Introduction to MPI

Lecture 1: Blocking & Collective communication

Liam Scarlett

liam.scarlett@curtin.edu.au

Class materials located at:

atom.curtin.edu.au/hpc

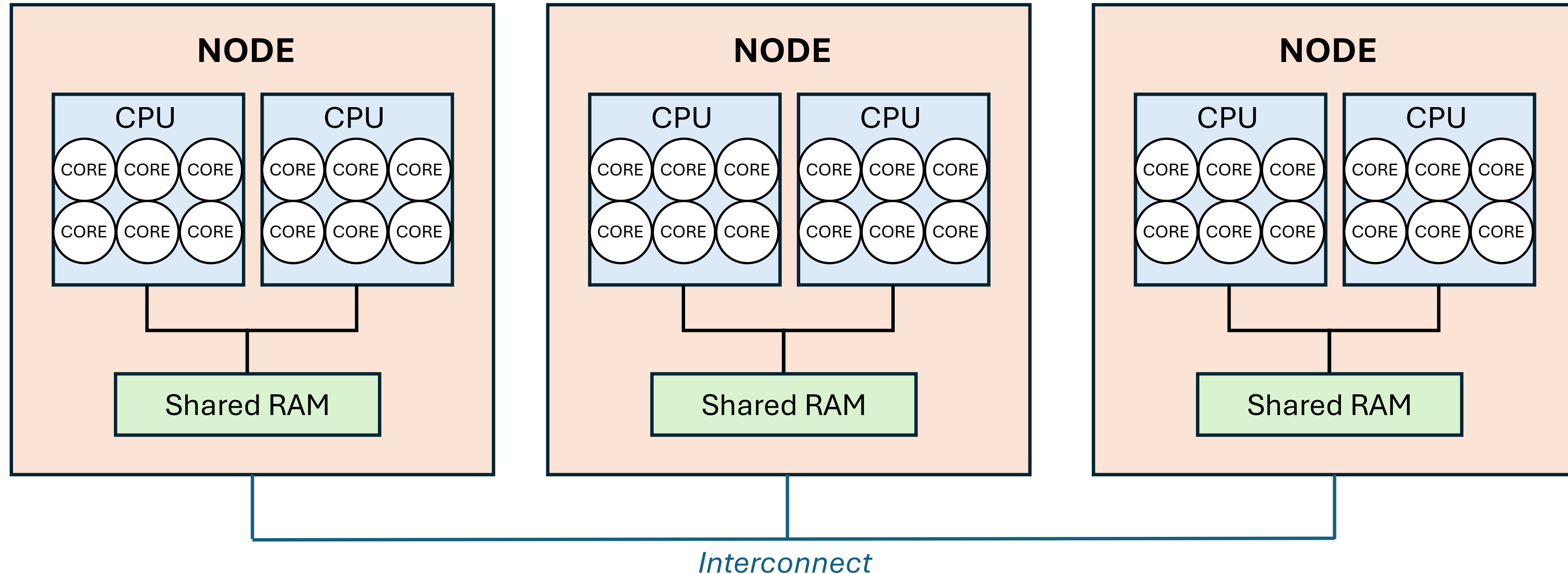
Or

```
git clone https://github.com/liamscarlett/intro-mpi
```

Workshop exercise: **Initial Setup**

(Just to make sure everyone can log in to Setonix)

Parallel computing on a supercomputer



Large-scale supercomputer jobs typically utilise two levels of parallelism:

- CPU cores within a single node work together doing calculations using data in their shared memory
- Different nodes communicate with each other and pass data between themselves as required.

Setonix has approx 400,000 GB of RAM and 200,000 CPU cores.
Each standard CPU compute node has two 64-core CPUs with shared access to 256GB of RAM.
There are approx 1,500 nodes.

Parallel computing on a supercomputer

- In this course you will learn two types of parallel programming for CPUs:

  - **OpenMP** (Open Multi-Processing) – *Later…*
  - **MPI** (Message Passing Interface) – *Now*

- These are not programming languages themselves, but rather libraries which provide special functions that enable parallel computing.

Parallel computing on a supercomputer

- Best practise in large-scale supercomputing is to combine the two:
  - **OpenMP** for ***intranode*** parallelism
  - **MPI** for *internode* parallelism

- It is possible to exclusively use **MPI** by treating each CPU core *as if* it did not share memory with other cores.

- Conversely, it is <u>not</u> possible to implement *internode* parallelism using **OpenMP** only.

- For educational purposes, we will learn **MPI** first, treating each CPU core in a single node as a distinct unit without shared memory.

How OpenMP works

- In **shared-memory multithreading** models (e.g. **OpenMP**) the main program is run by a single CPU core, with the remaining cores becoming active only when given work to do.

- A single instance of a program being run is called a **process**
  - **Not** to be confused with a *process*.

- Multithreading models invoke only a single process.

How MPI works

- **Message-passing** models (e.g. **MPI**) invoke **multiple processes**.
  Large-scale jobs might typically map each process to a single node, or a single CPU.

- We will associate each CPU core with a separate **process**.
  This means your single executable is being **run many times simultaneously**.
  (Note: you will often see the word "task" being used synonymously with "process" in the context of MPI)

- Supercomputers come with job-scheduling software which provides commands for launching multiple instances of the same program in separate processes.

  E.g. on *Setonix*, spawn 4 processes running the same executable with

```
srun -n 4 ./program.exe
```

Workshop exercise: **Hello World**

What was the point of that?

- Without including explicit instructions in the program for the parallelism, each process does *exactly the same thing*.

- The **MPI** library provides two primary things to help us out here:

  - Each **process** is assigned a number, starting from zero, called its **rank.** Each **process** is aware of its own **rank.**

  - You can instruct each process to do something different, depending on its **rank.**

  - **MPI** provides functions for communicating and transferring data between different **processes.**

Anatomy of a basic MPI program

<table>
<tr><th>Fortran</th><th>C</th></tr>
<tr><td>

```
program main
  implicit none
  include "mpif.h"
  integer :: ierr

  call MPI_Init(ierr)
  !Do computation here
  call MPI_Finalize(ierr)
end program
```

</td><td>

```
#include <mpi.h>

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  //Do computation here
  MPI_Finalize();
  return 0;
}
```

</td></tr>
</table>

- In Fortran, the final argument to all **MPI** subroutines is an integer which contains an error code on return:

  ```
  call MPI_Routine(x, y, z, ierr)
  ```

- In C, all **MPI** functions have that same error code as their return value:

  ```
  ierr = MPI_Routine(x,y,z);
  ```

- These are generally not of any importance as the **MPI** runtime prints error messages to the screen.

MPI header files

- Every Fortran routine or C function which makes MPI calls needs to include the appropriate header file:

```
include “mpif.h”  (Fortran)
#include <mpi.h>  (C)
```

- On *Setonix*, the header files are located at `${MPICH_DIR}/include`

- These header files define:
  - Constants that are used as arguments to MPI routines (datatypes, communicators, errors, etc.)

  - Interfaces to the MPI functions (C programs)

MPI header files

Important note for the Fortran programmers

- Many MPI subroutines are *overloaded* – the MPI library provides multiple versions of the subroutine with the same name which accept arguments of different types (integer, real, etc).

- The **"mpif.h"** header file does not provide interfaces for the MPI routines.

- This means that while your code is being compiled, the compiler does not know anything about the MPI subroutines. It is only when the MPI library is linked at the end that the compiler gets to check the subroutines you have called are actually provided by the MPI library.

- If the compiler sees you call the same subroutine multiple times with different arguments, it thinks it's a mistake.

- The solution is to use the compiler flag **`-fallow-argument-mismatch`**

Initialising and ending the MPI program

- To use any **MPI** functions/subroutines in your code, the MPI environment must be initialised:

```
call MPI_Init(ierr)          (Fortran)
MPI_Init(&argc, &argv);      (C)
```

- This is not responsible for spawning the separate **processes** – that was done by the `srun` command.

- `MPI_Init` assigns each process a **rank** and does some more "behind-the-scenes" stuff to facilitate communication between the processes.

Initialising and ending the MPI program

- At the end of your **MPI** program, the **MPI** environment needs to be terminated:

```
call MPI_Finalize(ierr)
```
(Fortran)

```
MPI_Finalize();
```
(C)

Note US spelling!

- The calls to **`MPI_Init`** / **`MPI_Finalize`** should be the very first / last thing in your program other than variable declarations and (in C) a final return statement.

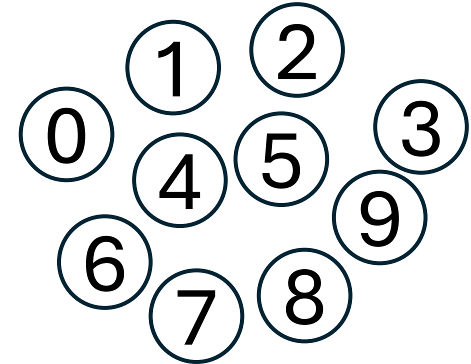
- On *Setonix* you can use **`man mpi_init`** and **`man mpi_finalize`** for details on these routines.

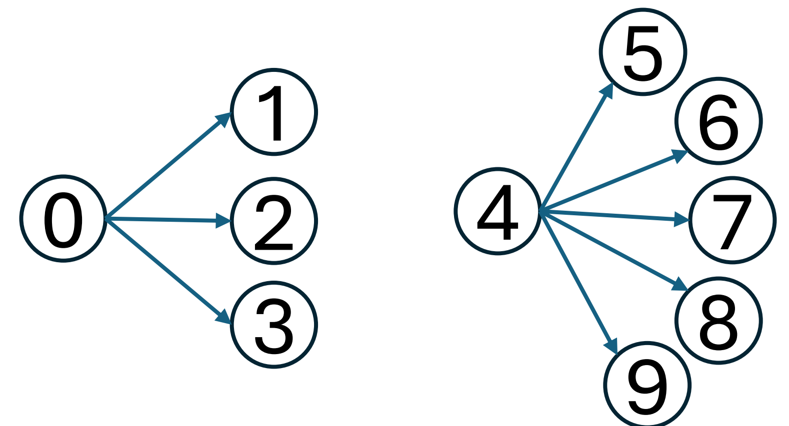
MPI Communicators

- Say we have 10 processes running our MPI program

- In some applications you may want to establish communication between subsets of the set of all processes.

- *E.g. perhaps **process 0** has some data it needs to share with **processes 1–3**, and **process 4** has some data it needs to share with **processes 5–9**.*





MPI Communicators

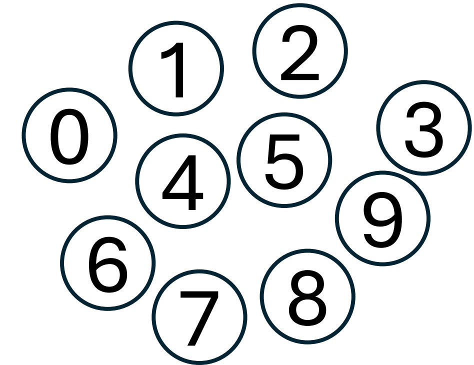
- It can be convenient to define ***communicators*** – objects that define a set of processes that can communicate with each other.

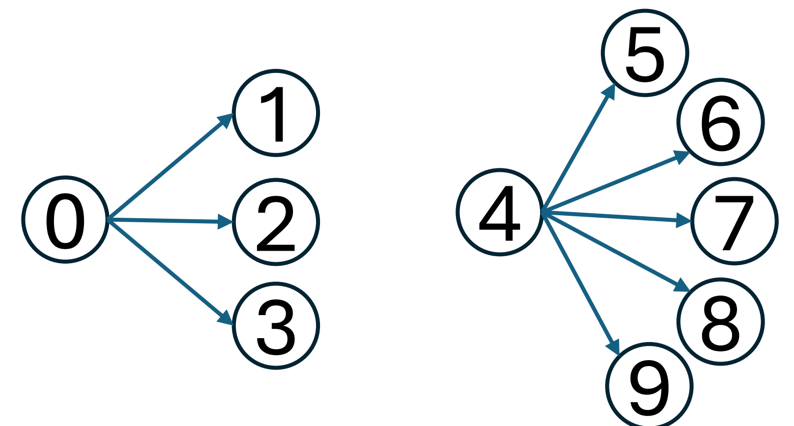
- When a **communicator** is set up, each of its **processes** is assigned a **rank** *within the communicator*.

- If a **process** is a member of multiple **communicators**, it will have a distinct **rank** within each one.

- All **MPI** communication routines have a **communicator** as one of their arguments.

  (e.g. in the diagrams here, **process 4** may be assigned **rank 0** within a **communicator** comprised of **processes 4–9**)





MPI_COMM_WORLD

- When the **MPI** environment is initialised by **`MPI_Init`**, it establishes a default **communicator** **`MPI_COMM_WORLD`**, comprised of **all MPI processes**.

Taking a peek inside the Fortran **MPI** header file:

```
INTEGER MPI_COMM_WORLD
PARAMETER (MPI_COMM_WORLD=1140850688)
```

we can see that `MPI_COMM_WORLD` is simply defined as an integer.

MPI_COMM_WORLD

- Defining **custom communicators** is advanced usage of **MPI**.

- We will stick exclusively to using the `MPI_COMM_WORLD` communicator.

- No need to think about this any further, but now we know why all our **MPI** function calls need to have `MPI_COMM_WORLD` as an argument ☺

MPI_Comm_size and MPI_Comm_rank

- **MPI** provides two routines to determine the **size** of a **communicator** and the **rank** of a **process**:

Fortran	C

```program main```
``` implicit none```
``` include "mpif.h"```
``` integer :: ierr, rank, nprocs```
 	
``` call MPI_Init(ierr)```
```→ call MPI_Comm_size(MPI_COMM_WORLD,nprocs,ierr)```
```→ call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)```
 	
``` call MPI_Finalize(ierr)```
```end program```
	```#include <mpi.h>```
	
	```int main(int argc, char *argv[]) {```
	``` int rank, nprocs;```
	
	``` MPI_Init(&argc, &argv);```
	```→ MPI_Comm_size(MPI_COMM_WORLD,&nprocs);```
	```→ MPI_Comm_rank(MPI_COMM_WORLD,&rank);```
	
	``` MPI_Finalize();```
	``` return 0;```
	```}```

- In the above example, each **process** will have the same value for **`nprocs`**, but a different value for **`rank`**.
- You can already implement some basic parallelism using **`if`** statements to perform different operations depending on the rank.

Workshop exercise: **Basic MPI**

The six basic MPI routines

```
MPI_Init
        Initialize MPI
MPI_Comm_rank
        Get the process rank
MPI_Comm_size
        Get the number of processes
MPI_Send
        Send data to another process
MPI_Recv
        Get data from another process
MPI_Finalize
        Finish MPI
```

- **MPI is *small***
  Full programs can be written using just these six routines.

- **MPI is *large***
  MPI has nearly 600 routines and is incredibly versatile.

On *Setonix*, use `man MPI_Routine` for info on any MPI routine.

MPI_Send

Fortran:

```
call MPI_Send(data, count, datatype, destination, tag, communicator, ierr)
```

C:

```
MPI_Send(&data, count, datatype, destination, tag, communicator);
```

- **data** – data to be sent (*in C, address of the data*)
- **count** – number of data items to be sent
- **datatype** – type of data (MPI_INTEGER, MPI_REAL, etc)
- **destination** – rank of the process to send data to
- **tag** – message label (an arbitrary integer)
- **communicator** – always MPI_COMM_WORLD for our purposes
- **ierr** – error return (*function return value in C, not required*)

MPI_Recv

Fortran:

```
call MPI_Recv(data, count, datatype, source, tag, communicator, status, ierr)
```

C:

```
MPI_Recv(&data, count, datatype, source, tag, communicator, &status);
```

- **data** – data to be sent (*in C, address of the data*)
- **count** – number of data items to be sent
- **datatype** – type of data (MPI_INTEGER, MPI_REAL, etc)
- **source** – rank of the process to receive data from
- **tag** – message label (an arbitrary integer)
- **communicator** – always MPI_COMM_WORLD for our purposes
- **status** – array/struct containing info about the message
- **ierr** – error return (*function return value in C, not required*)

MPI_Send / MPI_Recv

A valid communication between two **ranks** requires the use of **both** `MPI_Send` and `MPI_Recv`

If:

> **rank 0** calls `MPI_Send` with **rank 1** as the destination, and
> **rank 1** has not called a corresponding `MPI_Recv` with **rank 0** as the source

Then the data transfer does not complete (in fact the program will hang).

Send and receive program: Fortran

```
program send_receive
  include "mpif.h"
  integer :: rank, ierr, tag, source, destination, count, x
  integer :: status(MPI_STATUS_SIZE)  <------ Defined as an array

  call MPI_Init( ierr )
  call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
  tag=1234;  source=0;  destination=1; count=1
  if(rank == source)then
    x=5678
    call MPI_Send(x, count, MPI_INTEGER, destination, tag, MPI_COMM_WORLD, ierr)
    print*, "process ", rank ," sent ", x
  endif
  if(rank == destination)then
    call MPI_Recv(x, count, MPI_INTEGER, source, tag, MPI_COMM_WORLD, status, ierr)
    print*, "process ", rank, " got ", x
  endif
  call MPI_Finalize(ierr)
end program
```

Send and receive program: C

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[])
{
  int rank, numprocs, tag, source, destination, count, x;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  tag=1234;  source=0;  destination=1;  count=1;
  if(rank == source){
    x=5678;
    MPI_Send(&x,count, MPI_INT, destination, tag, MPI_COMM_WORLD);
    printf("process %d sent %d\n",rank,x);
  }
  if(rank == destination){
    MPI_Recv(&x,count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
    printf("process %d got %d\n",rank,x);
  }
  MPI_Finalize();
}
```

Uses a predefined struct

MPI_Send / MPI_Recv: **data, count, datatype**

- If sending a single value, **count** is always 1
- If sending an array, **count** is the number of elements in the array

Fortran MPI Types	
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	-

C MPI Types	
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_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

- The programmer is responsible for ensuring the MPI type matches the actual data type
- Using the wrong data type, e.g. **MPI_FLOAT** for a **double**, will lead to weird behaviour!

MPI_Send / MPI_Recv: **source, destination, tag**

```
call MPI_Send(data, count, datatype, destination, tag, communicator, ierr)
MPI_Send(&data, count, datatype, destination, tag, communicator);

call MPI_Recv(data, count, datatype, source, tag, communicator, status, ierr)
MPI_Recv(&data, count, datatype, source, tag, communicator, &status);
```

- **source** and **destination** are the **ranks** of the sending and receiving **processes** *within the specified **communicator***

- **tag** is an arbitrary integer used to distinguish between different messages.

  - It must be the same for both the **sender** and **receiver**.

  - Tags create an additional layer of verification in the handshake between **sender** and **receiver**.

  - Particularly important when multiple messages are being sent between the same two **ranks**.

MPI_Send / MPI_Recv: **status**

```
call MPI_Recv(data, count, datatype, source, tag, communicator, status, ierr)
MPI_Recv(&data, count, datatype, source, tag, communicator, &status);
```

Let's take a look in the C **MPI** header file (`$MPICH_DIR/include/mpi.h`):

```
typedef struct MPI_Status {
    int count_lo;
    int count_hi_and_cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

(`count_lo` and `count_hi_and_cancelled`:) Implementation specific and not for user access

`status.MPI_SOURCE`: **rank** of sender
`status.MPI_TAG`: **tag** of received message
`status.MPI_ERROR`: an error code

In most cases this information is redundant because we explicitly specified the **source** and **tag** in the `MPI_Recv` call.

BUT: In more advanced usage there are wildcards MPI_ANY_SOURCE and MPI_ANY_TAG

- **Receiver** can accept a message from **any source** and then determine later who it came from

MPI_Send / MPI_Recv: **status**

```
call MPI_Recv(data, count, datatype, source, tag, communicator, status, ierr)
MPI_Recv(&data, count, datatype, source, tag, communicator, &status);
```

In Fortran, the **status** is a basic array, which we need to define ourselves in the calling routine:

```
integer :: status(MPI_STATUS_SIZE)
```

The Fortran MPI header file (`$MPICH_DIR/include/mpif.h`) defines integers MPI_SOURCE, MPI_TAG, and MPI_ERROR, allowing the user to access the relevant info:

```
source = status(MPI_SOURCE)
  tag   = status(MPI_TAG)
```

This is all so you understand why we are passing these arguments to the MPI_Send and MPI_Recv but we will not be making use of the status struct/array in these workshops ☺

Blocking and non-blocking calls

- The standard `MPI_Send` / `MPI_Recv` routines are **blocking**
  They do not return control to the calling routine until *after* the send/receive process is complete.

- There is no guarantee that the receiver will call the `MPI_Recv` routine at the same time that the sender calls `MPI_Send`.

- The `MPI_Send` routine waits for confirmation from the receiving process that the message is received.
  - If the receiving process is delayed then the sender needs to sit and wait

- The `MPI_Recv` routine waits until the message has been received
  - If the sending process is delayed then the receiver needs to sit and wait

Blocking and non-blocking calls

- Even when the send/receive calls happen *at the same time* the routines do not return until the transfer is complete and for large messages this can take some time!

- Once the CPU has initiated the send/receive, the network interface card handles the data transfer, and the CPU can in principle continue doing other things.

- There are non-blocking variants of the send/receive routines: **Next week!**

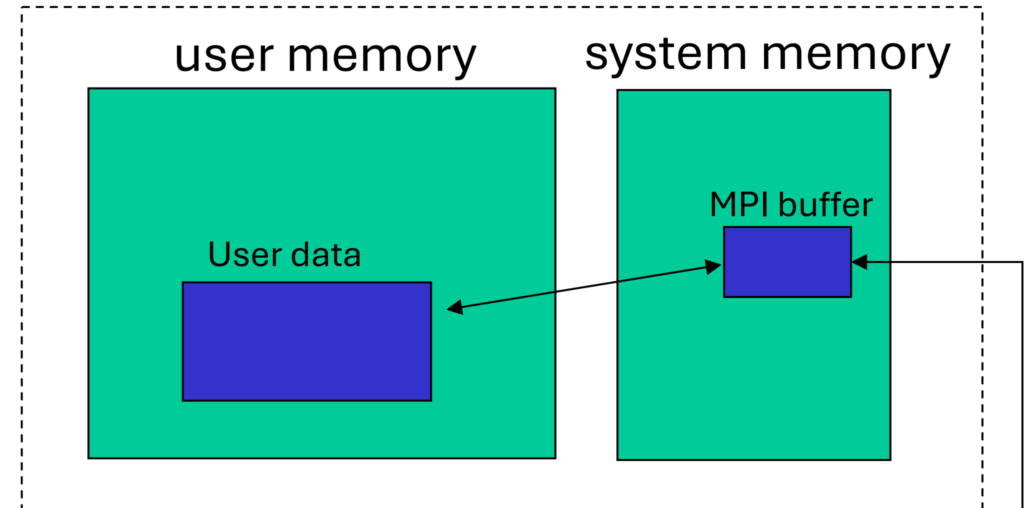
  `MPI_Isend, MPI_Irecv`

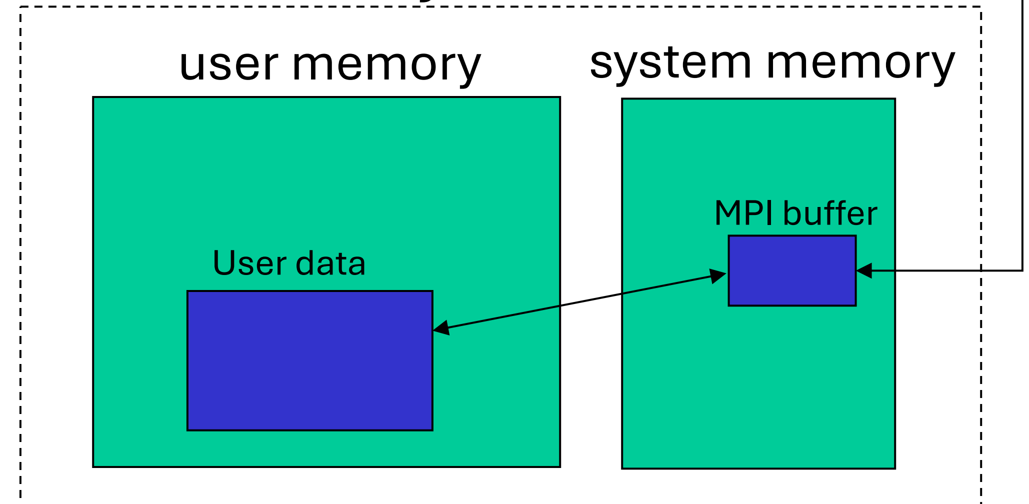
- Blocking is an important safety feature, preventing data from being modified before the transfer is complete.

Workshop exercise: **Point-to-Point communication**

Message buffering

- A caveat to the blocking/non-blocking discussion is **message buffering**
- In buffered sends, **MPI** first copies the data into a **buffer** and then it is the data in the buffer which is transferred.
- `MPI_Bsend` can be used to request buffered sends, using a message buffer you allocate yourself
  Uncommon – not recommended!
- **MPI** typically does its own default buffering, so we still need to understand it!





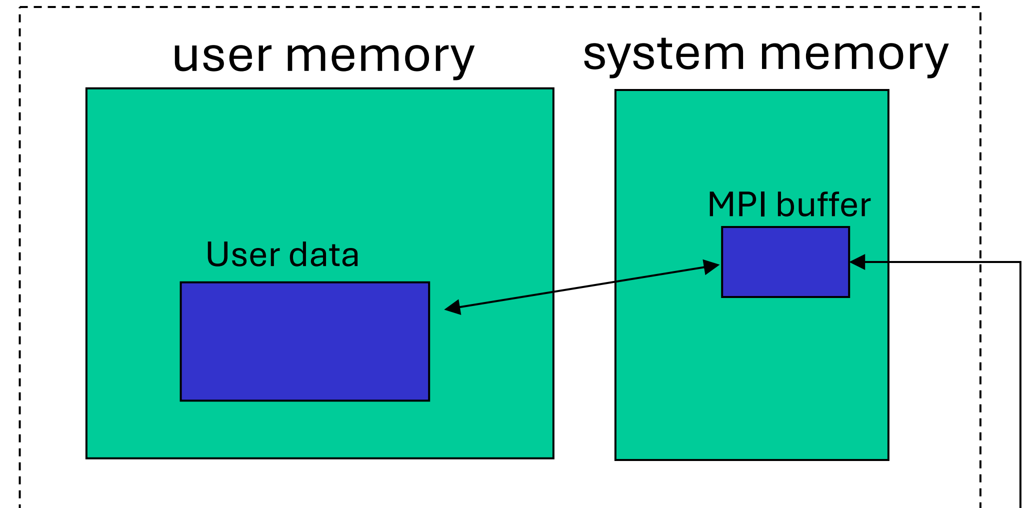
Message buffering

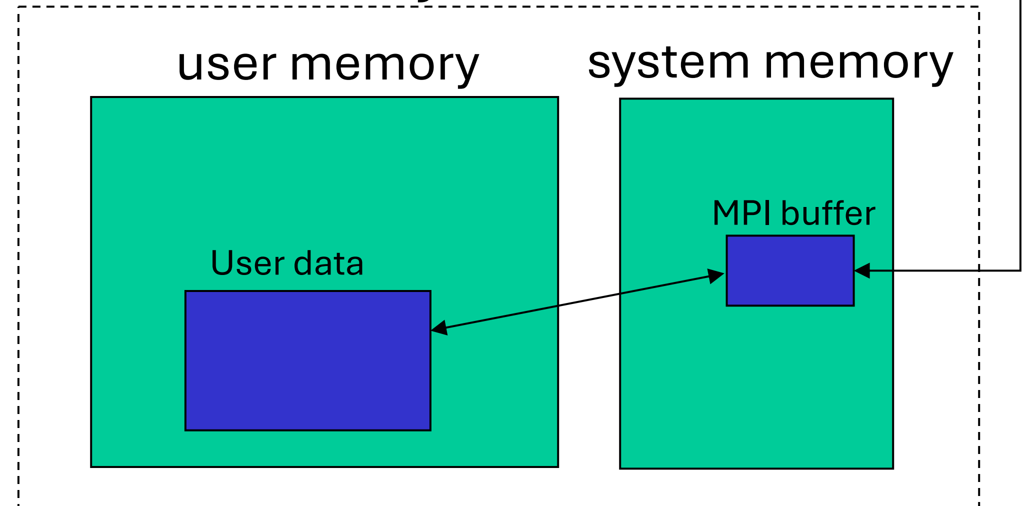
- Once the data is copied into the buffer, the **MPI_Send** routine registers this as a successful "receive" and returns.

- **MPI_Send** can then function as if it is **non-blocking**.

- It is safe for the **sender** to modify the data *before the transfer is complete* because the transfer is being made from a **copy**.

- The **MPI** library will decide when use buffering to enhance performance (depends on message size)



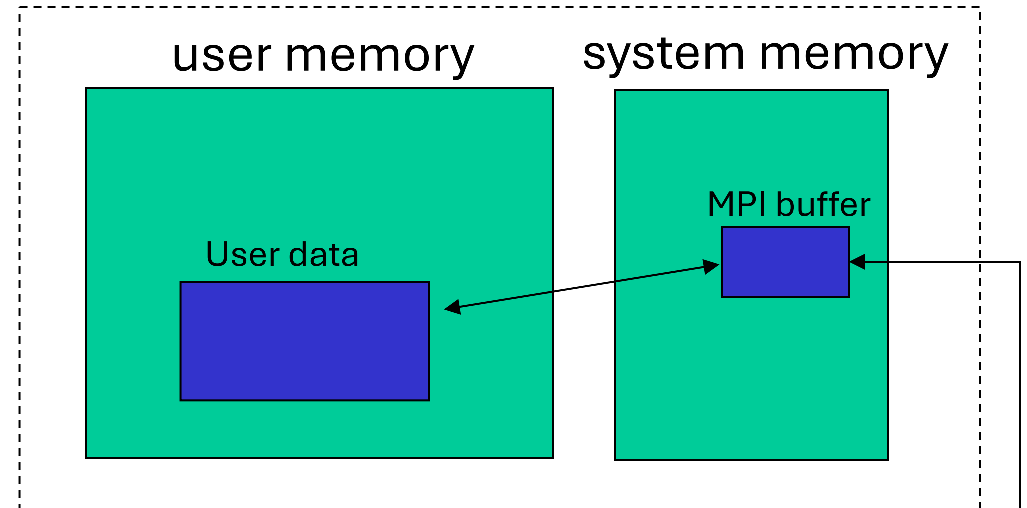


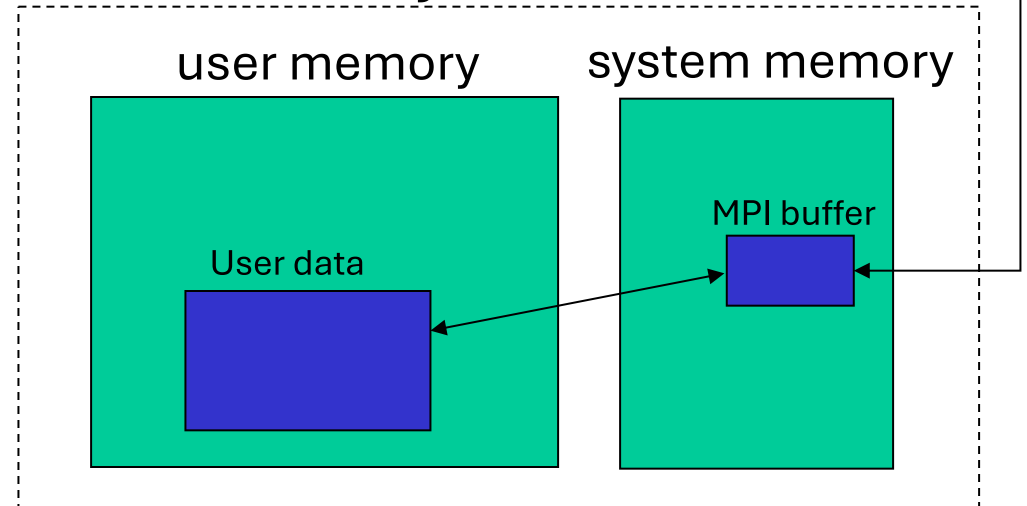
Message buffering

- Some implementations of **MPI** create a static pool of memory when `MPI_Init` is called, and use this for message buffers. This address space is visible only to the **MPI** library.

- Some implementations dynamically allocate buffers as they are required, in the same address space as the user data.

- There is typically an implementation-dependent **maximum** message size for buffering.





Message buffering

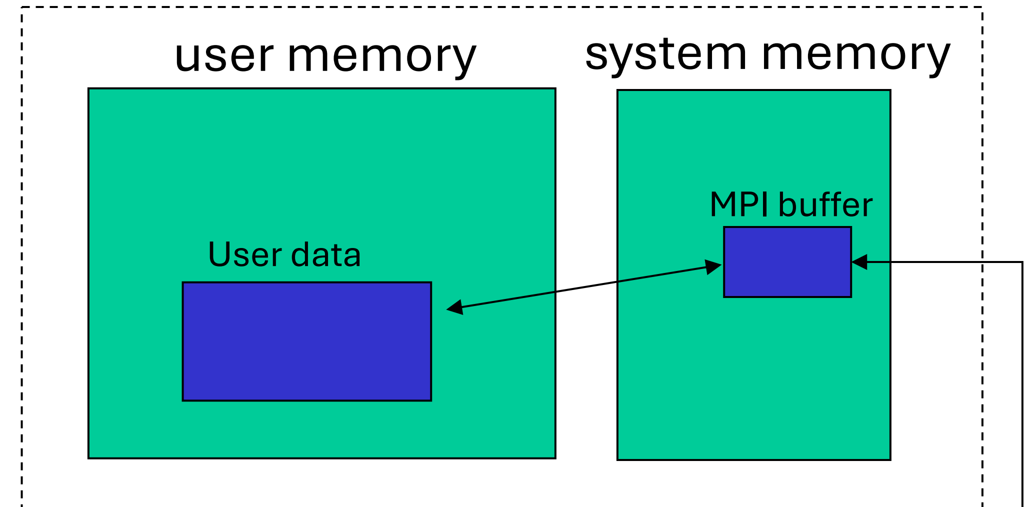
- In buffered **receives**, the data is *received into a buffer* before being **copied** into its final destination.

- The **MPI** library typically only uses buffered receives when the data arrives *before* the destination process calls the corresponding `MPI_Recv`

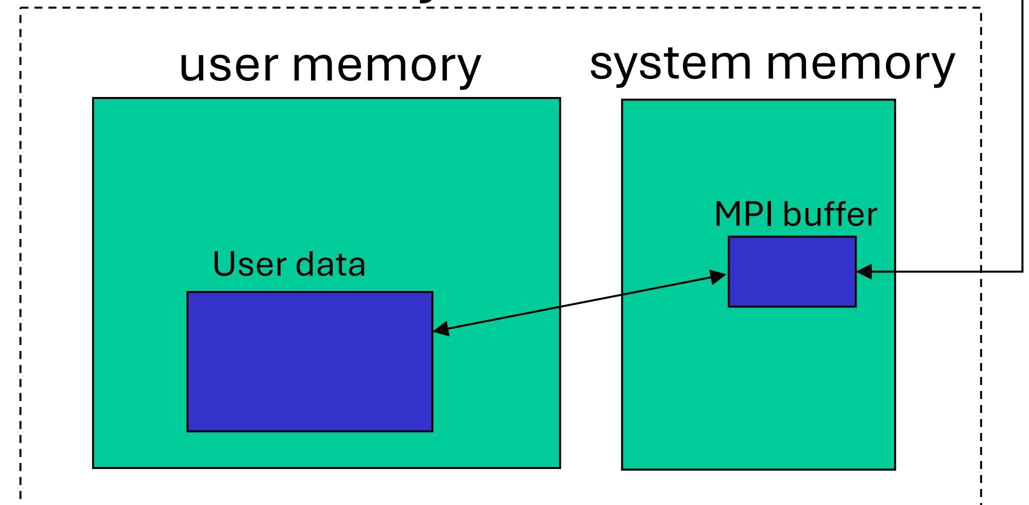
- We need to be aware that message buffering may happen behind the scenes, but **never** write code with the assumption it will happen.

Code that depends on implementation-dependent features may work just fine on one machine and fail on another!

MPI Process i:



MPI Process j:



Deadlock!

- **Deadlock** occurs when two (or more) process are each waiting for the other to finish before completing their own task
  - e.g. exchanging numbers between two processes

Fortran	C

`if (rank == 0) then`
`  call MPI_Send(y,1,MPI_REAL,1,...)`
`  call MPI_Recv(x,1,MPI_REAL,1,...)`
`endif`
 	
`if (rank == 1) then`
`  call MPI_Send(x,1,MPI_REAL,0,...)`
`  call MPI_Recv(y,1,MPI_REAL,0,...)`
`endif`
	`if (rank == 0) {`
	`  MPI_Send(y,1,MPI_REAL,1,...);`
	`  MPI_Recv(x,1,MPI_REAL,1,...);`
	`}`
	
	`if (rank == 1) {`
	`  MPI_Send(x,1,MPI_REAL,0,...);`
	`  MPI_Recv(y,1,MPI_REAL,0,...);`
	`}`

Rank 0 sends **y** to **rank 1** and receives **x** in return
Rank 1 sends **x** to **rank 0** and receives **y** in return

Both processes are **stuck** at their blocking send calls

Deadlock!

- In this example the problem is simply fixed by swapping the order of send/recv on rank 0:

Fortran	C

`if (rank == 0) then`
`  call MPI_Send(y,1,MPI_REAL,1,...)`
`  call MPI_Recv(x,1,MPI_REAL,1,...)`
`endif`
 	
`if (rank == 1) then`
`  call MPI_Recv(y,1,MPI_REAL,0,...)`
`  call MPI_Send(x,1,MPI_REAL,0,...)`
`endif`
	`if (rank == 0) {`
	`  MPI_Send(y,1,MPI_REAL,1,...);`
	`  MPI_Recv(x,1,MPI_REAL,1,...);`
	`}`
	
	`if (rank == 1) {`
	`  MPI_Recv(y,1,MPI_REAL,0,...);`
	`  MPI_Send(x,1,MPI_REAL,0,...);`
	`}`

- MPI's default message buffering can also "solve" the problem for you
  Faulty code can *appear* to work when the message size is small enough to be buffered but then fail when the message size is larger ☹

Workshop exercise: **Creating Deadlock**

Collective communication

- **MPI** has many routines which coordinate communication amongst a group of processes (typically MPI_COMM_WORLD).

- E.g. you can **broadcast** data from one rank to all other ranks in a communicator without calling `MPI_Send` many times.

- Message tags are not used.
- All processes in the communicator **must** call the collective operation.
- If not all processes call the collective operation the program will hang indefinitely.

- Three classes of collective operation:
  - Data movement
  - Collective computation
  - Synchronisation

MPI_Bcast

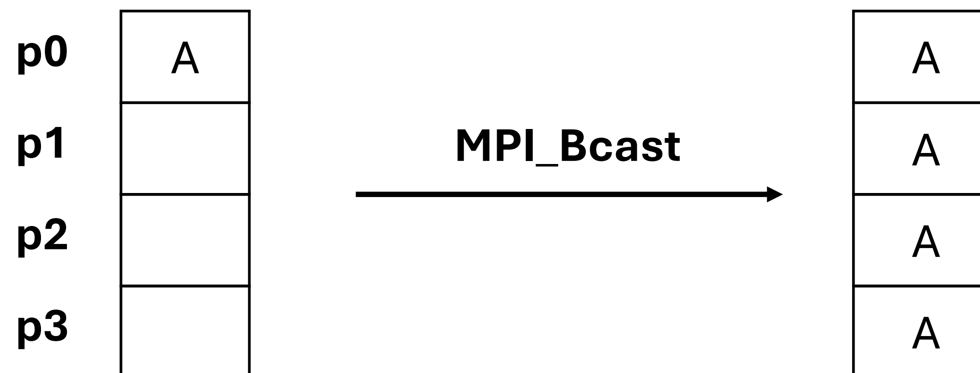
- Used to **broadcast** data from **one process** to **all other processes**

Fortran:

```
call MPI_Bcast(data,count,datatype,source,communicator,ierr)
```

C:

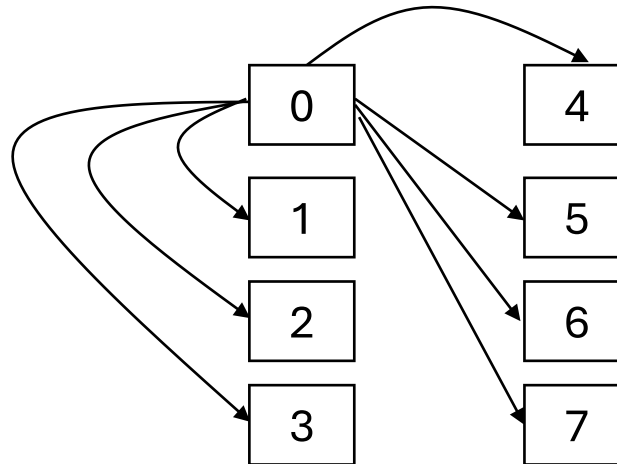
```
MPI_Bcast(&data,count,datatype,source,communicator);
```



MPI_Bcast

- The following *hand-coded* broadcast operation scales as $O(N)$, where N is the number of processes

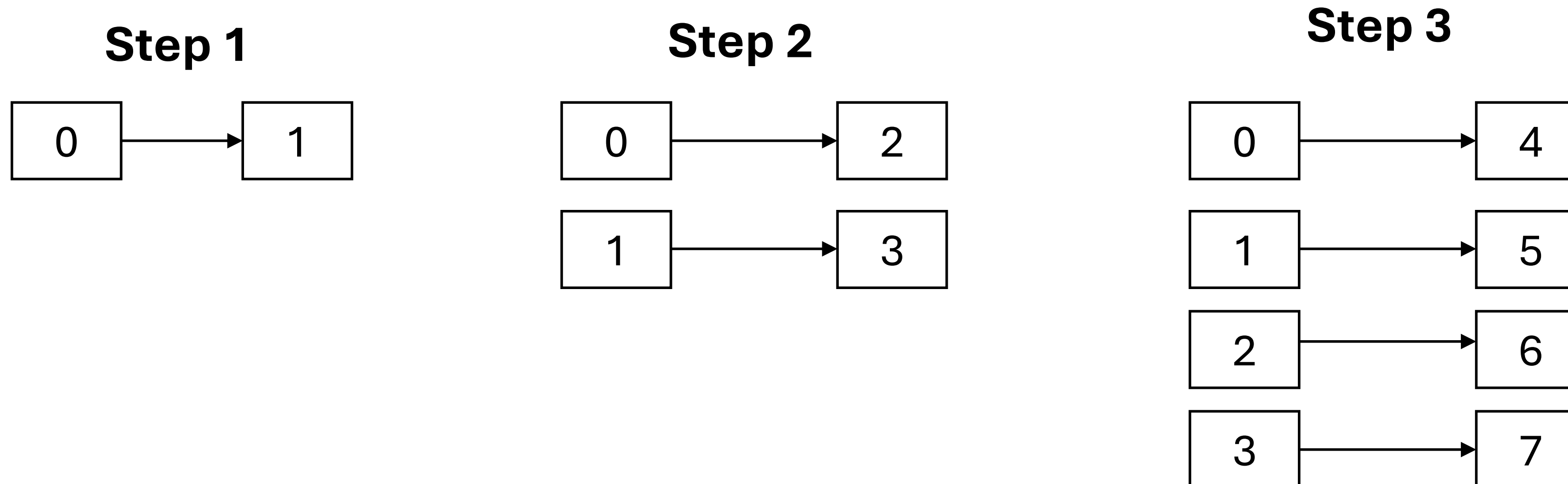
```
do i=1,nprocs-1
   if(rank == 0) call MPI_Send(x,1,MPI_REAL,i,..)
   if(rank == i) call MPI_Recv(x,1,MPI_REAL,0,..)
end do
```



MPI_Bcast

- When implemented properly, **`MPI_Bcast`** scales as $O(\log_2(N))$

- Although it's the usual case, we don't have to make rank 0 the source

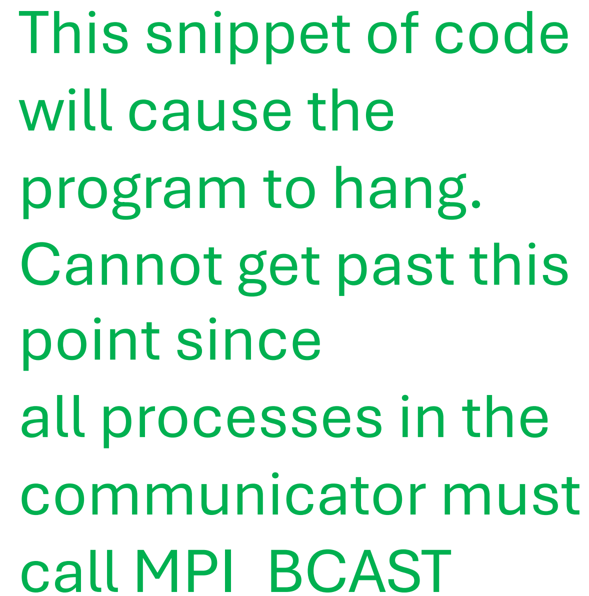


MPI_Bcast: a cautionary tale

- MPI_BCAST (like all other collective operations) is blocking and contains an implicit barrier
  - all processes in the group must call the same operation

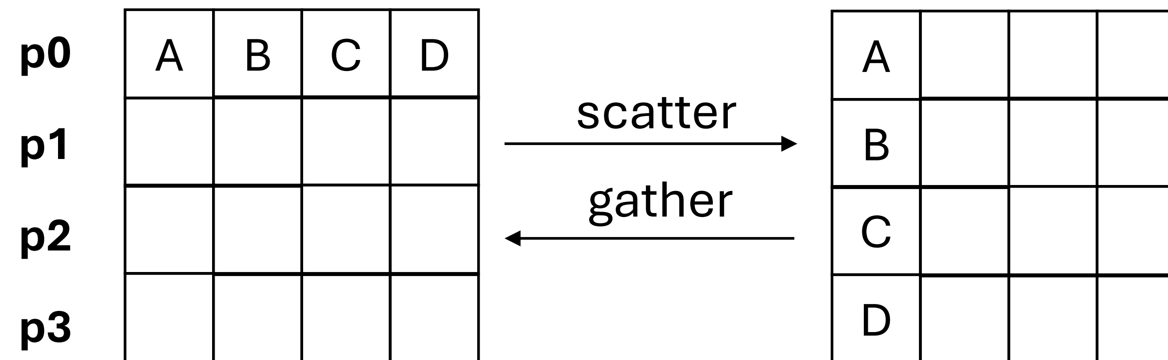
```
! BCAST to even processes
if(mod(rank,2) == 0)
   call MPI_Bcast(...)
endif
```

```
// BCAST to even procs
if (rank%2==0) {
   MPI_Bcast(...);
}
```



MPI_Scatter / MPI_Gather

- Scatter sends **different** data from the **source** process to all other **processes**
- Gather collects **different** data from all **other processes** onto the **destination** process



MPI_Scatter

Fortran:

```
call MPI_Scatter(senddata,count,datatype,
                 recvdata,count,datatype,
                 source,communicator,ierr)
```

C:

```
MPI_Scatter(&senddata,count,datatype,
            &recvdata,count,datatype,
            source,communicator);
```

- `senddata` is the array on the source process.
  Contains data to be sent to all other processes.

- `recvdata` is the receive array on all processes (including source process)

- `count` is the number of elements to be sent to **each** process.
  (Not the total number of elements to be sent from source)

MPI_Gather

Fortran:

```
call MPI_Gather(senddata,count,datatype,
                recvdata,count,datatype,
                destination,communicator,ierr)
```

C:

```
MPI_Gather(&senddata,count,datatype,
           &recvdata,count,datatype,
           destination,communicator);
```

- `senddata` is the array on each process (including destination)
  Contains data to be gathered on the destination process.

- `recvdata` is the receive array on the destination process.

- `count` is the number of elements to be sent *from* **each** process.
  (Not the total number of elements to be gathered on destination)

MPI_Scatter / MPI_Gather

Fortran:

```
call MPI_Gather(senddata,count,datatype,
                recvdata,count,datatype,
                destination,communicator,ierr)
```

C:

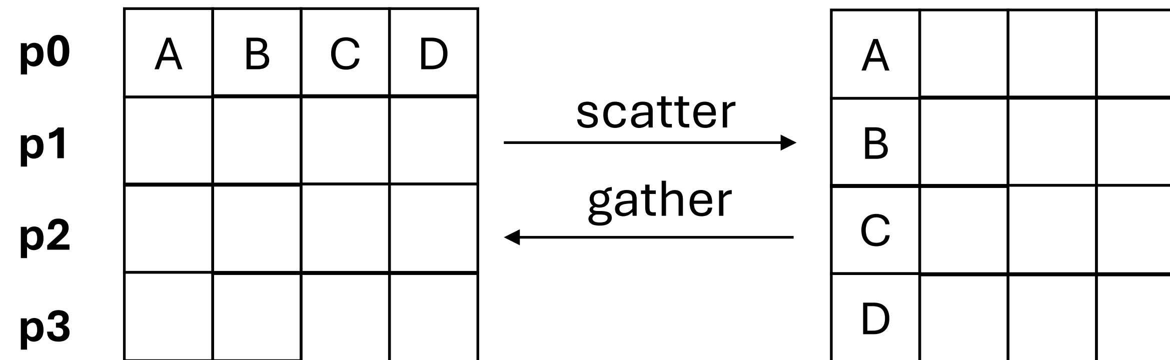
```
MPI_Gather(&senddata,count,datatype,
           &recvdata,count,datatype,
           destination,communicator);
```

Note that `count` and `datatype` appear twice in `MPI_Scatter` and `MPI_Gather`.

In principle you can specify a count and datatype to be received that is different from the count and datatype that is sent – advanced usage!

For most use cases we keep them the same and accept the redundancy.

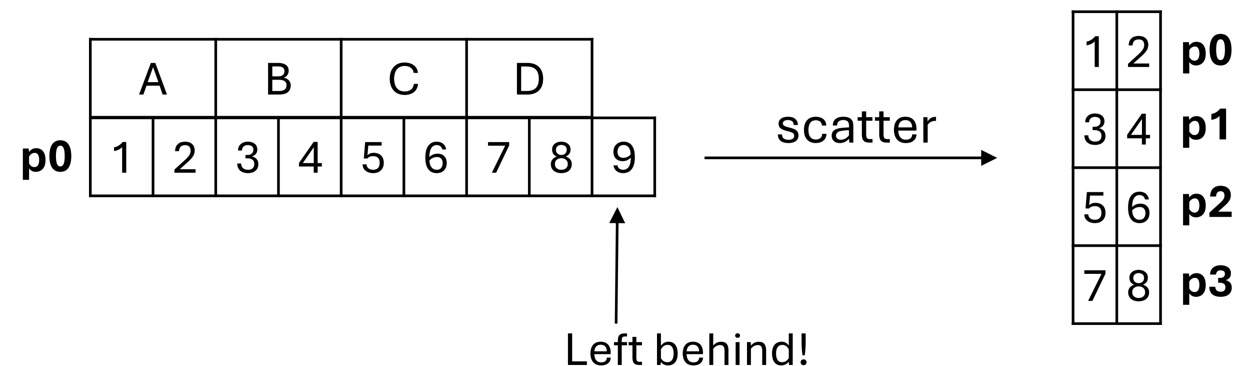
MPI_Scatter / MPI_Gather



A, B, C, D could be individual elements in the send array, or blocks of multiple elements each.

These routines make the most sense when the number of elements in the send array is a multiple of the number of processes.

Say we have nine elements in our send array and we scatter to four processes with **count=2**:



If the number of elements in the send array is **fewer** than the count times number of processes, bad things will happen, either:

- Segmentation fault (out-of-bounds memory access)
- Scatter without fault but some processes receive whatever random data was in the memory addresses next to the send array

MPI_Scatter / MPI_Gather

Fortran: MPI_Scatter

```
real*8, allocatable :: senddata(:), recvdata(:)
integer :: nprocs, count, ierr

allocate(senddata(nprocs*count))
allocate(recvdata(count))

call MPI_Scatter(senddata,count,MPI_DOUBLE_PRECISION, &
                 recvdata,count,MPI_DOUBLE_PRECISION, 0,MPI_COMM_WORLD,ierr)
```

C: MPI_Gather

```
int nprocs, count;
double senddata[count], recvdata[nprocs*count];

MPI_Gather(senddata,count,MPI_DOUBLE_PRECISION,
           recvdata,count,MPI_DOUBLE_PRECISION, 0,MPI_COMM_WORLD);
```

MPI_Scatter: Size of **senddata** is (at least) **nprocs*count**, size of **recvdata** is (at least) **count.**

MPI_Gather: The opposite.

MPI_Scatter / MPI_Gather: scaling behaviour

- Since each process requires a different set of data, can't use fan-out pattern of communication
  - Stuck with linear scaling, that is, $O(N)$

- If we have a small amount of data to send to each process, it might be faster to just broadcast **all** data to every process.

- **`MPI_Bcast`** is a simpler operation than **`MPI_Scatter`**, with smaller overheads.

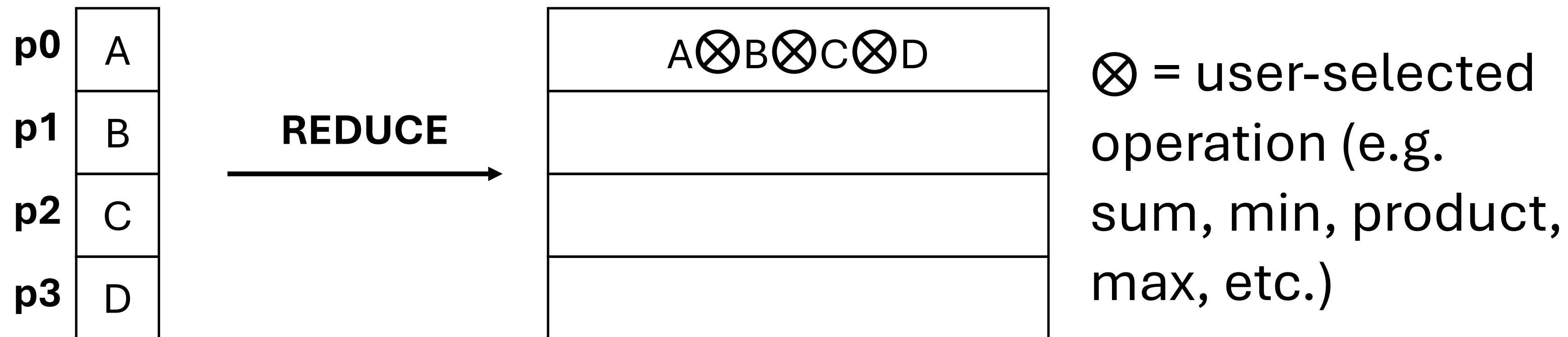
- As the size of the data increases, there is a point where the lower overheads are made irrelevant and **`MPI_Scatter`** is faster

MPI_Reduce

- Gathers data from all processes onto a single process (*while performing a mathematical operation*)

Fortran:
```
call MPI_Reduce(senddata,recvdata,count,
                datatype,operation,destination,communicator,ierr)
```

C:
```
MPI_Reduce(senddata,recvdata,count,
           datatype,operation,destination,communicator);
```



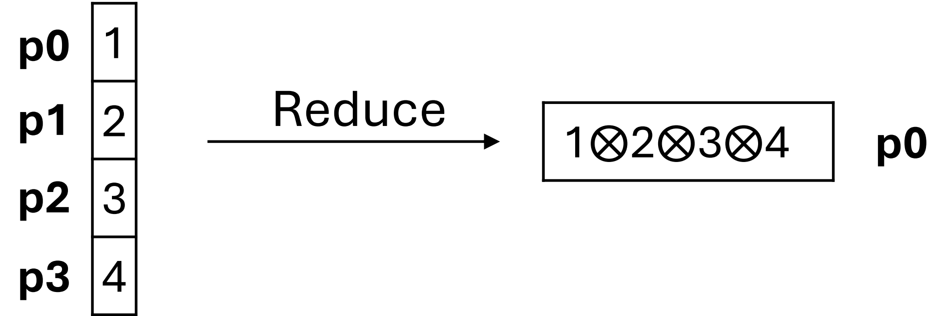
Same syntax as **MPI_Gather**, but:

- **count** and **datatype** specified only once (apply to both send and receive arrays)
- also specify the **operation**

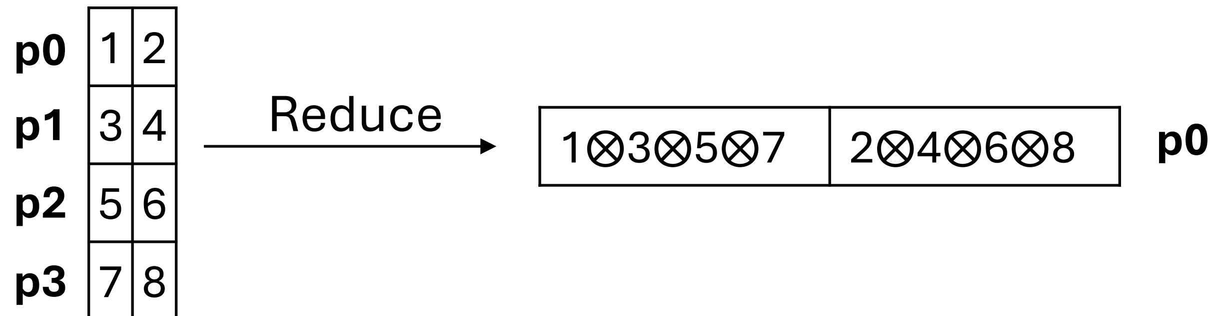
MPI_Reduce

The operation ⊗ is performed element-wise, e.g:

count=1:



count=2:

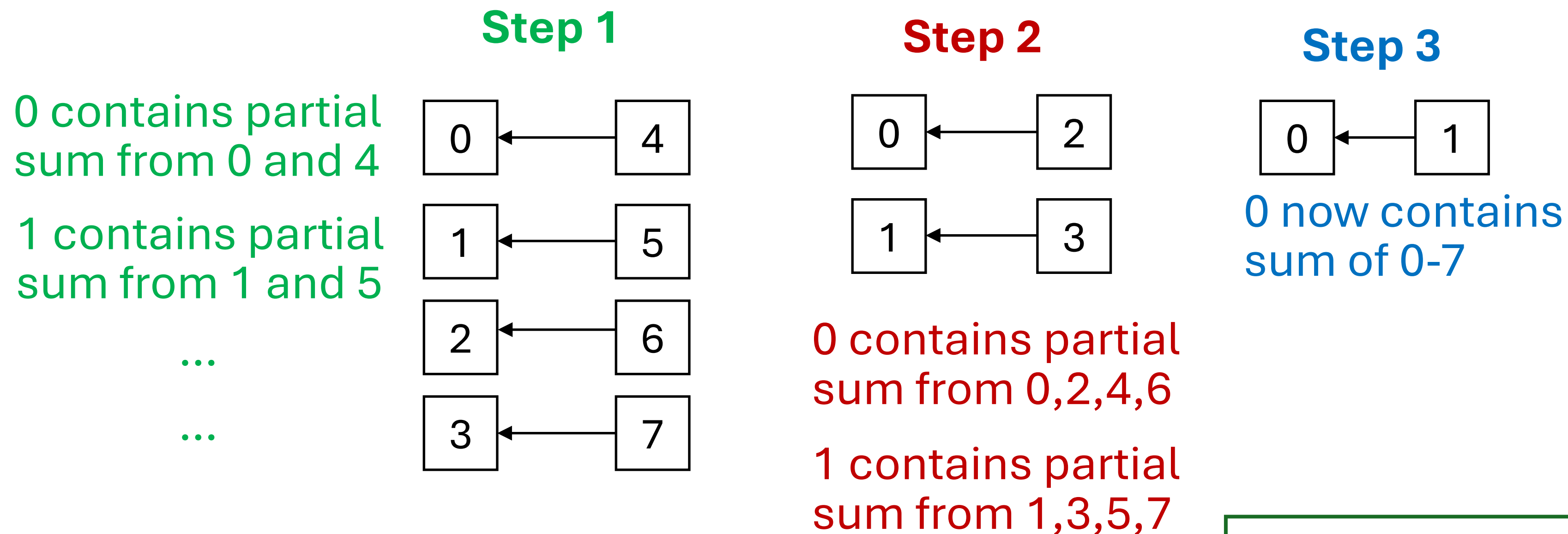


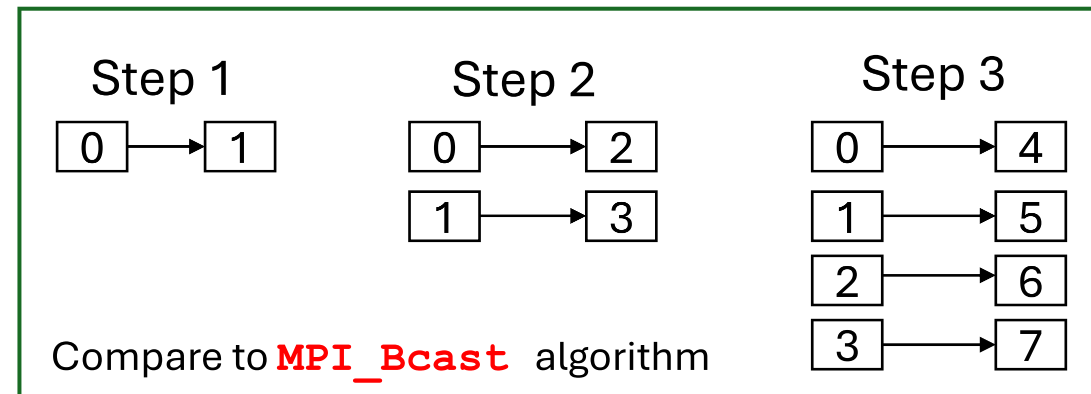
Defined operations:

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical "and"
MPI_LOR	Logical "or"
MPI_LXOR	Logical "xor"
MPI_BAND	Bitwise "and"
MPI_BOR	Bitwise "or"
MPI_BXOR	Bitwise "xor"
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

MPI_Reduce: scaling behaviour

- **MPI_Reduce** scales the same as MPI_Bcast
  - consider the example of summing across 8 processes:





MPI_Barrier

- Blocks the caller until all members in the communicator have called it

- Used as a synchronisation tool

```
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
MPI_Barrier(MPI_COMM_WORLD);
```

- Shouldn't be used excessively. If you find yourself programming with lots of barriers, you're probably writing inefficient code. Good uses include:
  - Debugging/developing applications
  - Getting accurate timings of parallel regions

MPI_Barrier to get accurate timings

```
! Different process may reach MPI_Bcast at
! various times. Timings may vary by process
tstart = MPI_Wtime()
call MPI_Bcast(...)
tend = MPI_Wtime()
t = tend - tstart
print*, "time = ",t

! Barrier call ensures synchronisation of timers
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
tstart = MPI_Wtime()
call MPI_BCAST(...)
tend = MPI_Wtime()
t = tend - tstart
print*, "time = ",t
```

MPI_Wtime() returns a double precision number