University of Leeds

Lecture 12: Non-blocking com

Previous lectures

So far we have only considered **blocking communication** in distri

- This *may* happen after the data is copied to a **buffer**.
- If the buffer is too small, will wait until it is being **d**.
- **Point-to-point communication**: `MPI_Send()`, `MPI_Recv()`.
- **Collective communication**: `MPI_Bcast()`, `MPI_Gather()`, `MPI_Scatter()`, `MPI_Reduce()`.

Today's lecture

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

-
-
- Can overlap communication with compu performance: **Latency hiding**.
- Useful in situations that require **d**
- Briefly look at **stencils**, a graphical representation of calculation locality.

Blocking communication

Definition

A communication is **blocking** if return of control to the calling

In MPI, **resources** primarily refers to the memory allocated for the message, such as the pointer `data` in th _ :


```
1 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 befo .

On ret
desti

For instance, a blocking call may return once the da copied to the buffer, **before it has even bee network** (and is therefore **not** synchro

[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 r **may**

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

Definition

Asynchronous communication does n between the sender(s) and the receiver(s).

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

Blocking ≠ synchronous

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

However, the distinction is more subtle:

- **Blocking** and **non-blocking** refer t
  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_`
- `MPI_`
- `MPI_`
- `MPI_`

return immediately).

The 'I' in `MPI_Isend()` and `MPI_Ire` te, 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**:

```
1  MPI_Request request;
2  MPI_S
3
4  // Start s
5  MPI_I
6
7  // Do other things not involving 'data'.
8  ...
9
10 // Wait until the communication is complete.
11 // (Can replace &status with MPI_STATUS_IGNORE.)
12 MPI_Wait( &request, &status );
13
14 // 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 goi**

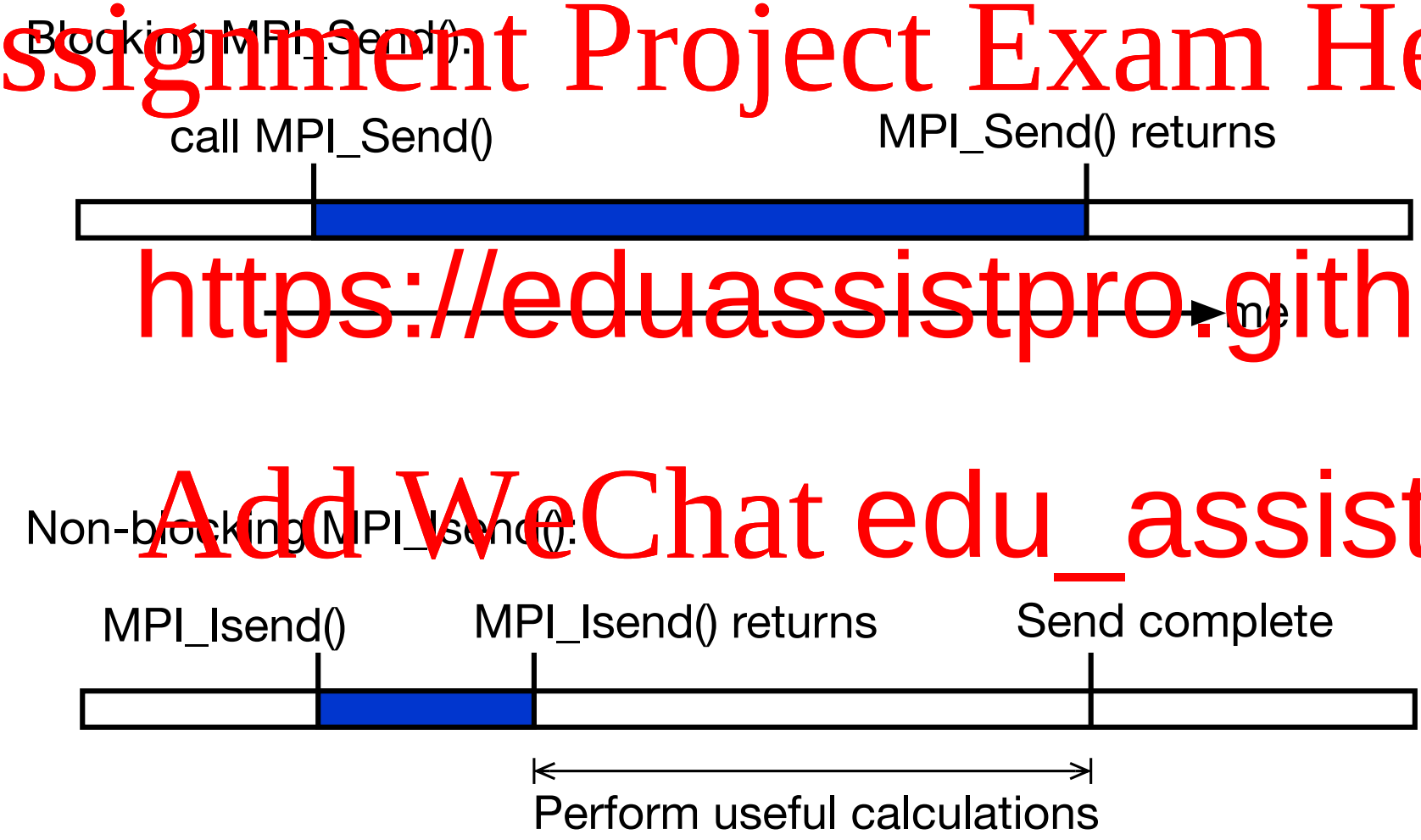
So rat
sequ

- Reduces total runtime, improving perfor

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)



Testing for completion or lock availability

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

```
regio
// Does no  r
```


The
is avai

Using test() could allow useful calculatio
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]:

Sign

- https://eduassistpro.github.i

Image processing: (2D data sets)

- Colour filtering, blurring, edge detection

Scientific and engineering modelling: (

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

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

Domain partitioning

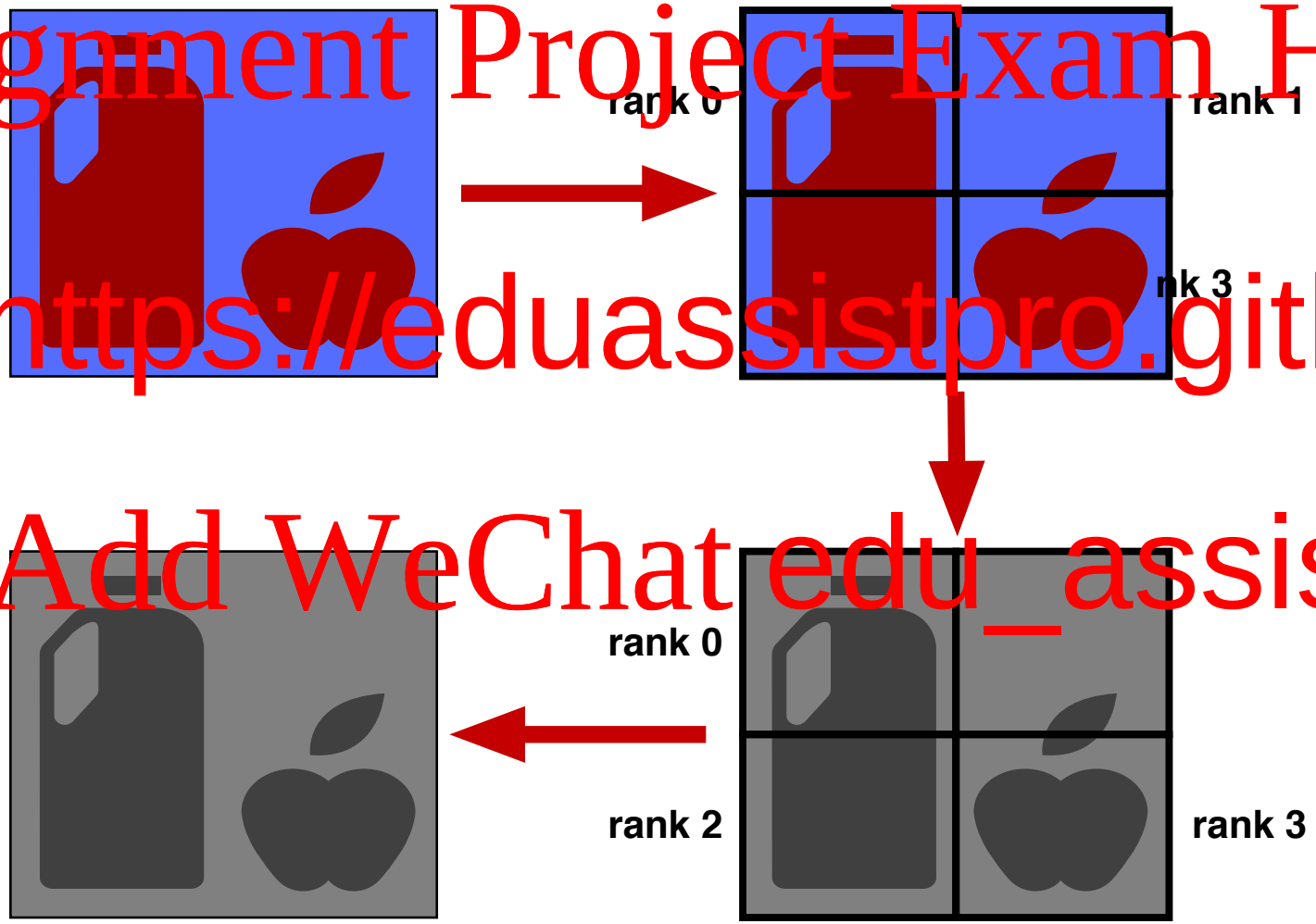
The standard way to parallelise such problems with **distributed** memory is to **partition the domain** between the processes, *i.e.*

Domain partitioning

Each processing unit is responsible for transfor **n**.

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

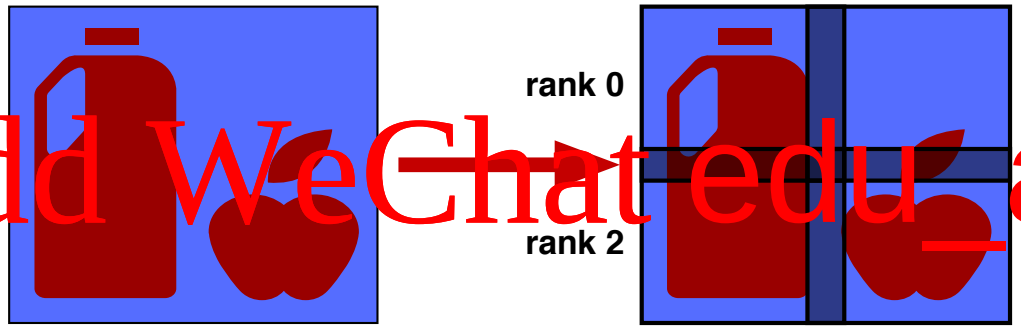
Map example: Colour transformation



Local transformations

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

- **Blurring** or **edge detection** in image processing.
-

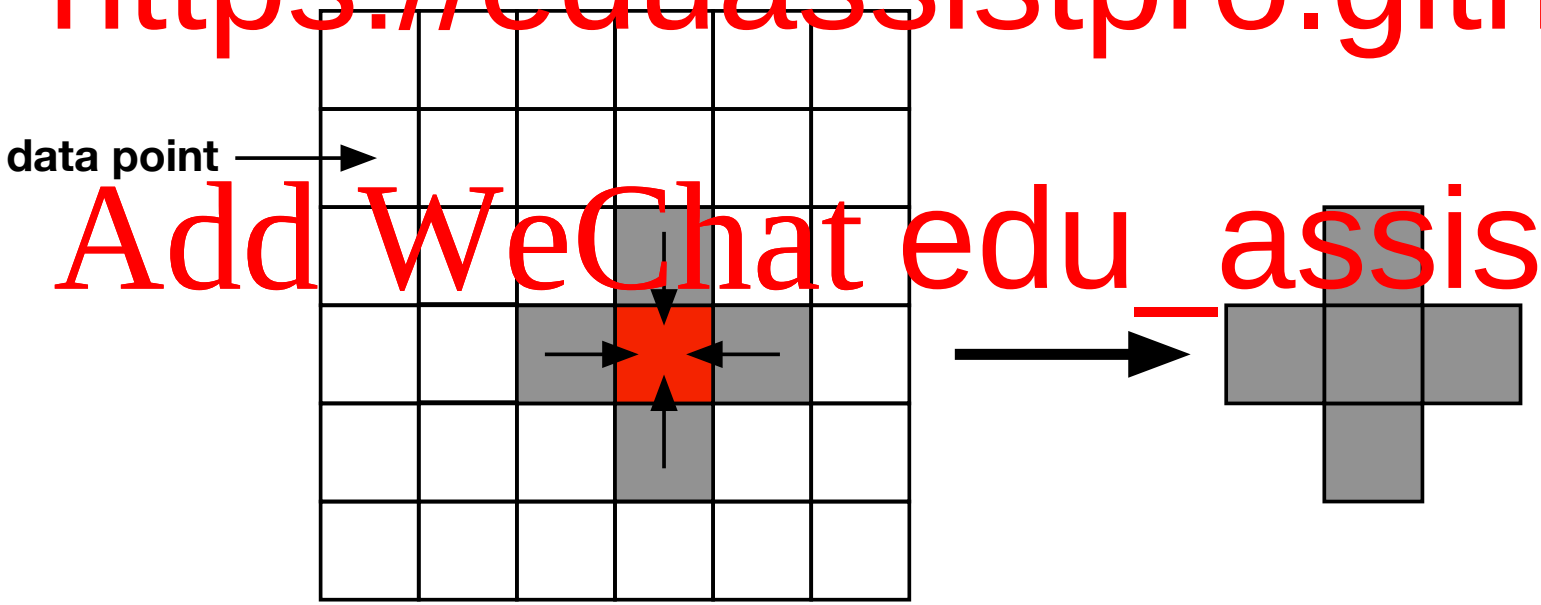


Need to **communicate** information lying at the **edges** of domains to perform the calculations correctly.

Stencils

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

This c

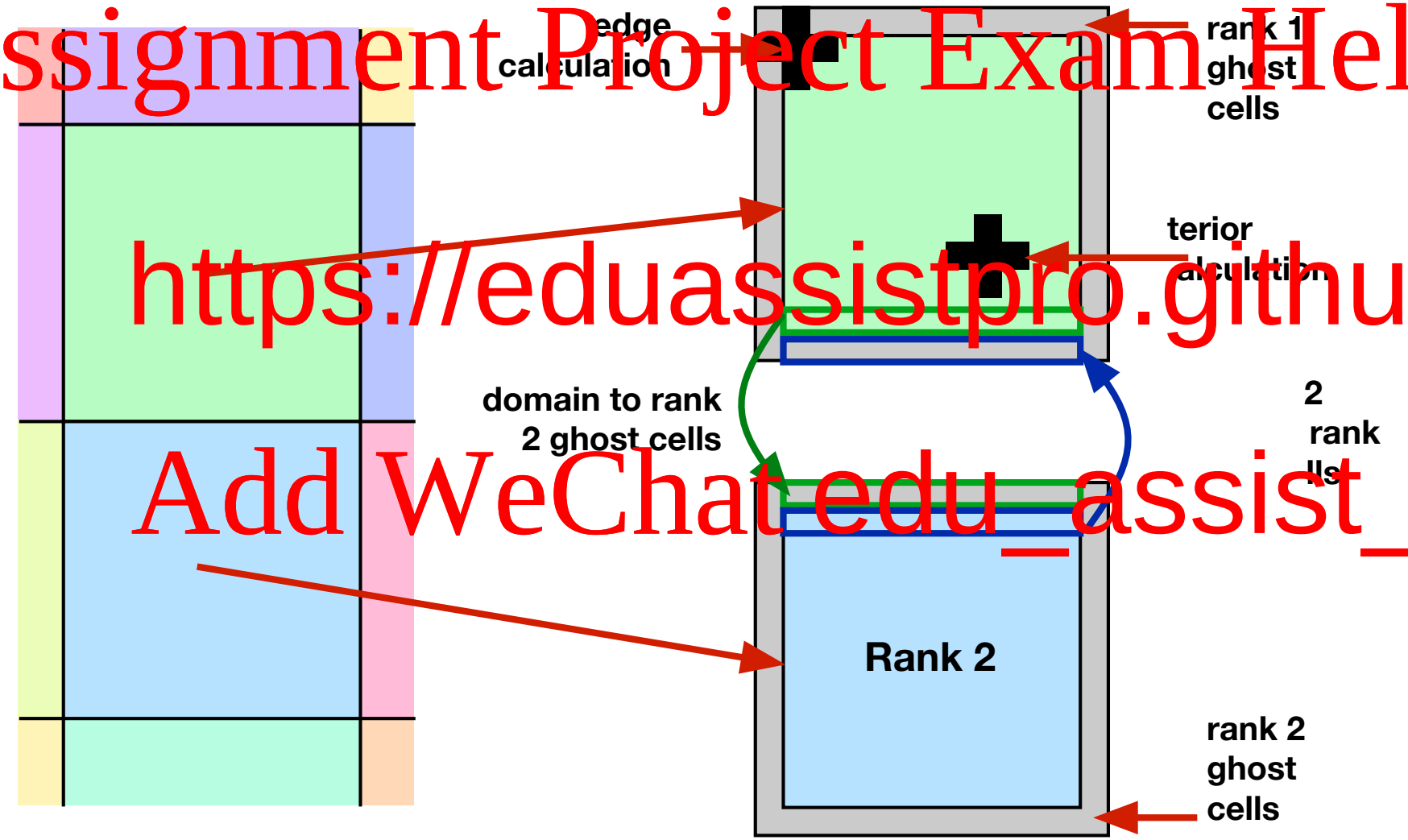


red cell calculation requires values of grey cells

Ghost cells

The standard way to communicate across boundaries is to use **ghos**

-
- neighbouring processes' domain.
- **Updated** after each iteration to match th by the neighbouring processes.
- Updating performed using **point-to-point communication**.



Implementation v1

Code on Minerva: `heatEqn.c`

In pseudocode the obvious implementation would be:

```
1 // Iterate multiple times.
2 for(i
3 {
4   // Se
5   co
6
7   // Update values within this rank's domain.
8   solveWithinDomain();
9 }
```


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:

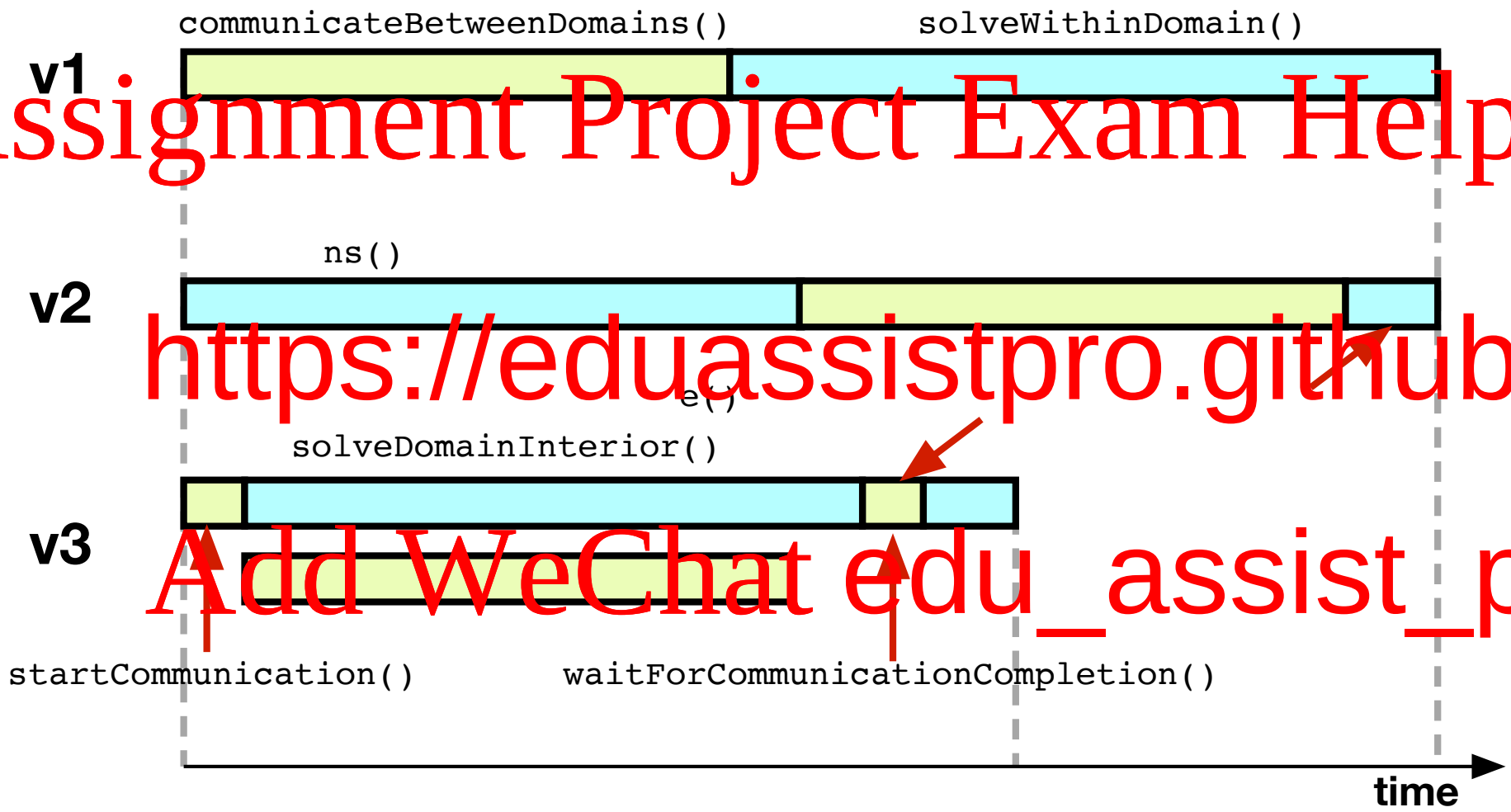
```
1  for(i
2  {
3    // Ca
4    so
5
6    // Send data at edge of domain to other processes.
7    communicateBetweenDomains();   // BLOCKING
8
9    // Now solve the remaining few points at the edge.
10   solveDomainEdge();
11 }
```


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:

```
1  for(iter=0;iter<NUM_ITERATIONS;iter++)
2  {
3    // St
4    st
5
6    // Calculate data points within the domain
7    // WHILE THE COMMUNICATION IS GOING ON!
8    solveDomainInterior();
9
10   // Wait until the communication has finished.
11   waitForCommunicationCompletion();
12
13   // Now solve the remaining few points at the edge.
14   solveDomainEdge();
15 }
```



Summary and next lecture

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

- Example of **domain partitioning**
- **Stencils** describe the locality of the calcu

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