You should see the same output as the left-hand pane in Figure 6.

To show the largest n clusters, change the final parameter in the call to `percwrite` 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 flexibility to help debug the development of the parallel program.

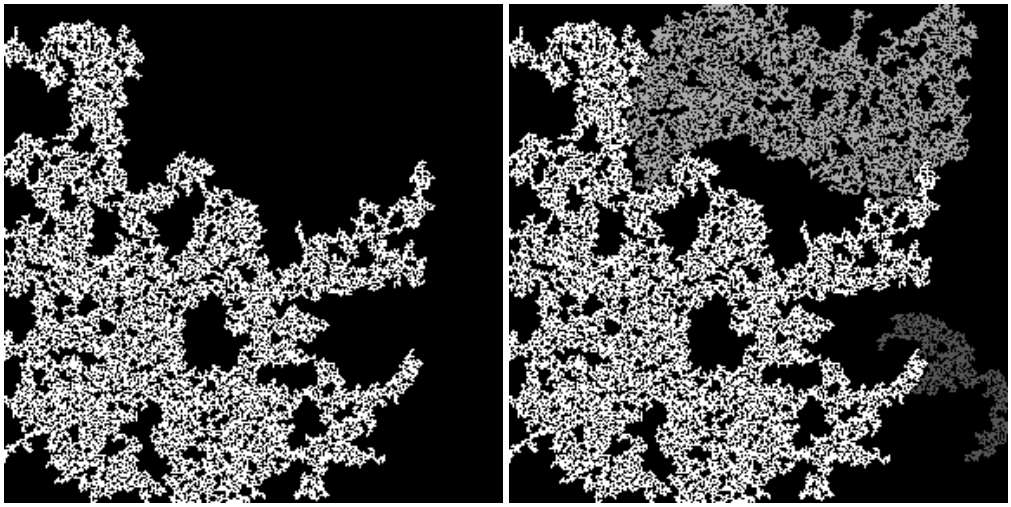

Figure 6: The default example showing the largest (left) and largest three (right) clusters. Note that the largest cluster does not percolate as it does not quite reach the top edge.

The code runs for a fixed number of steps (set to $16 \times L$). 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 (as implemented in the PS course), 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
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 master process. The map array is scattered to processes after initialisation, and gathered back before file IO. Note that this is not particularly efficient in terms of memory usage as we need enough space to store the whole map on a single process.

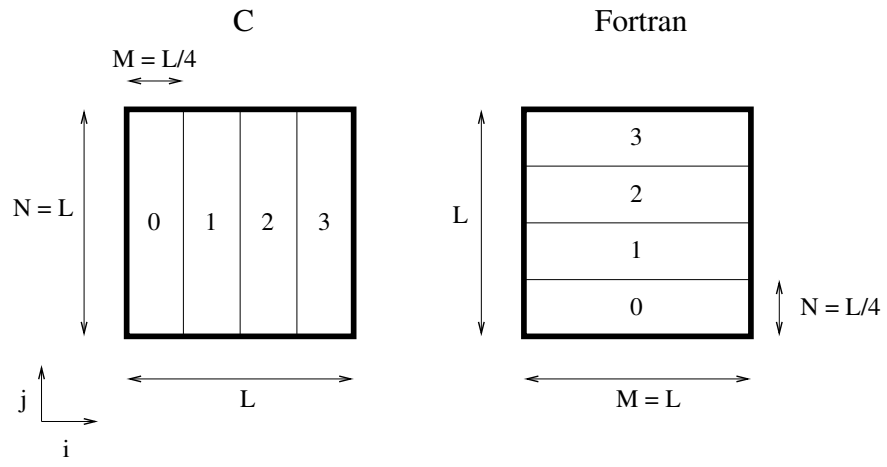

Figure 7: Decomposition strategies for 4 processes

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 master process only

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

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

Since we do the initialisation, check for percolation and perform file output in serial (i.e. on the master 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 Fortran 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.

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.