University of Leeds

Lecture 9: Point-to-point com

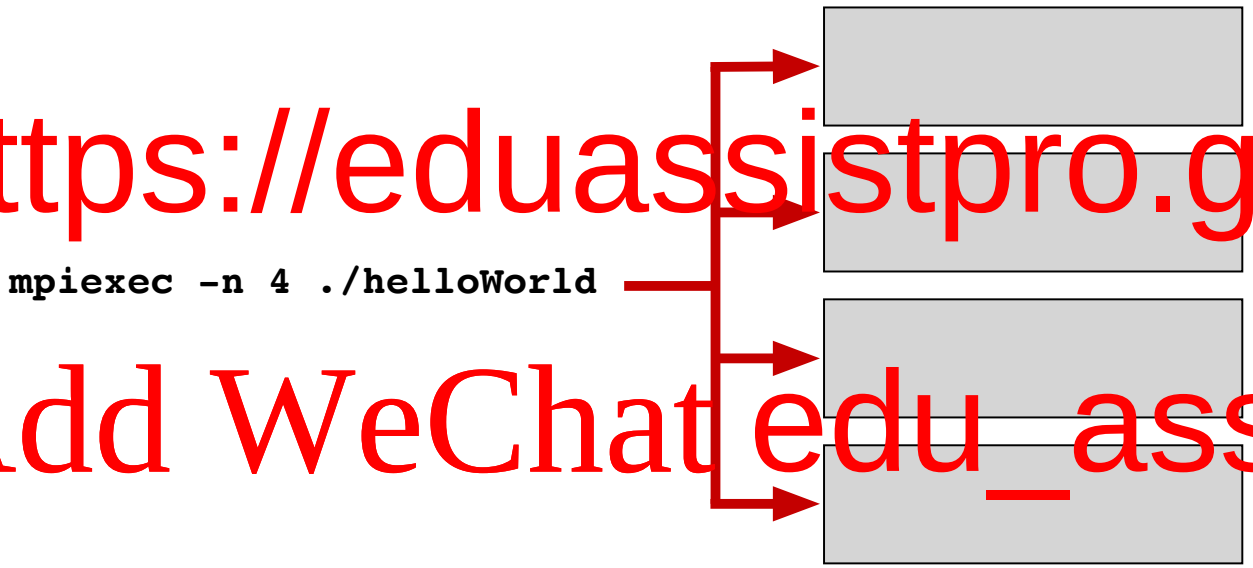
Previous lectures

Last lecture we started looking at **distributed memory systems**:

- Standard API for low-level programmin e
  Passing Interface
- Processing units are **processes** ra
- Saw a 'Hello World' program for MPI.

mpiexec or mpirun

Launches **multiple executables** simultaneously, possibly on different machines/nodes, which are identical in every way **except** their `rank`:



All processes exist for the **duration of the program run**.

- Creation or destruction of **processes** is **expensive** (compared to *threads* in *e.g.* shared memory systems).

Today's lecture

Today we will start looking at using MPI to solve real problems.

-
- **Vector addition**, the same problem we l
memory systems in Lecture 3.
- How exceeding the **buffer** size for so
patterns can lead to **deadlock**.

Vector addition

Recall vector addition can be written mathematically as

$$\mathbf{c} = \mathbf{a} + \mathbf{b}$$

and is


```
1 for( i=0
2   c[i] = a[i] + b[i];
```

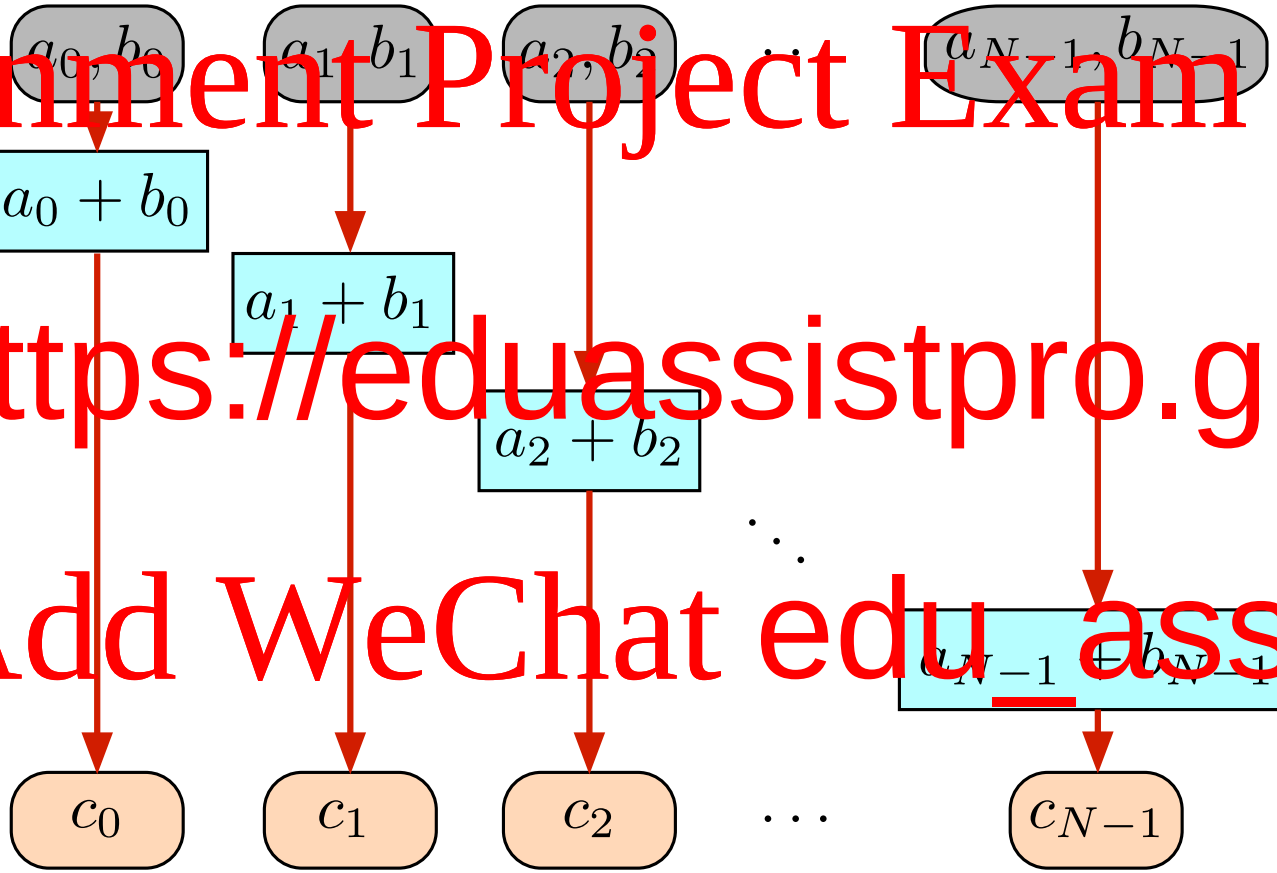
where vectors **a**, **b** and **c** all have N el

This is a **data parallel** problem, also kno

I.

[1] By convention, indexing starts from 1 for mathematics notation but 0 in code.

Vector addition as a map[1]



[1] McCool *et al.*, *Structured parallel programming* (Morgan-Kaufman, 2012).

Vector addition on a distributed memory system

Code on Minerva: `vectorAddition.c`

Suppose vectors **a** and **b** initially lie in the memory space of **one proc**

Therefore must:

1. **Distribute** vectors **a** and **b** acros
2. Perform the calculations in parallel ch working on a different **segment** of the arrays.
3. **Gather** the segments together on a single process.

Point-to-point communication

The simplest way to send data from one process to another is to use `MPI_Send()` and `MPI_Recv()`:

Recall from last lecture that after initialising MPI the total number of processes `numPro` process `rank` as follows:


```
1 int rank, numProcs;
2
3 MPI_Comm_size( MPI_COMM_WORLD, &numProcs );
4 MPI_Comm_rank( MPI_COMM_WORLD, &rank     );
```

MPI_Send()

For step 1, rank 0 **distributes** segments of the arrays a and b:

```
1  if( rank==0 )
2  {
3    fo
4
5
6                                  // Th   s
7      MPI_FLOAT,                  // The data type
8      p,                          // Destination process rank
9      0,                          // Tag: usually set to 0
10     MPI_COMM_WORLD              // Communicator
11   );
12 }
```

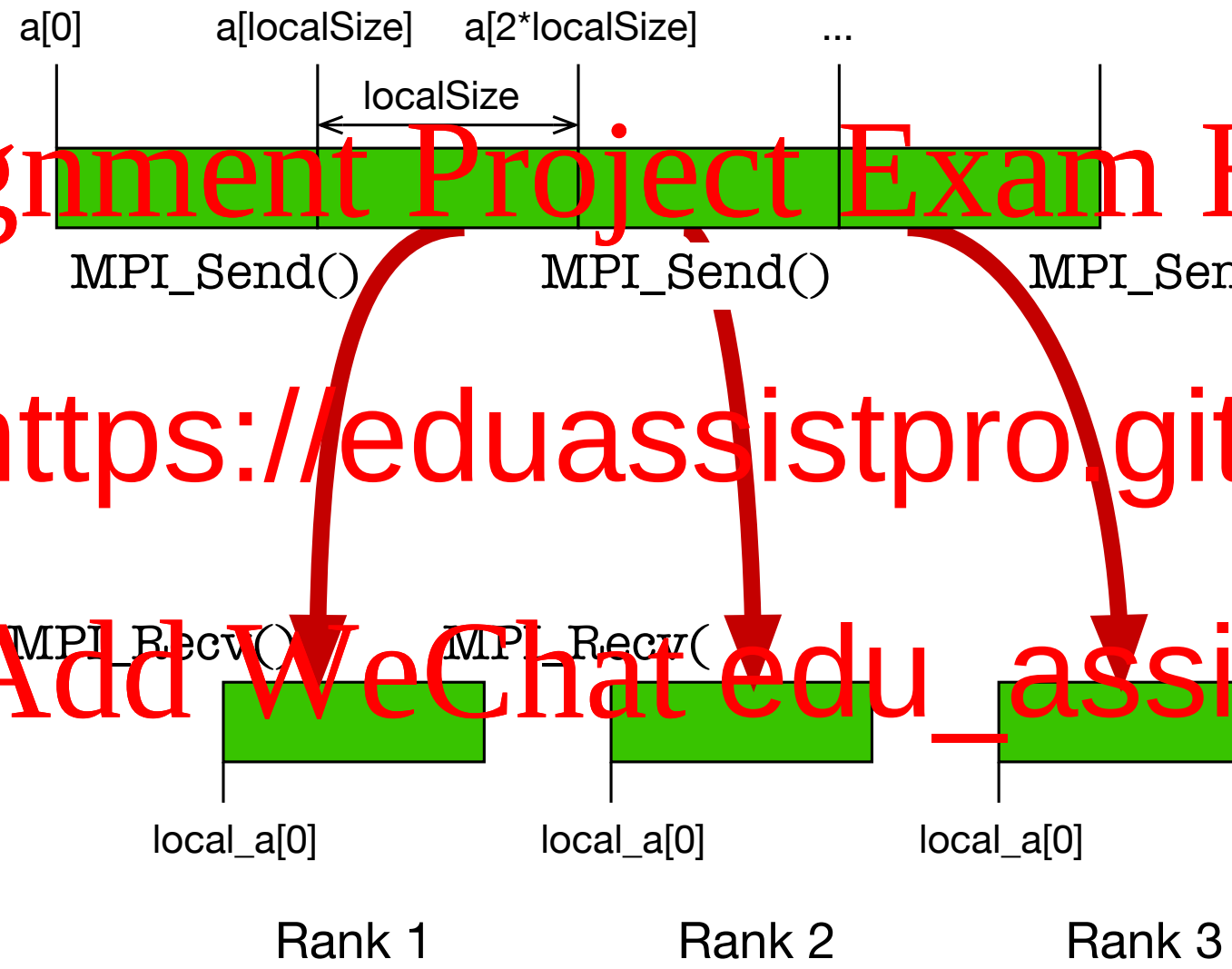
And similarly for b. Here, `localSize=N/numProcs` is the **problem size per process**, *i.e.* the size of the **local** arrays / array segments.

MPI_Recv()

All ranks except 0 need to receive the data:

```
1  MPI_Status status;
2  ...
3  if( ran
4  {
5    MP
6                          // Pointer to t
7      localSize,          // The size being sent
8      MPI_FLOAT,          // The data type
9      0,                  // Source process rank
10     0,                  // Tag (can set to 0)
11     MPI_COMM_WORLD,     // Communicator
12     &status             // MPI_Status object
13   );
14 }
```

And similarly for `local_b`.



Completing the calculation

After rank 0 has distributed the full arrays a and b to the local arrays `local_a` and `local_b` on all other ranks:

Note that in the code, **local** arrays are g the full arrays.

- *e.g.* `local_a` rather than `a`.

This is recommended (but not essential) to help keep track.

- The p-loop starts from 1, not zero. Sending 'to self' (*e.g.* from rank 0 to rank 0) is **undefined**.
  - Works in OpenMPI but *not* MPICH, so your code would not be portable.
- _ _ _OUBLE,
- rts at element `p*localSize` of `a`.
- Most MPI calls return `MPI_SUCCE` an error occurred.
- Can probe the `status` object to determine errors, rank of sending process *etc.*
- Can also replace `&status` with `MPI_STATUS_IGNORE`.

How is the communication performed?

The MPI standard does **not** specify **how** the communication is actually performed.

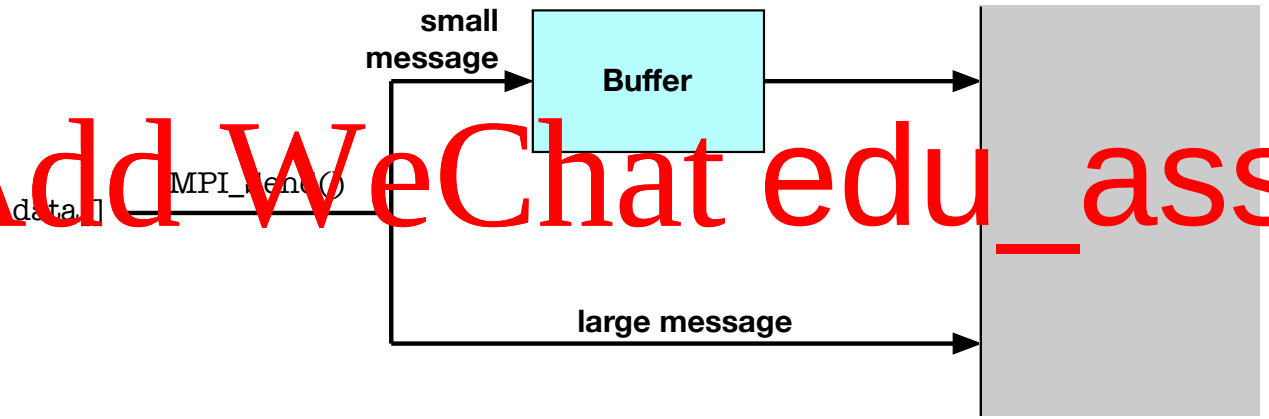
- For HPC machines (where nodes do not hav
could use **Link** layer protocols or besp

In this module we focus on **general** aspects of distributed system programming, not details of any MPI implementation.

- **Portable** code that should run on **any** implementation.

Common communication features

- Each data message has a **header** containing information such as the source and destination ranks[1].



[1] Maximum header size is `MPI_BSEND_OVERHEAD`, defined in `mpi.h`.

Blocking communication

`MPI_Send()` and `MPI_Recv()` are examples of **blocking** routines.

Blo

This **thout affecting the values sent**.

- Convenient from a programming perspe

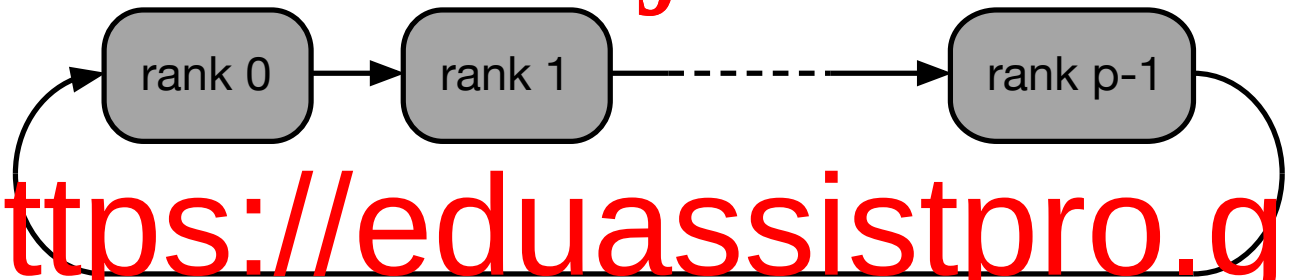
By contrast, **non-blocking** routines return 'immediately,' even though the data may still be being copied over.

- We will cover non-blocking communication in Lecture 12.

Cyclic communictation

Code on Minerva: `cyclicSendAndReceive.c`

Consider a problem where the communication pattern is **cyclic**:



Encode this concisely using the ternary operator ‘ ’ to handle the wrap-around:


```
1 // Send data 'to the right'.
2 MPI_Send( sendData, N, MPI_INT,
3   ( rank==numProcs-1 ? 0 : rank+1 ), ... );
4
5 // Receive data 'from the left'.
6 MPI_Recv( recvData, N, MPI_INT,
7   ( rank==0 ? numProcs-1 : rank-1 ), ... );
```

Use of buffering

If the data is small enough to fit on the buffer:

1. Each process calls `MPI_Send()` to send data 'to its right.'
2. [illegible] turns.
3. [illegible]

If the data is too large for the buffer, the application

1. `MPI_Send()` does **not** return until the receives the data.
2. **All processes are in the same situation** - none of them reach their call to `MPI_Recv()`.
3. As no data is received, no process returns from `MPI_Send()`.

Deadlock

This is another example of **deadlock** that we first saw in Lecture 7:

Dea

E

In this case the 'synchronisation event' is the block
receive that required the destination process to re

- Say more about the relationship between blocking and synchronisation in Lecture 12.

Resolving communication deadlocks

The buffer size is **not** specified by the MPI standard and varies between implementations.

There are various ways to resolve this deadlock pro

1. Change the **program logic** *here*
2. Use **non-blocking communication**
3. Allocate your own memory for a buffer and use **buffered send** `MPI_Bsend()`.

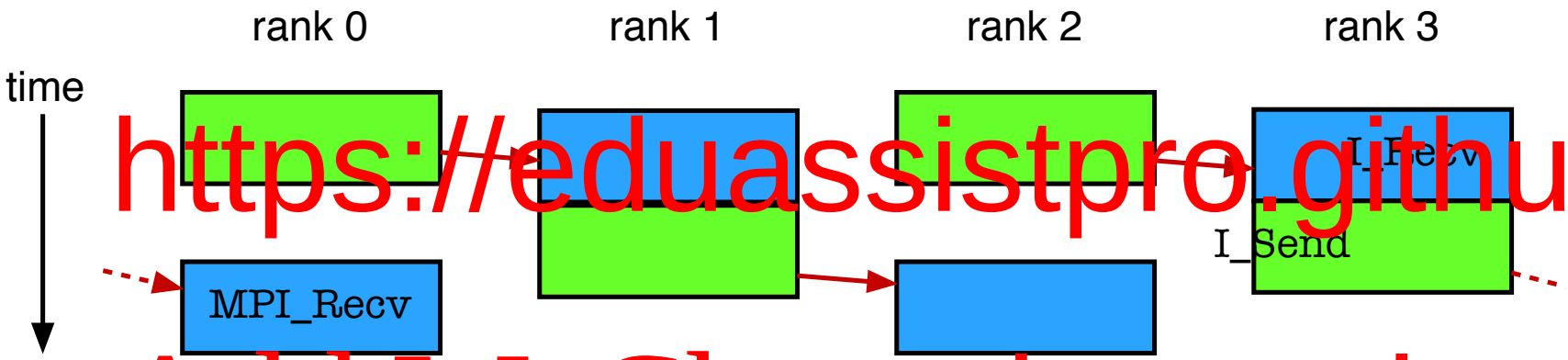
Staggering the send and receives

For this example, it is easiest to change the program logic to use **staggered** sends and receives[1]:


```
1  if( rank
2  {
3    MP
4    MP
5  }
6  else
7  {
8    MPI_Recv(recvData,N,MPI_INT,...);
9    MPI_Send(sendData,N,MPI_INT,...);
10 }
```


[1] Recall `i%2==0` if `i` is even, and 1 if `i` is odd.

Processes with even-numbered ranks **receive** first **then** send, breaking the deadlock.



Note the arguments with each `MPI_Send()` and `MPI_Recv()`, including the source and destination ranks, **have not been altered**.

Summary and next lecture

Today we have looked at **point-to-point communication** in a distri

-) using _ _

- These routines are **blocking**, a simil **synchronous communication**.
- Exceeding the buffer can lead to

Next time we will look at some **performance considerations**, and how they can be improved by using **collective communication**.