u1624423

PX425 Assignment 4 Report

Compilation and running in serial:

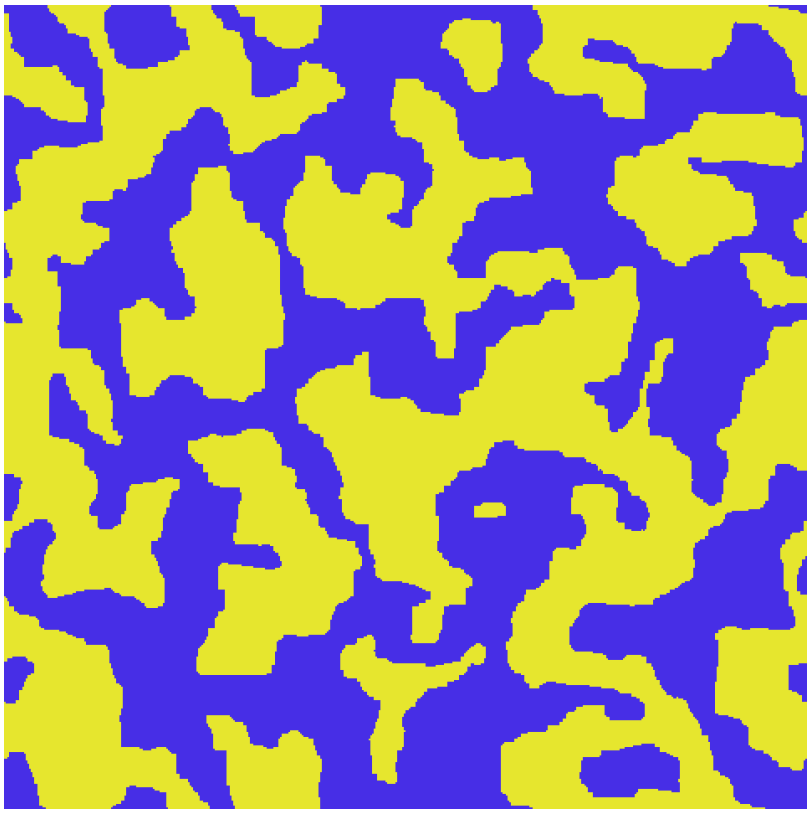
To control the number of MPI tasks running, within rfim.slurm the option "-n x" needs to be added to the srun command (where x is the number of MPI tasks desired). To allow this to be greater than 1, the value of "ntasks-per-node" needs to be set to something greater than or equal to x.

Running with 1 MPI task without making any changes to comms.c or rfim.c produces the following output to the terminal:

```
Size of each local processor grid :   480 x   480
Global magnetisation at cycle        0 :     0.001172
Global magnetisation at cycle      100 :     0.007569
Global magnetisation at cycle      200 :     0.020547
Global magnetisation at cycle      300 :     0.023186
Global magnetisation at cycle      400 :     0.023290
Global magnetisation at cycle      500 :     0.028247
Global magnetisation at cycle      600 :     0.033958
Global magnetisation at cycle      700 :     0.032517
Global magnetisation at cycle      800 :     0.033932
Global magnetisation at cycle      900 :     0.031745
Global magnetisation at cycle     1000 :     0.030156
End of simulation. Rank     0 accepted      3026981 moves.
```

(the time taken was about 16 seconds)

The final image produced:



u1624423

Getting MPI up and running:

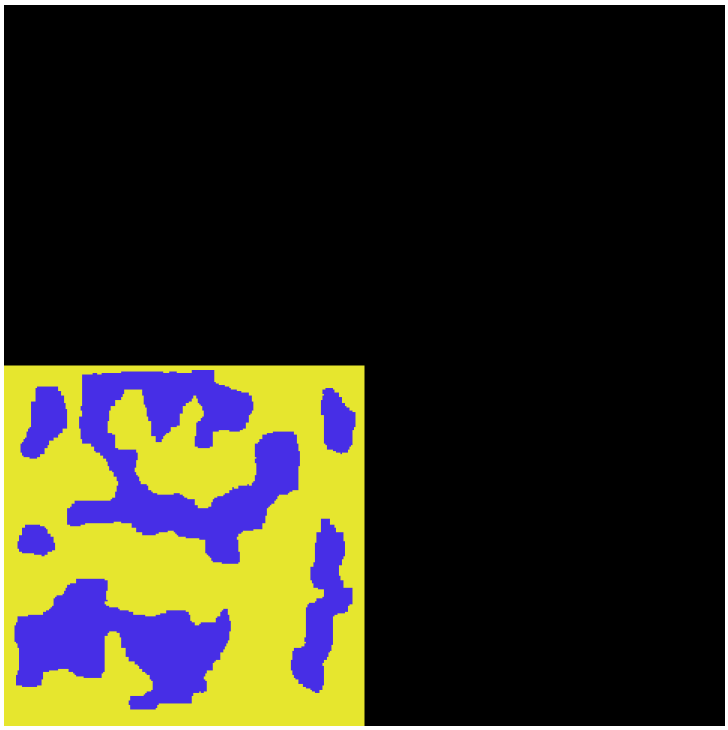
Standard initialisation implementation for MPI within comms_initialise, as well as getting the time, then standard finalisation within comms_finalise as well as getting the time at that point and calculating the difference.

Now the workload will be distributed equally between the x different MPI tasks, with the grid size of each task being:
process grid size = (total grid size)/ sqrt(x)

Output to terminal for 4 MPI tasks:

```
Size of each local processor grid :   240 x   240
Global magnetisation at cycle        0 :     0.003845
Global magnetisation at cycle      100 :     0.038646
Global magnetisation at cycle      200 :     0.052378
Global magnetisation at cycle      300 :     0.055720
Global magnetisation at cycle      400 :     0.060512
Global magnetisation at cycle      500 :     0.062144
Global magnetisation at cycle      600 :     0.066814
Global magnetisation at cycle      700 :     0.070573
Global magnetisation at cycle      800 :     0.071632
Global magnetisation at cycle      900 :     0.074913
End of simulation. Rank     1 accepted       775406 moves.
End of simulation. Rank     2 accepted       728889 moves.
Global magnetisation at cycle     1000 :     0.078012
End of simulation. Rank     0 accepted       753250 moves.
Total time elapsed since MPI initialised :      3.527363 s
End of simulation. Rank     3 accepted       749998 moves.
```

The final image produced:



As we can see, the time taken is much less than the original time taken. This is because ¼ of the work is being done (4 times at the same time). The time is in fact less than ¼ of the time taken for it to run originally. This extra time saved could be due to a difference in overhead of drawing the image as well as the fewer amount of “interactions” within the smaller region. To elaborate, since values depend on their neighbours, at the upper and right boundaries, there are no “interactions”, where there would usually be if the entire grid had been calculated.

The reason that only ¼ of the grid is shown is because all 4 tasks are being performed on the same region. This region is the first region, which is why the bottom left quarter is the one displayed. (i.e. the bottom left is the point (0,0) in the grid).

u1624423

Sharing the work – setting up a Cartesian topology:

The cartesian communicator is created with MPI_Cart_create(), the arguments are fairly self-explanatory and are already given except for the dimensions, which is just 2 since it is a 2d grid.

Setting the communicator grid rank is done using MPI_Comm_rank(), I created my_rank_cart to store this.

To get the coordinates of the current rank in the grid, I used MPI_Cart_coords(), and stored them in my_rank_coords.

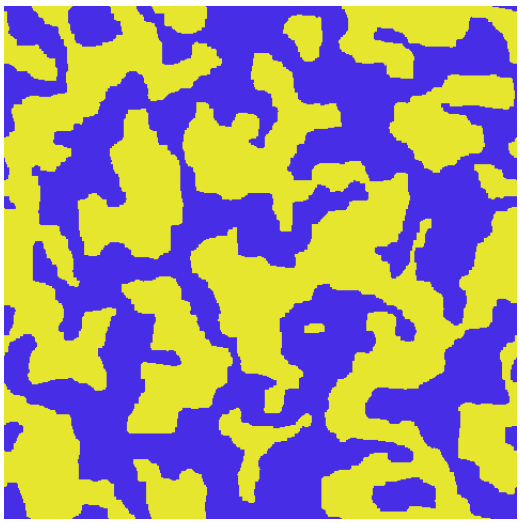
Finally to store the current rank's neighbours in a halo ring, two calls of MPI_Cart_shift are required (one shift left to right and one shift down to up). This is standard practice for creating a halo of neighbouring data.

The contents of the my_rank_coords and my_rank_neighbours arrays can easily be accessed and printed.

Terminal output for 1 MPI task:

```
my_rank: 0; my_rank_coords: (0,0)
my_rank: 0; my_rank_neighbours: left: 0, right=0, down=0, up=0
Size of each local processor grid :   480 x   480
Global magnetisation at cycle        0 :     0.001172
Global magnetisation at cycle      100 :     0.007569
Global magnetisation at cycle      200 :     0.020547
Global magnetisation at cycle      300 :     0.023186
Global magnetisation at cycle      400 :     0.023290
Global magnetisation at cycle      500 :     0.028247
Global magnetisation at cycle      600 :     0.033958
Global magnetisation at cycle      700 :     0.032517
Global magnetisation at cycle      800 :     0.033932
Global magnetisation at cycle      900 :     0.031745
Global magnetisation at cycle     1000 :     0.030156
End of simulation. Rank     0 accepted      3026981 moves.
```

Final image produced:

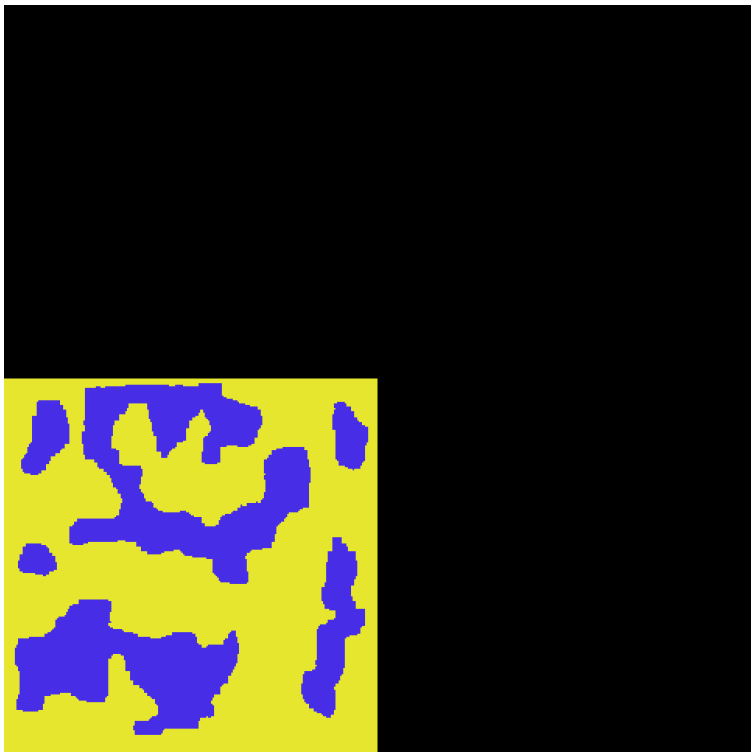


u1624423

Terminal output for 4 MPI tasks:

```
my_rank: 0; my_rank_coords: (0,0)
my_rank: 0; my_rank_neighbours: left: 2, right=2, down=1, up=1
Size of each local processor grid :   240 x   240
my_rank: 1; my_rank_coords: (0,1)
my_rank: 1; my_rank_neighbours: left: 3, right=3, down=0, up=0
my_rank: 2; my_rank_coords: (1,0)
my_rank: 2; my_rank_neighbours: left: 0, right=0, down=3, up=3
my_rank: 3; my_rank_coords: (1,1)
my_rank: 3; my_rank_neighbours: left: 1, right=1, down=2, up=2
Global magnetisation at cycle        0 :     0.003845
Global magnetisation at cycle      100 :     0.038646
Global magnetisation at cycle      200 :     0.052378
Global magnetisation at cycle      300 :     0.055720
Global magnetisation at cycle      400 :     0.060512
Global magnetisation at cycle      500 :     0.062144
Global magnetisation at cycle      600 :     0.066814
Global magnetisation at cycle      700 :     0.070573
Global magnetisation at cycle      800 :     0.071632
Global magnetisation at cycle      900 :     0.074913
End of simulation. Rank     2 accepted       728889 moves.
End of simulation. Rank     1 accepted       775406 moves.
End of simulation. Rank     3 accepted       749998 moves.
Global magnetisation at cycle     1000 :     0.078012
End of simulation. Rank     0 accepted       753250 moves.
Total time elapsed since MPI initialised :      3.461130 s
```

Final image produced:



u1624423

Terminal output for 9 MPI tasks:

```
my_rank: 1; my_rank_coords: (0,1)
my_rank: 1; my_rank_neighbours: left: 7, right=4, down=0, up=2
my_rank: 2; my_rank_coords: (0,2)
my_rank: 2; my_rank_neighbours: left: 8, right=5, down=1, up=0
my_rank: 3; my_rank_coords: (1,0)
my_rank: 3; my_rank_neighbours: left: 0, right=6, down=5, up=4
my_rank: 4; my_rank_coords: (1,1)
my_rank: 5; my_rank_coords: (1,2)
my_rank: 5; my_rank_neighbours: left: 2, right=8, down=4, up=3
my_rank: 6; my_rank_coords: (2,0)
my_rank: 6; my_rank_neighbours: left: 3, right=0, down=8, up=7
my_rank: 8; my_rank_coords: (2,2)
my_rank: 8; my_rank_neighbours: left: 5, right=2, down=7, up=6
my_rank: 0; my_rank_coords: (0,0)
my_rank: 0; my_rank_neighbours: left: 6, right=3, down=2, up=1
Size of each local processor grid :   160 x   160
my_rank: 7; my_rank_coords: (2,1)
my_rank: 7; my_rank_neighbours: left: 4, right=1, down=6, up=8
my_rank: 4; my_rank_neighbours: left: 1, right=7, down=3, up=5
Global magnetisation at cycle        0 :     0.001450
Global magnetisation at cycle      100 :     0.015816
Global magnetisation at cycle      200 :     0.021458
Global magnetisation at cycle      300 :     0.023194
Global magnetisation at cycle      400 :     0.024019
Global magnetisation at cycle      500 :     0.024297
Global magnetisation at cycle      600 :     0.023819
Global magnetisation at cycle      700 :     0.023993
Global magnetisation at cycle      800 :     0.024375
Global magnetisation at cycle      900 :     0.025035
End of simulation. Rank     6 accepted       319311 moves.
End of simulation. Rank     3 accepted       322690 moves.
End of simulation. Rank     2 accepted       278712 moves.
End of simulation. Rank     7 accepted       303911 moves.
End of simulation. Rank     5 accepted       289767 moves.
End of simulation. Rank     8 accepted       319958 moves.
End of simulation. Rank     1 accepted       304328 moves.
End of simulation. Rank     4 accepted       302669 moves.
Global magnetisation at cycle     1000 :     0.024748
End of simulation. Rank     0 accepted       310745 moves.
Total time elapsed since MPI initialised :      1.485225 s
```

u1624423

Final image Produced:

Unfortunately, we still have only 1 section of the grid.

Each dimension only uses 2 processors/tasks so
my_rank_neighbour[left]=my_rank_neighbour[right], and
my_rank_neighbour[down]=my_rank_neighbour[up].

Collecting data from all processors:

Within comms_get_global_grid(), we must use blocking MPI_Send(), MPI_Recv() (due to their order).

We send the starting point, grid_domain_start to remote_domain_start (which is to rank 0) a total of grid_domain_size number of times, we only need one MPI_Send to do this that many times (because it is blocking send it can be reused). So for each rank, we send the start point to rank 0. Similarly, for each iy (there is a grid_domain_size amount of these), we send grid_spin's data row by row to rank 0.

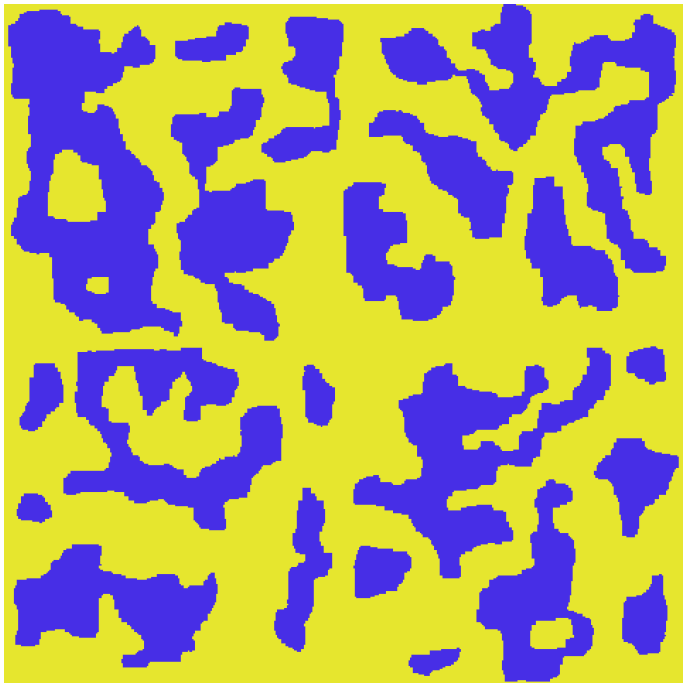
It is important that each set of send and receive have their own set of unique tags.

Within comms_get_global_mag, we use MPI_Reduce to reduce the values of each local_mag to a single value in global_mag (in rank 0).

Terminal output (without print statements from previous sections) on 4 MPI tasks:

```
Size of each local processor grid :   240 x   240
Global magnetisation at cycle        0 :     0.010668
Global magnetisation at cycle      100 :     0.115339
Global magnetisation at cycle      200 :     0.139106
Global magnetisation at cycle      300 :     0.150859
Global magnetisation at cycle      400 :     0.163142
Global magnetisation at cycle      500 :     0.171953
Global magnetisation at cycle      600 :     0.183437
Global magnetisation at cycle      700 :     0.194418
Global magnetisation at cycle      800 :     0.201849
Global magnetisation at cycle      900 :     0.208681
Global magnetisation at cycle     1000 :     0.216979
End of simulation. Rank     1 accepted       775406 moves.
End of simulation. Rank     3 accepted       749998 moves.
End of simulation. Rank     2 accepted       728889 moves.
End of simulation. Rank     0 accepted       753250 moves.
Total time elapsed since MPI initialised :      3.834653 s
```

Final image produced:



Unfortunately, it seems that at the boundaries the grid has not been correctly updated. Hence, we can see that, on the boundaries between the subdomains, there is only one colour.

Halo swapping:

Within comms_halo_swaps, there are 4 subroutines I have implemented to solve this issue. Firstly, the left boundary of grid_spin within the left neighbour's rank needs to go into the right boundary of grid_halo row by row in the current rank.

To do this, we first put the contents of the left hand boundary of grid_spin into sendbuf, then use nonblocking MPI_Isend, MPI_Irecv to transfer between tasks. (sendbuf -> recvbuf in receiving task) MPI_Wait must be called to ensure that the send and receive have been completed, then recvbuf is put into the right boundary of grid_halo row by row.

Similarly, this is done with the right boundary of grid_spin, leading to it being put into the left boundary of grid_halo to the corresponding task/grid section, row by row. Also for the lower boundary of grid_spin to the upper boundary of grid_halo column by column and the upper boundary of grid_spin to the lower boundary of grid halo column by column (to the corresponding grid sections).

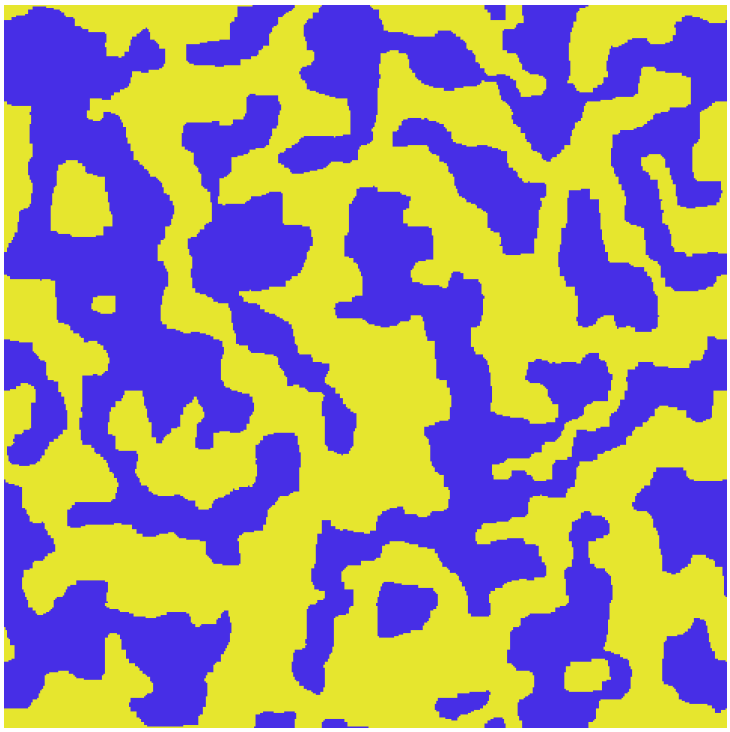
It is also worth mentioning that an MPI_Status and MPI_Request must be created for use in MPI_Isend, MPI_Irecv and the MPI_Wait. Looking into mpi.h, MPI_Request is just an integer, and MPI_Status is a struct containing 5 integers.

For each nonblocking send/recv we don't actually need to re-instantiate an MPI_Recv as long as they are completed before reusing them (i.e. via its use in MPI_Wait)

Terminal output on 4 tasks:

```
Size of each local processor grid :   240 x   240
Global magnetisation at cycle        0 :     0.004375
Global magnetisation at cycle      100 :     0.032161
Global magnetisation at cycle      200 :     0.037552
Global magnetisation at cycle      300 :     0.044262
Global magnetisation at cycle      400 :     0.042778
Global magnetisation at cycle      500 :     0.041953
Global magnetisation at cycle      600 :     0.046311
Global magnetisation at cycle      700 :     0.053759
Global magnetisation at cycle      800 :     0.057431
Global magnetisation at cycle      900 :     0.059236
Global magnetisation at cycle     1000 :     0.066493
End of simulation. Rank     3 accepted       806864 moves.
End of simulation. Rank     1 accepted       794655 moves.
End of simulation. Rank     2 accepted       747481 moves.
End of simulation. Rank     0 accepted       783269 moves.
Total time elapsed since MPI initialised :      3.666176 s
```

Final Image output:



We can see that the issue has been solved and that away from the boundaries there is little in terms of difference from the previous image. The main differences are at or around the boundaries.

Addendum to this section:

Currently the code for the first section of halo swapping is as shown below:

```
193   /* Send left hand boundary elements of grid_spin to my_rank_neighbours[left]
194       and receive from my_rank_neighbours[right] into the appropriate part
195       of grid_halo. Remember to use the appropriate communicator. */
196
197   //put left boundary of grid_spin into sendbuf
198   for(iy=0;iy<grid_domain_size;iy++) {
199     sendbuf[iy] = grid_spin[iy][0];
200   }
201
202   /* Insert MPI calls here to implement this swap. Use sendbuf and recvbuf */
203
204   //initialise mpi status,request for nonblocking Isend,Irecv
205   MPI_Status send_stat_0;
206   MPI_Status recv_stat_0;
207
208   MPI_Request send_req_0;
209   MPI_Request recv_req_0;
210
211   //actual sending and receiving (nonblocking)
212   MPI_Isend(sendbuf, grid_domain_size, MPI_INT, my_rank_neighbours[left], my_rank_neighbours[left]+600, MPI_COMM_WORLD, &send_req_0);
213   MPI_Irecv(recvbuf, grid_domain_size, MPI_INT, my_rank_neighbours[right], my_rank+600, MPI_COMM_WORLD, &recv_req_0);
214
215   //wait for these to be completed
216   MPI_Wait(&send_req_0, &send_stat_0);
217   MPI_Wait(&recv_req_0, &recv_stat_0);
218
219   //put elements of recvbuf into grid_halo as right hand boundary
220   for(iy=0;iy<grid_domain_size;iy++) {
221     grid_halo[right][iy] = recvbuf[iy];
222   }
```

Although MPI_Requests and MPI_Statuses are safe to reuse once MPI_Wait has been called and we can save resources (as tiny of a difference as it may be) by doing so instead of recreating them for all 4 sections of the halo swapping, there is a more efficient way to save on resources.

In fact, the entirety of the contents between lines 204 and 214 can be performed in a single line of code, calling MPI_Sendrecv, which performs the same operations.

```
225   //does the same as commented out section above, status can be reused
226   MPI_Sendrecv(sendbuf,grid_domain_size, MPI_INT, my_rank_neighbours[left], my_rank_neighbours[left]+600,recvbuf, grid_domain_size, MPI_INT, my_rank_neighbours[right], my_rank+600, MPI_COMM_WORLD, &status);
227
```

MPI_Sendrecv isn't like a blocking MPI_Send followed by MPI_Recv, it is like an MPI_Isend followed by MPI_Irecv, followed by the two MPI_Waits, so our nonblocking sending and receiving is achieved (which is essential to avoid deadlocking).

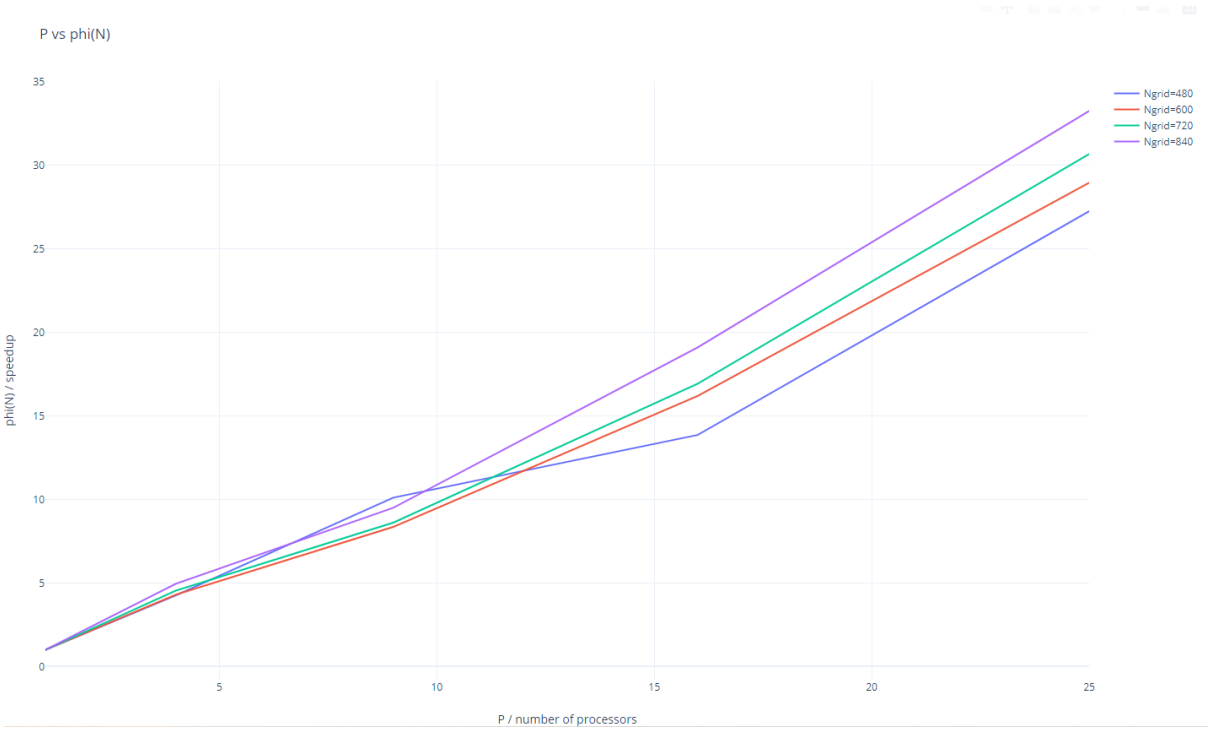
The code has been changed to use MPI_Sendrecv for all 4 sections. Of course, it is important that the result is the same, and looking at the images produced, we can see that it is indeed the same.

Timing wise, one would expect the time for completion to be negligible. To test the timings, it is best to compare the longest tasks as any time improvement/deterioration in the time would be most apparent then. As expected, the time difference is tiny, on the order of 10^-3s over the mean of 5 runs. In some cases, the performance was better and in some it was worse, it varied with the number of processors, but this is almost guaranteed to be entirely due to the inconsistency of orac. Running the same code multiple times itself has a range of greater than 10^-3s.

u1624423

Timings (recalculated after previous addendum):

P	Ngrid	Time 1	Time 2	Time 3	Time 4	Time 5	Mean Time	Task Grid Size	φ(N)
1	480	14.5145	14.39653	14.41133	14.42836	14.41045	14.4322352	480	1
4	480	3.396233	3.401181	3.379339	3.377912	3.407911	3.3925152	240	4.254142
9	480	1.447872	1.418052	1.405782	1.461427	1.406149	1.4278564	160	10.10763
16	480	1.053215	1.058852	1.108192	0.937368	1.049579	1.0414412	120	13.85795
25	480	0.533403	0.532194	0.527472	0.529505	0.52525	0.5295648	96	27.25302
1	600	23.79866	24.09404	23.72175	23.99608	23.85059	23.8922258	600	1
4	600	5.730299	5.502603	5.334159	5.467216	5.723915	5.5516384	300	4.303636
9	600	2.920286	2.752294	2.767123	2.864178	2.997323	2.8602408	200	8.353223
16	600	1.415488	1.516783	1.522451	1.38413	1.539097	1.4755898	150	16.19165
25	600	0.826747	0.815257	0.812063	0.836812	0.834317	0.8250392	120	28.9589
1	720	37.09955	36.22767	36.67323	36.84009	36.98554	36.7652148	720	1
4	720	8.197228	8.07324	7.97626	7.966777	8.258649	8.0944308	360	4.542038
9	720	4.263325	4.194163	4.263002	4.166484	4.467005	4.2707958	240	8.608515
16	720	2.187208	2.157373	2.159623	2.164343	2.18784	2.1712774	180	16.93253
25	720	1.238804	1.185282	1.192729	1.185697	1.190665	1.1986354	144	30.67255
1	840	55.80826	55.1879	55.13415	55.33241	55.49812	55.392169	840	1
4	840	11.15306	11.09824	11.26072	11.20071	11.24435	11.1914174	420	4.949522
9	840	6.064352	5.828801	5.944316	5.277779	6.039292	5.830908	280	9.49975
16	840	2.880853	2.968562	2.675491	3.057231	2.916018	2.899631	210	19.10318
25	840	1.693323	1.693977	1.641241	1.651674	1.649721	1.6659872	168	33.24886



(NB – the timings for 1 processor were not being printed as the print statement within comms_finalise only prints when p>1, so it has been changed to p>=1 to print the timings for when it is being run in serial.)

As expected, the more processors, the faster the completion is. Also, as Ngrid increases, so does the speedup for each P.

There is, however, a strange, unexpected occurrence. On multiple processors, say x, one would think that ideally, the speedup would itself be x, but realistically, you would expect it to be lower than x due to various overheads. Strangely though, the speedup exceeds the number of processors. This seems to increase the higher the resolution.

The improvement over increasing the number of processors is simple to explain, the more processors there are, the smaller the workload each does and so each processor does less work, resulting in a shorter completion time. Amdahl's law proposes that the higher the proportion of code that is parallelised, the higher the increase in performance gained by increasing the number of processors for highly parallelised code (up to a certain point).

It shouldn't be possible theoretically for the same task being parallelised to have a speedup faster than the number of processors being used. Upon inspecting the images produced, they are different to that of when they are run in serial. This could mean that my implementation is incorrect and so when in parallel, each processor somehow ends up doing less work than it's meant to. Alternatively, it could be that there is somehow less work to do when using more processors due to the boundaries.

I think the most reasonable assumption to make is that when running in serial, there is an overhead that occurs only in serial, perhaps due to the MPI calls that might waste time.