XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 12: Non-blocking communication

Previous lectures

So far we have only considered **blocking communication** in distributed memory systems:

- Do not return until it is safe to use the resources, *i.e.* the memory allocated for the data.
- This **may** happen after the data is copied to a **buffer**.
- If the buffer is too small, will wait until it is being **received**.
- **Point-to-point communication**: `MPI_Send()`, `MPI_Recv()`.
- **Collective communication**: `MPI_Bcast()`, `MPI_Gather()`, `MPI_Scatter()`, `MPI_Reduce()`.

This lecture

Now we will look at **non-blocking communication**:

- Non-blocking calls return **'immediately'**.
- Require **extra coding** to determine when it is safe to re-use the resources.
- Can overlap communication with computation to improve performance: **Latency hiding**.
- Useful in situations that require **domain partitioning**.
- Briefly look at **stencils**, a graphical representation of calculation locality.

Blocking communication

Definition

A communication is **blocking** if return of control to the calling process **only** occurs after **all resources** are safe to re-use.

In MPI, **resources** primarily refers to the memory allocated for the message, such as the pointer `data` in this `MPI_Send()` example:

```
MPI_Send( data, size, MPI_INT, ... );
```

Note this only refers to the viewpoint of the **calling** process; the **receiving** process is not mentioned.

Synchronous communication

Definition

Communication is **synchronous** if the operation does not complete before **both** processes have started their communication operation.

On return, all resources can be re-used, *and* we know the destination process has started receiving[1].

For instance, a blocking call may return once the data has been copied to the buffer, **before it has even been sent to the network** (and is therefore **not** synchronised with the receiver).

[1]MPI supports synchronised communication with `MPI_Ssend()`. A common use is **debugging**: If replacing `MPI_Send()` with `MPI_Ssend()` results in **deadlock**, the original code would have deadlocked **when the data exceeded the buffer**.

Non-blocking and asynchronous communication

Definition

A **non-blocking operation** may return before it is safe to re-use the resources. In particular, changing `data` after returning **may change the data being sent**.

Essentially, such calls only **start** the communication.

Definition

Asynchronous communication does not require any co-operation between the sender(s) and the receiver(s).

e.g. a send that **doesn't expect a corresponding receive**.

Blocking $\neq$ synchronous

Sometimes the terms **blocking** and **synchronous**, and **non-blocking** and **asynchronous**, are used interchangeably.

- Blocking **can** act as a form of synchronisation.
- *e.g.* `MPI_Recv()` will not return until the data has been received.

However, the distinction is more subtle:

- **Blocking** and **non-blocking** refer to a single process's view, *i.e.* 'what the programmer needs to know.'
- **Synchronous** and **asynchronous** refer to a more global view involving at least two processes.

Non-blocking communication in MPI

The key routines are:

- `MPI_Isend()` : Start a **non-blocking** send.
- `MPI_Irecv()` : Start a **non-blocking** receive.
- `MPI_Wait()` : Will not return until the communication is complete.
- `MPI_Test()` : Test to see if the communication is complete (but return 'immediately').

The '`I`' in `MPI_Isend()` and `MPI_Irecv()` stands for **immediate**, because they return (almost) immediately.

There are other routines, including non-blocking collective communication in MPI v3 [`MPI_Ibcast()`, . . .], but these will not be covered here.

MPI_Request

To link each MPI_Isend() or MPI_Irecv() with its corresponding MPI_Wait() or MPI_Test(), MPI uses **requests**:

```
MPI_Request request;
MPI_Status status;

// Start the communication.
MPI_Isend( data, size, ..., &request );

// Do other things not involving 'data'.
...

// Wait until the communication is complete.
// (Can replace &status with MPI_STATUS_IGNORE.)
MPI_Wait( &request, &status );

// Can now safely re-use 'data'.
```

Why use non-blocking communication?

Since a non-blocking communication call returns immediately, we can perform other useful calculations **while the communication is going on** – as long as they do not involve the resources.

So rather than performing calculations and communications sequentially, some may be performed **concurrently**.

- Reduces total runtime, improving performance.

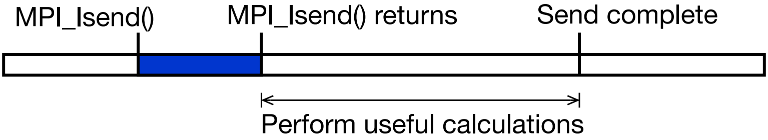
Latency hiding

The primary reason to use non-blocking communication is to **overlap** communication with computation or other communications. This is known as **latency hiding**.

Schematic (sending)

Blocking MPI_Send():

Non-blocking MPI_Isend():



Testing for completion or lock availability

In Lecture 7 we saw how **locks** can be used to synchronise **threads** in a shared memory system:

```
regionLock.lock();
// Does not return until lock acquired
```

The `lock()` method is **blocking** - it does not return until the lock is available.

Using `test()` could allow useful calculations to be performed while waiting for the lock to become available:

```
while( !regionLock.test() )
  { ... /* Do as many calculations as possible */ }
```

The MPI function `MPI_Test()` performs a similar role for non-blocking communication.

Potential applications of non-blocking communucation

Many applications require **large data sets** to be modified according to some rules. Examples include[1]:

Signal processing: (1D data sets)

- Frequency analysis, noise filtering, ...

Image processing: (2D data sets)

- Colour filtering, blurring, edge detection, ...

Scientific and engineering modelling: (1D, 2D or 3D)

- Fluid dynamics, elasticity/mechanics, weather forecasting, ...

[1]Wilkinson and Allen, *Parallel programming* (Pearson, 2005).

Domain partitioning

The standard way to parallelise such problems with **distributed** memory is to **partition the domain** between the processes, *i.e.*

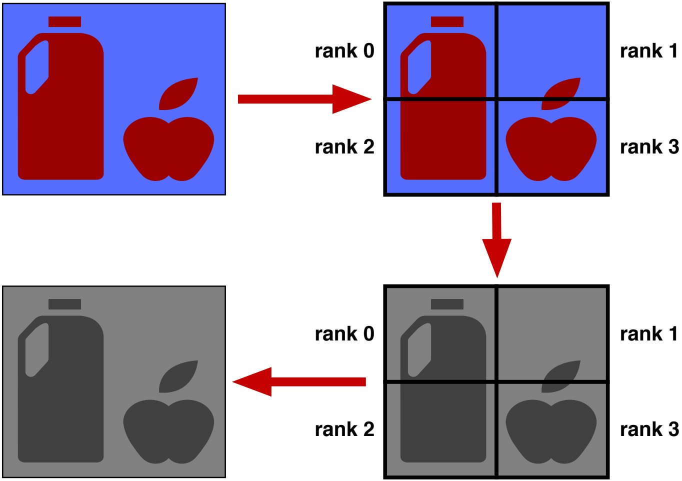
- Segments of a time series.
- Regions of an image *[next slide]*.
- ...

Domain partitioning

Each processing unit is responsible for transforming **one partition**.

If the transformation only depends on each data point **in isolation**, this is a **map**; also an **embarrassingly parallel problem**.

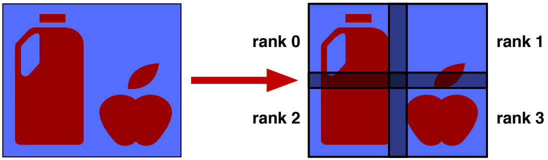
Map example: Colour transformation



Local transformations

More commonly, however, the transformation depends on **nearby information**.

- **Blurring** or **edge detection** in image processing.
- Most scientific and engineering applications solve equations with **gradient** terms (*i.e.* changes in quantities).

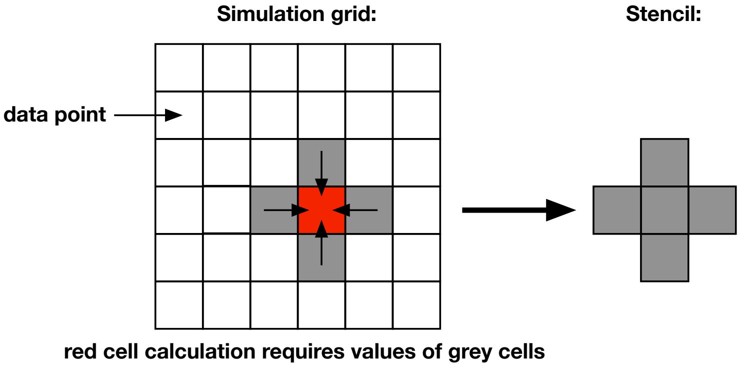


Need to **communicate** information lying at the **edges** of domains to perform the calculations correctly.

Stencils

A **stencil** is a graphical representation of where the required data exists relative to point being calculated.

This common stencil arises in many scientific applications:



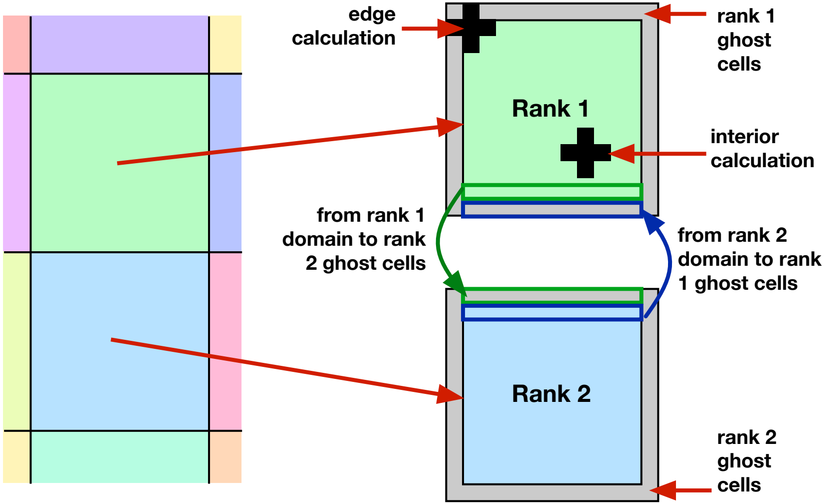
Ghost cells

The standard way to communicate across boundaries is to use **ghost cells**, sometimes known as a **halo**.

- Layer(s) of data points around each process's domain.
- Contains **read-only copy** of corresponding points from neighbouring processes' domain.
- **Updated** after each iteration to match the values calculated by the neighbouring processes.
- Updating performed using **point-to-point communication**.

Conceptual simulation domains:

Implemented simulation domains:



Implementation v1

Code on Minerva: heatEqn.c

In pseudocode the obvious implementation would be:

```
// Iterate multiple times.
for(iter=0;iter<NUM_ITERATIONS;iter++)
{
  // Send data at edge of domain to other processes.
  communicateBetweenDomains();    // BLOCKING.

  // Update values within this rank's domain.
  solveWithinDomain();
}
```

However, this ignores the fact that **only** the data points **near** to the edge of the domain require other processes' data.

- **Interior sites** can be calculated **prior to communication**.
- This is normally the **bulk** of the calculation.

Implementation v2

Since the ghost cells need only be updated before the edge cell calculations, can in principle solve the interior cells first:

```
for(iter=0;iter<NUM_ITERATIONS;iter++)
{
  // Calculate data points within the domain.
  solveDomainInterior();

  // Send data at edge of domain to other processes.
  communicateBetweenDomains();    // BLOCKING.

  // Now solve the remaining few points at the edge.
  solveDomainEdge();
}
```

But this is still using **blocking** communication.

Implementation v3

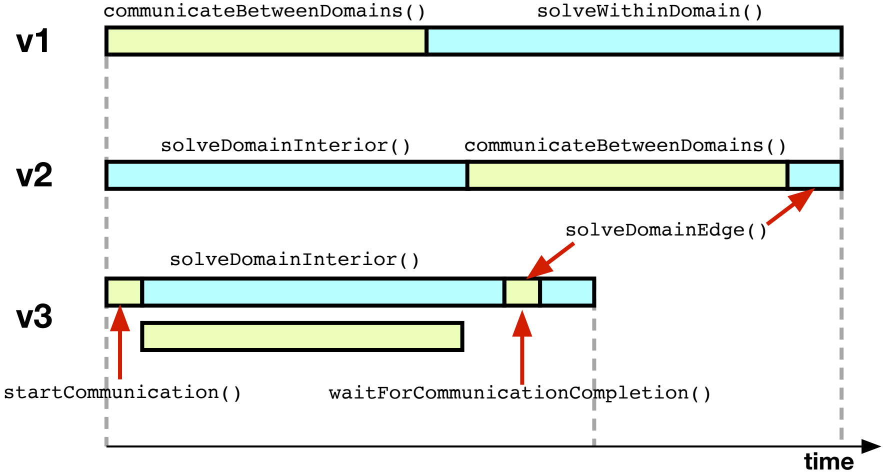
Use non-blocking communication to overlap the **calculation** of interior points with the **communication** of ghost cells:

```
for(iter=0;iter<NUM_ITERATIONS;iter++)
{
  // Start communication with other processes.
  startCommunication();    // NON-BLOCKING.

  // Calculate data points within the domain
  // WHILE THE COMMUNICATION IS GOING ON!
  solveDomainInterior();

  // Wait until the communication has finished.
  waitForCommunicationCompletion();

  // Now solve the remaining few points at the edge.
  solveDomainEdge();
}
```


communicateBetweenDomains()
solveWithinDomain()
v1
solveDomainInterior()
communicateBetweenDomains()
v2
solveDomainEdge()
solveDomainInterior()
v3
startCommunication()
waitForCommunicationCompletion()
time

Summary and next lecture

Today we have looked at the difference between **blocking** and **non-blocking** communication, and **synchronous** and **asynchronous** communication:

- Non-blocking can overlap communication with computation: **Latency hiding**.
- Example of **domain partitioning** using **ghost cells**.
- **Stencils** describe the locality of the calculation.

Next time we will look in detail at a *very* important concept for **all** parallel systems - **load balancing**.