Introduction to MPI

Lecture 2: Non-Blocking Communication & Master/Worker Parallelism

Liam Scarlett

liam.scarlett@curtin.edu.au

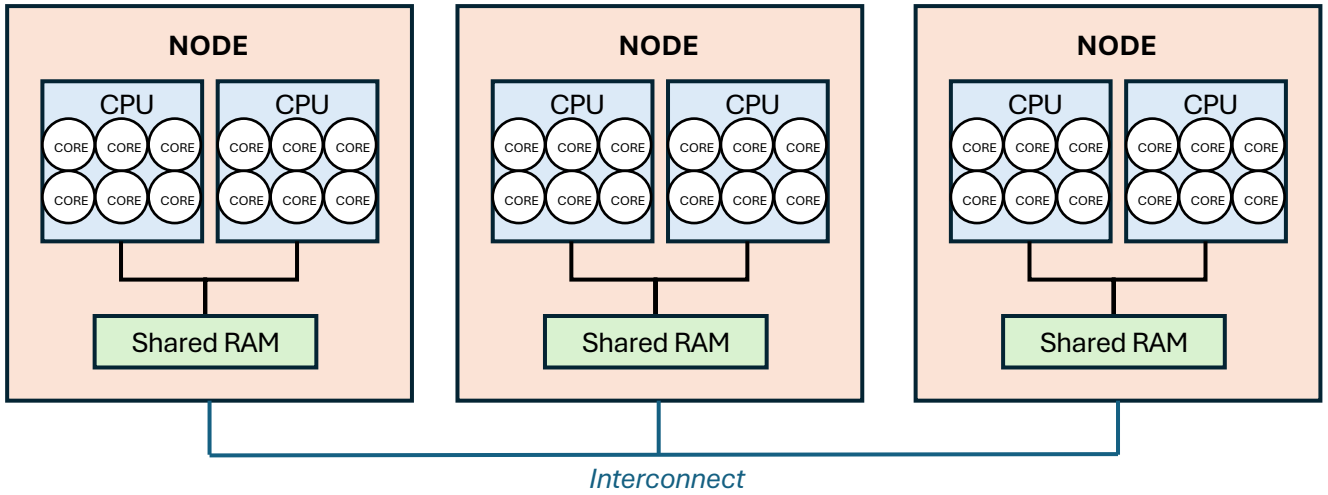
Class materials located at:

atom.curtin.edu.au/hpc

Or

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

Recap on how MPI works



- **MPI** (Message Passing Interface) allows work to be distributed over many process**ors** which do not share memory.

- We introduced the concept of a **process** – a single instance of a program being run.

- Generally in a large job you may have process**es** running on multiple **nodes**.
  - But there is flexibility in the way process**es** are mapped to CPU cores.
  - You can run an MPI program with multiple process**es** within a single node.

Recap on how MPI works

*What is a **process**? A deeper look*

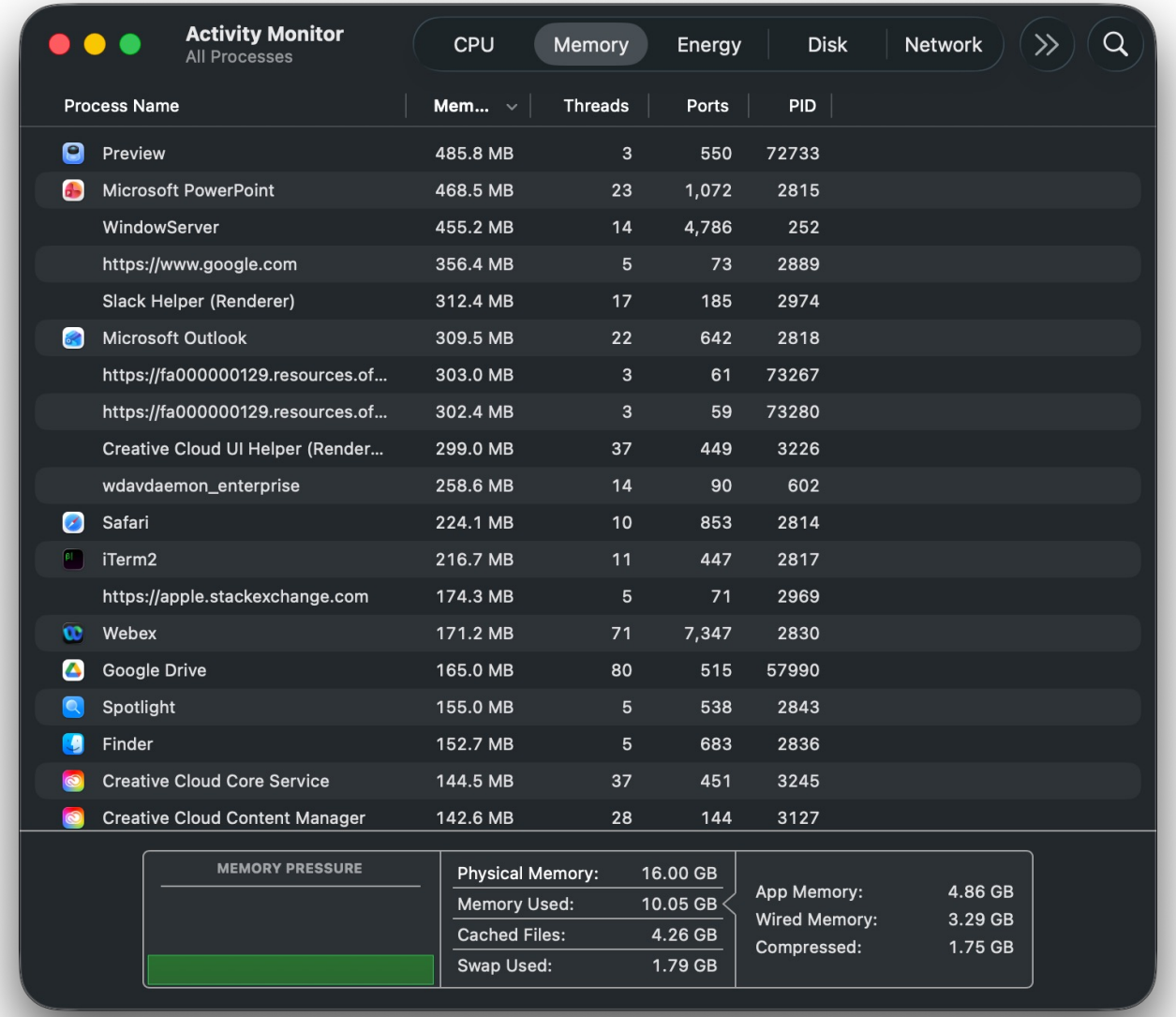
- If I check Activity Monitor on my iMac, I can see many active process**es**.

- Each **process** may have multiple **threads**. Multithreading will be taught later in the OpenMP module, but briefly:

  A **thread** is an *independent sequence of instructions that the operating system can schedule to run concurrently with other threads*.

- One CPU **core** can execute instructions from one thread at a time (caveat: hyperthreading)

- My iMac has **10** physical CPU **cores**. The huge number of active **threads** we see here have to be juggled by the operating system.



Recap on how MPI works

*What is a **process**? A deeper look*

- Now I run an MPI program **`main`** using 10 process**es,** by executing the command:

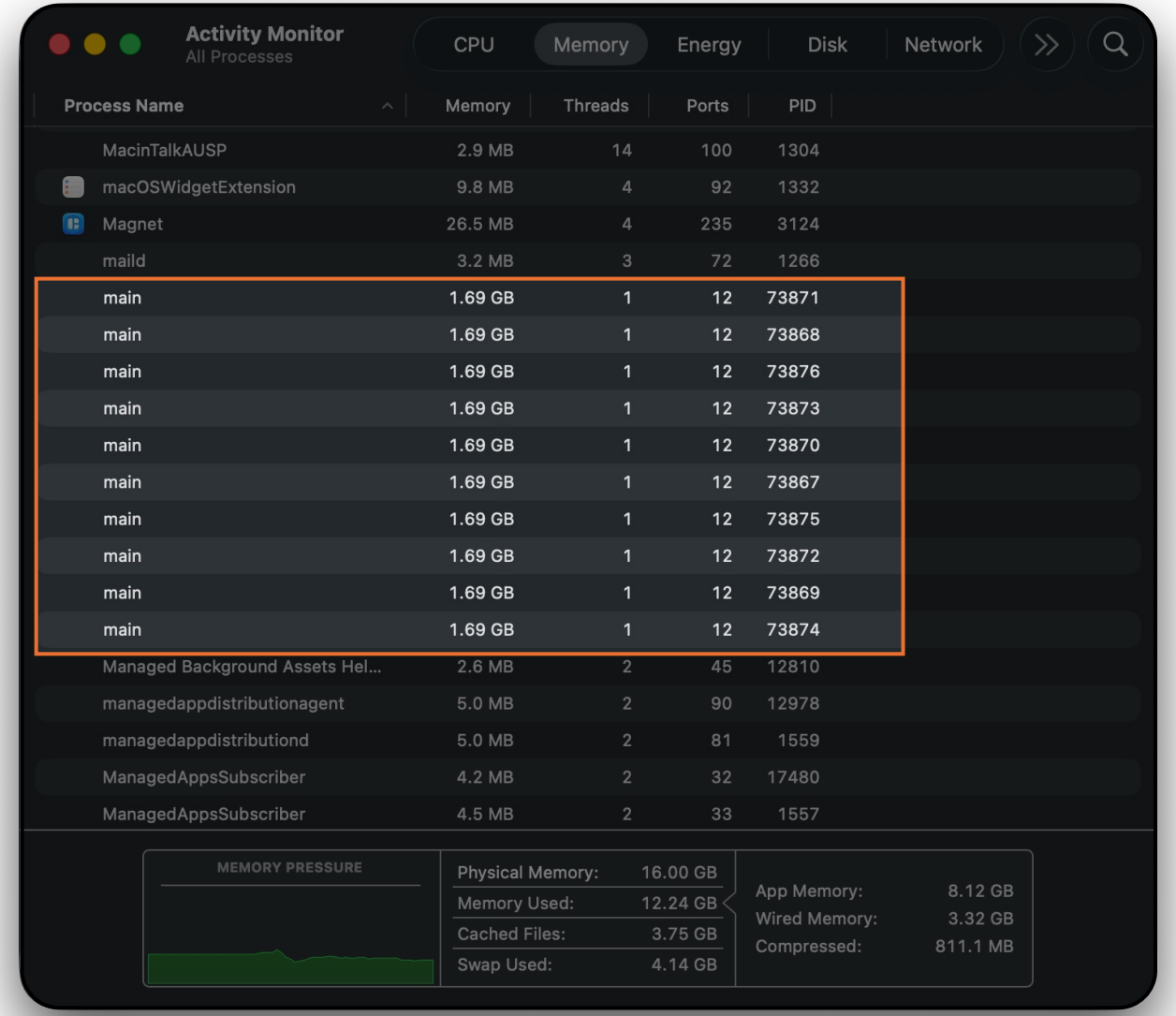
```
mpirun -n 10 ./main
```

Equivalent of **`srun`**
(I don't have SLURM on my iMac)

- Because I have not written a multithreaded code, each **process** has one thread associated with it.

- I could also run an MPI+OpenMP program with 5 process**es** and two threads *per process* to fully utilise my 10 CPU cores.

- When you schedule a job on a compute node of a supercomputer, the only process**es** that will be running are your program and some lightweight background OS services.



Recap on how MPI works

*What is a **process**? A deeper look*

- If I try to run **`main`** using 15 process**es,** I get an error:

- By default the MPI launcher (**`mpirun`** or **`srun`**) wants each **process** to have at least one CPU core dedicated to it.

- This can be overridden (oversubscription) but then the OS needs to juggle multiple process**es**.

- In typical use you will always have
  $(\text{number of processes}) \times (\text{threads per process}) = (\text{total number of CPU cores})$.

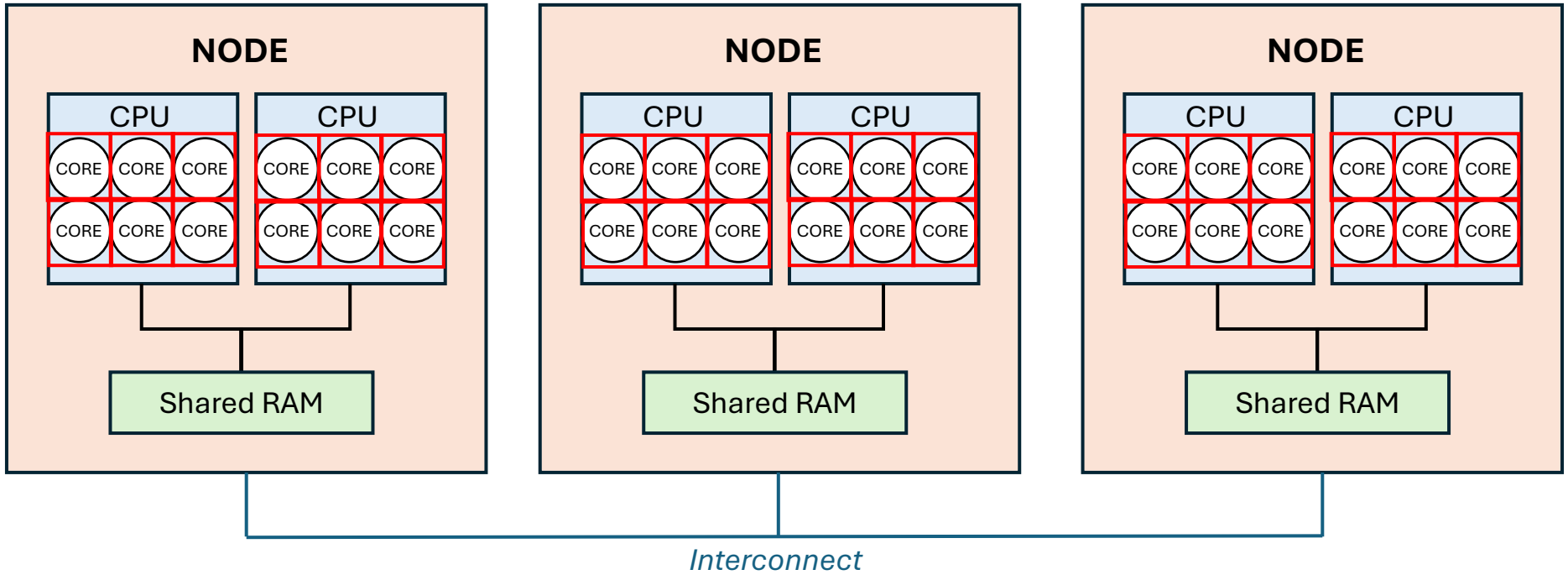
```
❯  mpirun -n 15 ./main
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 15
slots that were requested by the application:

  ./main

Either request fewer procs for your application, or make more slots
available for use.

A "slot" is the PRRTE term for an allocatable unit where we can
launch a process.  The number of slots available are defined by the
environment in which PRRTE processes are run:

  1. Hostfile, via "slots=N" clauses (N defaults to number of
     processor cores if not provided)
  2. The --host command line parameter, via a ":N" suffix on the
     hostname (N defaults to 1 if not provided)
  3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
  4. If none of a hostfile, the --host command line parameter, or an
     RM is present, PRRTE defaults to the number of processor cores

In all the above cases, if you want PRRTE to default to the number
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --map-by :OVERSUBSCRIBE option to ignore the
number of available slots when deciding the number of processes to
launch.
--------------------------------------------------------------------------
```

Recap on how MPI works

*What is a **process**? A deeper look*

Some of the ways I can associate process**es** ☐ with CPU cores:

<u>One process per core</u>



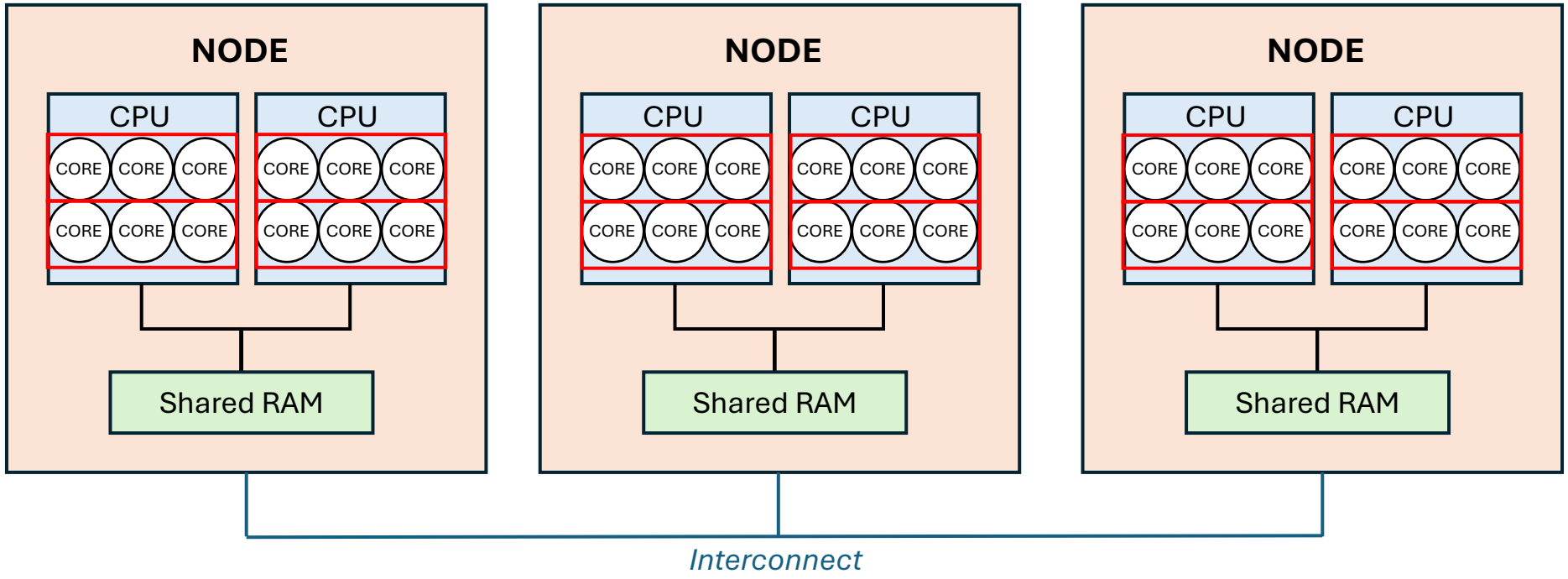
Recap on how MPI works

*What is a **process**? A deeper look*

Some of the ways I can associate process**es**  with CPU cores:

<u>Fewer processes per CPU</u>

Each process has access to multiple cores – requires multithreading to utilise.



Recap on how MPI works

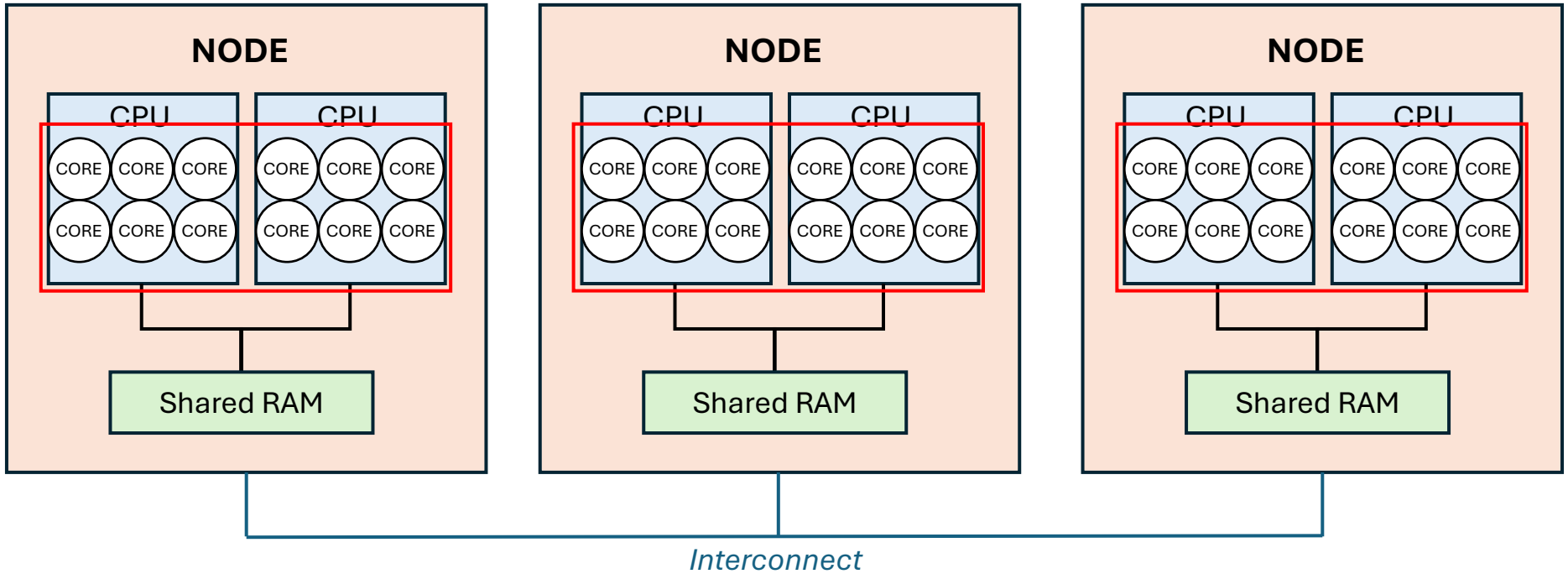
*What is a **process**? A deeper look*

Some of the ways I can associate process**es**  with CPU cores:

<u>Processes spanning cores over multiple CPUs</u>

ONLY if these CPUs have shared memory!

All threads in a process need access to the same memory space to read the program instructions (binary) and access global variables, etc.

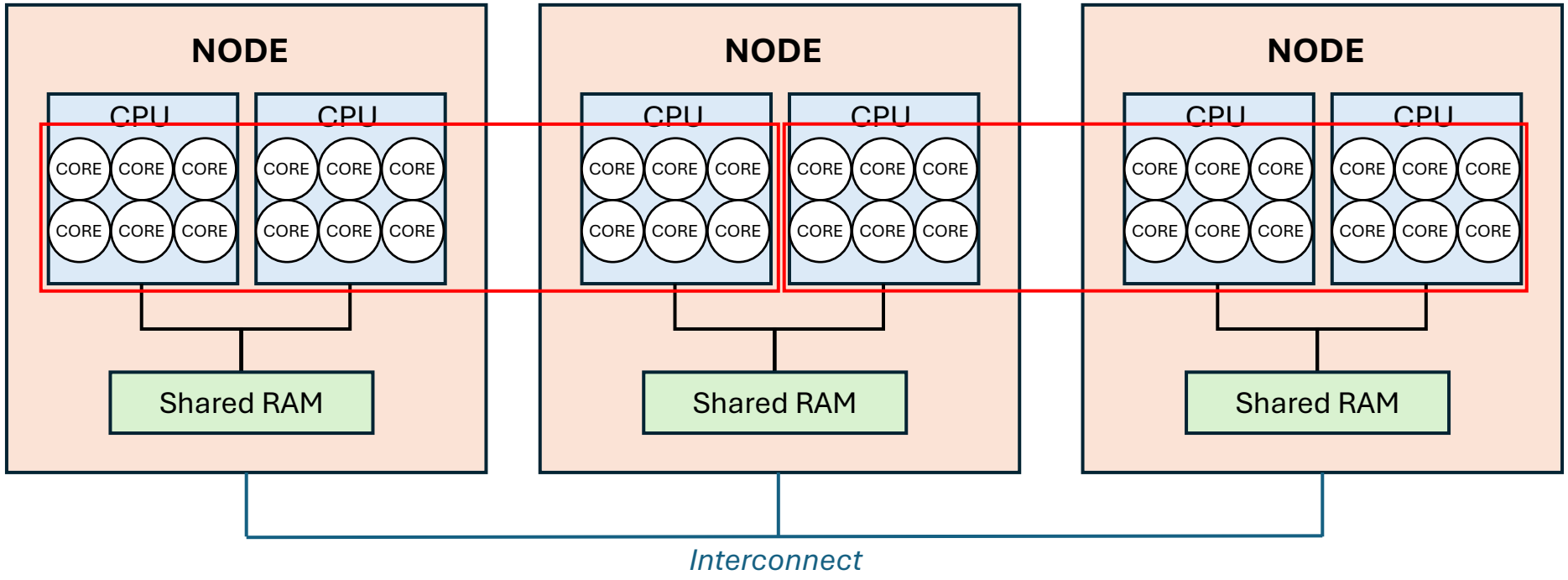


Recap on how MPI works

*What is a **process**? A deeper look*

I cannot do this!

All threads in a process need access to the same memory space to read the program instructions (binary) and access global variables, etc.

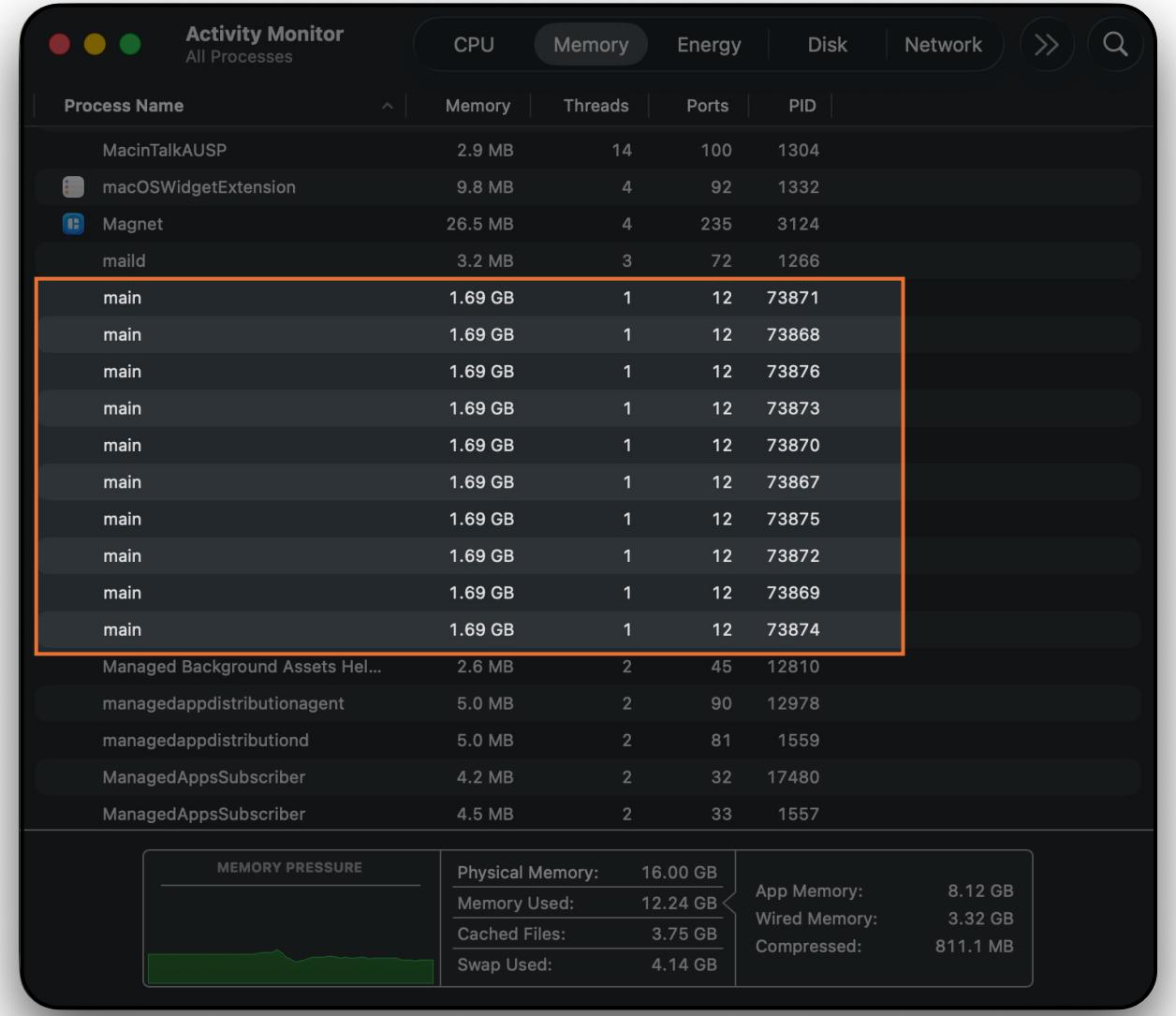


Recap on how MPI works

- These separate process**es** all run an identical program binary and execute the same set of instructions, unless we explicitly tell them to do otherwise.

- The MPI environment associates a **rank** with each processes, which we can check the value of in our program to tell different process**es** to execute different instructions.

- The MPI library provides functions to enable communication and synchronisation between different process**es**



Process/rank/task terminology

- **Process**
  - A single instance of the program being run
- **Task / MPI task**
  - Used synonymously with process
- **Rank**
  - A number assigned to each process to distinguish it from the others.
  - When we are not using custom communicators, **rank** and **process** can essentially be used interchangeably because each process is assigned only one **rank** for the duration of the program.
  - If you start using custom communicators (we won't), you just need to be careful to distinguish **rank** and **process** because one process can be assigned a different rank in different communicators.

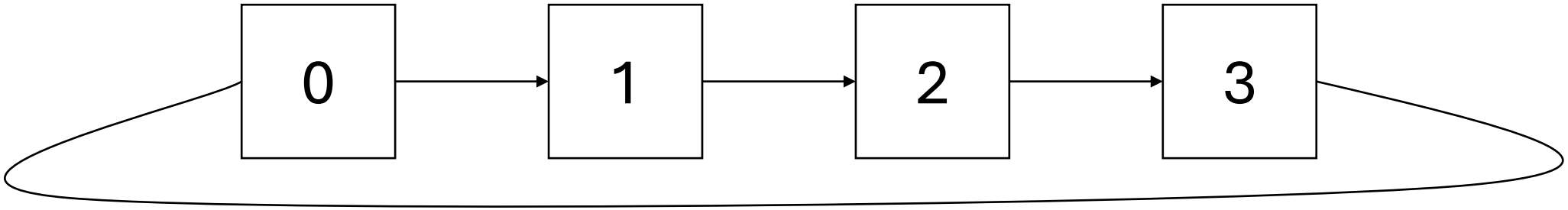
Some of the MPI functions we've learnt already

`MPI_Init`	Initialise 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_Bcast`	Broadcast same data from one process to all processes
`MPI_Scatter`	Scatter chunks of an array from one process over all processes
`MPI_Gather`	Gather chunks of an array from all processes back to one
`MPI_Reduce`	Gather chunks of an array while performing an operation
`MPI_Finalize`	Finish MPI

	Must be called by ALL ranks
	No restriction
	Must be called in send/recv pairs
	Must be called by all ranks in a communicator

Communication ring

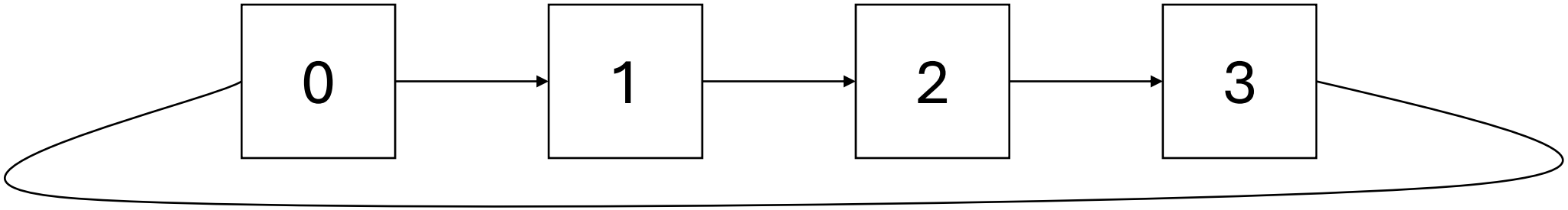
Last session we had an exercise where a single number was passed around the different processes in a ring. All ranks needed to call `MPI_Recv` and then `MPI_Send` except for **rank 0** which needed to start the communication going by calling `MPI_Send` and then `MPI_Recv`.
If not, **rank 1** would not receive the data it then needs to send on to **rank 2**, and so on.



Now lets consider a communication ring where each process sends some data to the next, and receives some other data from the previous. (Not passing the same data around as we did last week).

The send and receive operations on a single process are now uncoupled – the data being received is different to what is being sent.

What if everyone posts Send then Recv?



It may appear that each process can call `MPI_Send` and then `MPI_Recv`, but let's see what happens:

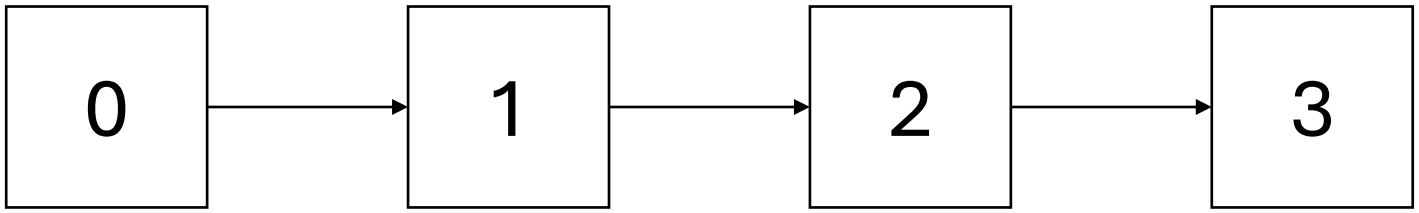
0 waits until data has been received by 1
1 waits until data has been received by 2
2 waits until data has been received by 3
3 waits until data has been received by 0

But process 0 never calls `MPI_Recv` because it is still waiting for process 1 to signal it received its message.

Deadlock! **(Provided message size exceeds the size of the internal MPI buffer).**

What if everyone posts Send then Recv?

If the final process does not send a message back to process 0 then deadlock is avoided:



0 waits until data has been received by 1
1 waits until data has been received by 2
2 waits until data has been received by 3
3 signals it received the message from 2
2 signals it received the message from 1
1 signals it received the message from 0
Program continues

Non-blocking Communication

- Non-blocking send
  - send call returns immediately, send actually occurs later
  - if the sent data is to be modified, call a wait subroutine
- Non-blocking receive
  - receive call returns immediately.
  - when received data is needed, call a wait subroutine
- Non-blocking communication can be used in attempts to overlap communication with computation
- Can help prevent deadlock

Caveat on compute/communication overlap

- Although non-blocking communications appear to give you the ability to overlap computation and communication, it may not *actually* happen:
  - On some parallel machines, the processor is actively involved in the communications and may not really be available to do other work
  - May only be able to perform reasonable computation for extremely large messages

Non-blocking Send with MPI_Isend

MPI_Isend (I for immediate return)

Fortran:

```
integer :: request
call MPI_Isend(data,count,datatype,destination,tag,communicator,request,ierr)
```

C:

```
MPI_Request request;
MPI_Isend(&data,count,datatype,destination,tag,communicator,&request);
```

The new output quantity **`request`** is a “handle” you can pass to other functions for checking the status of the communication, among other things.

You must not **alter** the information being sent (**`data`**) until the communication is complete.
The precise meaning of *complete* will depend on how MPI transmits the message (e.g. buffered or not)

Non-blocking Receive with MPI_Irecv

MPI_Irecv (I for immediate return)

Fortran:

```
integer :: request
call MPI_Irecv(data,count,datatype,source,tag,communicator,request,ierr)
```

C:

```
MPI_Request request;
MPI_Irecv(&data,count,datatype,source,tag,communicator,&request);
```

No **`status`** argument like there is for **`MPI_Recv`** as the status information is not known at the time we return from this function. Now the **`request`** handle can be used to extract status information later.

You must not **access** the information being received (**`data`**) until the communication is complete.

MPI_Wait

Input is a **request** from an **Isend** or **Irecv**

Fortran:

```
integer :: status(MPI_STATUS_SIZE)
Integer :: request
call MPI_Wait(request,status,ierr)
```

C:

```
MPI_Status status;
MPI_Request request;
MPI_Wait(&request,&status);
```

MPI_Wait blocks until the message specified by **request** completes

Completion of a **send** indicates the sender is now free to alter the data being sent

Completion of a **receive** indicates that the data array contains the received message

The **status** output contains the same information you'd get from the **status** output from an **MPI_Recv**. If you've called **MPI_Wait** on an **MPI_Isend** then the **status** is not of much use.

MPI_Waitall / MPI_Waitany

The idea is you have called multiple non-blocking communication routines and used request handles `requests[0]`, `requests[1]`, etc.

The output is an array of `status`es, each one corresponding to the request handle with the same index in the `requests` array.

Fortran:

```
integer :: reqcount, index
integer :: statuses(MPI_STATUS_SIZE,count)
integer :: requests(reqcount)
call MPI_Waitall(reqcount,requests,statuses,ierr)
call MPI_Waitany(reqcount,requests,index,statuses,ierr)
```

C:

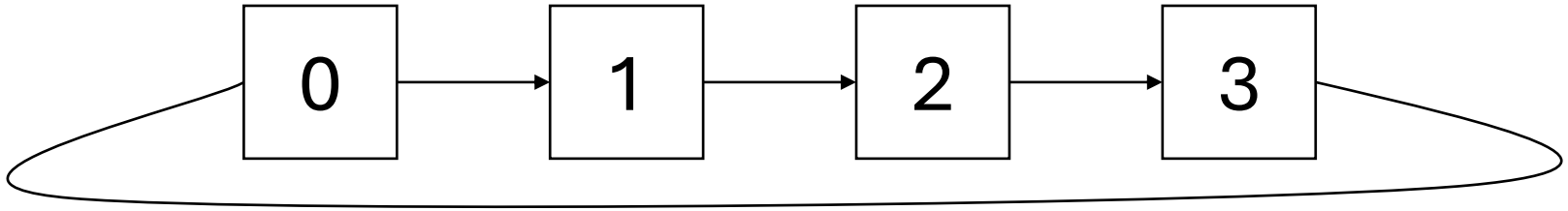
```
int reqcount, index;
MPI_Status statuses[reqcount];
MPI_Request requests[reqcount];
MPI_Waitall(reqcount,&requests,&statuses);
MPI_Waitany(reqcount,&requests,&index,&statuses);
```

`MPI_Waitall` blocks until **all** operations specified in the `requests` array have completed.

`MPI_Waitany` blocks until **the first** operation specified in the `requests` array has completed.
- the output `index` specifies which operation completed.

Communication ring deadlock: Solution 1

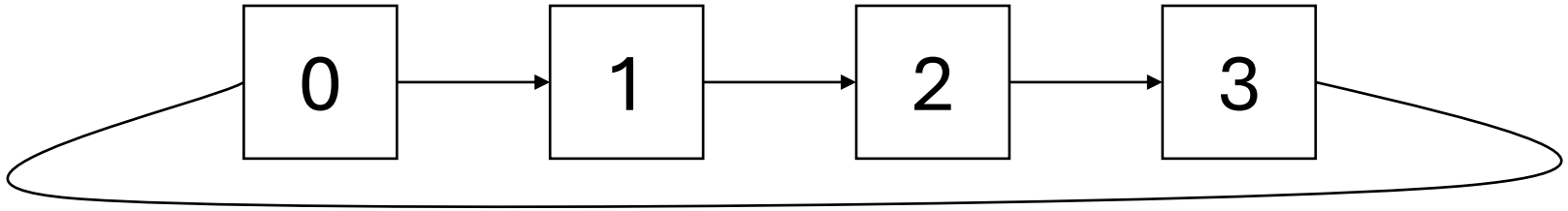
```
#include <mpi.h>

MPI_Request requests[2];
MPI_Status statuses[2];

int next = (rank + 1) % nprocs;
int prev = (rank - 1 + nprocs) % nprocs;  //ensure non-negative result

MPI_Isend(&x, datacount, MPI_INT, next, tag, MPI_COMM_WORLD, &requests[0]);
//can do work here that doesn't modify x or use y
MPI_Irecv(&y, datacount, MPI_INT, prev, tag, MPI_COMM_WORLD, &requests[1]);

MPI_Waitall(reqcount, request, statuses);
//can do work here that modifies x and uses y
```



Why does this work? Now rank 0 can call its **MPI_Irecv** before it receives confirmation that the send operation completed. Once it receives the message from the final rank the confirmation of successful send is passed through the ring back to rank 0.

Communication ring deadlock: Solution 2

```
#include <mpi.h>

MPI_Request request;
MPI_Status status;

int next = (rank + 1) % nprocs;
int prev = (rank - 1 + nprocs) % nprocs;  //ensure non-negative result

MPI_Isend(&x, datacount, MPI_INT, next, tag, MPI_COMM_WORLD, &request);
//can do work here that doesn't modify x or use y
MPI_Recv(&y, datacount, MPI_INT, prev, tag, MPI_COMM_WORLD, &status);
//can do work here that uses y

MPI_Wait(request, status);
//can do work here that modifies x and uses y
```



All that we really needed was for the **send** operation to be non-blocking.
Using a **blocking receive** and then **wait** only on the **non-blocking send** is equivalent to the previous solution (except we can use y earlier!)

MPI_Test

```
logical :: flag
call MPI_Test(request,flag,status,ierr)
```

```
int flag;
MPI_Test(&request,&flag,&status);
```

- Similar to **`MPI_Wait`**, but does not block
- Allows for computation while the message is being received
- Value of **`flag`** signifies whether or not a message has been delivered
- In C: **`flag=0`** is false and **`flag=1`** is true

MPI_Test

```
MPI_Request request;
MPI_Status status;

MPI_Isend(&x,datacount,MPI_INTEGER,next,tag,MPI_COMM_WORLD,request,ierr);

int flag = 0;
while (!flag) {
  // sit in this loop and do some other
  // work waiting for send to complete

  MPI_Test(&request,&flag,&status)
}
```

There are also **MPI_Testany** and **MPI_Testall** routines for performing tests on collections of operations.

These are analogous to the **MPI_Waitany** and **MPI_Waitall** routines.

Workshop exercise:
Non-Blocking Communication

Synchronous send with MPI_Ssend

Fortran:

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

C:

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

The same as standard **`MPI_Send`**, but it *always* blocks until the receiver posts confirmation that the message was received.

Perhaps the MPI environment will still buffer the message for transport efficiency.

But from our perspective the communication proceeds the same as if there is no buffering

- The **`MPI_Ssend`** function does not return until the send/recv handshake is complete.

Synchronous send with MPI_Ssend

Fortran:

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

C:

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

Synchronous communications can be used to guarantee your code is not relying on default buffering. If your code is correctly written, you should be able to replace all calls to **`MPI_Send`** with **`MPI_Ssend`**.

This is an excellent way to make sure that your program is avoiding possible deadlocks. Will help you avoid any surprises when you port your code to another platform, in particular ones without default buffering for small messages

Non-blocking synchronous send with MPI_Issend

Fortran:

```
integer :: request
Call MPI_Issend(data,count,datatype,destination,tag,communicator,request,ierr)
```

C:

```
MPI_Request request;
MPI_Issend(&data,count,datatype,destination,tag,communicator,&request);
```

The same syntax as **`MPI_Isend`**, but *synchronous*.

Wait, what?

Let's clarify some terminology:

- **Blocking** operations **do not** return until the operation is ***complete*.**

- **Non-blocking** operations return immediately.
  You must check to see when the operation is *complete* later.

The above says nothing about what defines the ***completion*** of an operation.

- In an `MPI_Bsend` (user-requested buffered send), or an `MPI_Send` with default buffering, the operation is ***complete*** as soon as the data is copied into the MPI buffer. The sender is safe to modify the data without corrupting the message, but there is no guarantee if/when the message has been received.

- In an `MPI_Ssend`, or an `MPI_Send` with no default buffering, the operation is ***complete*** once the message is received by the destination rank.

Wait, what?

Let's clarify some terminology:

- **Blocking** operations **do not** return until the operation is ***complete*.**

- **Non-blocking** operations return immediately.
  You must check to see when the operation is *complete* later.

The above says nothing about what defines the ***completion*** of an operation.

- In an **`MPI_Isend`**, the definition of *complete* is the same as for **`MPI_Send`**. If the MPI environment decides to buffer the message, then **`MPI_Wait`** or **`MPI_Test`** will say the communication is complete as soon as the data is copied into the MPI buffer.

- In an **`MPI_Issend`**, the definition of *complete* is the same as for **`MPI_Ssend`**. Even if the MPI environment buffers the message, **`MPI_Wait`** and **`MPI_Test`** will not say the communication is complete until the message is received by the destination rank.

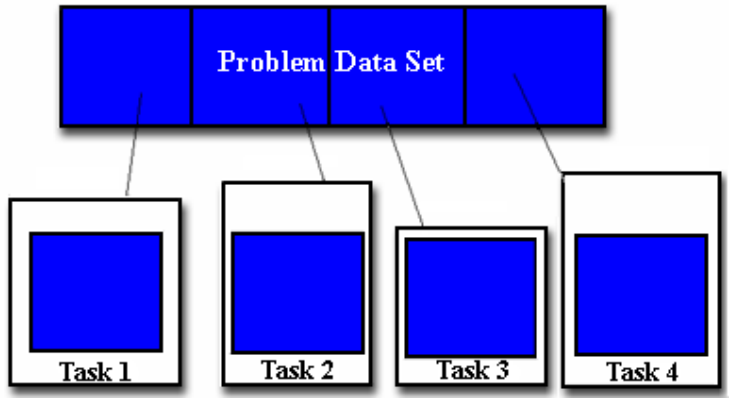
Workshop exercise:
Synchronous Communication

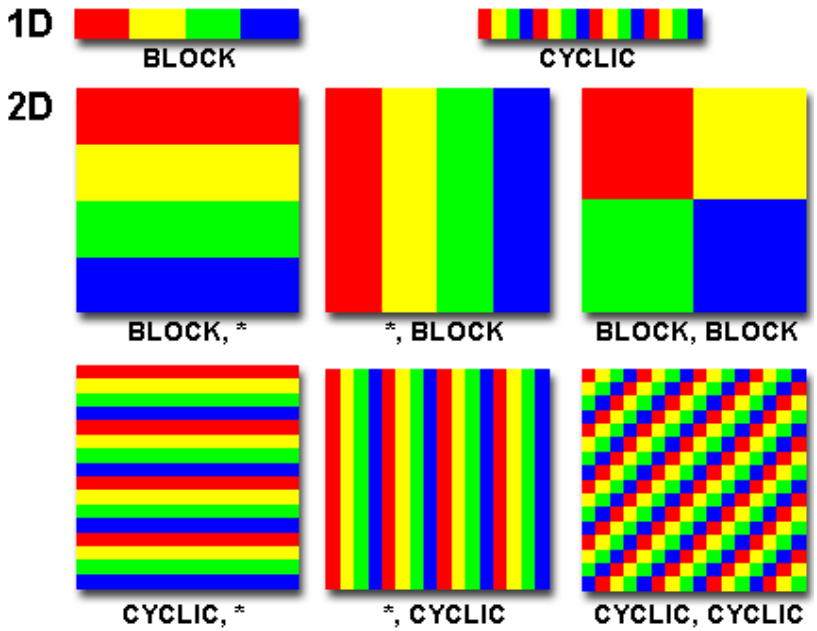
Designing parallel programs: Partitioning

- **Partitioning**
  - One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple processes. This is known as decomposition or partitioning.
  - There are two basic ways to partition computational work among parallel processes:
    - ***domain decomposition***
    - ***functional decomposition***

Designing parallel programs: Partitioning

- **Domain Decomposition**
  - In this type of partitioning, the *data* associated with a problem is decomposed.
  - Each parallel process then works on a separate portion of the data.
  - There are different ways to partition data
    - e.g. block, cyclic



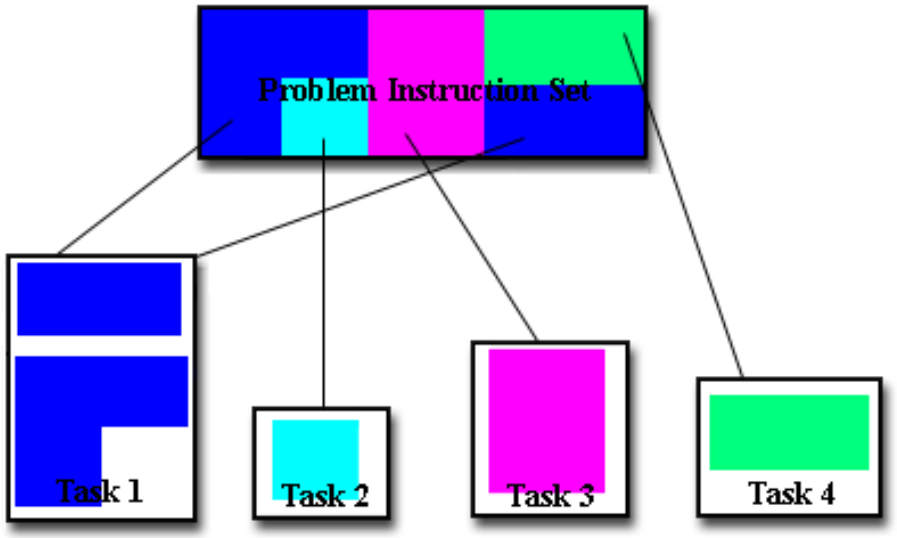


Designing parallel programs: Partitioning

- **Functional Decomposition**

  - In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
    The problem is decomposed according to the work that must be done.
    Each process then performs a portion of the overall work.

  - Functional decomposition lends itself well to problems that can be split into different sub-problems.



Designing parallel programs: Communications

- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for ranks to share data.

    - E.g. imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can be distributed to multiple processes that then act independently of each other to do their portion of the work.

  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward.
    Very little inter-process communication is required.

Designing parallel programs: Communications

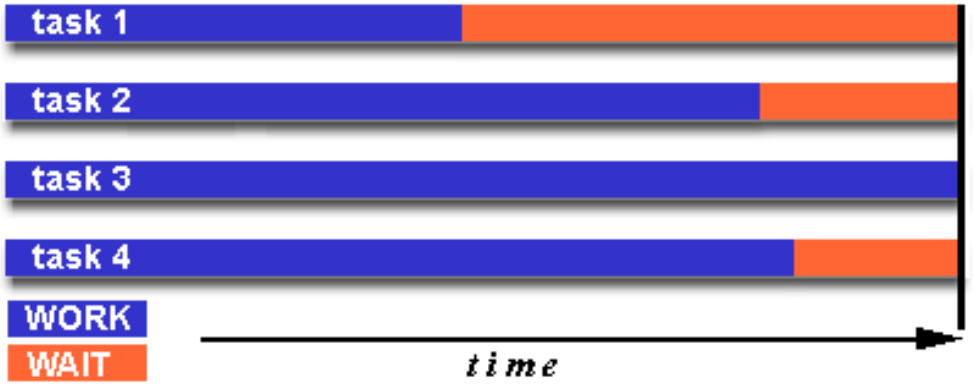
- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require processes to share data with each other. For example, a 3-D heat diffusion problem requires that each process knows the value of data held by the neighboring processes.

- **Cost of communications**
  - Inter-process communication often implies overhead.
  - Machine cycles that could be used for computation are instead used to package and transmit data.
  - Communications frequently require synchronisation between processes, which can result in processes spending time "waiting" instead of doing work.

Designing parallel programs: Load balancing

- **Load Balancing**

  - the practice of distributing work among processes so that *all* processes are kept busy *all* of the time. It can be considered a minimisation of process idle time.



  - Load balancing is important to parallel programmes for performance reasons. For example, if all processes are subject to a barrier synchronisation, the slowest process will determine the overall

Designing parallel programs: Load balancing

- **Achieving Load Balance: Equally partition the work each process receives**

- For array/matrix operations where each process performs similar work, evenly distribute the data set among the processes.

- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the processes.

- Load imbalance can be detected using performance analysis tools, or you can use timing functions (e.g. `MPI_Wtime`) to measure how long each process spends doing useful work.

Designing parallel programs: Load balancing

- **Load Balance: dynamic work assignment**

  - Certain classes of problems result in load imbalances even if data is evenly distributed
    - *Sparse arrays* - some processes will have actual data to work on while others have mostly zeros
    - *Adaptive grid methods* - some processes may need to refine their mesh while others don't
    - *N-body simulations* - some particles may migrate to/from their original domain to another process's domain; or where the particles owned by some processes require more work than those owned by other processes.

  - Sometimes helpful to use a ***Master–Worker*** approach. As each process finishes its work, it queues to get a new piece of work.

Designing parallel programs: Speedup

- ***Speedup*** is the time taken by the sequential solution divided by the time taken by the parallel solution
  - e.g. the serial version of a code takes 100 secs to run, while the parallel version takes 25 secs → speedup=4
  - Greater than 1. Ideally equal to number of processes

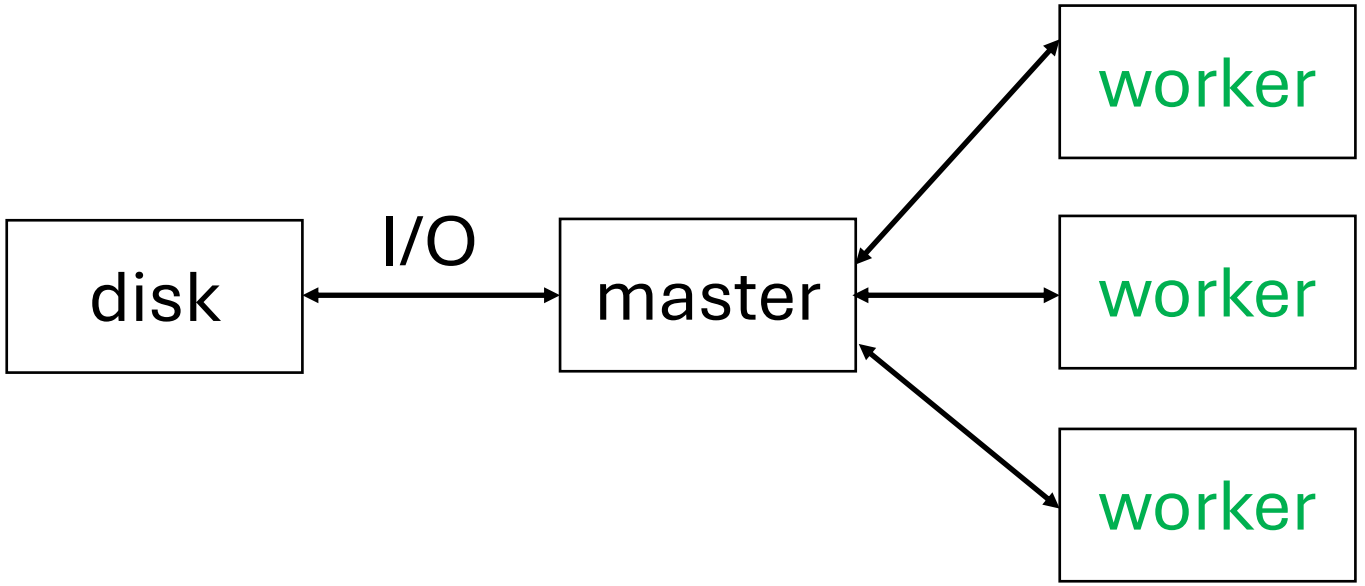
- **Amdahl's Law**
  - The maximum speedup is given by $$\frac{1}{(P/N) + S}$$

    where P = parallel fraction,
    N = number of processes and S = serial fraction.

  - Ultimately the speedup of your program is limited by the serial part.

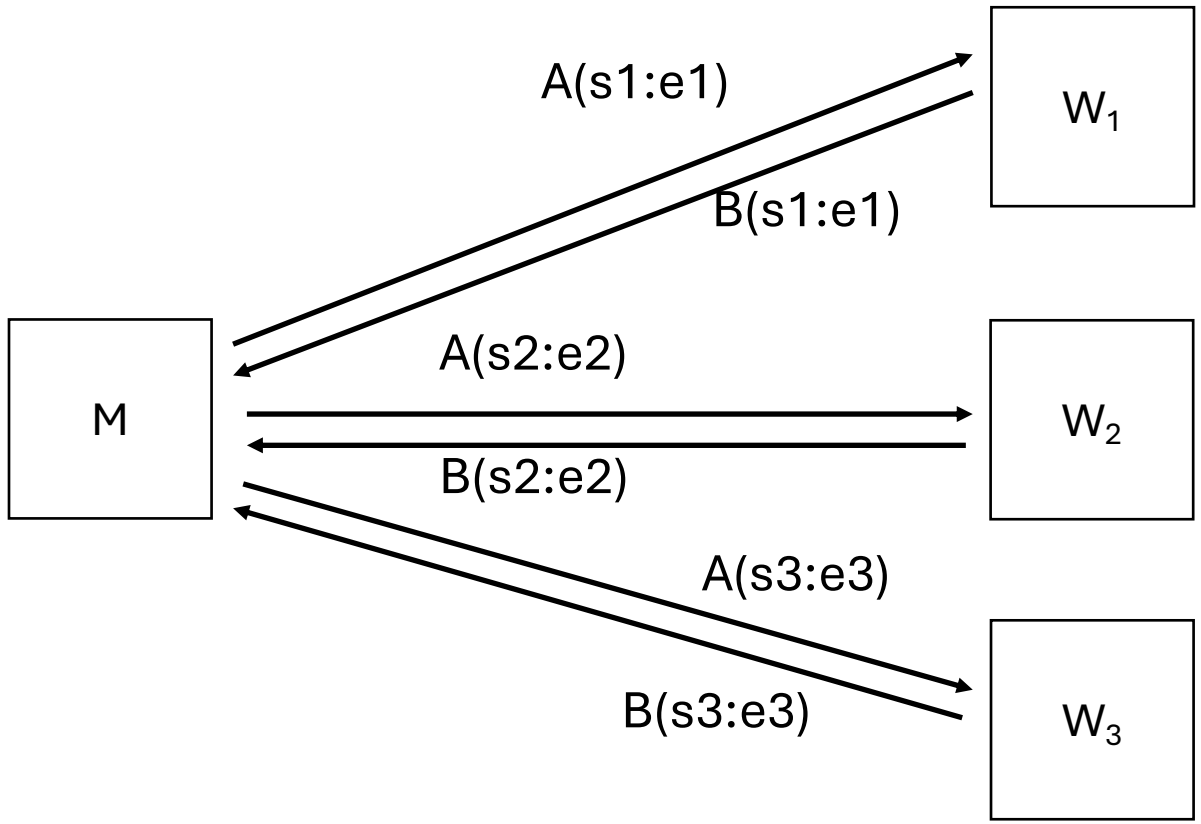
Master/worker parallelism

- A simple programming model where one process acts as the "*master*" and distributes work to the "*worker*" processes



Master/worker parallelism: example

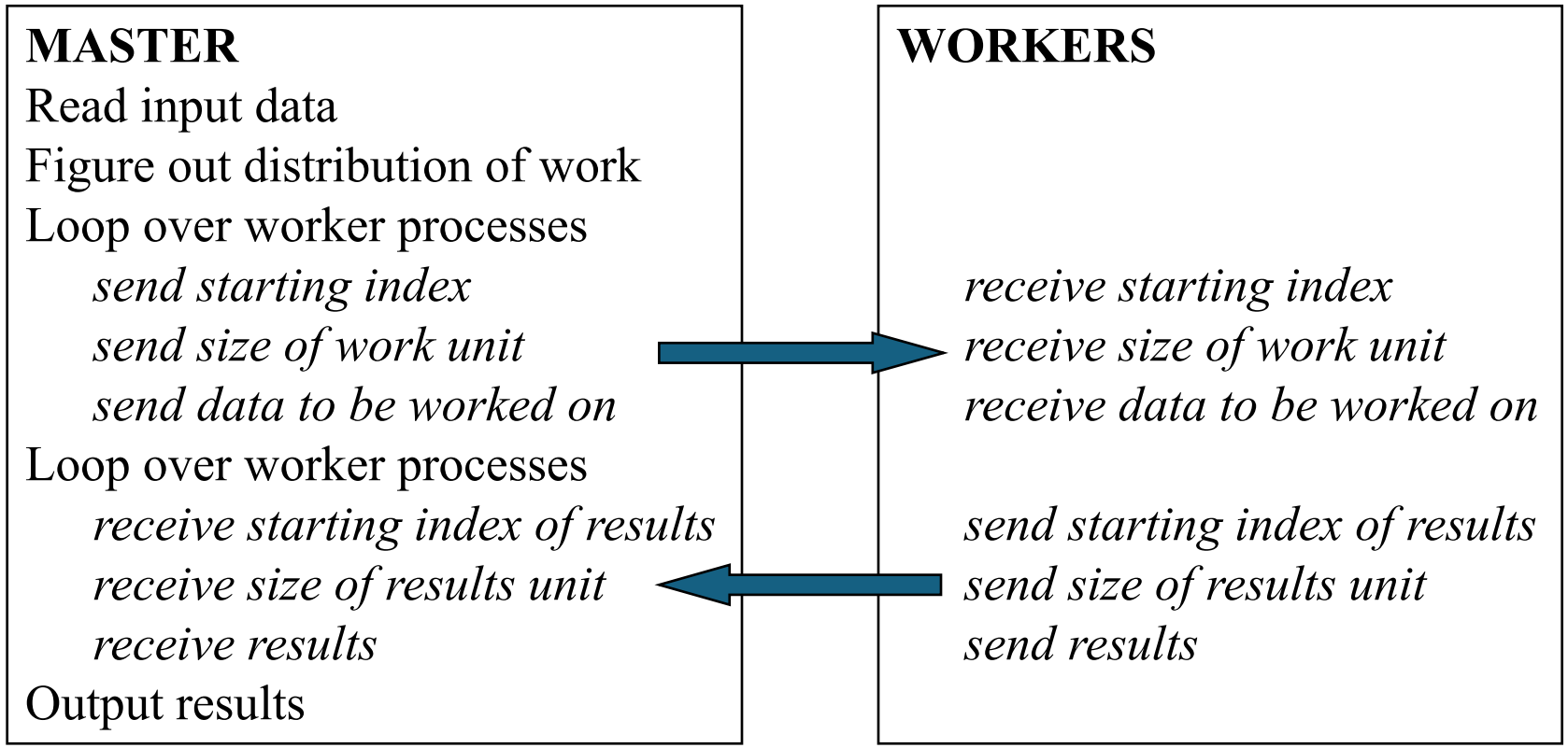
- Elements of A_i are distributed to workers which evaluate $B_i=f(A_i)$ and then pass results back to the master




```
master = 0
if (rank == master) {
   perform I/O, distribute work, collect
   results, etc.
}

if (rank != master) {
   receive work, do calculations, return
   results, etc.
}
```

Master/worker parallelism: Static decomposition

- Master process decides on the distribution of work at the start
- Excellent where the amount of time required for each parcel of work is known at the start of the simulation



Master/worker parallelism: Static decomposition

Calculating the work

```
if (rank == 0) { //only master determines work allocation
   numworkers = nprocs - 1;
   //how much work to give each process:
   chunksize = datasize/numworkers; //integer division

   remainder = datasize%numworkers;

   for(int dest=1; dest<=numworkers; dest++) {
      if (dest <= remainder) {
            //these processes get extra work to cover the remainder
            portion[dest] = chunksize + 1;
      }
      else {
            portion[dest] = chunksize;
      }
   }
```

Master/worker parallelism: Static decomposition

Distributing the work

```
if (rank == 0) {
  index = 0;
  for(int dest=1; dest<=numworkers; dest++) {

    MPI_Send(&index,1,MPI_INT,dest,itag,MPI_COMM_WORLD);

    MPI_Send(&portion[dest],1,MPI_INT,dest,ptag,MPI_COMM_WORLD);

    MPI_Send(&x[index],portion[dest],MPI_INT,dest,atag,MPI_COMM_WORLD);

    index = index + portion[dest]

  }
}
```

Master/worker parallelism: Static decomposition

Collecting the results

```
if (rank == 0) {
  index = 0;
  for(int source=1; source<=numworkers; source++) {

    MPI_Recv(&index,1,MPI_INT,source,itag,&status,MPI_COMM_WORLD);

    MPI_Recv(&portion[dest],1,MPI_INT,source,ptag,&status,MPI_COMM_WORLD);

    MPI_Recv(&y[index],portion[dest],MPI_INT,source,atag,&status,MPI_COMM_WORLD);

    index = index + portion[dest]

  }
}
```

Master/worker parallelism: Static decomposition

On the worker processes

```
if (rank != 0) {
  MPI_Recv(&index,1,MPI_INT,0,itag,MPI_COMM_WORLD,&status);
  MPI_Recv(&portion[dest],1,MPI_INT,0,ptag,MPI_COMM_WORLD,&status);
  MPI_Recv(&x[index],portion[dest],MPI_INT,0,atag,MPI_COMM_WORLD,&status);

  //operate on x[index]..x[index+portion-1]

  MPI_Send(&index,1,MPI_INT,0,itag,MPI_COMM_WORLD);
  MPI_Send(&portion[dest],1,MPI_INT,0,ptag,MPI_COMM_WORLD);
  MPI_Send(&y[index],portion[dest],MPI_INT,0,atag,MPI_COMM_WORLD);
}
```

Master/worker parallelism: Static decomposition

Ways to simplify

- We could have all processors calculate their portion size and avoid one set of matching sends and receives.

- Do the workers really need to know the starting index of their array chunks?
  - master can keep a list of the index for each worker

- We also could use …
  - `MPI_Scatter` to distribute the work
  - `MPI_Gather` to collect the work
  - Need to use `MPI_Scatterv` and `MPI_Gatherv` when the data isn't distributed equally to each process

Master/worker parallelism: Dynamic decomposition

- Master processor distributes out new work as workers finish their previous work
  - Excellent for calculations where the work needed for solving each individual problem varies significantly.
  - Example: iterative schemes where number of steps required for convergence can have a large variation.
- Make sure that parcels of work are large enough so that you don’t spend all of your time in communications
- Make sure that parcels of work are small enough so that you achieve load balancing

Master/worker parallelism: Dynamic decomposition

Overview

MASTER
Read input data
Establish a chunksize
Loop over worker processes
    hand out initial parcel of work
While work remains to do
    get results from finished worker
    and hand out new parcel of work
Loop over worker processes
    tell all workers to quit

WORKERS

While work remains to do
    get work from master
    do work
    send results to master

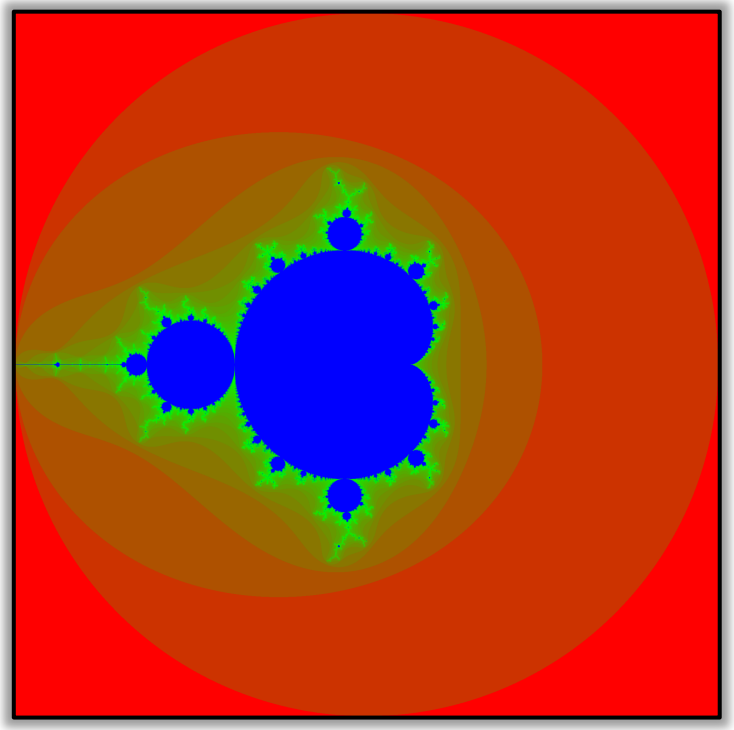
Master/worker parallelism: Dynamic decomposition

Receiving the results

- The heart of the dynamic load balancing scheme:
  The master receives work back from whoever finishes first by posting a receive from any source

```
MPI_Recv(&indexrecv,1,MPI_INT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,&status);
source = status.MPI_SOURCE;

. . receive results from source ...
. . send new work to source ...
```

Assessable task: Mandelbrot set



- For every point κ on the complex plane define an iterative sequence

$$z_{i+1} = (z_i)^2 + \kappa$$

- Starting value is $z_0 = \kappa$, then:

$$z_1 = (z_0)^2 + \kappa = \kappa^2 + \kappa$$
$$z_2 = (z_1)^2 + \kappa = (\kappa^2 + \kappa)^2 + \kappa$$
$$z_3 = (z_2)^2 + \kappa = [(\kappa^2 + \kappa)^2 + \kappa]^2 + \kappa$$
$$\vdots$$

- Keep in mind all quantities are complex when defining the squaring operation
- A point κ is **in** the Mandelbrot set if the sequence is bounded as $i \to \infty$

- It is known that the Mandelbrot is bounded by the disk $|\kappa| < 2$, so we only need to iterate until $|z_i| > 2$ to determine that a point is not in the set.

- In practice, have a maximum number of iterations **`MAXITER`**, and if the sequence does not leave the $|\kappa| < 2$ disk within that many iterations consider it in the set.
  - As you increase **`MAXITER`** you get more detail in the rendering

mandelbrot.c - provided

```
// Iterate over 2-D grid: i covers N cols and j covers N rows
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {

    //kappa is the coordinate in the complex plane of the (i,j)th pixel
    //both the real and imaginary axis range from -2 to 2 (4 units)
    kappa = (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;

    //z_0 = kappa, initial value of z before iteration begins
    z = kappa;

    //now we iterate until either the max number of iterations is reached
    //OR z leaves the circle of radius 2 centred at the origin
    k=1;
    while ((cabs(z)<=2) && (k++<MAXITER)) {
      z = z*z + kappa;
    }

    // Store the number of iterations required for this point to leave the Mandelbrot set.
    // If this point IS IN the Mandelbrot set, then k=MAXITER.
    x[i][j]=log((float)k) / log((float)MAXITER);
  }
}
```

Provided code will handle evaluating the Mandelbrot set and printing to file

`x` is a 2D array representing the complex plane. The number of iterations it took to leave the Mandelbrot set is used to colour the output

mandelbrot2.c - provided

This code collapses the 2D grid into a 1D array for ease of distributing work to different MPI ranks

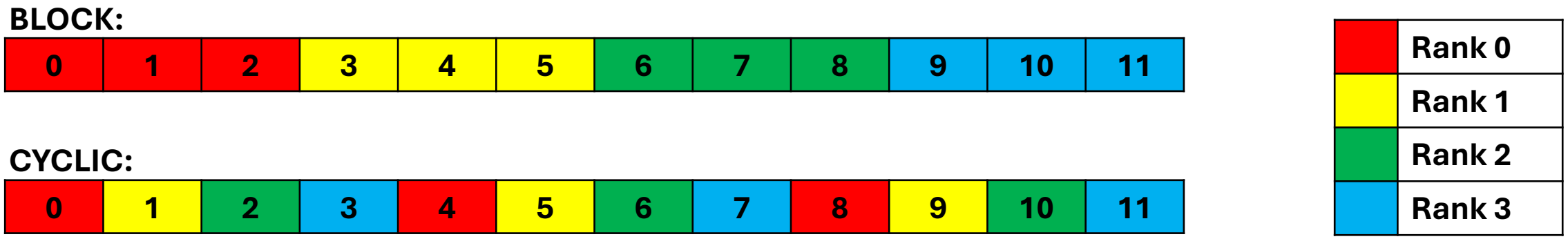
```
//Loop over N^2 pixels
for (loop=0; loop<N*N; loop++) {
  // i:cols and j:rows

  // i=mod(loop,N) means i iterates from 0...N-1 then back to 0...N-1 etc.
  i=loop%N;

  // j=loop/N is integer division.
  // For the first  round of i=0...N-1, loop < N, and hence j=0.
  // For the second round of i=0...N-1, N <= loop < 2*N, and hence j=1.
  // This way we fix the row and iterate over all columns
  // and then increment the row the iterate over all columns again, etc...
  j=loop/N;

  //In here is the same evaluation of the Mandelbrot set as in mandelbrot.c

  x[loop]= log((float)k) / log((float)MAXITER);
}
```

`x` is a 1D array now

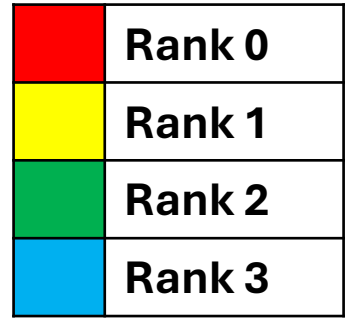
Partitioning

- In two of the assignment tasks you will be required to implement a static domain decomposition.
- The work of evaluating the 1D array `x` should be distributed over the different MPI ranks.
- The different parts of `x` should then be compiled on rank 0 using **`MPI_Gather`**
- You will need to implement both BLOCK and CYCLIC distributions of the `x` array

E.g. let's say `x` has 12 elements which we distribute over 4 ranks with a chunk size of 3:

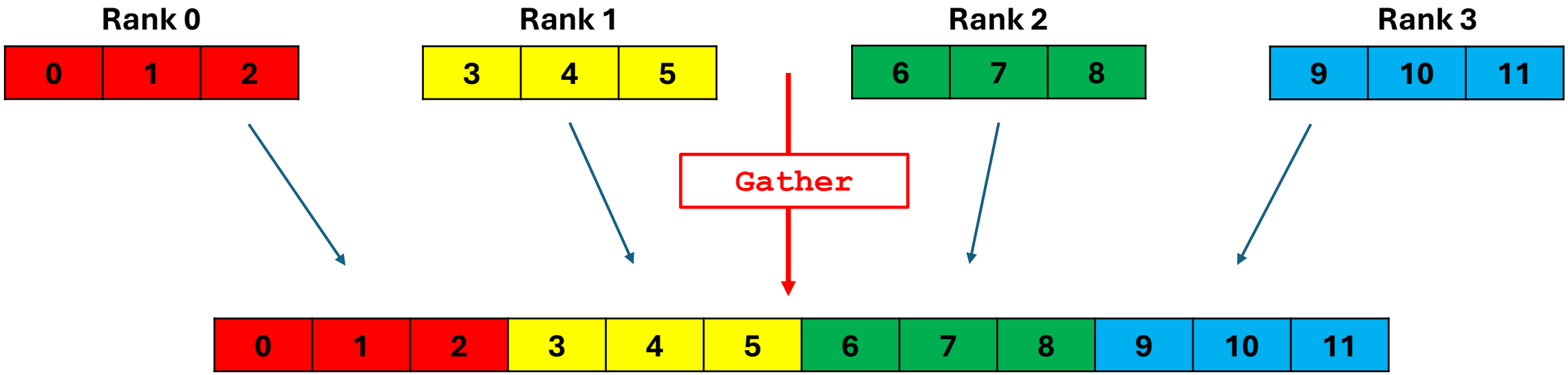


BLOCK partitioning



BLOCK partitioning is simple to implement.

Each rank works on a contiguous section of the array:



Then **`MPI_Gather`** brings the sections back *in the proper order*

BLOCK partitioning **Some suggestions...**

In the provided serial code there is an iterator called `loop` that iterates over the elements of `x`:

	0	1	2	3	4	5	6	7	8	9	10	11
loop:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*

You may implement the **block** partitioning with each rank having an iterator `l` that goes from 0 to chunksize-1

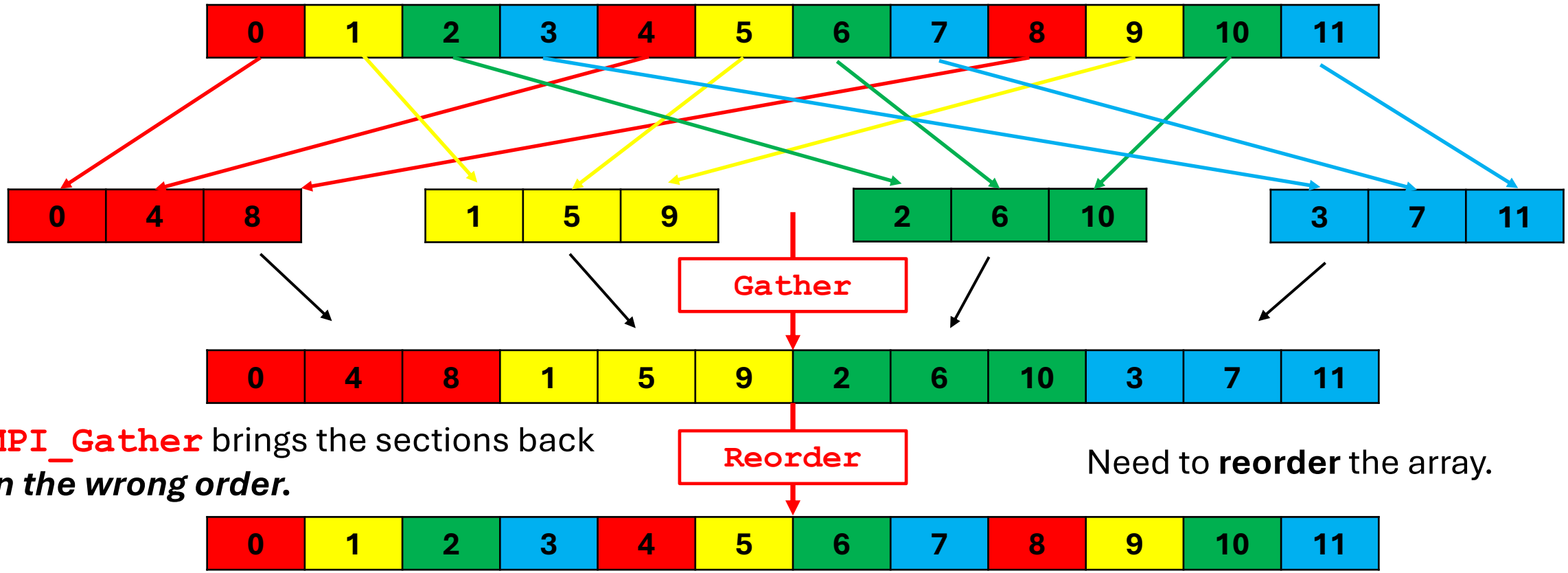
	0	1	2	3	4	5	6	7	8	9	10	11
rank	*0*	*0*	*0*	*1*	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*
l:	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*
loop:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*

It is straightforward to see the relationship between the three variables, which you can code so that each rank knows the value of the index **`loop`** in the global array for each index **`l`** in its local array.

Then you can reuse **`i=loop%N; j=loop/N;`** from the serial code to determine the pixel coordinates.

CYCLIC partitioning **requires some more thought…**

Each rank works on a **non-contiguous** section of the array.
In our case we don't need to **scatter** data for the ranks to work on.
Just need a way for each rank to know which parts of the array it is assigned



`MPI_Gather` brings the sections back ***in the wrong order.***

Need to **reorder** the array.

CYCLIC partitioning **Some suggestions…**

You may implement the **cyclic** partitioning with each rank having an iterator `l` that goes from 0 to chunksize-1

`rank`	*0*	*0*	*0*	*1*	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*
`l:`	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*
`loop:`	*0*	*4*	*8*	*1*	*5*	*9*	*2*	*6*	*10*	*3*	*7*	*11*

Although it is more complicated now, you can still determine the relationship between the three variables

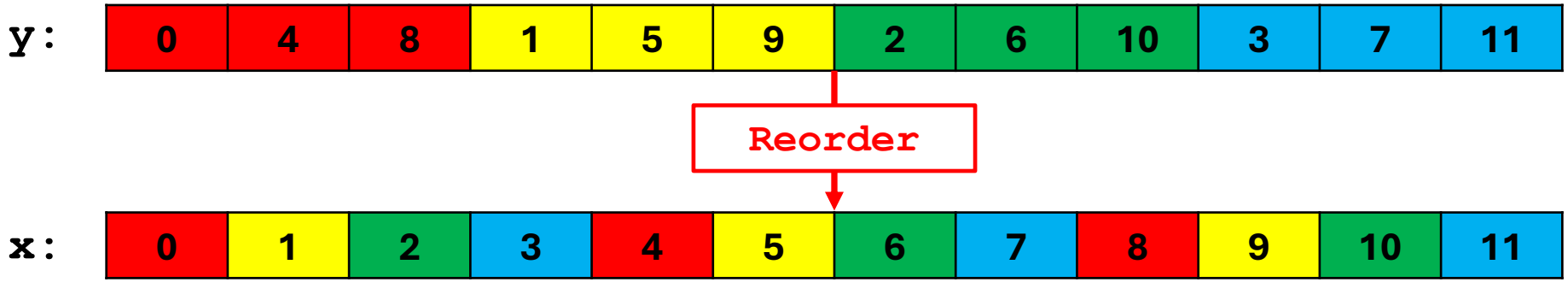
Then each rank knows the value of the index **`loop`** in the global array for each index **`l`** in its local array.

Then you can reuse **`i=loop%N; j=loop/N;`** from the serial code to determine the pixel coordinates.

CYCLIC partitioning

Some suggestions...

The reordering process is simplest if you just **MPI_Gather** into a secondary array **y** and then copy elements into **x** in the right order. (This all happens on rank 0 only now).



You may like to have a nested loop over ranks and the index in the local arrays:

```
for (int r=0; r<nprocs; r++) {
  for (int l=0; l<chunksize; l++) {
    loop = ???; //same relationship between rank and local index you already found
    yind = ???;
    x[loop] = y[yind];
  }
}
```

yind is the index in the y array for a given rank and local index. This is a simple offset relation accounting for the local arrays being stacked side-by-side in the **y** array.

Master/worker

You will also be asked to implement a master/worker solution.
The simplest way you can approach this is:

- All worker ranks initially start work on the chunk starting at `(rank-1)*chunksize`

- Master rank sits in a while loop, and at each iteration:
  - Master rank receives a chunk from MPI_ANY_SOURCE and saves it.
  - Master rank then sends the index of the next chunk to start work on to whichever worker sent last chunk.
  - The master rank therefore must keep track of and increment this index each time it sends a new request.

- Worker ranks are also sitting in a while loop, and at each iteration:
  - Send the chunk they have finished working on to the master rank.
  - Receive the index of the new chunk to work on from the master rank.

Master/worker

What should the conditions be on the `while` loops?

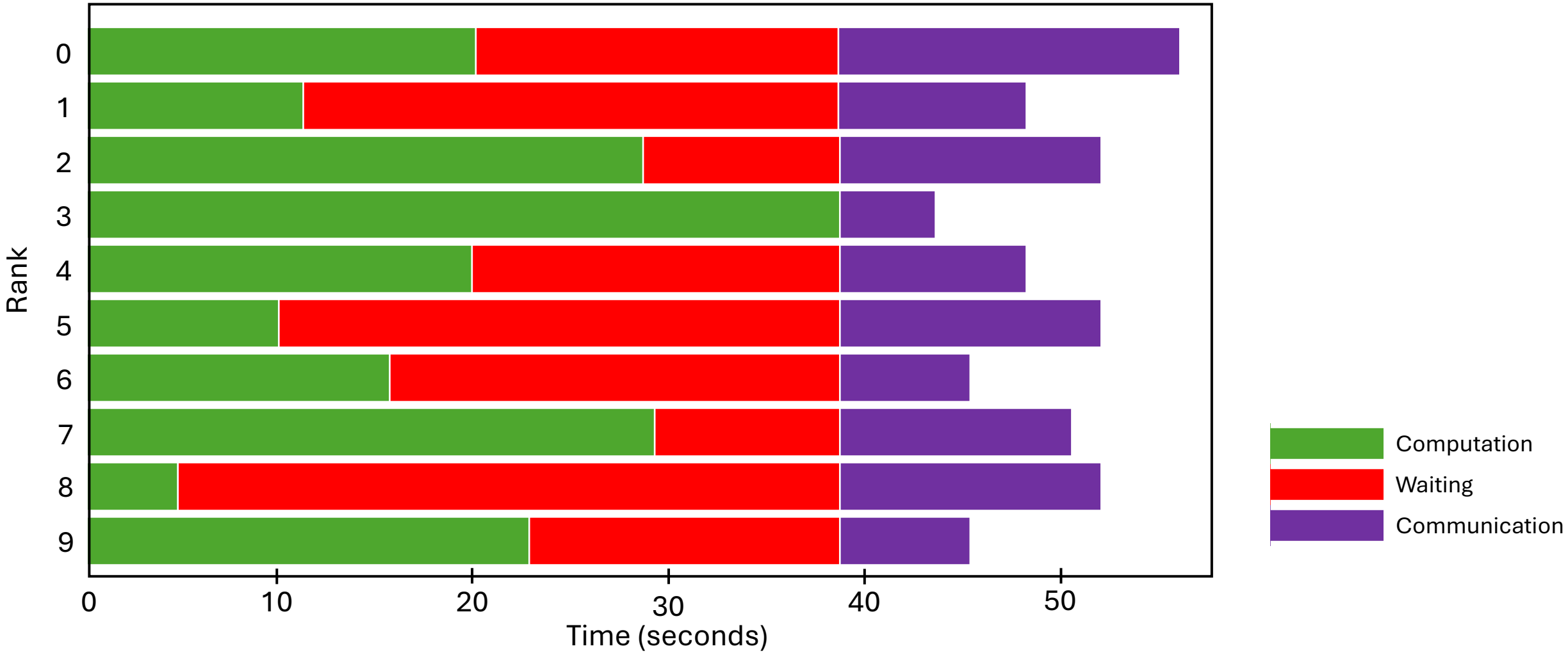
- If the master rank is updating the value of the chunk starting index every time it sends a new request, it will eventually reach a point where the index exceeds the number of pixels to work on.
  - When a worker receives an index outside the bounds of the pixel array it can interpret this as a message that the work is complete and exit its `while` loop.

- The master rank cannot simply exit its while loop once the index goes out of bounds, because it needs to send that “out of bounds index message” to all worker ranks to signal that work is finished.
  - The master rank needs to keep track of how many workers it has told to stop work and then exit its own while loop once everyone has been informed.

Timing visualisation

You'll be asked to make figures in the same format as shown here to indicate how long each process spends on:

- **Computation**: Performing the Mandelbrot algorithm
- **Waiting**: Waiting for other processes to finish their Mandelbrot algorithm.
- **Communication**: Time spend in the blocking `MPI_Gather` routine.



Workshop exercise:
Understanding Memory

Then start on assignment