XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 9: Point-to-point communication

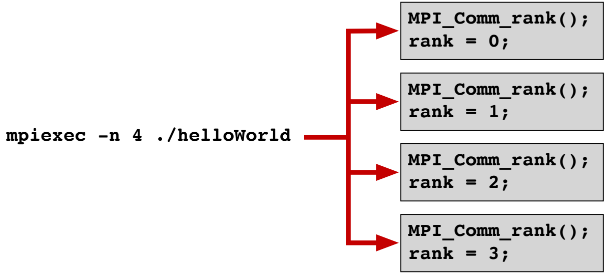
Previous lectures

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

- Each **processing unit** can only see a fraction of memory.
- *e.g.* **clusters** of machines for HPC = High Performance Computing, where each node has its own memory.
- Standard API for low-level programming is MPI = Message Passing Interface.
- Processing units are **processes** rather than threads.
- 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).

This lecture

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

- The simplest form of communication: **point-to-point**.
- Implemented in the MPI standard with `MPI_Send()` and `MPI_Recv()` *(plus variations)*.
- **Vector addition**, the same problem we looked at for shared memory systems in Lecture 3.
- How exceeding the **buffer** size for some communication patterns can lead to **deadlock**.

Vector addition

Recall vector addition can be written mathematically as

$$\mathbf{c} = \mathbf{a} + \mathbf{b} \quad \text{or} \quad c_i = a_i + b_i \;, \; i = 1 \ldots N,$$

and in serial code as

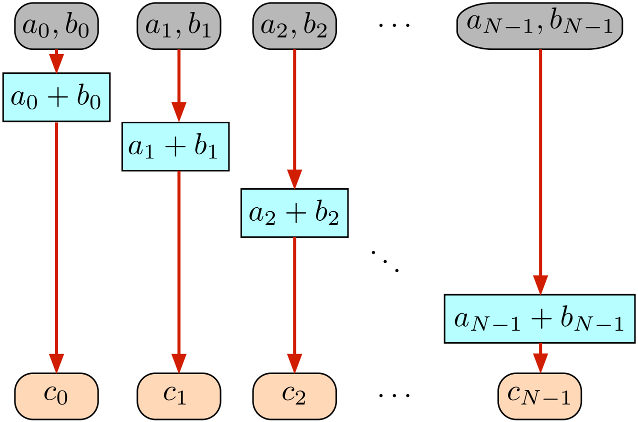
```
for( i=0; i<N; i++ )
  c[i] = a[i] + b[i];
```

where vectors **a**, **b** and **c** all have N elements[1].

This is a **data parallel** problem, also known as a **map** *[Lecture 3]*.

[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 process only**, *e.g.* rank 0.

- Unlike in a shared memory system, the other processes **cannot see the data**.

Therefore must:

1. **Distribute** vectors **a** and **b** across multiple processes.
2. Perform the calculations in parallel **across all processes**, each 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()`:

- The process **sending** the data calls `MPI_Send()`.
- The process **receiving** the data calls `MPI_Recv()`.

Recall from last lecture that after initialising MPI, we determine the total number of processes `numProcs`, and the **rank** of 'this' process `rank` as follows:

```c
int rank, numProcs;

MPI_Comm_size( MPI_COMM_WORLD, &numProcs );
MPI_Comm_rank( MPI_COMM_WORLD, &rank     );
```

MPI_Send()

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

```
if( rank==0 )
{
  for( p=1; p<numProcs; p++ )
    MPI_Send(
      &a[p*localSize],      // Pointer to the data
      localSize,            // The size to be sent
      MPI_FLOAT,            // The data type
      p,                    // Destination process rank
      0,                    // Tag; usually set to 0
      MPI_COMM_WORLD        // Communicator
    );
}
```

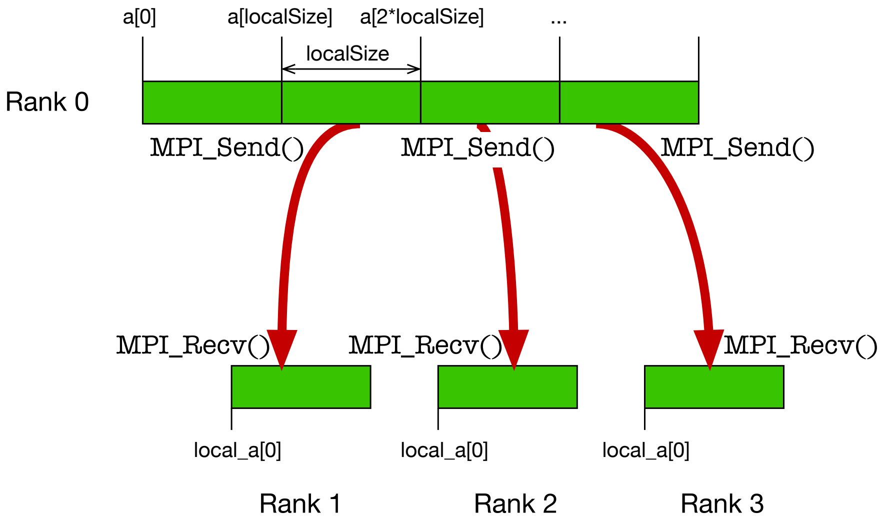
And similarly for b. Here, `localSize=N/numProcs` is the **problem size per process**, *i.e.* the size of the **local** arrays / array segments.

MPI_Recv()

All ranks except 0 need to receive the data:

```
MPI_Status status;
...
if( rank>0 )
{
  MPI_Recv(
    local_a,            // Pointer to the data
    localSize,          // The size being sent
    MPI_FLOAT,          // The data type
    0,                  // Source process rank
    0,                  // Tag (can set to 0)
    MPI_COMM_WORLD,     // Communicator
    &status             // MPI_Status object
  );
}
```

And similarly for `local_b`.


a[0]
a[localSize]
a[2*localSize]
...
localSize
Rank 0
MPI_Send()
MPI_Send()
MPI_Send()
MPI_Recv()
MPI_Recv()
MPI_Recv()
local_a[0]
local_a[0]
local_a[0]
Rank 1
Rank 2
Rank 3

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:

- They all perform vector addition using their local arrays.
- The `local_c` arrays on each `rank`>0 are sent to rank 0 using the same procedure as before.

Note that, in the code, **local** arrays are given separate names to 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, so we never try to 'send to self,' *i.e.* send from rank 0 to rank 0 **[see next slide]**.
- The **data type** is one of `MPI_FLOAT`, `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, ...
- `&a[p*localSize]` is a pointer to a **sub-array** that starts at element `p*localSize` of `a`.
- Most MPI calls return `MPI_SUCCESS` if successful, otherwise an error occurred.
- Can probe the `status` object to determine errors, rank of sending process *etc.*
- Can also replace `&status` with `MPI_STATUS_IGNORE`.

Sending to 'self'

What happens if you try to send a message from a rank **to itself**?

```
if(rank==0)
  MPI_Send(data,size,MPI_FLOAT,0,...)
```

This is not covered by the MPI standard and so the behaviour is **undefined**

- OpenMPI will **copy** the data within rank 0's heap memory.
- MPICH will **hang** and your application will fail.

Therefore, to ensure your code is **portable**, it is essential not to deviate from the MPI standard (ideally you should also test using MPICH – but this may not always be available to you)

How is the communication performed?

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

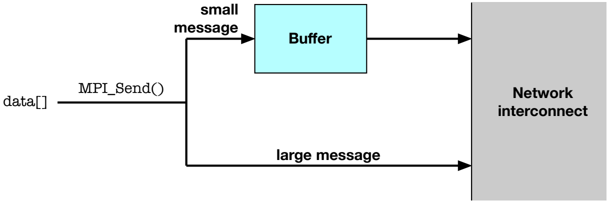
- If the nodes have IP addresses, could use standard internet protocols (*e.g.* sockets); the **Network** layer *[cf. COMP2221 Networks]*.
- For HPC machines (where nodes do not have IP addresses), could use **Link** layer protocols or bespoke methods.

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].
- Message placed on a **buffer** ready to send.
- If it is too large, will send **direct** to the destination process.



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

Blocking communication

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

Blocking routines do not return until all resources can be reused

This means *e.g.* values in the data array can be altered **without affecting the values sent**.

- Convenient from a programming perspective.

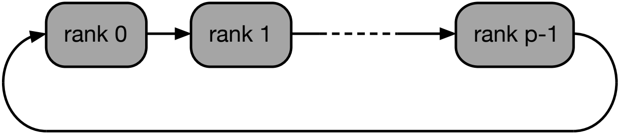
By contrast, **non-blocking** routines return 'immediately,' even though the data may still be being copied over.

- 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 '`(a?b:c)`' to handle the wrap-around:

```
// Send data 'to the right'.
MPI_Send( sendData, N, MPI_INT,
  ( rank==numProcs-1 ? 0 : rank+1 ), ... );

// Receive data 'from the left'.
MPI_Recv( recvData, N, MPI_INT,
  ( 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. The data is **copied to the buffer** and `MPI_Send()` returns.
3. Each process calls `MPI_Recv()` and receives data from the process 'to its left.'

If the data is too large for the buffer, the application **hangs**:

1. `MPI_Send()` does **not** return until the destination process 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:

Deadlock

Each process is waiting for a synchronisation event that never occurs.

In this case the 'synchronisation event' is the blocking send and receive that required the destination process to receive the data.

- 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.

- Even allowed to be zero size!

Need to write **portable** code that works for **any** buffer size.

There are various ways to resolve this deadlock problem:

1. Change the **program logic** *[here]*.
2. Use **non-blocking communication** *[Lecture 12]*.
3. Allocate your own memory for a buffer and use **buffered send** `MPI_Bsend()`.

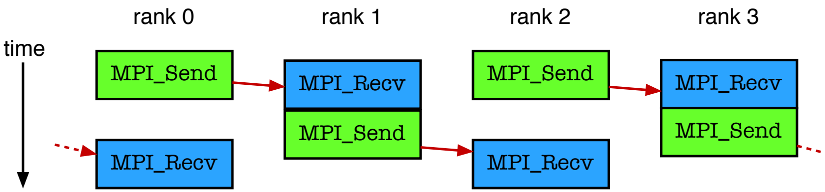
Staggering the send and receives

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

```
if( rank%2 )
{
  MPI_Recv(recvData,N,MPI_INT,...);
  MPI_Send(sendData,N,MPI_INT,...);
}
else
{
  MPI_Send(sendData,N,MPI_INT,...);
  MPI_Recv(recvData,N,MPI_INT,...);
}
```

[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 distributed memory system:

- How to implement a **data parallel problem** (or **a map**) using `MPI_Send()` and `MPI_Recv()`.
- These routines are **blocking**, a similar concept to **synchronous communication**.
- Exceeding the buffer can lead to **deadlock**.

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