Parallelism

Threads and OpenMP

- In a reduction, a local copy of each list (target) variable is made and initialised for each thread
- Updates occur on the local copy
- Local copies are reduced to a single value and combined with the original global value
- The same effect as a reduction can be achieved using other directives eg. `#pragma omp critical` (see next)
- However, reduction is *more scalable*

Parallelism

Threads and OpenMP

- Other ways of controlling thread access are using:
  - Barriers - force each thread to wait until all of the threads have finished executing

    `#pragma omp barrier`

  - Mutual exclusion - define a block of the code that only one thread at a time can execute (sort of defeats the point of adding the parallelism…)

    `#pragma omp critical` or

    `#pragma omp atomic` (for memory updates only)

    - We can also use `#pragma omp critical` to stop false sharing
    - All of the above can be used to prevent race conditions

Parallelism

Threads and OpenMP

```
#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

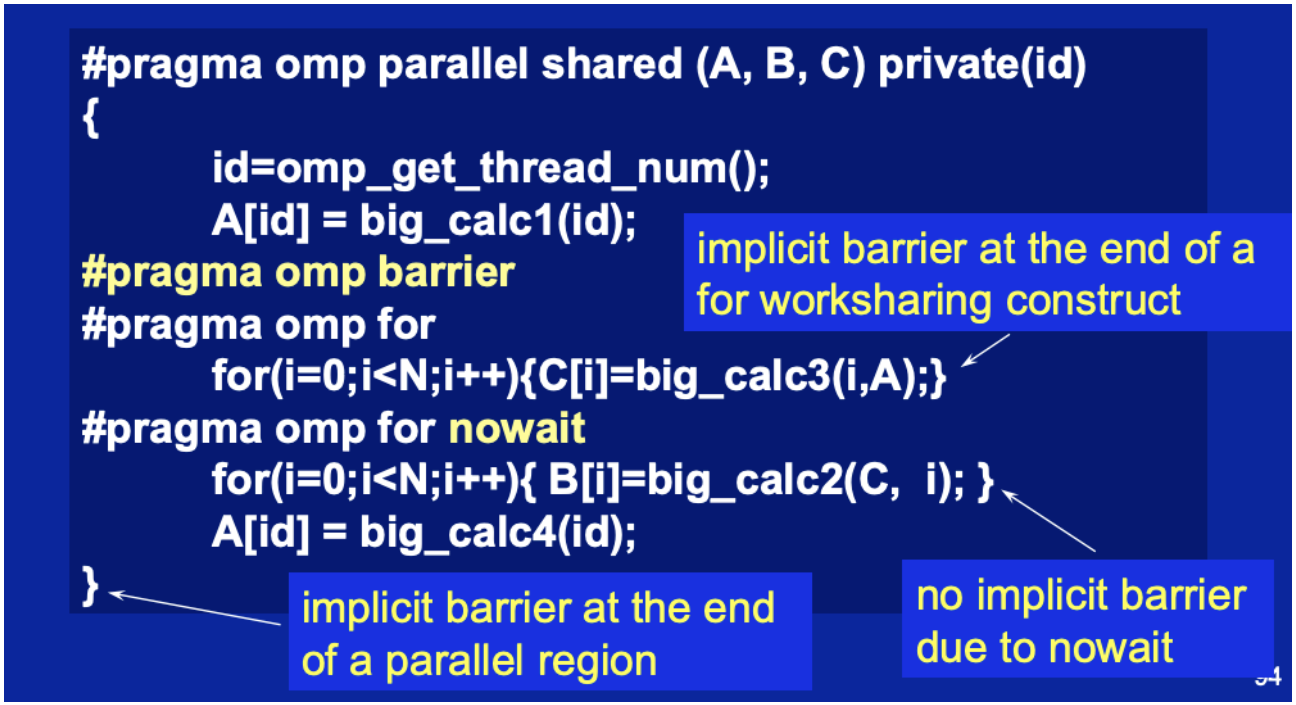
```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
           B =  big_job(i);
#pragma omp critical
           res += consume (B);
     }
}
```

Parallelism

Threads and OpenMP

- Barriers are *implicit* at the end of work sharing constructs such as `#pragma omp for`, and at the end of parallel regions

- To get rid of this implicit barrier, we add `nowait`, eg.



Parallelism

Threads and OpenMP

- Variables within a worksharing construct (eg. `#pragma omp for`) are `private` by default
- This can be achieved explicitly using eg. `#pragma omp for private(x)`, where `x` are the variables inside the construct - this creates a new local (uninitialised) copy of `x` for each thread
- Outside working constructs, variables are `shared`
- Default attributes can be overridden using `default(private|shared|none)`
- We can also use:
  - `firstprivate` - initialises from shared variable
  - `lastprivate` - passes last value out to shared variable

Parallelism

Threads and OpenMP

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Parallelism

Threads and OpenMP

- Other useful features of OpenMP that we will not cover in detail include
  - `#pragma omp master` denotes a structured block that is only executed by the master thread
  - `#pragma omp single` - only executed by one thread (not necessarily the master thread)
  - `#pragma omp sections` - gives a different structured block to each thread
  - Simple and nested locks - these produce a memory fence, flushing all thread visible variables

Parallelism

Threads and OpenMP

Cheat Sheet

Directives:

```
#pragma omp parallel
#pragma omp parallel for (or parallel do for FORTRAN)
#pragma omp parallel for collapse(number of loops)
#pragma omp parallel for reduction (operation:variable name)
#pragma omp parallel shared(A,B,C) private(x)
#pragma omp sections
#pragma omp nowait
```

To avoid if possible (but sometimes necessary):

```
#pragma omp barrier
#pragma omp critical
#pragma omp atomic
#pragma omp master
#pragma omp single
```

Runtime Library functions:

```
omp_get_num_threads()
omp_get_thread_num()
omp_set_num_threads()
omp_get_max_threads()
omp_num_procs()
```

Parallelism

Threads and OpenMP

- In summary:
  - OpenMP can be a relatively simple way to parallelise your program (see last slide), but…
  - THINK before you implement
  - Race conditions and global/private variables can create strange bugs that can be difficult to find
  - False sharing can degrade your performance if not mitigated against

Parallelism

Threads and OpenMP

```
#include <omp.h>
static long num_steps = 100000;        double step;

void main ()
{          int i;    double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
           for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any executable code and by adding 2 simple lines of text!

Parallelism

Message Passing Interface (MPI)

Parallelism

Message Passing Interface (MPI)

- Some useful resources:
  - ARCHER2 Youtube courses (https://www.archer2.ac.uk/training/courses/200514-mpi/)
  - James' git repository - MPI in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)
  - https://mpitutorial.com/tutorials/ (some examples from here)
  - Several other online lecture courses

Parallelism

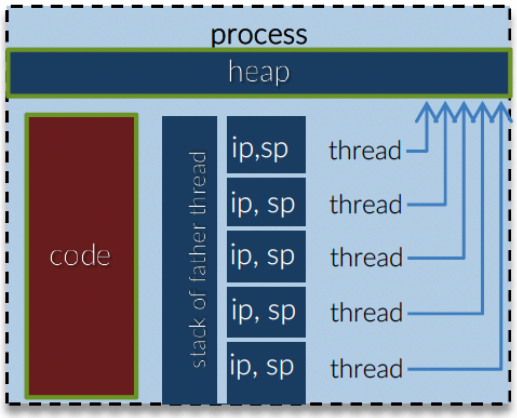
Message Passing Interface (MPI)

- If we want to parallelise our code over multiple cores *without* shared memory, we need to use MPI - recall:

Shared-Memory

(e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



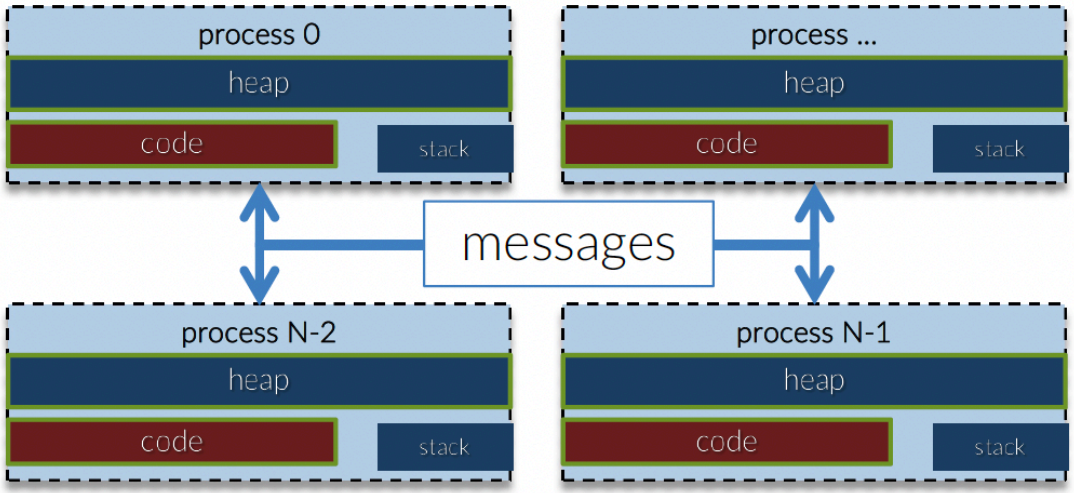
Distributed-Memory

(e.g. MPI)

N processes are created, each with its own copy of the code and its own memory space.

A process ***can not*** access the memory space of another process.

The processes communicate through *messages*.



Parallelism

Message Passing Interface (MPI)

- This means that once we have split up the problem, we need to explicitly *communicate* between cores
- This is done using the routines:
  - `MPI_Init` initialise MPI
  - `MPI_Comm_size` get number of processes
  - `MPI_Comm_rank` get the process ID (rank)
  - `MPI_Send` send data from core
  - `MPI_Recv` receive data to core
  - `MPI_Finalize` close MPI
- This is almost all you need to know to use MPI

Parallelism

Message Passing Interface (MPI)

Example: Compile and run `mpi.cpp`

- Compile with `mpicc mpi.cpp -o whatever_name`
  - `mpicc` is a wrapper around a certain compiler(s)
  - You can see what commands are run using `mpicc -show` (for me, it uses `gcc`)
- To run, use the command `mpirun -np 4 ./whatever_name`
  - The flag `-np` sets the number of *processes*
  - You must ensure that the number of ranks does not exceed the number of available cores
  - If you do assign more ranks than cores, the program will probably run, but the performance will be poor

Parallelism

Message Passing Interface (MPI)

- `MPI_Init` initialises execution environment, takes command line arguments (always keep these arguments)
- `MPI_COMM_WORLD` is defined by `mpi.h` and designates processes in the MPI job
- Each statement executes independently in each process
- As with OpenMP, there is no defined output order
- Note we have also included the library `<mpi.h>` and installed `mpich` (see Dockerfile)

Parallelism

Message Passing Interface (MPI)

- The previous example performed the same command for each rank - usually, we want to split a given problem between several cores
- We therefore require the processes running on different cores to communicate using:
  - `MPI_Send(void* data, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm communicator)`
  - `MPI_Recv(void* data, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm communicator, MPI_Status* status)`
- Although this looks like a lot to remember, most MPI calls use the same syntax (see example)

Parallelism

Message Passing Interface (MPI)

- The MPI data types are mostly just the normal data types you use in the language you are coding, with `MPI_` in front
  - Eg. `MPI_INT`, `MPI_FLOAT`
- We can even choose which ranks will send/receive certain chunks of work

Example 1: Compile and run `mpi_send_recv.cpp`

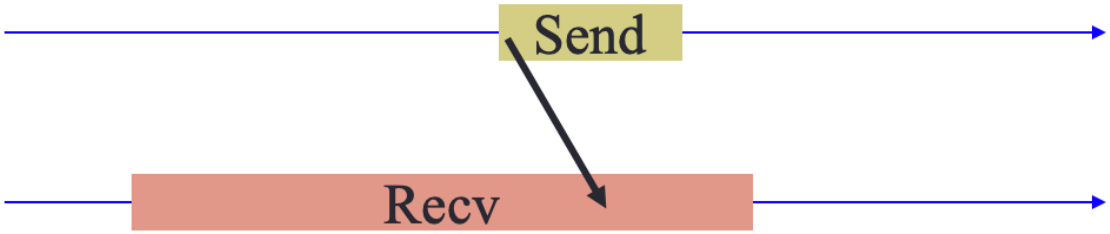
- Change the code so that process 0 sends a message to all other processes (not just 1)

Example 2: Compile and run `mpi_ping_pong.cpp`

Parallelism

Message Passing Interface (MPI)

- In the ping pong example, we notice that some signals are `received' before they are sent - how is this possible?



- If the (blocking) receive is posted before its matching send, the receiving task must wait until the data is sent

  - We will come onto blocking…

Parallelism

Message Passing Interface (MPI)

- With `MPI_Send` and `MPI_Recv`, the instruction is complete when it is *safe to change/access the data we sent/received*
- We can choose either to
  - Start a communication and wait for it to complete - *blocking*
  - Start a communication and return control to the main problem - *non-blocking*
    - This requires us to *check for completion* before we can change/ access the data we sent/received
- `MPI_Send` and `MPI_Recv` are both blocking operations
- These can be unsafe and lead to deadlocks if the message passing cannot be completed
  - Eg. `MPI_Send` is called before `MPI_Recv` for a previous exchange

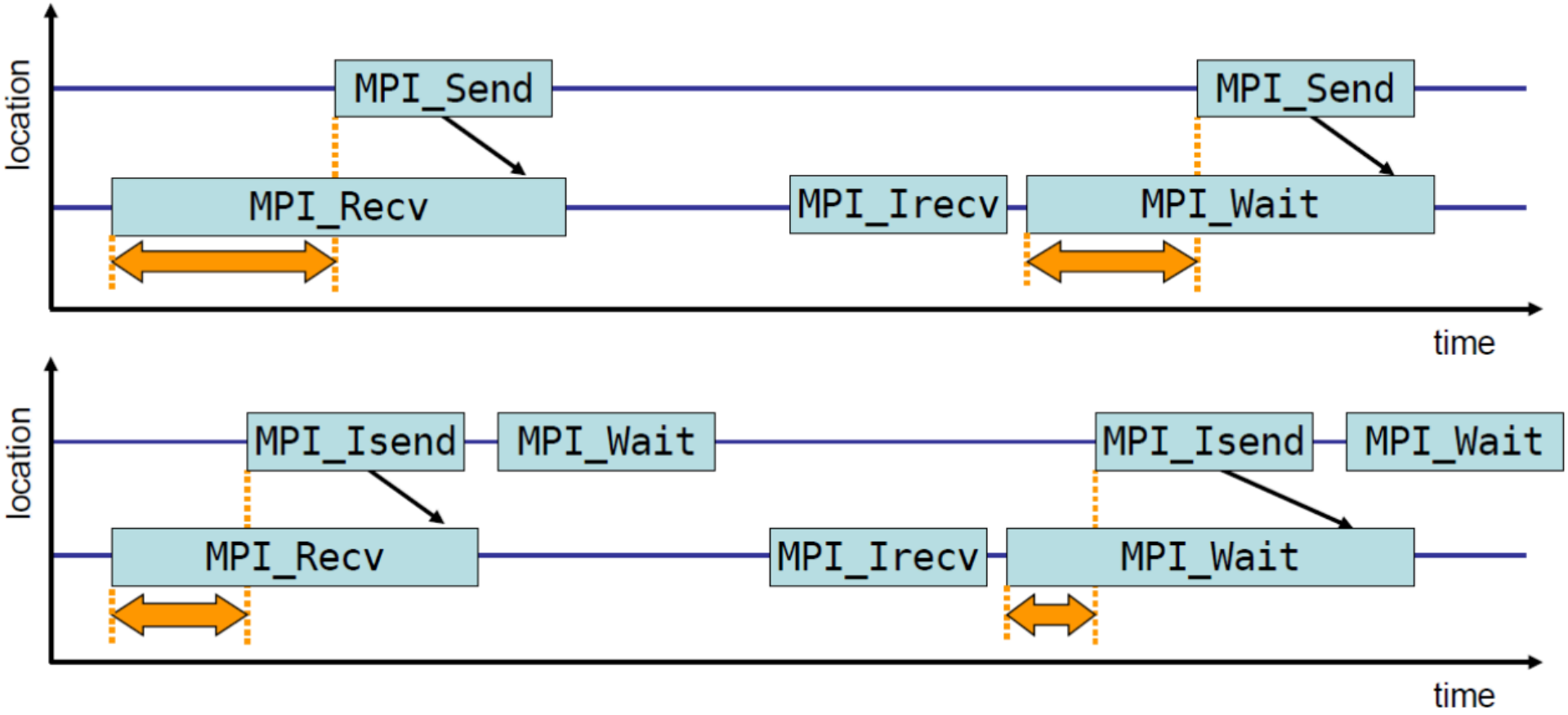
Parallelism

Message Passing Interface (MPI)

- Non-blocking allows separation between the initiation of a communication and the completion
- This can be more efficient but the programmer must also make sure to check for completion
- For these, we instead use `MPI_Isend` and `MPI_Irecv`
  - `MPI_Isend(void* data, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm communicator, MPI_Request *req)`
  - `MPI_Irecv(void* data, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm communicator, MPI_Status* status, MPI_Request *req)`
- The I stands for `immediate’

Parallelism

Message Passing Interface (MPI)



http://www.morrisriedel.de/wp-content/uploads/2019/10/High-Performance-Computing-Lecture-5-1-Understanding-MPI-Communicators-and-Data-Structures.pdf

Parallelism

Message Passing Interface (MPI)

- If using `MPI_Isend` and `MPI_Irecv`, we can use

  ```
  MPI_Wait(MPI_Request *req, MPI_Status *status)
  ```

  to wait until the communication pointed to by `req` is complete

- We can also use

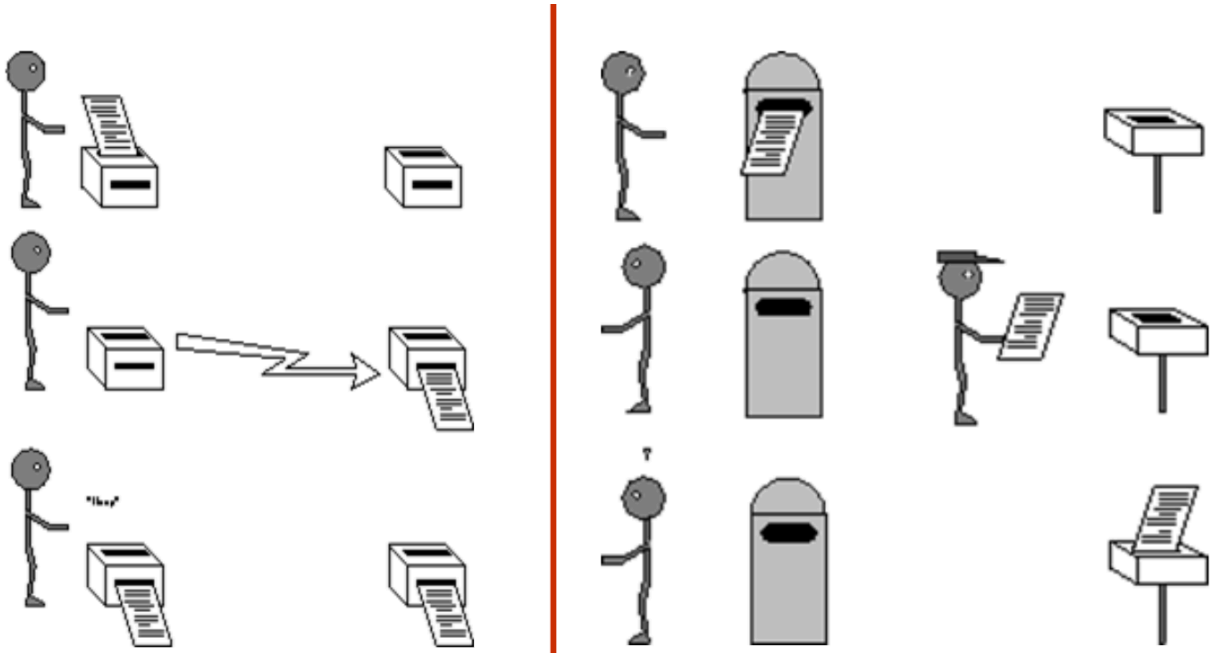
  ```
  MPI_Test(MPI_Request *req, int *flag, MPI_Status
  *status)
  ```

  to test whether the message has been completed

Parallelism

Message Passing Interface (MPI)

- `MPI_Isend` and `MPI_Irecv` can use either:
  - *Synchronous* communication - the sender blocks further operations until an acknowledgement or response is received
  - *Asynchronous* communication - the sender continues execution without waiting for an acknowledgement



Parallelism

Message Passing Interface (MPI)

- Some other options are:
  - `MPI_Bsend` (buffer send) - always completes, irrespective of receiver
  - `MPI_Ssend` (synchronous send) - only completes when the receive has completed, guarantees the buffer passed can be safely reused
  - `MPI_Rsend` (ready send) - always completes, irrespective of whether the receive has completed
- For comparison, standard send and receives
- All of the above can be used as blocking or non-blocking