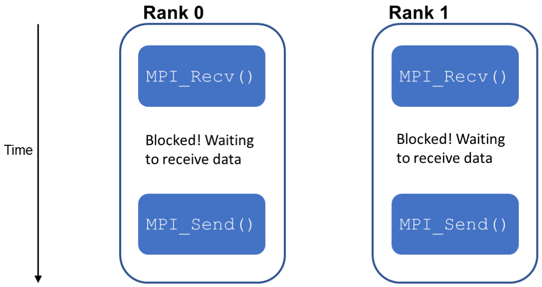
Message exchange between two processes



- Consider a simple program with two processes sending a message to each other
- Which operation should be called first?

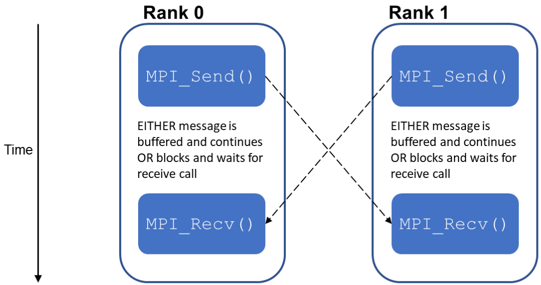
Receive first

- Both processes call `MPI_Recv` first
- In both processes, the send would not be called until after the receive completes
- But the receive cannot complete if a send is not called from the other process
- This causes the program to hang, also known as **deadlock**


Rank 0
Rank 1
Time
MPI_Recv()
Blocked! Waiting to receive data
MPI_Send()
MPI_Recv()
Blocked! Waiting to receive data
MPI_Send()

Send first

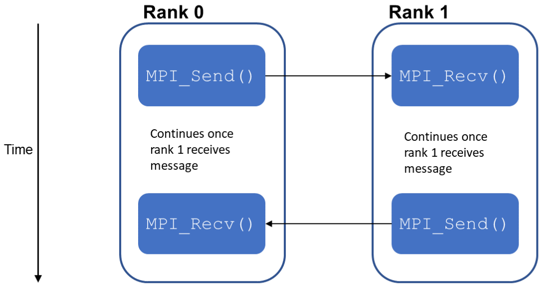
- Both processes call `MPI_Send` first
- MPI will either copy the message to a temporary buffer or wait for the receive call
- If none of the two messages is buffered (e.g. if the messages are large), the send calls will not complete and program will hang
- Therefore, calling the send function first is not safe either



Rank 0
Rank 1
Time
MPI_Send()
EITHER message is buffered and continues OR blocks and waits for receive call
MPI_Recv()
MPI_Send()
EITHER message is buffered and continues OR blocks and waits for receive call
MPI_Recv()

Alternate send/receive

- Rank 0 calls MPI_Send first and Rank 1 calls MPI_Recv first
- The first message arrives, allowing operation of the second send-receive pair
- The messages are safely passed and deadlock is avoided!



Rank 0
Rank 1
MPI_Send()
MPI_Recv()
Continues once rank 1 receives message
Continues once rank 1 receives message
MPI_Recv()
MPI_Send()
Time

Example: alternate send/receive

- Alternate send/receive can be implemented as follows:

```
if (rank == 0)
{
  MPI_Send(sendbuf, sendcount, sendtype, 1, sendtag, comm);
  MPI_Recv(recvbuf, recvcount, recvtype, 1, recvtag, comm, status);
}
else if (rank == 1)
{
  MPI_Recv(recvbuf, recvcount, recvtype, 0, recvtag, comm, status);
  MPI_Send(sendbuf, sendcount, sendtype, 0, sendtag, comm);
}
```

- This can get complicated when dealing with more processes and more complex communication
- One way round this: the `MPI_Sendrecv` function

MPI_Sendrecv(): Send and receive operations

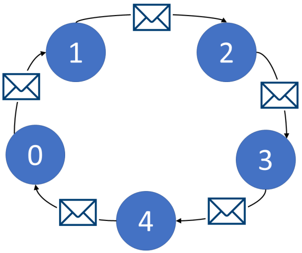
```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,\
    int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,\
    int source, int recvtag, MPI_Comm comm, MPI_Status *status);
```

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, &
    recvtype, source, recvtag, comm, status, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Executes a send-receive operation: sends a message to one destination and receives another message from another (or the same) process
- Both send and receive use the same communicator, but possibly different tags, counts and datatypes
- The send and receive buffers should be different
- MPI_Sendrecv effectively places the responsibility for correct communication execution to the MPI library, avoiding deadlocks

Send-Receive: circular chain

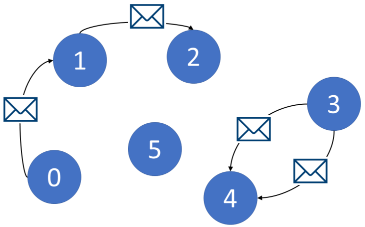
- We've seen how a call to MPI_Sendrecv can be used for communication between two processes
- This can be easily extended to larger circular chains, for example



```
MPI_Sendrecv(sendbuf, sendcount, sendtype, (rank + 1) % size, sendtag, recvbuf,\
    recvcount, recvtype, (rank + size - 1) % size, recvtag, comm, status);
```

Send-Receive: multiple chains

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount,\
    recvtype, source, recvtag, comm, status);
```



Rank	dest	source
0	1	MPI_PROC_NULL
1	2	0
2	MPI_PROC_NULL	1
3	4	4
4	3	3
5	MPI_PROC_NULL	MPI_PROC_NULL

- MPI_PROC_NULL is a dummy process
- When used as a destination or source argument in a send/receive call, then the communication has no effect - i.e. the call succeeds and returns as soon as possible without taking any action
- Example use: to simplify code for dealing with boundaries

MPI_Sendrecv_replace(): Send and receive with message replacement

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest,\
    int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status);
```

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,&
    comm, status, ierror)
  TYPE(*), DIMENSION(..) :: buf
  INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Executes a blocking send-receive operation, with the same buffer used for both the send and the receive
- So the message sent is replaced by the message received
- For example, if used in a circular chain, this will simply permute messages around

Example application: Estimating π

- Let's apply what we learnt so far to something more useful: calculating the value of π, using the Leibniz formula:

$$\pi = 4 \sum_{i=0}^{\infty} \left((-1)^i \frac{1}{2i+1} \right) = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots \right]$$

- We will truncate the infinite sum to some integer n:

$$\pi \approx 4 \sum_{i=0}^{n} \left((-1)^i \frac{1}{2i+1} \right) = S$$

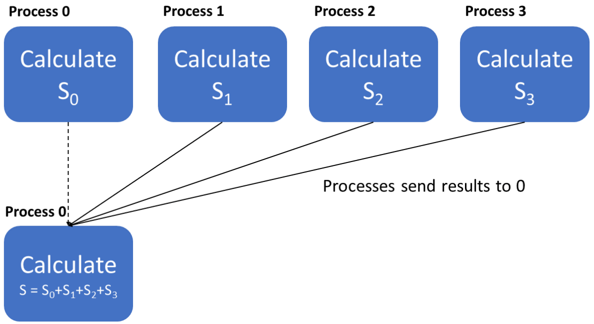
Summation code: serial

serial_sum.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>

int main(int argc, char *argv[])
{
  long n = std::stol(argv[1]);
  double sum = 0.0;
  for (long i = 0; i < n; i++)
  {
    double recip = 1.0/(2.0 * i + 1.0);
    if (i%2 == 0)
    {
      sum += recip;
    }
    else
    {
      sum -= recip;
    }
  }
  sum *= 4.0;

  std::cout << std::setprecision(10);
  std::cout << "Pi ~ " << sum << " with error " << M_PI - sum << std::endl;
  return 0;
}
```

Parallelisation

- Increasing n gives greater accuracy in our approximation
- Each term in the sum is independent of the others, so we can parallelise
- A possible way to parallelise with nproc processes:
    - Split the total sum, S, into nproc sums: $S = S_0 + S_1 + ... + S_{\mathtt{nproc}-1}$
    - Process i calculates S_i and sends it to process 0
    - Process 0 computes S by adding the S_i together



Summation code: parallel

parallel_sum.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <mpi.h>

int main(int argc, char *argv[])
{
MPI_Init(argc, argv);

long n = std::stol(argv[1]);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

double sum = 0.0;
long i_start = n * rank / size;
long i_end = n * (rank + 1) / size;

for (long i = i_start; i < i_end; i++)
{
  double recip = 1.0/(2.0 * i + 1.0);
  if (i%2 == 0)
  {
    sum += recip;
  }
  else
  {
    sum -= recip;
  }
}
```

```cpp
  //Continued code
  if (rank != 0)
  {
    //Send result to rank 0 (tag=0)
    MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }
  else
  {
    double global_sum = sum;
    for (int i = 1; i < size; i++)
    {
      double local_sum;
      //Receive results from all other ranks (tag=0)
      MPI_Recv(&local_sum, 1, MPI_DOUBLE,\
      MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,\
      MPI_STATUS_IGNORE);

      //Update global sum
      global_sum += local_sum;
    }
    global_sum *= 4.0;

    std::cout << std::setprecision(10);
    std::cout << "Pi ~ " << global_sum << " with error "\
        << M_PI - global_sum << std::endl;
  }

  MPI_Finalize();

  return 0;
}
```

Results

```
$ g++ serial_sum.cpp -o serial_sum
$ ./serial_sum 10
Pi ~ 3.041839619 with error 0.09975303466
$ ./serial_sum 10000
Pi ~ 3.141492654 with error 9.999999976e-05
$ ./serial_sum 10000000
Pi ~ 3.141592554 with error 1.000000016e-07

$ mpicxx parallel_sum.cpp -o parallel_sum
$ mpiexec -n 4 parallel_sum 10
Pi ~ 3.041839619 with error 0.09975303466
$ mpiexec -n 4 parallel_sum 10000
Pi ~ 3.141492654 with error 9.999999975e-05
$ mpiexec -n 4 parallel_sum 10000000
Pi ~ 3.141592554 with error 1.000000505e-07
```

Note

We get slightly different errors when running on more processes. This is expected. It is because the order of floating point summation changes when introducing more processes

Summation code performance

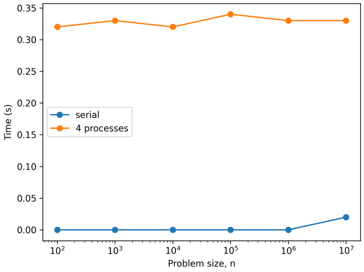
- The simplest way to time our program is to use the Unix shell command `time`
- This outputs the time taken to run a program from start to finish.

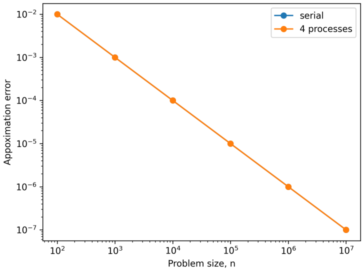
```
$ time ./serial_sum 10000000
Pi ~ 3.141592554 with error 1.000000016e-07
real    0m0.028s
user    0m0.026s
sys     0m0.000s
```

- tells us that the program runs in 28ms
- Timing a command only once may give misleading results. One could use `multitime` instead:

```
$ multitime -n 2 ./serial_sum 10000000
Pi ~ 3.141592554 with error 1.000000016e-07
Pi ~ 3.141592554 with error 1.000000016e-07
===> multitime results
1: ./serial_sum 10000000
Mean        Std.Dev.    Min         Median      Max
real        0.029       0.002       0.027       0.029       0.031
user        0.028       0.002       0.027       0.028       0.030
sys         0.000       0.000       0.000       0.000       0.000
```

Serial vs. parallel

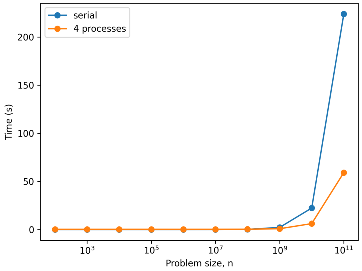


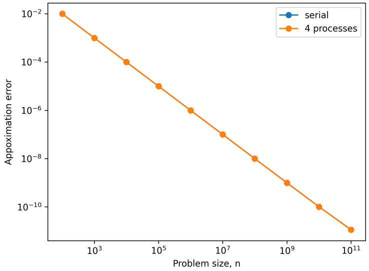


- It appears to take longer to run in parallel (with 4 processes) than serially! Why?
- Because of the **parallel overhead**, typically coming from communication

$$T_{parallel}(n, p) = T_{serial}(n)/p + T_{overhead}(n, p)$$

Serial vs. parallel

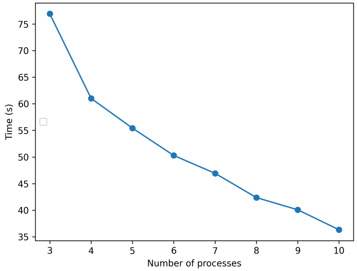




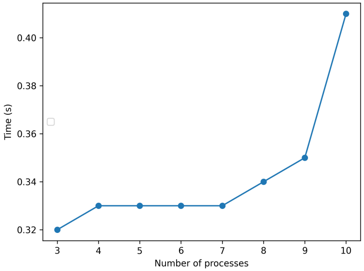
- The efforts of parallelisation only pay off when the problem size is big enough
- The parallel overhead does increase with the problem size, but at a much slower rate compared to the execution time
- For $n = 10^{11}$, we get a speed up $S(n,p) = \frac{T_{serial}(n)}{T_{parallel}(n,p)}$ of ~ 3.8

Number of processes

$n = 10^{11}$

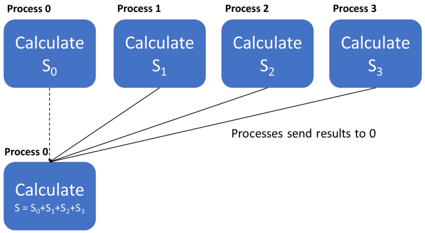


$n = 10^{4}$



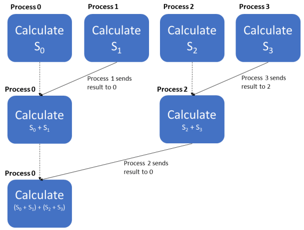
- For a large problem size, run time decreases with more processes, as one would expect
- This is not the case for a smaller problem size. When using more processes, the communication overheads become increasingly significant and cause an overall increase in run time

Simple structure



- In our algorithm to estimate π, each process calculated a value which was then sent to process 0 to compute the global sum
- While process 0 is calculating this sum, the other processes are doing nothing
- We could try to distribute the workload among the processes more evenly

Tree-structure



- Iterations of odd-positioned processes sending their results to even-positioned processes to do the sum
- Assuming that the message-passing and calculations occur simultaneously at each stage, the scheme requires 2 receives and 2 additions (compared with 3 receives and 3 additions in previous scheme)
- If using 16 processes, this improves the original scheme by a factor of 4

Which communication structure?

- A tree-like communication structure is much harder to implement than the original, simple structure
- There are many other possible candidates
- The best structure might depend on the number of processes we use, or even worse, the problem we're solving
- How do we choose the optimal one?
- We don't have to: MPI's collective communication functions come to the rescue

Collective communication

Collective communication

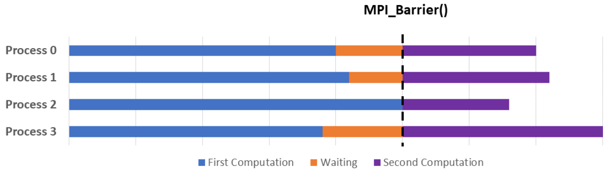
- Collective communication refers to communication that involves a group or groups of processes
- MPI provides a number of collective functions
- Collective functions involve **all** processes within a communicator
- They can do two kinds of operations:
  - Data distribution, e.g. broadcasting a message to all processes
  - Collective computation, e.g. calculating the minimum, maximum, sum etc.
- These effectively remove the responsibility of optimising communication from the programmer and place it on the MPI implementation developer
- They should be used in place of point-to-point communication, whenever possible

MPI_Barrier(): Process synchronisation

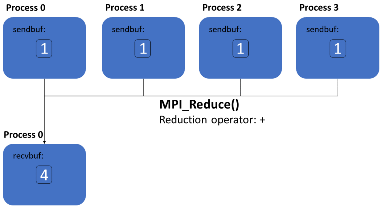
```c
int MPI_Barrier(MPI_Comm comm);
```

```fortran
MPI_Barrier(comm, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- **Barrier** is the simplest collective operation: it synchronises the processes within the communicator, comm
- No process can return from the call to `MPI_Barrier` before all processes have entered
- This is the only collective operation which guarantees synchronisation!
- Note: this does not guarantee that all processes will exit the call at the same time



Reduction operation



- **Reduction**: one of the most commonly used collective function
- It takes values from all processes within the communicator and combines them into a scalar result, according to the operator:

MPI reduction operator	Meaning
`MPI_MAX` / `MPI_MIN`	Maximum / Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND` / `MPI_LOR`	Logical AND / OR
`MPI_BAND` / `MPI_BOR`	Bitwise AND / OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
`MPI_MAXLOC` / `MPI_MINLOC`	Maximum / minimum and location

- It is also possible to define your own reduction operator

MPI_Reduce(): Reduction operation

```c
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype\
    datatype, MPI_Op op, int root, MPI_Comm comm);
```

```fortran
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- The send and receive buffers must be different to avoid **aliasing**
- The receive buffer is only relevant in the root (destination) process - it is a dummy buffer in all other processes
- By setting count > 1, MPI_Reduce effectively operates on arrays, instead of scalars

Estimating π - revisited

parallel_reduce_sum.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <mpi.h>

int main(int argc, char *argv[])
{
  MPI_Init(NULL, NULL);

  long n = std::stol(argv[1]);

  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  double sum = 0.0;
  long i_start = n * rank / size;
  long i_end = n * (rank + 1) / size;

  for (long i = i_start; i < i_end; i++)
  {
    double recip = 1.0/(2.0 * i + 1.0);
    if (i%2 == 0)
    {
      sum += recip;
    }
    else
    {
      sum -= recip;
    }
  }
```

```cpp
  //Continued code
  double global_sum;
  MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE,\
      MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank == 0)
  {
    global_sum *= 4.0;
    std::cout << std::setprecision(10);
    std::cout << "Pi ~ " << global_sum << " with error "\
        << M_PI - global_sum << std::endl;
  }

  MPI_Finalize();

  return 0;
}
```

Collective communication rules

- **All** processes in the communicator must call the same collective function - trying to match an MPI_Reduce with an MPI_Recv is erroneous
- The calls to the collective functions must have compatible arguments - e.g. same destination process, datatype, count, reduction operation
- When a collective function returns the output to a single process, all other processes must still pass a (dummy) receive buffer argument - this could be just NULL
- Collective communications don't use the concept of a tag - they are matched **according to the order in which they are called**

Order of collective calls matters

Process 0

```c
int a = 1, b = 2;
int c, d;
MPI_Reduce(&a, &c, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&b, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Process 1

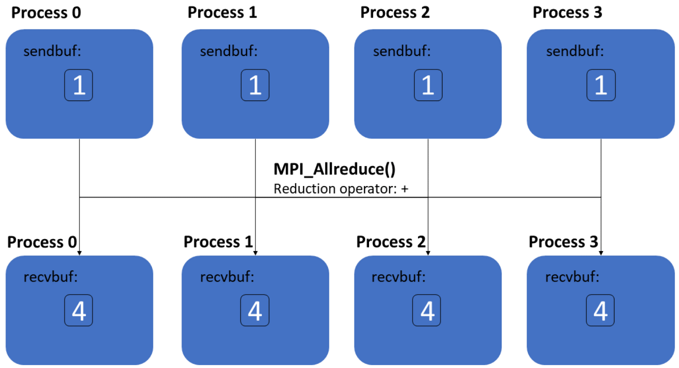
```c
int a = 1, b = 2;
int c, d;
MPI_Reduce(&b, &d, ...);
MPI_Reduce(&a, &c, ...);
```

Process 2

```c
int a = 1, b = 2;
int c, d;
MPI_Reduce(&a, &c, ...);
MPI_Reduce(&b, &d, ...);
```

- We have two summation reductions:
  - `MPI_Reduce(&a, &c, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`
  - `MPI_Reduce(&b, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`
- One would expect the reductions to sum variables a and b and store the results in variable c and d of rank 0
- **But** we would only get this outcome if **all processes called these functions in the same order**
- **Whereas now**:
  - The first collective reduction operator adds variables a of ranks 0 and 2 with variable b of rank 1 and stores in rank 0: `c = 1 + 2 + 1`
  - The second collective reduction operator adds variables b of ranks 0 and 2 with variable a of rank 1 and stores in rank 0: `d = 2 + 1 + 2`

Reduction operation with result sent to all processes



MPI_Allreduce(): Reduction to all processes

```c
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,\
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

```fortran
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Like the reduction operation, but now the result is returned to all processes in the group (hence the omission of a root process argument)

MPI_Wtime(): Taking timings

- In our previous example, we used the Unix shell command time to get timings for our program
- It is usually more helpful to time certain parts of our code, excluding time taken for input/output
- We are generally interested in **wall-clock time** (total elapsed time), rather than **CPU time**: the latter does not include idle time (e.g. a MPI_Recv waiting for a matching MPI_Send)
- MPI defines a convenient timer, MPI_Wtime:

```
double MPI_Wtime(void);
```

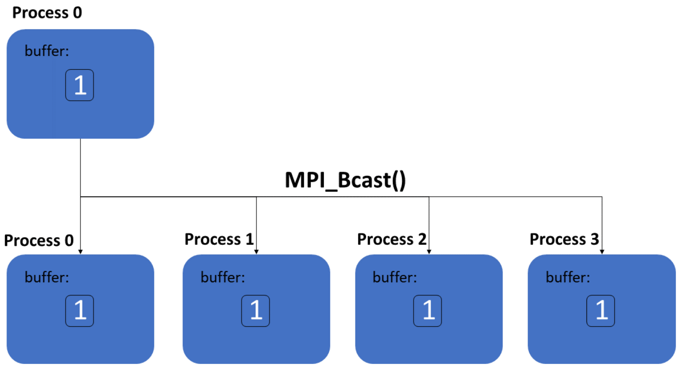
```
DOUBLE PRECISION MPI_Wtime()
```

- MPI_Wtime returns the time in seconds since some origin point in the past
- This origin is guaranteed to be constant within a process, but need not be the same for each process
- The function MPI_Wtick() returns the precision of MPI_Wtime in seconds (depends on the hardware)

Example: synchronised timing

```
// Perform some computations

double start, end, elapsed, max, min, avg;

// Barrier makes sure all processes have finished earlier computation
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

// Perform computation we want to time

end = MPI_Wtime();
elapsed = end - start;

// Reductions to get minimum, maximum and average timings
MPI_Reduce(&elapsed, &max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&elapsed, &min, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&elapsed, &avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0)
{
  std::cout << "Maximum elapsed time: " << max << " s\nMinimum elapsed time: " <<\
      min << " s\nAverage elapsed time: " << avg / size << " s" << std::endl;
}
```

Broadcast operation



- **Broadcast** operation: Sends a message from a root process to all processes of the group including itself

MPI_Bcast(): Broadcast operation

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

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
  TYPE(*), DIMENSION(..) :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- The root process sends the contents in `buffer` to all processes in the communicator `comm`
- By using a count greater than 1, we can send arrays of data

Example: User input

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin
- This avoids having to decide which process gets which part of the input data

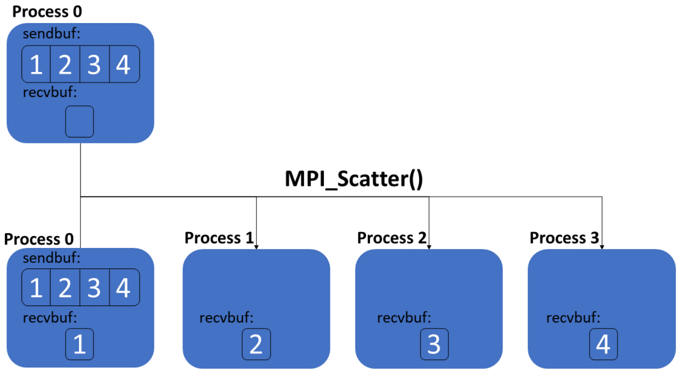
```
double x;
int n;
if (rank == 0)
{
  //Read user input
  std::cout << "Enter x: " << std::endl;
  std::cin >> x;
  std::cout << "Enter n: " << std::endl;
  std::cin >> n;
}

//Broadcast x and n to all processes
MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

//Perform calculation using x and n
```

Example: Input file reading

```c
int file_size;
char *input_string

if (rank == 0)
{
  //Open file and get its size
  getFileAndSize(input_string, &file_size);
}

//Broadcast file size (count of chars)
MPI_Bcast(&file_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (rank != 0)
{
  //Allocate enough memory to input buffers
  input_string = (char*) malloc(file_size * sizeof(char));
}

MPI_Bcast(input_string, input_size, MPI_CHAR, 0, MPI_COMM_WORLD);
```

- This technique should only be done for small input files
- MPI provides other ways for parallel input/output file operations (outside of this course's scope)

Scatter operation



- **Scatter** operation: distributes array blocks among the processes in the communicator, in the order of their ranks
- In the above example, the array blocks have size 1

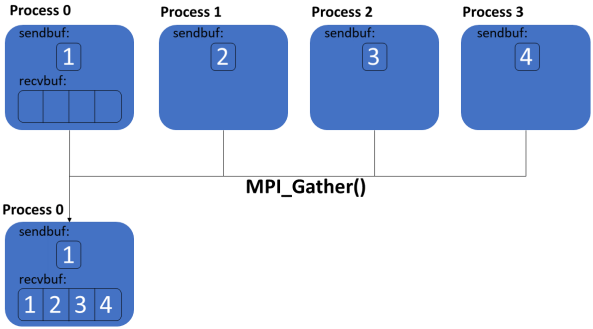
MPI_Scatter(): Scatter operation

```c
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,\
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```fortran
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,  root, &
    comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount, root
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- This sends the first block of sendcount elements in sendbuf of the root process to process 0, the next block of sendcount to process 1, etc. - i.e. we are scattering sendcount $\times$ nproc elements in total
- The arguments sendbuf, sendcount and sendtype are only significant at the root
- On the root, the send and receive buffers should be different
- The receive count must match the corresponding send count on the root (unlike point-to-point communication)

Gather operation



- **Gather** operation is the opposite of scatter
- It brings together blocks of data from all processors in the group and stacks it into a single array in the root process, in the order of the sending processes' ranks
- Be cautious of the memory requirements on the root, especially when sending large amounts of data from each process

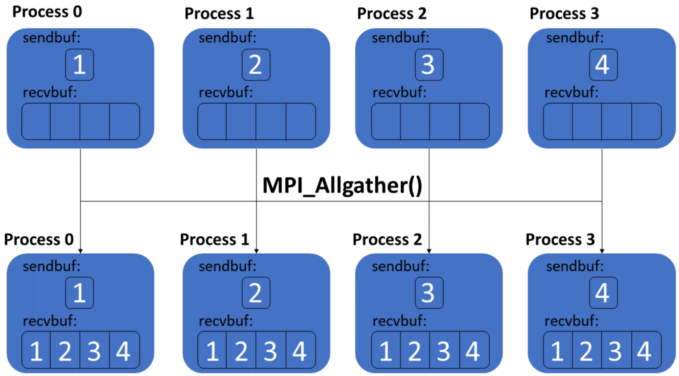
MPI_Gather(): Gather operation

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,\
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, &
    comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount, root
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- The blocks of sendcount elements in sendbuf of all processes in comm are concatenated and stored in the recvbuf of the root process
- The arguments recvbuf, recvcount and recvtype are only significant at the root
- On the root, the send and receive buffers should be different
- The send count must match the corresponding receive count on the root (unlike point-to-point communication)

Gather operation with result sent to all processes



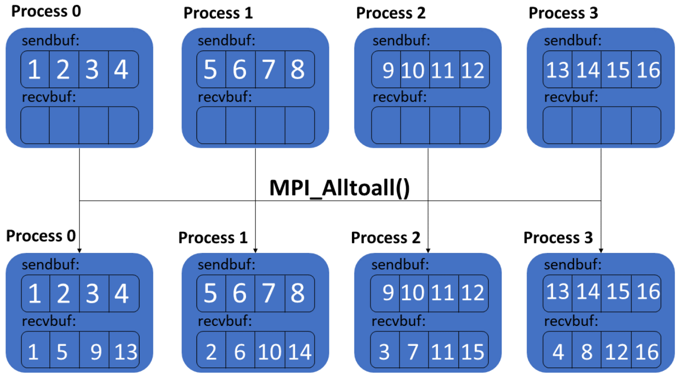
MPI_Allgather(): Gather to all processes

```c
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype,\
    sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,\
    MPI_Comm comm);
```

```fortran
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,&
    recvtype, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Like the gather operation, but now the result is returned to all processes in the group (hence the omission of a root process argument)

All-to-all operation



- The **All-to-All** operation is an extension to the all-gather operation, but now each process sends distinct data to each of the receivers
- Specifically, the j^{th} block of process i is placed as the i^{th} block in process j - like a matrix transpose operation

MPI_Alltoall(): All-to-all operation

```c
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype,\
    sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,\
    MPI_Comm comm);
```

```fortran
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,&
    comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- The send and receive buffers should be different to avoid aliasing
- The send and receive counts should be the same (unlike point-to-point communication)

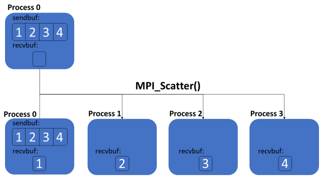
MPI_IN_PLACE: In-place communication

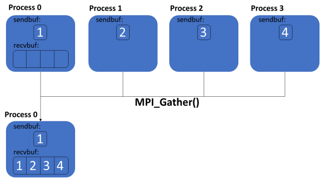
- Most of the collective communication functions introduced so far restrict the use of the same send and receive buffers to avoid aliasing
- We can still safely perform the operations in place, if we pass the special value `MPI_IN_PLACE` as one of the buffers
- For `MPI_Reduce` and `MPI_Gather`, `MPI_IN_PLACE` is used in the `sendbuf` argument at the root (or at all processes, if using `MPI_Allreduce`, `MPI_Allgather` or `MPI_Alltoall`)
- For `MPI_Scatter`, it is used in the `recvbuf` argument at the root
- For example, back to our code for estimating π, we could do the reduction in place, by replacing the `MPI_Reduce` call by:

```
if (rank == 0)
  MPI_Reduce(MPI_IN_PLACE, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
else
  MPI_Reduce(&sum, NULL, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Scatter/Gather count

- So far, when scattering or gathering data, we used the same count of data elements in each process

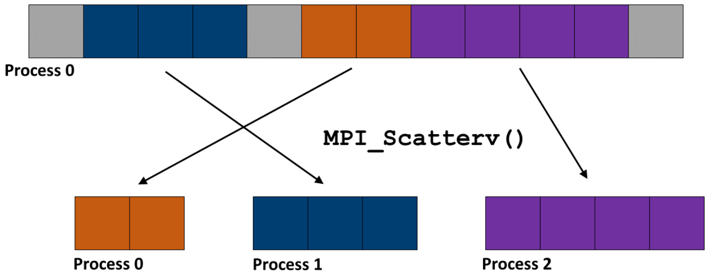




- What if we want to scatter or gather an array with size that's not divisible by the number of processes?
- MPI also defines the variants `MPI_Scatterv`, `MPI_Gatherv`, `MPI_Allgatherv` (and `MPI_Alltoallv`), which allow having a different send/receive count on each rank
- These are even more flexible, in that they allow the scattered/gathered block data to be non-contiguous in memory

Scatter operation with variable element count

```
sendcounts = [2, 3, 4]
displs = [5, 1, 7]
```



MPI_Scatterv(): Variable count scatter operation

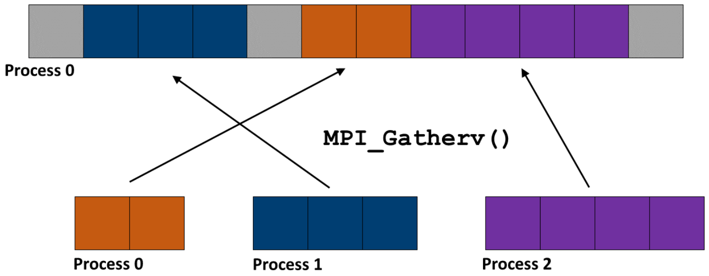
```
int MPI_Scatterv(const void *sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype,\
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- This extends the functionality of `MPI_Scatter` by allowing irregular message sizes to be sent - hence the sendcount argument of `MPI_Scatter` is now replaced by the array sendcounts
- The data can be sent in any order: the displacement (relative to sendbuf) of the block of data to be sent to process i, is the i^{th} entry of the array `displs`
- It is not allowed to read any data from the root more than once
- Fortran users: beware of conversion from array indices to ranks
- All other rules of `MPI_Scatter` also apply here

Gather operation with variable element count

```
recvcounts = [2, 3, 4]
displs = [5, 1, 7]
```



MPI_Gatherv(): Variable count gather operation

```c
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,\
    void *recvbuf, const int recvcounts[], const int displs[],\
    MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```fortran
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, &
    recvtype, root, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Does exactly the opposite of `MPI_Scatterv`
- Gathers a message of count `recvcounts[i]` from rank i and stores it into `recvbuf` of root, beginning at an offset of `displs[i]` elements
- The user needs to make sure that no location on the root is written more than once
- But the `recvbuf` need not be filled in a contiguous manner

MPI_Allgatherv(): Variable count gather to all processes

```c
int MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype\
    sendtype, void *recvbuf, const int recvcounts[], const int\
    displs[], MPI_Datatype recvtype, MPI_Comm comm);
```

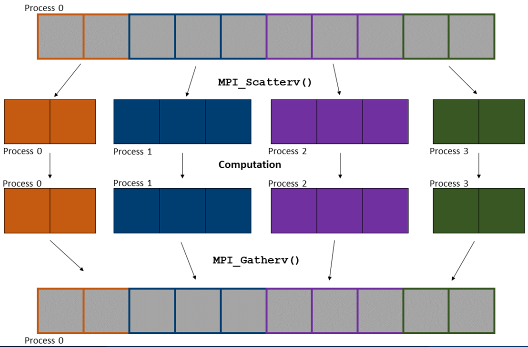
```fortran
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, &
    recvtype, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(*), DIMENSION(..) :: recvbuf
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Like MPI_Gatherv, but now all processes receive the result, instead of just the root

Example: Distributing and collecting data

Suppose

- We have a large array of data on rank 0 (potentially read from input file)
- We want to distribute this array (approximately) evenly to all processes
- Processes will perform some kind of computation on each element of the array
- And send their solutions back to rank 0 (e.g. to output results to file)



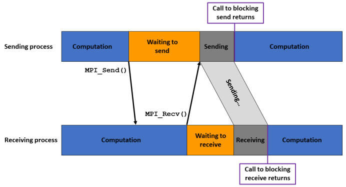
Example: Distributing and collecting data

```c
int arraylength = 100000;
double *global_array;
if (rank == 0)
{ // Set up data on rank 0 (e.g. by reading a file)
  getData(global_array);
}
// Compute the size of the array on every process
long istart = n * rank / size;
long iend = n * (rank + 1) / size;
int length = (int) (iend - istart);
// Get array sizes and displacements for communication
int lengths[size], displs[size];
MPI_Gather(&length, 1, MPI_INT, lengths, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0)
{
  displs[0] = 0;
  for (int i = 1; i < size; i++)
  {
    displs[i] = displs[i-1] + lengths[i-1];
  }
}
// Allocate enough memory to data buffers
double *array = (double*) malloc(length * sizeof(double));

// Distribute data from 0 to all processes
MPI_Scatterv(global_array, lengths, displs, MPI_DOUBLE, array, length, MPI_DOUBLE, 0, MPI_COMM_WORLD);

//Do some computation

// Collect updated array in rank 0
MPI_Gatherv(array, length, MPI_DOUBLE, global_array, lengths, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

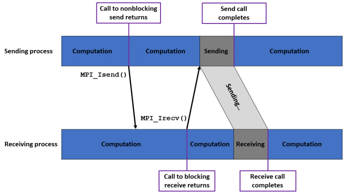
More Point-To-Point

Blocking and Non-blocking communication

- **Blocking communication**: send/receive operations which are guaranteed to only return after the corresponding buffers can be safely used



- **Non-blocking communication**: send/receive functions which start the corresponding operation but return immediately, without waiting for its completion (or freeing of buffers)



Blocking vs. Non-blocking communication

Blocking

+ Provides synchronisation: safe re-use of buffer is guaranteed with the return of the function call
− Can cause deadlocks
− The programmer should carefully order the send and receive calls - can be very difficult when using complex or dynamic communication patterns

Non-blocking

+ Avoids deadlocks
+ Allows different communication operations to overlap (e.g. preventing serialisation)
+ Allows communication and computation operations to overlap
− Requires explicit checking for communication completion (through MPI-defined routines) before re-using send/receive buffers

Sending communication modes

- To provide more flexibility, MPI defines four communication modes for sending messages:
  - **Standard mode**: After the send call, the message can be either temporarily stored to a local buffer, or wait for a matching receive to be posted. It is up to the MPI implementation to decide, based on performance and memory considerations
  - **Buffered mode** (B): If no matching receive is posted, the message is temporarily stored in some buffer space set by the programmer
  - **Synchronous mode** (S): The send call waits for a matching receive to be posted and sends the message to the receive buffer
  - **Ready mode** (R): The send call may be started only if the matching receive is already posted, otherwise the operation is erroneous

Communication modes

- All sending communication modes are available in both blocking and non-blocking form (this course covers the blocking and standard non-blocking sends)

	Communication mode	**MPI routine**
Blocking	Standard	`MPI_Send`
	Buffered	`MPI_Bsend`
	Synchronous	`MPI_Ssend`
	Ready	`MPI_Rsend`
Non-blocking	Standard	`MPI_ISend`
	Buffered	`MPI_Ibsend`
	Synchronous	`MPI_Issend`
	Ready	`MPI_Irsend`

- There are two receive operations:
  - `MPI_Recv` is the blocking receive
  - `MPI_Irecv` is the non-blocking receive
- All sending routines can be matched with either of the two receives

MPI_Bsend(): Blocking buffered send

```c
int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,\
    int tag, MPI_Comm comm);
```

```fortran
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Blocking send in buffered mode
- Can be used to avoid deadlocks
- But not ideal if memory is an issue
- Takes the same arguments as the standard send function `MPI_Send()`
- The programmer should specify some buffer space on the sending process using `MPI_Buffer_attach` before calling `MPI_Bsend`

MPI_Buffer_attach(): Setting a buffer

```
int MPI_Buffer_attach(void *buffer, int size);
```

```
MPI_Buffer_attach(buffer, size, ierror)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
  INTEGER, INTENT(IN) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Provides a `buffer`, of size `size` (in bytes) in the user's memory, to be used for buffer-mode outgoing messages
- If not specified, a zero-sized buffer is associated with the process
- Only one buffer can be attached to a process at a time
- So the size of the buffer should be big enough to cover all pending buffered sends from the process

MPI_Buffer_detach(): Removing a buffer

```c
int MPI_Buffer_detach(void *buffer_addr, int *size)
```

```fortran
MPI_Buffer_detach(buffer_addr, size, ierror)
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
  TYPE(C_PTR), INTENT(OUT) :: buffer_addr
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Detaches the buffer currently associated with this process
- Returns the buffer address and size
- Blocks until all messages in the buffer have been transmitted
- Once this call returns, the programmer can safely reuse or deallocate the space taken by the buffer

Buffer size

- The call to `MPI_Pack_size(count, datatype, comm, &data_size)` calculates an upper bound for the amount of space (in bytes) needed to pack a message, and returns this value in `data_size`
- The constant `MPI_BSEND_OVERHEAD` gives an upper bound on the space needed for additional information (e.g. tag, destination, communicator)

Example: buffered sends

- So, if we wanted to use a buffer-mode send to transmit an integer from rank 0 to all other processes, we could do the following:

```c
int comm_size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0)
{
  int a = 10; // integer to be sent

  // Calculate buffer size in bytes
  int data_size;
  MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &data_size);
  int buf_size = (comm_size - 1) * (data_size + MPI_BSEND_OVERHEAD);

  int* buf = (int*) malloc(buf_size);
  MPI_Buffer_attach(buf, buf_size);

  for (int i = 1; i < comm_size; i++)
  {
    MPI_Bsend(&a, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }

  //Do other work

  //When all sends are expected to have completed, detach and free the buffer
  MPI_Buffer_detach(&buf, &buf_size);
  free(buf);
}
```

MPI_Ssend(): Blocking synchronous send

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

```
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Blocking send in synchronous mode
- Takes the same arguments as MPI_Send()
- Can be used in place of standard sends to check whether a program is *safe*: i.e. if it does not depend on message buffering for its completion
- Safe programs are more portable because they don't depend on the the communication protocol used or the amount of buffer space available

MPI_Rsend(): Blocking ready send

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

```
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Blocking send in ready mode
- Requires careful programming! It should only be used when it's certain that the receive has already been posted, otherwise the operation is erroneous and the outcome is undefined
- A ready send could improve performance by saving the overhead of checking for a matching receive call
- Takes the same arguments as `MPI_Send()`

MPI_Isend(): Non-blocking send

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

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Initiates a non-blocking send in standard mode
- Arguments are as in the blocking sends, except from the final request argument
- MPI_Request is used as an identifier for the immediate send
- It can then be used in completion calls, to request information about the send operation status (before re-using the send buffer)

MPI_Irecv(): Non-blocking receive

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

```fortran
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, source, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Initiates a non-blocking receive
- Arguments are as in the blocking receive, except from the `status` argument which is now replaced by the `request` argument
- This can then be used in completion calls, to request information about the receive operation status (before using the receive buffer)

Communication completion operations

- With non-blocking communication, it is essential to ensure that communication has completed before re-using the send/receive buffers
- There are two types of completion testing functions:
  - **Wait** returns when the operation is complete - this is blocking: it waits until the operation is complete
  - **Test** returns a flag which is set to `true` if the operation has completed, or `false` otherwise - this is non-blocking: it returns immediately
- Both of these take the `request` argument passed in the non-blocking send/receive call to identify which operation to wait on or test

MPI_Wait(): Wait for completion

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

```
MPI_Wait(request, status, ierror)
  TYPE(MPI_Request), INTENT(INOUT) :: request
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Waits until the operation identified by `request` is complete
- Recall: a send/receive operation is complete when the send/receive buffers are safe to be re-used
- When the call returns, it also deallocates the request and sets `request` to `MPI_REQUEST_NULL` (a null request handle)
- If used for a receive operation, the call also returns information on the completed operation in `status` (see slide 42)

MPI_Test(): Test for completion

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

```fortran
MPI_Test(request, flag, status, ierror)
  TYPE(MPI_Request), INTENT(INOUT) :: request
  LOGICAL, INTENT(OUT) :: flag
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Tests to find out if the operation identified by `request` is complete
- If so, it returns `flag = true`, it deallocates the request and sets `request` to `MPI_REQUEST_NULL`
- Otherwise it returns `flag = false`
- If used for a receive operation, the call also returns information on the completed operation in `status` (see slide 42)

Example: Non-blocking point-to-point communication

```
if (rank == 0)
{
  MPI_Isend(a, 10, MPI_DOUBLE, 1, tag, comm, request);

  // Perform some other computation without altering variable a

  MPI_Wait(request, status);
  // Can now safely change variable a
}
else (if rank == 1)
{
  MPI_Irecv(a, 15, MPI_DOUBLE, 0, tag, comm, request);

  // Perform some other computation without using variable a

  MPI_Wait(request, status);
  // Can now safely read and use a
}
```

Multiple completions

- It is possible to test or wait for the completion of a number of non-blocking operations at the same time
- MPI defines variants of the `MPI_Wait` and `MPI_Test` for waiting/testing whether all/some/any of the operations in a given list have completed
- `MPI_Waitall`/`MPI_Testall`: Waits/tests for completion of **all** operations
- `MPI_Waitsome`/`MPI_Testsome`: Waits/tests for completion of **at least one** operation
- `MPI_Waitany`/`MPI_Testany`: Waits/tests for completion of **any one** of the listed operations

Requests

- It is good practice to initialise requests to `MPI_REQUEST_NULL`:
    - If a process accidentally calls a communication completion function with an uninitialised request handle, the outcome is undefined (possible a segfault)
    - But communication functions called with `MPI_REQUEST_NULL` are simply null operations
- If we want to test for completion without deallocating the request, we can use `MPI_Request_get_status(request, flag, status)`
- We can deallocate requests and set them to `MPI_REQUEST_NULL` by calling `MPI_Request_free(request)`
- **Caution**: This should only be used if we know communication has completed, for example if we called `MPI_Request_get_status` or if we match `MPI_Isend` with `MPI_Recv`. If an error occurs during a communication after the request object has been freed, the error code cannot be returned to the user

Message Probing

- Probing: MPI allows us to check for incoming messages, without actually receiving them
- This also provides information that is useful for knowing *how* to receive them (e.g. message type, count etc.)
- There is a blocking and non-blocking probing function, `MPI_Probe` and `MPI_Iprobe`

MPI_Probe(): Blocking probe

```c
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

```fortran
MPI_Probe(source, tag, comm, status, ierror)
  INTEGER, INTENT(IN) :: source, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- MPI_Probe waits until an incoming message with matching source, tag and communicator comm has been found and returns its status
- The returned status argument is the same value that would have been returned by MPI_Recv

MPI_Iprobe(): Non-blocking (immediate) probe

```c
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

```fortran
MPI_Iprobe(source, tag, comm, flag, status, ierror)
  INTEGER, INTENT(IN) :: source, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  LOGICAL, INTENT(OUT) :: flag
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- MPI_Iprobe is like MPI_Probe, but it is non-blocking
- It checks to see if there is an incoming message with matching source, tag and communicator comm and returns immediately
- If such a message can be found, it returns flag = true and fills the status argument, otherwise it returns flag = false and leaves status undefined

Error handling

Error handler

- So far, we've completely ignored the error code returned from MPI routines
- This is the return value of the MPI function in C/C++, or the `ierror` argument in Fortran
- The reason for this is that by default, MPI sets the error handler to `MPI_ERRORS_ARE_FATAL`, which means that when an error occurs, the program aborts
- If we want to know more about the error and let the program continue, we can reset the error handler to `MPI_ERRORS_RETURN` by using `MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);`
- This should be used with caution: if unstopped, errors could propagate and cause more errors further down

Error codes

- MPI defines `MPI_SUCCESS`, which is returned when the routine finishes with no error
- All other return codes are implementation-defined
- But MPI defines a small set of error classes to divide the error codes into categories depending on the type of the error, including:

Error class	**Meaning**
`MPI_SUCCESS`	No error
`MPI_ERR_BUFFER`	Invalid buffer pointer
`MPI_ERR_COUNT`	Invalid count argument
`MPI_ERR_TYPE`	Invalid datatype argument
`MPI_ERR_TAG`	Invalid tag argument
`MPI_ERR_COMM`	Invalid communicator
`MPI_ERR_RANK`	Invalid rank
`MPI_ERR_REQUEST`	Invalid request handle
`MPI_ERR_ROOT`	Invalid root
`MPI_ERR_OP`	Invalid operation

Retrieving errors

- The MPI routine `MPI_Error_class` converts each error code into the corresponding error class

```c
int MPI_Error_class(int errorcode, int *errorclass);
```

```fortran
MPI_Error_class(errorcode, errorclass, ierror)
  INTEGER, INTENT(IN) :: errorcode
  INTEGER, INTENT(OUT) :: errorclass
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- To obtain a human readable description of the error, we can use the function `MPI_Error_string`, where the output argument `resultlen` gives the length of the error string

```c
int MPI_Error_string(int errorcode, char *string, int *resultlen);
```

```fortran
MPI_Error_string(errorcode, string, resultlen, ierror)
  INTEGER, INTENT(IN) :: errorcode
  CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
  INTEGER, INTENT(OUT) :: resultlen
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Example: Error handling

```
// Change default behaviour of error handler, so that error codes are returned
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

// Error: send to an invalid rank of index -1
int errcode;
errcode = MPI_Send(&x, 1, MPI_DOUBLE, -1, 0, MPI_COMM_WORLD);
if (errcode != MPI_SUCCESS)
{
  /* The error code will either be MPI_SUCCESS, or some error code that falls
  under an error class, which we can retrieve */
  int errclass;
  MPI_Error_class(errcode, &errclass);
  if (errclass == MPI_ERR_RANK)
  {
    // The string must be at least MPI_MAX_ERROR_STRING characters long
    char errstring[MPI_MAX_ERROR_STRING];
    int resultlen;
    MPI_Error_string(errcode, errstring, &resultlen);
    std::cout << errstring << std::endl;

    // Then either find some fix and continue the application or call MPI_Abort
  }
}
```

Common mistakes

General mistakes

- **Mistake**: Doing things before `MPI_Init` or after `MPI_Finalize`

  Cause: The MPI standard does not say much about the region before `MPI_Init` or after `MPI_Finalize` - not even how many processes are running. It does require that rank 0 of `MPI_COMM_WORLD` returns from `MPI_Finalize`, but that's only if it hasn't been terminated before (e.g. aborted). Any other behaviour will be implementation dependent.

  Solution: If possible, avoid doing any tasks (other than specific MPI routines like `MPI_Get_version`) before initialising or after finalising MPI

- **Mistake**: Using a different MPI implementation in compile, link or run time

  Cause: Different implementations make different choices in the way they are written

  Effect: Link-time errors, MPI program failing to start, or multiple serial versions running instead of multiple communicating ones

Deadlock

Deadlock

When processes are waiting for an event that will never happen and the program hangs

- Mistake: Posted send and receive operations do not match

  Cause: This might be due to the operations having different tags, or datatypes, or the receive buffer being shorter than the message size, for example

  Effect: Deadlock

  Solution: Make sure all send-receive pairs follow the rules of slide 41
- Mistake: Calls to send and receive posted in wrong order

  Cause: For example, posting the receives first, on both processes

  Effect: Deadlock

  Solution: Use one of: Non-blocking operations, blocking buffered send, (`MPI_Bsend`), or `MPI_Sendrecv`

Mistakes using point-to-point communication

- **Mistake**: Not making programs safe
  Cause: We are relying on message buffering when sending with `MPI_Send`
  Effect: Program works for small message sizes, but will cause deadlock for larger messages
  Solution: Use one of: Non-blocking operations, blocking buffered send, (`MPI_Bsend`), or `MPI_Sendrecv`

Race condition

When two or more operations attempt to read/write the same location in memory at the same time, resulting in corrupted data

- **Mistake**: Re-using send/receive buffers of non-blocking operations before the send/receive is complete
  Cause: Non-blocking operations return immediately, without waiting for communication completion
  Effect: Results change if the program is run multiple times
  Solution: Use a communication completion testing routine, like `MPI_Wait` or `MPI_Test`

Mistakes using point-to-point communication

- Mistake: Assuming that the dealing of messages is fair

  Cause: Messages are non-overtaking, but non necessarily fair. If two messages are sent **from the same source to the same destination** and a receive operation matches both messages, the message posted first will be received first (the same applies for 2 receives matching the same message: the receive posted first will receive the message). **But** messages might be overtaken by matching messages coming from another process, even if they were sent later

  Effect: Wrong results if sends and receives are unintentionally matched, potentially different every time we run the program

  Solution: Avoid using `MPI_ANY_SOURCE` or `MPI_ANY_TAG` too much. If using non-blocking communication, one can use `MPI_Testsome` and handle all complete communications

Mistakes using collective communication

- **Mistake**: Not calling collective operations on all processes in the communicator, or calling them in different orders

  Effect: Deadlock, or wrong results

  Solution: Take care, especially when using `if` statements
- **Mistake**: Expecting all processes to return from collective calls at (roughly) the same time

  Solution: If it is essential that processes synchronise, use `MPI_Barrier`
- **Mistake**: Trying to match an `MPI_Bcast` with an `MPI_Recv`

  Cause: Even though acts as a multiple-send, `MPI_Bcast` is a collective operation

  Solution: Call `MPI_Bcast` on all processes in the communicator, with no call to `MPI_Recv`
- **Mistake**: Using the same input and output buffer (e.g. in `MPI_Reduce`, `MPI_Gather` etc)

  Cause: This is not allowed by the MPI standard, to avoid aliasing

  Effect: This may work in some cases, but may fail in others

  Solution: Use `MPI_IN_PLACE` (see slide 90)

Performance Tips

Performance tips

- Use collective, rather than point-to-point communication, where possible
- Instead of sending lots of small messages, it will be faster to send less messages of a larger size (e.g. using `count` $\gg$ 1, or MPI-derived datatypes)
- Don't use more processes than cores
- If the application is not big or complex enough, consider running in serial first, with compiler optimisations turned on

Further topics

Further topics

- MPI is a large and evolving standard
- We have only covered a small part of it
- There are hundreds of MPI routines which provide additional functionality, suited to different applications and needs

User-defined data types

- We have introduced and used basic MPI data types (slide 43)
- It is possible to combine data into a single, user-defined datatype
- The advantage is that we can send this data using just one message
- There are various ways to do this - we will very briefly mention one of them

User-defined data types in practice

- If we want to combine an array of 10 doubles, stored in variable x and an integer stored in y into a new data type
- And then broadcast from 0 to all other processes

```
MPI_Datatype newtype;

int array_of_blocklenghts[2] = {10, 1};
MPI_Datatype array_of_types[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint x_addr, y_addr;
MPI_Aint array_of_displacements[2]; // Displacement of elements from first element (in bytes)

array_of_displacements[0] = 0;
MPI_Get_address(&x, &x_addr); // Returns address of x on the system
MPI_Get_address(&y, &y_addr);
array_of_displacements[1] = y_addr - x_addr;

// Build the derived data type:
MPI_Type_create_struct(2, array_of_blocklengths, array_of_displacements, array_of_types, &newtype);
// Before using the datatype, we should first commit it:
MPI_Type_commit(&newtype);

MPI_Bcast(&x, 1, newtype, 0, MPI_COMM_WORLD);

// After using it, free any additional storage used to create the data type
MPI_Type_free(&newtype);
```

User-defined communicators

- **Recall**: A communicator is a collection of processes that can send messages to each other
- `MPI_COMM_WORLD` is the default communicator consisting of all processes in the program
- `MPI_COMM_SELF` is also provided - this only includes the process itself (not as useful)
- Within their `MPI_COMM_SELF`, all processes have rank 0
- The user can also define their own communicators
- This is particularly useful when the program has multiple levels of parallelism, i.e. can be split into a number of further paralellisable sub-tasks
- Using user-defined communicators insures that data relating to one sub-task are not accidentally sent to another

User-defined communicators

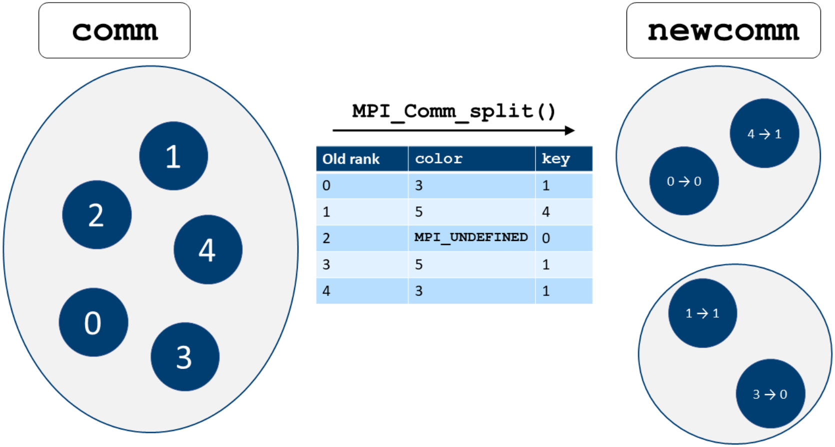
- New communicators can only be built upon existing communicators
- There are different ways to construct new communicators
- MPI_Comm_split is a collective call which partitions an existing communicator into disjoint sub-communicators

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

```
MPI_Comm_split(comm, color, key, newcomm, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: color, key
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

  - comm is the existing communicator
  - color is a non-negative integer which determines to which new communicator each process will belong (or MPI_UNDEFINED if process is joining none)
  - key determines the ordering (ranks) within the new communicators - i.e. process with smallest key receives rank 0 etc.
  - newcomm is the new communicator
- We can remove this by calling MPI_Comm_free(newcomm)

Splitting a communicator



Old rank	color	key
0	3	1
1	5	4
2	MPI_UNDEFINED	0
3	5	1
4	3	1

Input/Output

- The MPI standard does not require that all nodes can perform I/O (even though our examples have assumed that all processes can do output and process 0 can also do input)
- To find which processes can do I/O, one can call `MPI_Comm_get_attr(MPI_COMM_SELF, MPI_IO, attribute_val, flag)`
- `attribute_val` is a pointer, which will hold the returned value:

Returned value	Meaning
`MPI_ANY_SOURCE`	all ranks can do I/O
Processes' rank	the current process can do I/O (but not all)
Another rank	some rank can do I/O
`MPI_PROC_NULL`	no rank can do I/O

- So we can check our assumption that rank 0 can do I/O by running this function on rank 0 and checking for a return value of either `MPI_ANY_SOURCE` or 0
- MPI also provides functions for parallel I/O (e.g. to read from and write to a file in parallel), which are outside of the scope of this course

Using MPI with OpenMP

- Need even more speed-up?
- Nothing stops us from using thread-based parallelisation (e.g. OpenMP) within a node and MPI between nodes, in a *hybrid* approach
- **But** we should do this in a *thread safe* manner
- The call to `MPI_Init` should be replaced by `MPI_Init_thread`, which initialises the thread environment to the `required` level of thread support, if possible (depends on the MPI implementation)

```c
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided);
```

```fortran
MPI_Init_thread(required, provided, ierror)
  INTEGER, INTENT(IN) :: required
  INTEGER, INTENT(OUT) :: provided
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Thread support level	Meaning
MPI_THREAD_SINGLE	Only one thread
MPI_THREAD_FUNNELED	Multiple threads - a master thread makes all MPI calls
MPI_THREAD_SERIALIZED	Multiple threads - only call MPI one at a time
MPI_THREAD_MULTIPLE	Multiple threads - may all call MPI with no restrictions

Using MPI with OpenMP

- MPI_Init_thread should be called once on every process
- The thread which calls it becomes the main thread of that process
- The two routines below are callable from any thread at any time (unlike most MPI routines):
    - MPI_Is_thread_main(int *flag) determines whether the current thread is the main thread (the one that called MPI_Init_thread)
    - MPI_Query_thread(int *provided) returns the current level of thread support - i.e. the same as the one provided by MPI_Init_thread

Course overview

In this course, we

- gave a brief introduction to parallel programming
- introduced the message passing interface and its syntax
- looked at point-to-point communication for sending messages from one process to another (and the potential issues arising from mismatched/hanging calls)
- looked at MPI functions for collective communication between all processes in a communicator, which simplified the programming, increased performance and eliminates some of the risks associated with point-to-point communication
- introduced other modes of sending messages (e.g. blocking vs non-blocking), which allow the overlapping of calculation and communication
- looked at the errors MPI functions may return,
- as well as mistakes which are not caught by MPI
- mentioned some further topics (user-defined data types, user-defined communicators and how to use MPI with OpenMP)

Further reading

- There is nothing wrong with using this small subset of MPI
- Keeping it simple may prove to be less prone to bugs and equally fast as something more sophisticated
- But there might be applications which require further functionality
- You can always refer to the MPI standard, found at https://www.mpi-forum.org/docs/, or the books suggested in slide 3
- Happy parallelising! :)