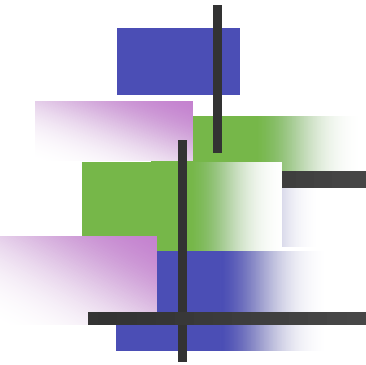


5. Data Decomposition and MPI

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

Goals

- Will discuss, with examples, how data may be distributed across the cooperating processes of a distributed memory parallel program.
  - Illustrate with a possible Mandelbrot Set implementation.
- Move on to consideration of data decomposition in a simulation – the Game of Life again
- Adding MPI communication to the Life program, to make it work in parallel.

DATA DECOMPOSITION

Data Decomposition

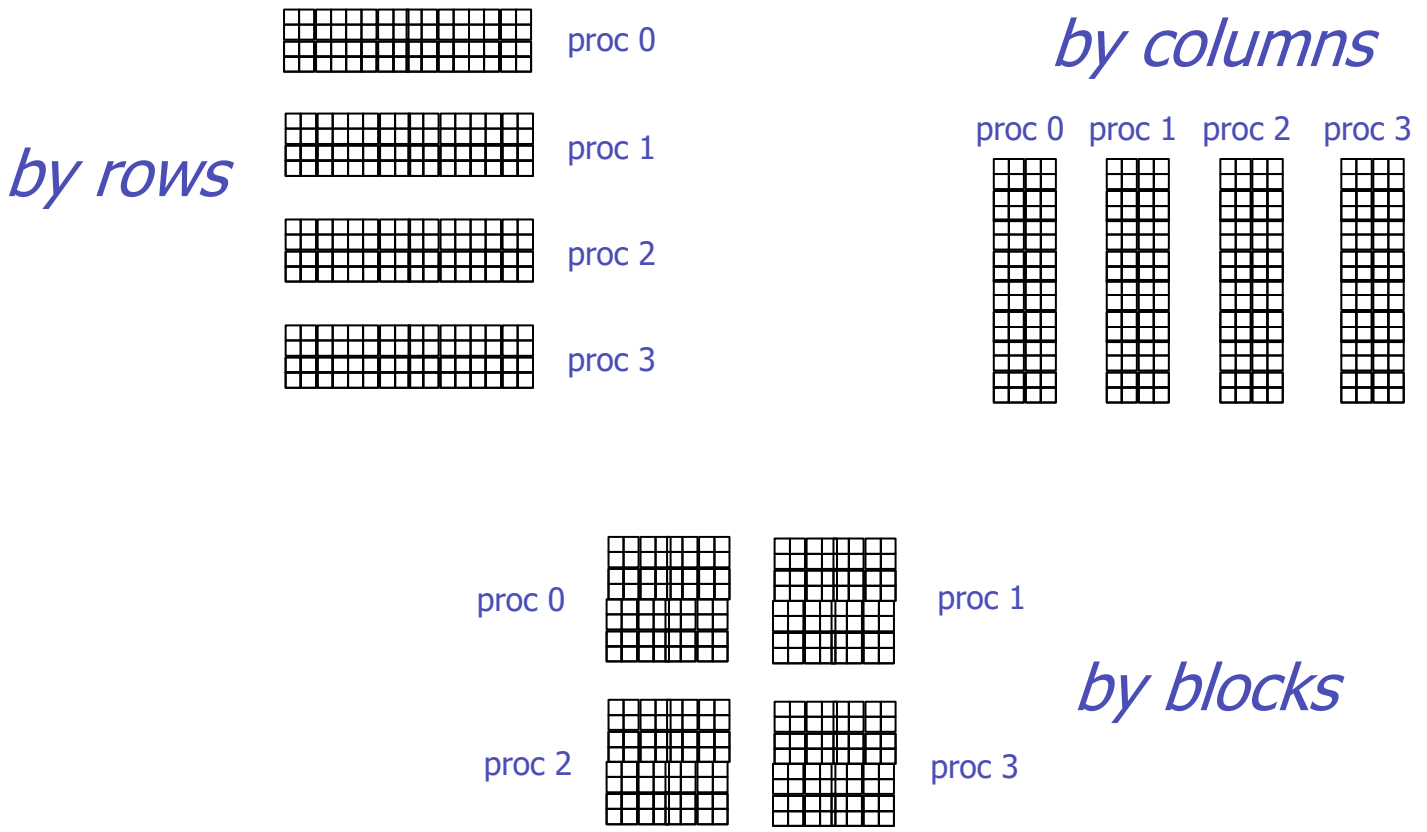
- Earlier lectures discussed *workload decomposition*
  - process of dividing *computations* across cores
  - e.g. dividing range of iterations of loops.
- In distributed memory programming, also need to consider *data decomposition*
  - How the major *data structures* of a program will be divided across processors.
  - e.g. breaking arrays into subsections held by each process.

Data Ownership

- In several examples from lectures and labs, main data structures are *arrays*.
- We have typically divided up work so that *each thread* modifies *one particular section* of an array.
- With *shared memory*, this isn't essential (though *may* make best use of caches).
- With *distributed memory*, it is very often the best approach – a process *owns* and *locally stores* the particular section of the array it primarily works on.

Array Decompositions

- Examples of decomposition of a 2d array:



Domain Decompositions

- Each process holds one "domain" of the whole, logical array
  - Sometimes referred to as a *distributed array*.
- Note similar considerations apply to higher dimensional arrays – e.g. *3d* volume of atmosphere for weather forecasting.
- Can also apply to irregular data structures like *trees*, but then the domains don't usually have any regular shape.

Storage of Domains

- In shared memory, may divide *responsibility for updating* along such lines, but, *programmatically*, remains part of a *single* array.
- Not so with distributed memory:
  - *Programmatically*, each process creates a small array that holds the local domain.
  - *Algorithmically*, may view as part of a single large array.
  - But no representation of this *global* view in ordinary programming languages assumed by MPI! We just have to write a program that manipulates the local part of the array.

Shared Memory Mandelbrot Set

Globally:

```
int set [] [] = new int [N] [N] ;
```

Per thread:

```
int begin = me * B ;    // B is "block size"
int end = begin + B ;
for(int i = begin ; i < end ; i++) {
   for(int j = 0 ; j < N ; j++) {
      double x = step * i - 2.0 ;    // -2 ≤ x ≤ 2
      double y = step * j - 2.0 ;    // -2 ≤ y ≤ 2

      ... inner loop to calculate k as a
          function of x, y ...

      set [i] [j] = k ;
   }
}
```

Distributed Memory Mandelbrot

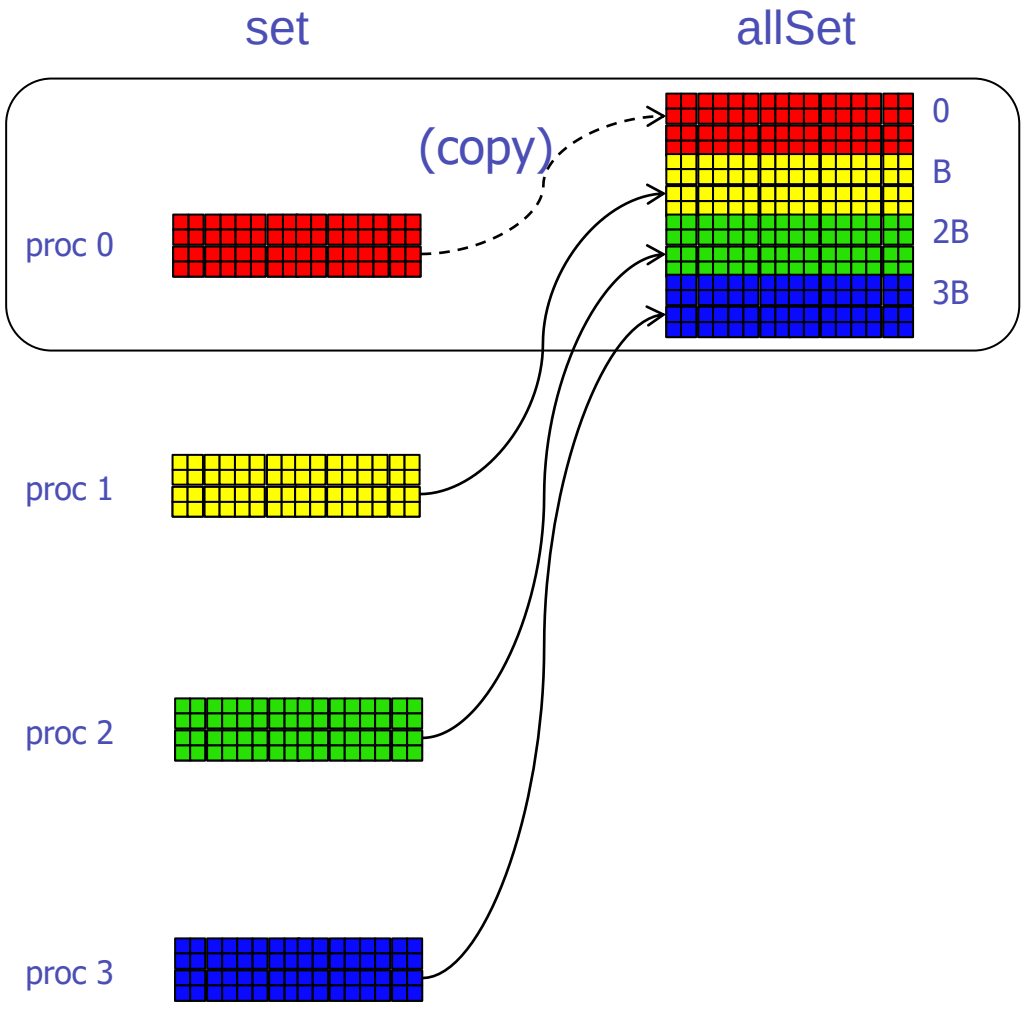
Per process:

```
int set [] [] = new int [B] [N] ;
                    // B is "block size"

int begin = me * B ;

for(int i = 0; i < B; i++) {
   for(int j = 0 ; j < N ; j++) {

      double x = step * (begin + i) - 2.0 ;   // -2 ≤ x ≤ 2
      double y = step * j - 2.0 ;             // -2 ≤ y ≤ 2

      ... inner loop to calculate k as a
          function of x, y ...

      set [i] [j] = k ;
  }
}
```

MPI Mandelbrot

- One MPI version of the Mandelbrot set could follow structure on previous slide.
- In this case (similar to π calculation) only communication is at end, where process 0 must collect together ("gather") segments from all processes, for display or storage.
- Following code assumes process 0 (only) creates an $N \times N$ temporary array allSet, into which results are collected ready for display.

Communication Pattern



Communications in MPI Mandelbrot

```
if(me > 0) {
    MPI.COMM_WORLD.Send(set, 0, B, MPI.OBJECT, 0, 0) ;
}
else {    // me == 0
    for(int i = 0 ; i < B ; i++) {    // copy local `set' to start of `allSet'
        for(int j = 0 ; j < N ; j++) {
            allSet [i] [j] = set [i] [j] ;
        }
    }
    for(int src = 1 ; src < P ; src++) {
        MPI.COMM_WORLD.Recv(allSet, src * B, B, MPI.OBJECT, src, 0) ;
    }
     ... display allSet ...
}
```

Aside: 2d Arrays in MPJ

- In MPJ, message buffers are *1d arrays*.
- But in Java a *2d array* is an *array of arrays*...
- ... can *regard* a 2d array as a 1d array, *but* elements of this array don't have simple type (elements are *themselves* arrays).
- In MPJ, any *non-primitive* element is sent as MPI.OBJECT – here each "object" sent as an element of buffer is a 1d array representing a *row* of the original 2d array.
- Note: communicating MPI.OBJECTs has big serialization overheads – use it judiciously!

Aside: Distribution Format

- Notice we reverted to a block-wise decomposition of work here.
- We know this isn't good for load balancing in the Mandelbrot set.
- Like workload, data arrays *can* be decomposed *cyclically* - would lead to better load balancing.
  - It involves a little more work with collecting the data together at the end
  - One could recv data segments into a receive buffer before copying rows to rightful places in allSet...
  - ... or use advanced features of MPI/MPJ to recv directly into non-consecutive parts of allSet.

INTERACTING PROCESSES

Interacting Processes

- As previously observed, *π* and *Mandelbrot* are *embarrassingly parallel* - only communication is to collect results together after all work has been done.
- Now must consider a more challenging example where processes must communicate *during* the computation – how, for example, do we write a *Life* simulation in MPI?

Sequential Life - Schematic

```
int cells [] [] = new int [N][N] ;
int sums [] [] = new int [N][N] ;

while(true) {
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] =
                cells [i-1][j-1]  + cells[i-1][j] + cells[i-1][j+1] +
                cells [i]    [j-1] +                    cells[i]   [j+1] +
                cells [i+1][j-1] + cells[i+1][j] + cells[i+1][j+1] ;
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
}
```

Notes

- Have ignored the problems at the edges of the board.
- In the lab version we slightly modified calculation of $i \pm 1$, $j \pm 1$ to be modulo N - thus board "wrapped around" at edges.
- Assume this has been done here – it doesn't alter things except at extreme edges of the board.
- See earlier notes/lab scripts for definition of update rule – doesn't much affect parallelization.

First Sketch of Distributed Memory Life

- Per process (*B* is *N* / *P* – the block size):

```
int cells [] [] = new int [B][N] ;
int sums [] [] = new int [B][N] ;

while(true) {
   for(int i = 0 ; i < B ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] =
               cells [i-1][j-1]  + cells[i-1][j] + cells[i-1][j+1] +
               cells [i]    [j-1] +                cells[i]   [j+1] +
               cells [i+1][j-1] + cells[i+1][j] + cells[i+1][j+1] ;
   ... update loop ...
}
```

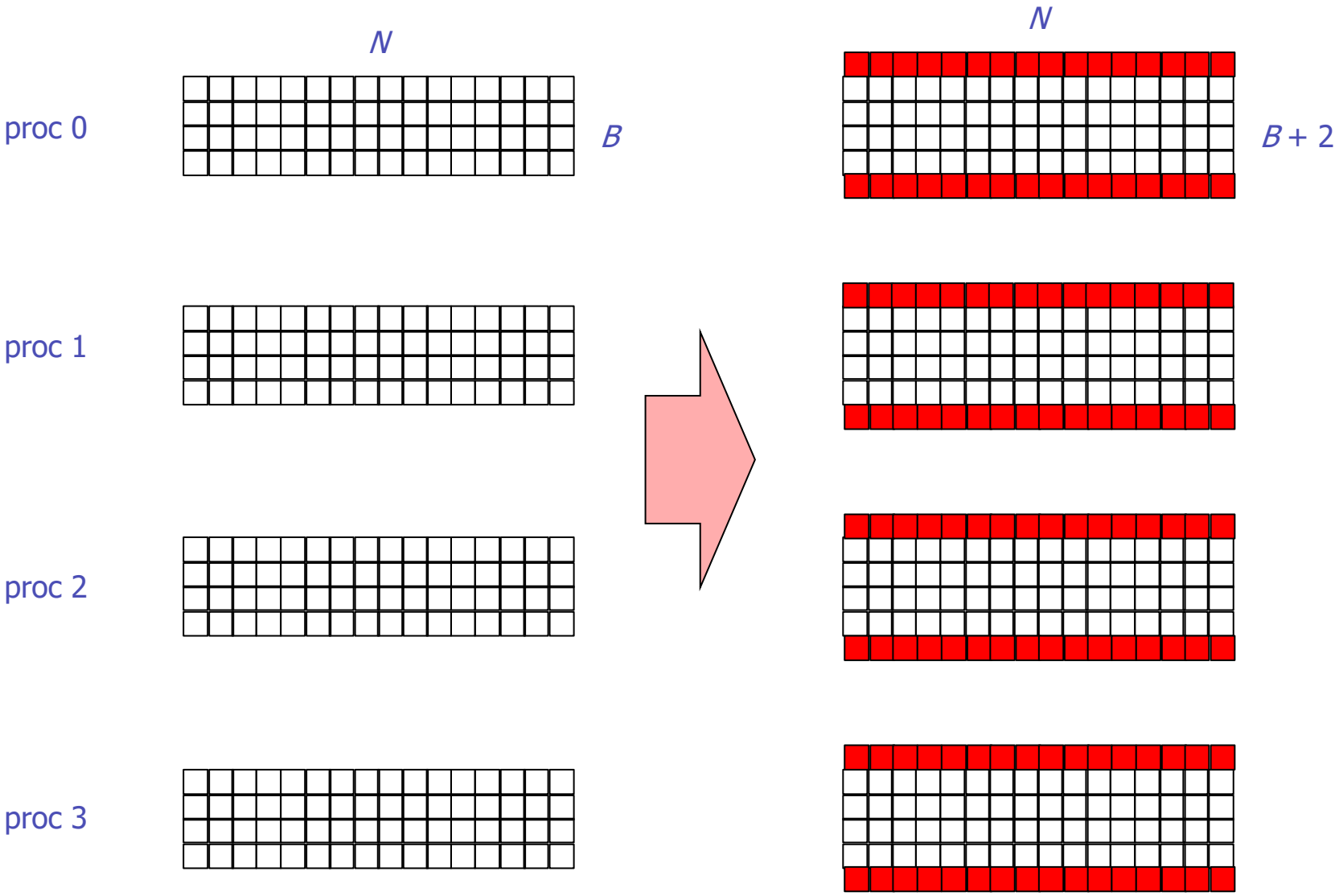
Edge of Block Problem

- This is the beginning of the right structure.
- But clearly there is a problem here:
  - if i = 0 then all accesses like cells[i-1][...] produce an indexing exception
  - likewise for cells[i+1][...] accesses when i = B-1.
- We *cannot* "wrap around" within block held by local process – for correct results we need to get elements belonging to adjacent processes!

Ghost Regions

- We *could* put some kind of test inside the loops and do something different when we are at the edge of the block.
- But experience shows that it is nearly always best to keep the structure of the *inner loops* as close as possible to their structure in the sequential program.
- So the first thing we do is "grow" the local arrays by one row on either side of the blocks...

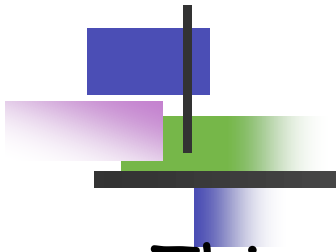
Adding Ghost Regions



Distributed Life with Ghost Regions

- Per process (*B* is *N* / *P* – the block size):

```
int cells [] [] = new int [B+2][N] ;

while(true) {
   for(int i = 1 ; i < B+1 ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] =
               cells [i-1][j-1]  + cells[i-1][j] + cells[i-1][j+1] +
               cells [i]    [j-1] +                  cells[i]   [j+1] +
               cells [i+1][j-1] + cells[i+1][j] + cells[i+1][j+1] ;
   ... update loop ...
}
```

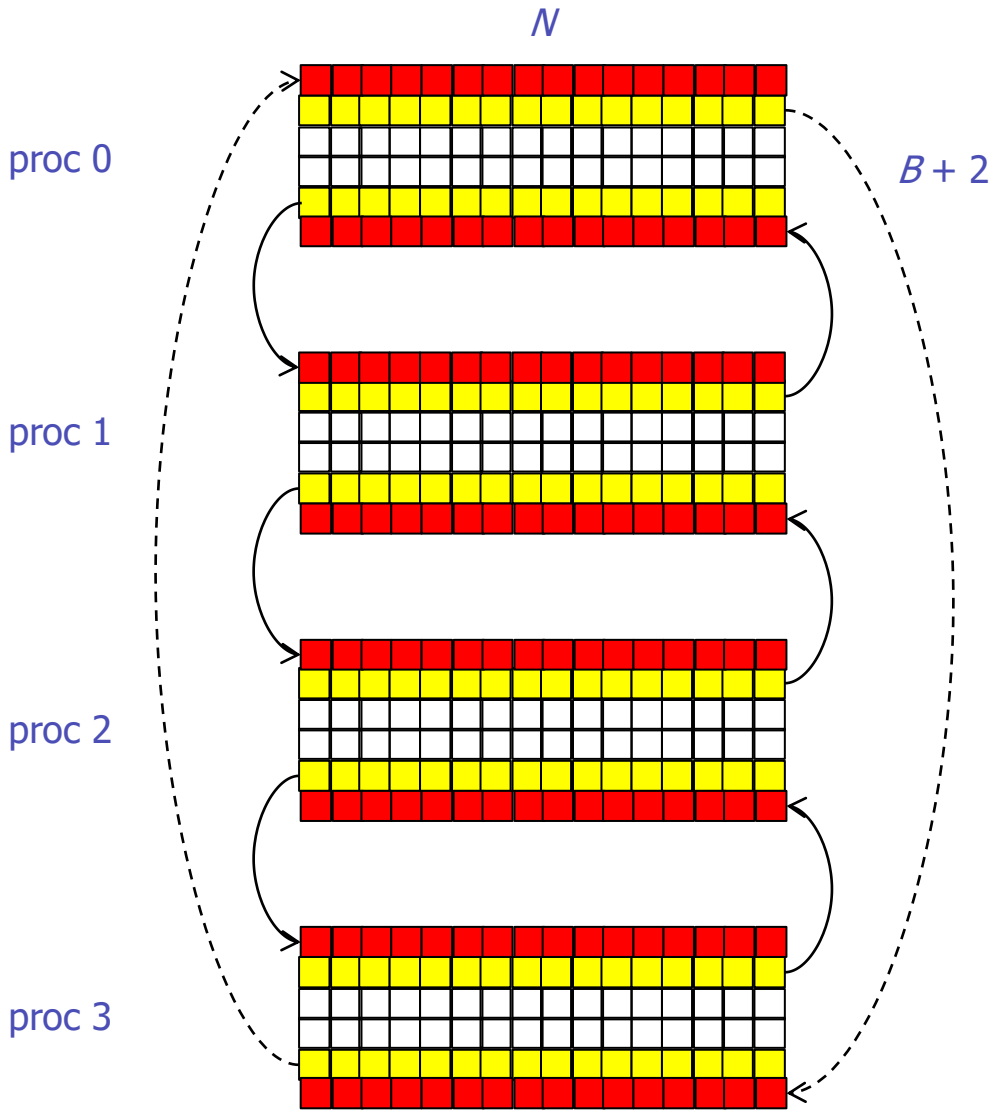


Notes

- This eliminates the array indexing exceptions.
- But only gets *correct* results if ghost region contains value from adjacent (non-ghost) row in neighbouring process.

SWAPPING EDGES

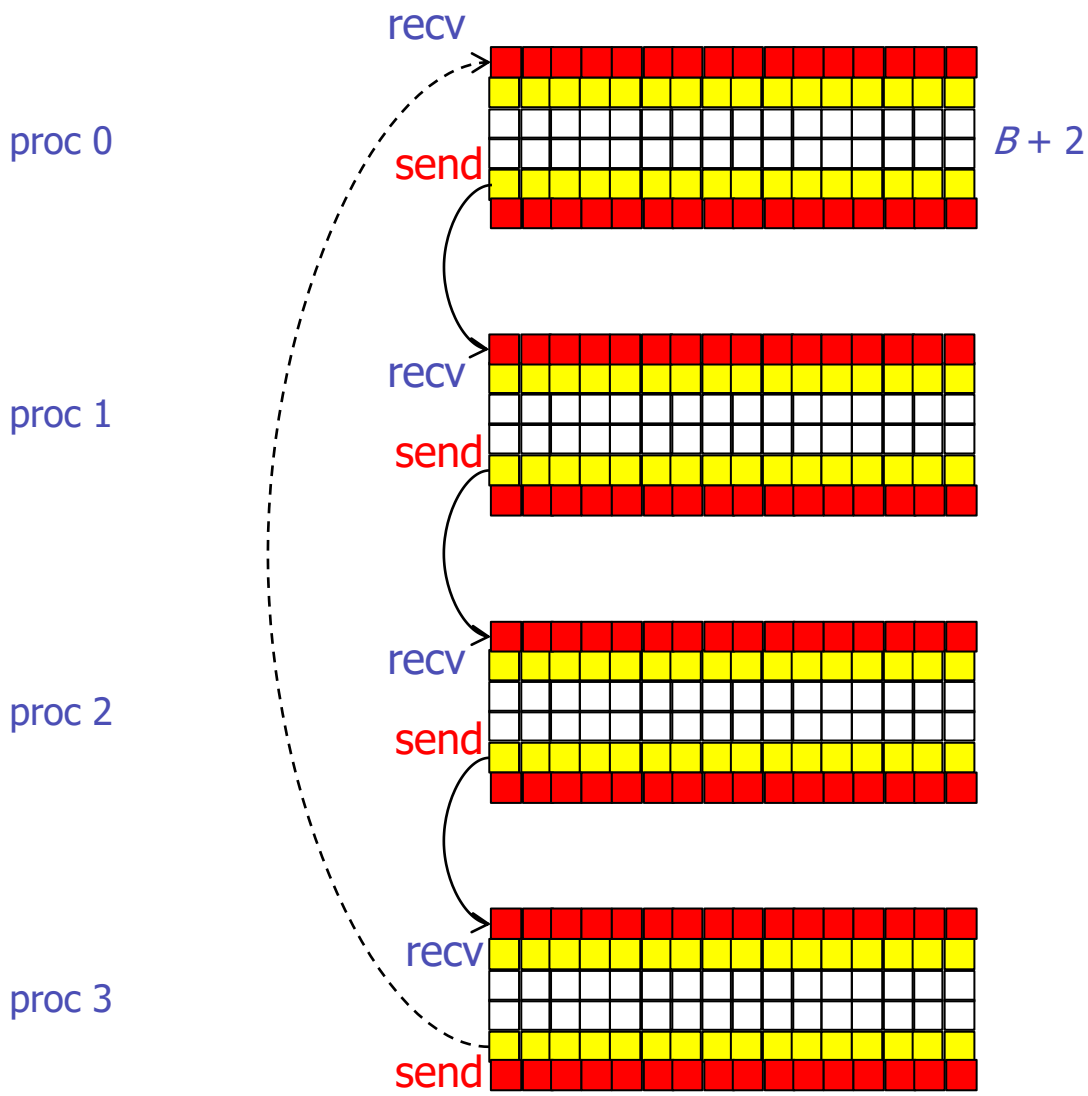
What Ghost Regions *Should* Contain



Edge Swaps

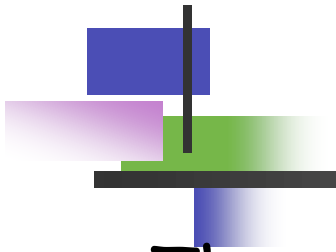
- Immediately before starting the "sum of neighbours" loop, we should perform an MPI communication to get the values from the yellow cells to the neighbouring ghost cells – then everything should come out right.
- This communication is called an "*edge swap*".
- The communication must be repeated in every iteration of the main loop over generations.

Communication Pattern for top red regions



Attempt at Getting –ve Neighbours

```
int next = (me + 1) % P ;
int prev = (me - 1 + P) % P ;

MPI.COMM_WORLD.Send (cells[B], 0, N, MPI.INT, next, 0) ;
MPI.COMM_WORLD.Recv (cells[0], 0, N, MPI.INT, prev, 0) ;
```



An Old Problem

- The method Send() implements what is called in MPI *standard mode send*.
- Standard mode gives no guarantee of buffering of messages; Send() may block until Recv() is called in destination neighbouring process.
- But in our logic *every* process calls Send() first, and if these calls block, *no* process can reach Recv().
- Thus every process may be waiting for another process – *what is this condition called*?
  - Hint: there is a cycle here!

Getting –ve Neighbours - Correctly

- The easiest way to avoid this kind of *deadlock* is by using the MPI Sendrecv() call, which initiates a send and receive "at the same time":

```
int next = (me + 1) % P ;
int prev = (me - 1 + P) % P ;

MPI.COMM_WORLD.Sendrecv(cells[B], 0, N, MPI.INT, next, 0,
                                                cells[0], 0, N, MPI.INT, prev, 0) ;
```

Full Edge Exchange

- Populating ghost regions from *both* neighbours:

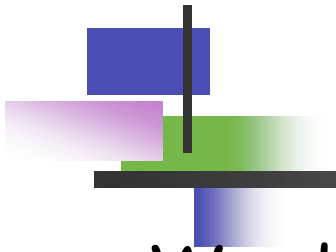
```
int next = (me + 1) % P ;
int prev = (me - 1 + P) % P ;

MPI.COMM_WORLD.Sendrecv(cells[B], 0, N, MPI.INT, next, 0,
                                          cells[0], 0, N, MPI.INT, prev, 0) ;

MPI.COMM_WORLD.Sendrecv(cells[1], 0, N, MPI.INT, prev, 0,
                                          cells[B+1], 0, N, MPI.INT, next, 0) ;
```

Complete Distributed Life

```
int cells [] [] = new int [B+2][N] ;

while(true) {
   ... Edge exchange – see previous slide
   for(int i = 1 ; i < B+1 ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] =
                cells [i-1][j-1]  + cells[i-1][j] + cells[i-1][j+1] +
                cells [i]    [j-1] +                cells[i]   [j+1] +
                cells [i+1][j-1] + cells[i+1][j] + cells[i+1][j+1] ;
   ... Local update loop ...
}
```

Notes

- Finding neighbours in j direction ($j \pm 1$) still done modulo N.
- No longer need this for neighbours in i direction ($i \pm 1$), because now taken care of by ghost regions plus cyclic edge swap.



Summary

- We discussed how in practice to make an MPI parallel program by decomposing the data structures and inserting MPI communication.
- You now know enough about MPI to do basic parallel programming on clusters!
- Next week: *The Rest of MPI*.

Further Reading

- William Gropp, Ewing Lusk and Anthony Skjellum, *Using MPI*, 2nd Edition MIT Press, 1999.
  - Standard text on MPI, but examples are in C and Fortran.
  - Available as an electronic book through the library.