MPI Casestudy: Parallel Percolation Simulation

David Henty

Contents

List of Figures

1 Introduction

The aim of this exercise is to write a complete MPI parallel program that does a simple simulation of water percolating through porous rock. We will start with a serial code that performs the required calculation but is also designed to make subsequent parallelisation as straightforward as possible. The rock is represented by a large two-dimensional array, so the natural parallel approach is to use regular domain decomposition. As the algorithm we will use involves nearest-neighbour interactions between grid points, this will require boundary swapping between neighbouring processes and adding halos to the arrays. The exercise also utilises parallel scatter and gather operations. For simplicity, we will use a one-dimensional process grid (i.e. decompose the problem into slices).

The solution can easily be coded using either C or Fortran. The only subtlety is that the natural direction for the parallel decomposition of the arrays, whether to slice up the 2D grid over the first or second dimension, is different for the two languages. A sample serial solution, including a function to write the result to an image file, can be found in `MPP-percolate.tar` on the MPP course web pages.

If you are following the *Programming Skills* (PS) course then you will be familiar with the problem and the algorithm for solving it. If you want, you can skip to Section 4 to read about the provided serial code. Note that this serial code implements the algorithm in a slightly different way and is written differently, both to make parallelisation simpler.

2 The Percolation Problem

The program solves the following problem. Suppose we have a grid of squares (e.g. 5×5) in which a certain number of squares are “filled” (grey) and the rest are “empty” (white). The problem to be solved is whether there is a path from any empty (white) square on the top row to any empty square on the bottom row, following only empty squares – this is similar to solving a maze. The connected empty squares form clusters: if a cluster touches both the top and bottom of the grid, i.e. there is a path of empty squares from top to bottom through the cluster, then we say that the cluster *percolates*.

See Figure 1 for an example of a grid that has a cluster which percolates. See Figure 2 for an example of a grid with no cluster that percolates.

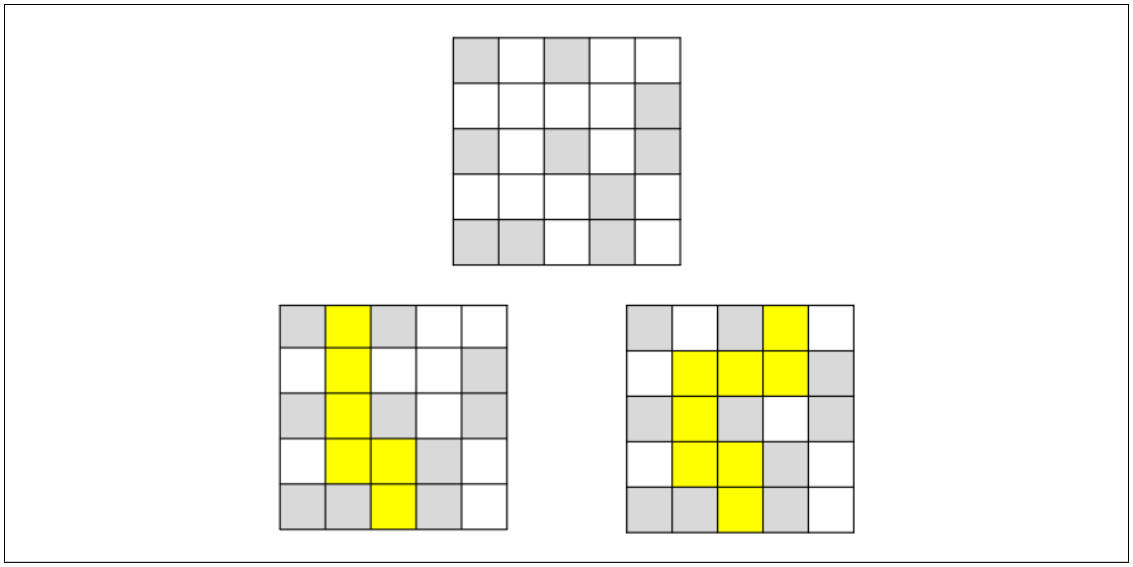


Figure 1: The grid at the top has a cluster which has paths through empty squares from top to bottom. Two example paths are highlighted in yellow. This grid *has a cluster that percolates*.

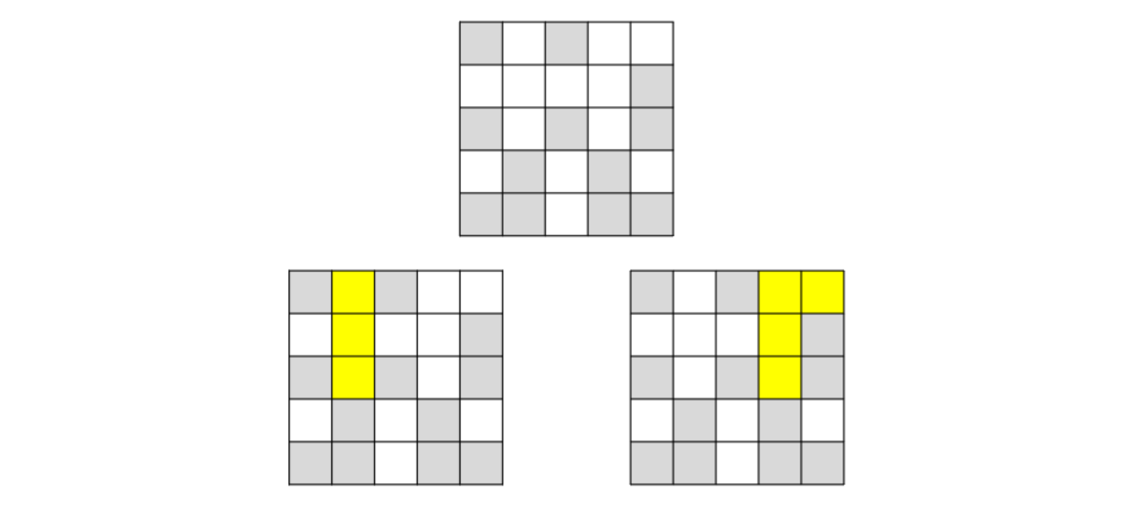


Figure 2: The grid at the top has no cluster with a path through empty squares from top to bottom. Two examples of the longest paths are highlighted in yellow. This grid *does not have a cluster that percolates*.

3 Cluster Identification Algorithm

The algorithm identifies all the connected clusters – the largest clusters are the ones of interest for percolation. In the grid of Figure 1 the largest cluster is 13 squares, and the second largest cluster is 2 squares. In the grid of Figure 2 the largest cluster is 9 squares and the second largest cluster is 2 squares.

Now, consider an $L \times L$ grid where we vary the number of filled squares. The question we want to answer is: will we get percolation if the grid is filled with a given density ρ (Greek letter, pronounced "rho") where $0.0 \leq \rho \leq 1.0$, i.e. where the density ranges from empty to completely solid rock? In the grid of Figure 1, $L = 5$ and $\rho = 0.40$ (10/25, as 10 squares are filled). In the grid of Figure 2, $L = 5$ and $\rho = 0.48$ (12/25, as 12 squares are filled). For a given ρ there are many possible grids so we generate a random grid and see if a cluster percolates. To check the reliability of our answer we would re-run many times with different grids, i.e. use different random number seeds.

The simplest cases are $\rho = 0.0$ (all squares are empty) when we always get percolation and $\rho = 1.0$ (solid rock where all squares are filled) when we never get percolation. But what about $\rho = 0.5$? Or $\rho = 0.4$ or 0.6? What is the critical value of ρ which separates percolating rocks from non-percolating rocks? Does the answer depend on the size of the simulation, i.e. the value of L?

We can answer these questions using computer simulation.

First, we initialise the grid:

- declare an integer array of size $L \times L$;
- each of the white squares (which represent empty space) are set to a *unique* positive integer;
- all of the filled squares, representing solid rock, are set to zero.

Then, we loop over all the non-zero squares in the grid for many steps and successively *replace each square with the maximum of its four neighbours*. The largest positive numbers gradually fill the gaps so that each cluster eventually contains a single, unique number (the filled squares are automatically ignored as they are set to zero). If we monitor how many values change at each step then we can tell when we have identified all the clusters as nothing will change between successive steps.

We can then count and identify the clusters. If we find a part of the same cluster on both the top and bottom row of the grid then we can conclude that the cluster percolates – see Figure 3.

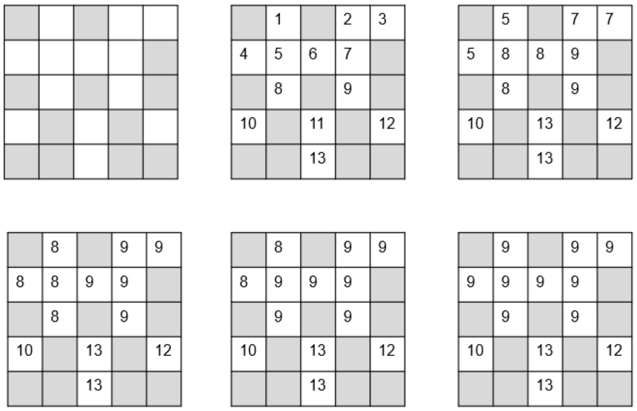


Figure 3: Solving percolate in five steps after initialisation. No squares would be updated on a subsequent step. All the filled (grey) squares can be assumed to be zero.

Our solution involves replacing each square with the maximum of its four neighbours, but what about squares at the edges of the grid which have no neighbours? We do not want to write extra code to handle these as a special case, so a common solution to this problem is to define a “halo” around the grid: we surround our grid with a layer of solid rock (to stop the water leaking out!).

For an $L \times L$ percolation problem we therefore use an $(L+2) \times (L+2)$ grid and set the halos to be filled (i.e. zero). Zeros block cluster growth and they will never propagate to other squares – see Figure 4.

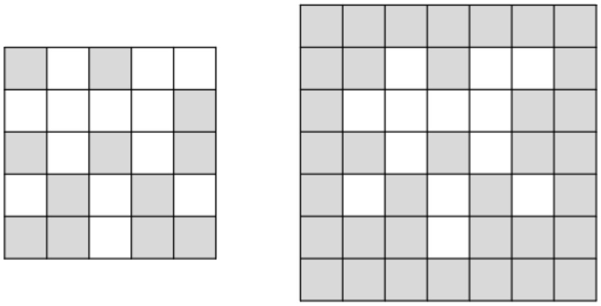


Figure 4: To process a 5×5 grid we add a halo of filled squares round the border of the grid to produce a 7×7 grid. See Figure 5 for the example of Figure 3 but with the use of a halo.

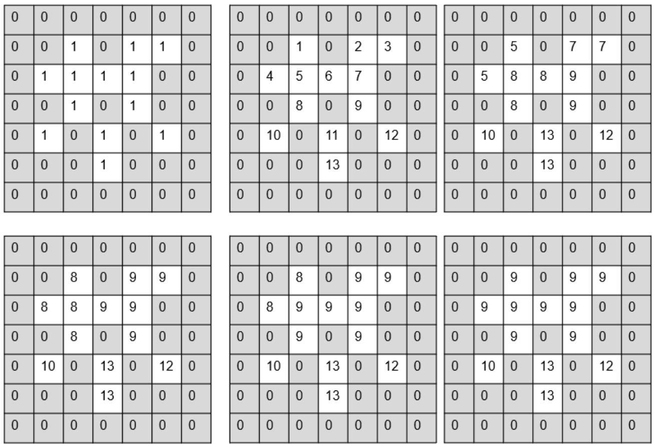
4 Serial Code

Download the serial code `MPP-percolate.tar` from the MPP course web pages. It should compile and run as supplied. **Note** for this simulation we define percolation as **a cluster that spans the grid from left to right** and **not from top to bottom** as in the explanation above.

The default parameters are:

- system size $L = 432$;
- rock density $\rho = 0.411$.

It takes a single command-line parameter – the random number seed. To reproduce the results shown here use $seed = 6543$.



0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	1	1	1	0	0
0	0	1	0	1	0	0
0	1	0	1	0	1	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	0	1	0	2	3	0
0	4	5	6	7	0	0
0	0	8	0	9	0	0
0	10	0	11	0	12	0
0	0	0	13	0	0	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	0	5	0	7	7	0
0	5	8	8	9	0	0
0	0	8	0	9	0	0
0	10	0	13	0	12	0
0	0	0	13	0	0	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	0	8	0	9	9	0
0	8	8	9	9	0	0
0	0	8	0	9	0	0
0	10	0	13	0	12	0
0	0	0	13	0	0	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	0	8	0	9	9	0
0	8	9	9	9	0	0
0	0	9	0	9	0	0
0	10	0	13	0	12	0
0	0	0	13	0	0	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	0	9	0	9	9	0
0	9	9	9	9	0	0
0	0	9	0	9	0	0
0	10	0	13	0	12	0
0	0	0	13	0	0	0
0	0	0	0	0	0	0

Figure 5: Solving percolate using a halo

On Cirrus:

```
[user@cirrus]$ tar -xvf MPP-percolate.tar
...
[user@cirrus]$ cd percolate/C-SER
[user@cirrus]$ make
icc -O3 -c percolate.c
icc -O3 -c percwrite.c
icc -O3 -c uni.c
icc -O3 -o percolate percolate.o percwrite.o uni.o

[user@cirrus]$ ./percolate 6543
percolate: L = 432, rho = 0.411000, seed = 6543, maxstep = 1728
percolate: rho = 0.411000, actual density = 0.410928
percolate: number of changes on step 100 is 27742
percolate: number of changes on step 200 is 14288
percolate: number of changes on step 300 is 13375
percolate: number of changes on step 400 is 11911
percolate: number of changes on step 500 is 12977
percolate: number of changes on step 600 is 10948
percolate: number of changes on step 700 is 10426
percolate: number of changes on step 800 is 6132
percolate: number of changes on step 900 is 3875
percolate: number of changes on step 1000 is 1507
percolate: number of changes on step 1100 is 0
percolate: number of changes on step 1200 is 0
...
percolate: number of changes on step 1700 is 0
percolate: cluster DOES NOT percolate
percwrite: only visualising the largest cluster
percwrite: cluster sizes are 40109, 6571, 4861
percwrite: opening file <map.pgm>
percwrite: writing data ...
percwrite: ... done
percwrite: file closed

[user@cirrus]$ display map.pgm
```

You should see the same output as the left-hand pane in Figure 6 (to access the `display` program on Cirrus you need to issue the command `module load ImageMagick`).

To show the largest n clusters, change the final parameter in the `percwrite` call in the main program from 1 to n (up to a maximum of $n = 9$), e.g. the right-hand pane in Figure 6 shows the output using `percwrite("map.pgm", map, 3)`. We will use this feature to help debug the parallel program.

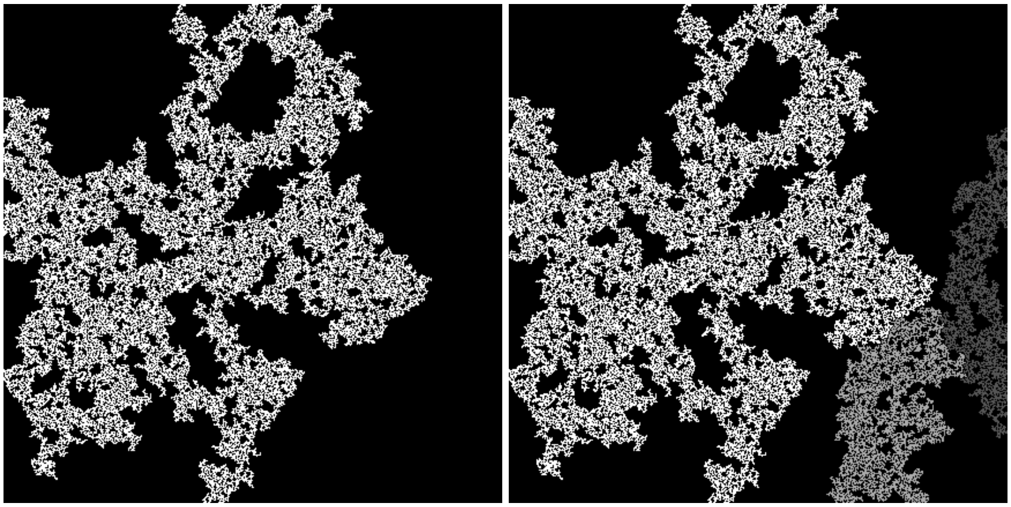


Figure 6: The default example showing the largest (left) and largest three (right) clusters. Note that the largest cluster does not percolate as it does not quite reach the right-hand side.

The code runs for a fixed number of steps (set to $4 \times L$). Although it calculates the number of changes at each step, it does not stop when this becomes zero. This will prove useful when measuring the performance of the parallel program. Even if the clusters are found very quickly, we can run for a large number of steps to get an execution time long enough to enable us accurately to estimate the time per step. Note that this simple algorithm performs the same amount of work per step even after all the clusters have been identified: it still compares every non-zero grid point to its four nearest neighbours.

4.1 Method

Using the algorithm illustrated in Figure 5, values are changed one-by-one as we loop over the grid, i.e. the update is done *in-place*. This means that a parallel implementation would not be exactly the same, step-by-step, as the serial algorithm (although it would eventually give the same answer) – do you understand why? In our implementation, at each step we create a new grid based purely on the values of the old grid, then copy new back to old in preparation for the next step.

We need to have halos on the main arrays. It is simplest if the indices range from 0 to $L + 1$, where the grid itself is represented by indices $1, 2, 3, \ldots, L$ and the halos are indices 0 and $L + 1$. We can do this in C and Fortran by using:

```
int      old[M+1][N+2]       // C
integer old(0:M+1, 0:N+1)   !  Fortran
```

For initialisation and IO we also use an $L \times L$ array called map which *does not have any halos*.

In the serial code, although we have separate constants M and N for the two dimensions of the arrays, these are both set to L. However, in the parallel algorithm, we will have to alter these definitions as the array sizes on each process will be smaller by a factor of the number of processes P.

We will see that, for C programs, $M = \frac{L}{P}$ and $N = L$; for Fortran, $M = P$ and $N = \frac{L}{P}$. This is why we distinguish between L, M and N in the serial algorithm even though these values are all the same - it is to make it easier to parallelise.

4.2 Algorithm in pseudocode

```
1. declare $M \times N$ integer arrays *old* and *new* with halos, and an $L \times L$ array *map* without halos
2. seed the random number generator from command-line option and initialise *map*
3. loop over $i = 1, M; j = 1, N$
    • $old_{i,j} = map_{i-1,j-1}$ (in C)
    • $old_{i,j} = map_{i,j}$ (in Fortran)
4. zero the bottom, top, left and right halos of *old*
5. begin loop over steps
    • loop over $i = 1, M; j = 1, N$
        – set $new_{i,j} = max(old_{i-1,j}, old_{i+1,j}, old_{i,j-1}, old_{i,j+1})$
        – increment the number of changes if $new_{i,j} \neq old_{i,j}$
    • report the number of changes every 100 steps
    • set the $old$ array equal to $new$, making sure that *the halos are not copied*
6. end loop over steps
7. loop over $i = 1, M; j = 1, N$
    • $map_{i-1,j-1} = old_{i,j}$ (in C)
    • $map_{i,j} = old_{i,j}$ (in Fortran)
8. test $map$ to see if any clusters percolate all the way from left to right
9. write out the result by passing $map$ to *percwrite*
```

5 Initial Parallelisation

For the initial parallelisation we will simply use trivial parallelism, i.e. each process will work on different sections of the image but with no communication between them. Although this will not identify the clusters correctly, since we are not performing the required halo swaps, it serves as a good intermediate step to a full parallel code. Most importantly it will have exactly the same data decomposition and parallel IO approaches as a fully working parallel code.

It is essential that you complete this initial parallelisation before continuing any further

The entire parallelisation process is made much simpler if we ensure that the slices of the grids operated on by each process are contiguous pieces of the whole image (in terms of the layout in memory). This means dividing up the image between processes over the first dimension i for a C array `old[i][j]`, and the second dimension j for a Fortran array `old(i,j)`. Figure 7 illustrates how this would work on 4 processes where the slices are numbered according to the rank of the process that owns them.

Again, for simplicity, we will always assume that L is exactly divisible by the number of processes P. It is easiest to program the exercise if you make P a compile time constant (a `#define` in C or a `parameter` in Fortran).

- for C: $M = L/P$ and $N = L$
- for Fortran: $M = L$ and $N = L/P$

The simplest approach for initialisation and output is to do it all on one controller process. The map array is scattered to processes after initialisation, and gathered back before file IO. Note this is not particularly efficient in terms of memory usage as we need enough space to store the whole map on a single process.

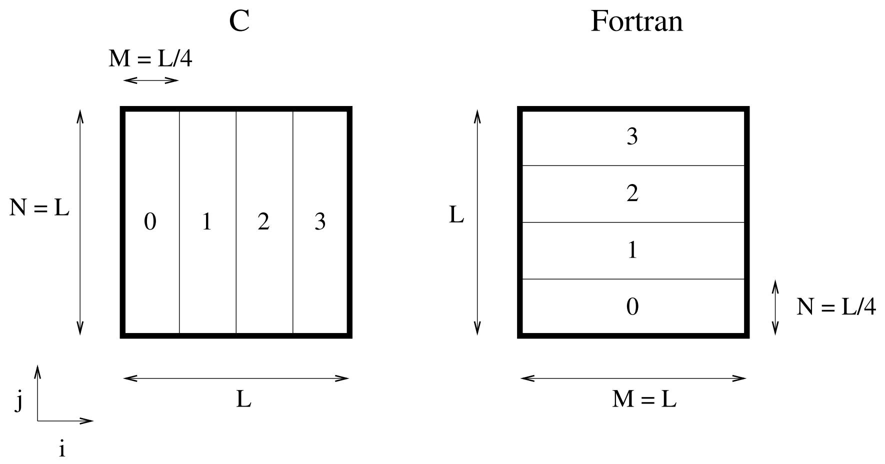


Figure 7: Decomposition strategies for 4 processes

5.1 The parallel program

The steps to creating the first parallel program are as follows.

a) set M and N appropriately for the parallel program

b) create a new $M \times N$ array called $smallmap$ without any halos

c) initialise MPI, compute the $size$ and $rank$, and check that $size$ = P

d) initialise map on the controller process only

e) scatter the map from the controller to all other processes using `MPI_Scatter` with $sendbuf = map$ and $recvbuf = smallmap$

f) in steps 3 and 7, replace map with $smallmap$

g) follow steps 4 through 6 of the original serial code exactly as before.

h) gather the map back to the controller from all other processes using `MPI_Scatter` with $sendbuf = smallmap$ and $recvbuf = map$

i) check for percolation and write out the result on the controller process only

Since we do the initialisation, check for percolation and perform file output in serial (i.e. on the controller process only), these parts do not require to be changed.

5.2 Testing the parallel program

It will be easier to check things are correct if you visualise more than just the largest cluster. For example, if you are running on 4 processes, call `percwrite` with the number of clusters set to 8. You should also re-run the serial code to visualise the largest 8 clusters so you have a correct reference result.

How does your output compare to the correct (serial) answer? Does the algorithm take the same number of steps until there are no changes? Do you understand what is happening?

As you increase the number of processes, does the total execution time decreasing as you expect? In terms of Amdahl's law, what are the inherently serial and potentially parallel parts of your (incomplete) MPI program?

6 Full Parallel Code

The only change now required to get a full parallel code is to add halo swaps to the old array (which is the only array for which there are non-local array references of the form $old_{i-1,j}$, $old_{i,j+1}$ etc). This should be done once every step immediately after the start of the loop and before any other computation has taken place.

To do this, each process must know the $rank$ of its neighbouring processes. Referring to the C decomposition as shown in Figure 7, these are given by $rank - 1$ and $rank + 1$. However, since we have fixed boundary conditions on the edges, $rank$ 0 does not send any data to (or receive from) the left and $rank$ 3 need not send any data to (or receive from) the right. This is best achieved by defining a 1D Cartesian topology with non-periodic boundary conditions - you should already have code to do this from the previous "message round a ring" exercise. You also need to ensure that the processes do not deadlock by all trying to do synchronous sends at the same time. Again, you should re-use the code from the same exercise.

The communications involves sending and receiving entire horizontal or vertical lines of data (depending on whether you are using C or Fortran). The process is as follows - it may be helpful to look at the appropriate decomposition in Figure 7.

For C:

- send the N array elements (`old[`M`][j]; j = 1,N`) to $rank + 1$
- receive N array elements from $rank - 1$ into (`old[0][j]; j = 1,N`)
- send the N array elements (`old[1][j]; j = 1,N`) to $rank - 1$
- receive N array elements from $rank + 1$ into (`old[`M`+1][j]; j = 1,N`)

For Fortran:

- send the M array elements (`old(i,`N`); i = 1,M`) to $rank + 1$
- receive M array elements from $rank - 1$ into (`old(i,0); i = 1,M`)
- send the M array elements (`old(i,1); i = 1,M`) to $rank - 1$
- receive M array elements from $rank + 1$ into (`old(i,`$N + 1$`); i = 1,M`)

Each of the two send-receive pairs is basically the same as a step of the ring exercise except that data is being sent in different directions (first clockwise then anti-clockwise). Remember that you can send and receive entire halos as single messages due to the way we have chosen to split the data amongst processes: you **should not** issue N separate calls, one for every element.

6.1 Testing the complete code

Again, run your program and compare the output images to the serial code. Are they exactly the same? Does the algorithm take the same number of steps until there are no changes?

How do the execution times compare to the serial code and the previous (incomplete) parallel code? How does the time scale with P?

Plot parallel scaling curves for a range of problem sizes. You may want to insert explicit timing calls into the code so you can exclude the IO overheads.

7 Further Work

Congratulations for getting this far! Here are a number of suggestions for extensions to your current parallel code, in no particular order.

7.1 Stopping criterion

Change your code so that it stops computation as soon as there are no changes, rather than doing a fixed number of steps.

7.2 Overlapping communication and calculation

One of the tutorial problems concerns overlapping communication and calculation, i.e. doing calculation that does not require the communicated halo data at the same time as the halo is being sent using non-blocking routines. Calculations involving the halos are done separately after the halos have arrived.

See if you can implement this in practice in your code. Does it improve performance?

7.3 Derived Data Types for IO

The initialisation stage proceeds in three phases:

1. initialise map on the controller
2. scatter the data from map to $smallmap$
3. copy $smallmap$ into old on all processes

and in the reverse order to gather data before file output. By defining a derived datatype that maps onto the internal region of old, excluding the halos, see if you can transfer data directly between map and old.

7.4 Alternative decomposition

The way that the slices were mapped onto processes was chosen to simplify the parallelisation. How would the IO and halo-swap routines need to be changed if you wanted to parallelise your code by decomposing over the other dimension (over j for C and i for Fortran). You should be able to make only minor changes to your existing code if you define appropriate derived datatypes for the halo data, although the IO will be more complicated. What is the effect on performance?