PX425 Assignment 4

Introduction

This assignment presents you with a computer program which is structured to allow distributed memory parallelism (domain decomposition) via MPI. All of the MPI calls are missing, and your task is to add them.

This program implements a simulation of the 2D random field Ising model. Non-physicists don't need to be too concerned about the details, however this model consists of a square grid of 'spins' σ_i, each of which can be oriented in two different ways, represented by σ_i taking values of -1 or $+1$. In addition each spin is subject to a random magnetic field h_i which is drawn from a uniform distribution between $\pm\zeta$. The energy of the system is

$$H(\sigma) = -J \sum_{\langle ij \rangle} \sigma_i, \sigma_j - \sum_i \sigma_i h_i \qquad (1)$$

where the sum is taken over all spins in the grid i, and j runs over the four nearest neighbour spins (left, right, down, up) of spin i. The simulation consists of Monte Carlo (MC) moves in which a randomly selected spin i is given a new random σ_i, and this new situation is accepted with a probability which depends on the temperature of the system via a Boltzmann distribution:

$$P_{\text{old}\to\text{new}} = \min(e^{-\beta(E_{\text{new}} - E_{\text{old}})}, 1) \qquad (2)$$

Compilation and running in serial

First, visit the assignment 4 section of the moodle page and download `rfim.zip`. This task is intended to be run mainly on `orac`, where for this assignment we have a special "reservation" by the name `PX425` to ensure your jobs run quickly. so you will need to copy over the to your space on `orac`. Remember that the SCRTP home and storage drives are available on `orac`, under `/desktop/home/` and `/desktop/storage` respectively, but these are not visible from the compute nodes, so please make a folder on the `orac` `home` drives for compiling and running this assignment. Uncompress this file using the `unzip` command in a terminal window.

As before I have provided you with a makefile to ease compilation of the code. This one is set up for `orac`: it might work elsewhere if you have the right tools available, but I can't offer tech support for that.

On `orac` there will not necessarily be any compilers loaded when you first log in, so load up `intel`, `impi` and `libpng` modules as discussed in the workshop (check with `module list` that they are loaded). Compile the code there using the makefile (`make`), being sure Examine the script `rfim.slurm` and ensure you are requesting exactly one processor (`procs=1`) for no more than one minute. Submit the script to the batch system, referring back to workshop 3 material if you are not comfortable with using the SLURM commands such as `sbatch` and `squeue`. If you get the message "sbatch: error: Batch job submission failed: Access denied to requested reservation" then it might mean you have signed up to Orac in the name of a different research group (eg for a project), and have not automatically been given access to the PX425 reservation. If so, let the module leader know by email so we can get it sorted.

The code as provided should complete in less than one minute. It performs a simulation on a `Ngrid`×`Ngrid` grid of spins (with periodic boundary conditions) for 1000 MC cycles. Each cycle consists of `Ngrid`^{2} MC moves. `Ngrid` should be set to 480 initially. You will notice that the program dumps a `.png` image file of the simulation every 100 cycles. Here each grid point is represented by a block, the colour of which corresponds to σ. You can open these images by copying them back to the SCRTP system and viewing them in a web browser, or in any image viewer. You should see an evolution of the original random field of spins into larger single-colour domains.

Currently, a single processor is performing all MC moves. Your goal is to implement a domain decomposition such that the grid is divided equally over MPI tasks.

Getting MPI up and running

Examine `comms.c` in your editor. Putting all the communications routines in one place is a common way of structuring MPI codes. All of the interaction with MPI is handled by the set of functions/subroutines within this source file. The current file is a 'stub' i.e. a set of routines which implement dummy parallelism, such that an MPI code can correctly run in serial on machines which have no MPI libraries available. We want this to run on many processors, so you will need to modify each of these routines to create an effective MPI code.

Start by editing `comms_initialise`. Remove the two dummy lines and replace with appropriate calls to initialise MPI, store the rank of the current processor in `my_rank` and the size of the communicator in `p`. You should also call `MPI_Wtime()` which has no arguments and returns a double precision real number. Store this in `t1`.

Next edit `comms_finalise`. Use `MPI_Wtime()` again to set t2, *before* the time elapsed in calculated and printed. Then shut down MPI with the appropriate call. Also remove the extra `return` statement at the top of this routine.

Compile the resulting program and submit it to the `PX425` reservation on `orac` requesting four (`procs=4`) processors. The program should now run substantially faster, and report that each processor has a *local* grid size (the grid on each processor) one quarter the size of the total grid. However all is not well. By examining the `.png` images produced you'll notice that only one quarter of the grid is printed. In fact all processors are simulating the same quarter of the grid (replicating rather than sharing work), and there is no communication between them. Ooops.

Save the final image for your report and briefly explain its context.

Sharing the work - setting up a Cartesian topology

The reason all processors are working on the same grid square is that each processor relies on the array `my_rank_coords` which has two elements, the x and y coordinates of the grid section to be simulated. Currently this is set to $(0, 0)$ on all MPI tasks. Your next task is to modify the routine `comms_processor_map` to set this array appropriately. I recommend creating a Cartesian communicator as discussed in lectures. Some variables are pre-defined at the head of this routine to assist you in calling the appropriate routine.

Once this communicator is in place, you should also (in the same routine) make appropriate MPI calls to identify the four neighbouring MPI tasks of the current task, within the newly defined Cartesian topology. Remember that the overall (global) grid has periodic boundary conditions. You should store the neighbours of the current MPI task in the four elements of `my_rank_neighbours` in the order left, right, down and up.

Constants allowing you to reference elements in these arrays by name (rather than index) are defined in `grid.c`. For example you can reference the left-hand neighbour of the current MPI task with `my_rank_neighbours[left]`. You should add appropriate print statements (to be removed later) which result in each MPI task printing the contents of both these arrays in a well-formatted manner.

Compile the resulting code, and run on 1, 4, and 9 processors. Include the output of these print statements in your report, and satisfy yourself that each processor is working on a different part of the grid and has the correct neighbour information. In your report, briefly describe and explain the what MPI calls you have had to add to set up the appropriate communicators.

Collecting data from all processors

You'll see that we still only have a fraction of the output image. The next step is to modify `comms_get_global_grid`. This routine is intended to collect the array `grid_spin`, i.e. the local per-processor grid, from each MPI task and assemble these into a single array `global_grid_spin` on MPI task 0 only. This will allow the whole image to be output. Appropriate loops are already in place to help you. After filling out its own part of the global grid, rank zero loops over all other processors and first receives the two element integer array `remote_domain_start`. Grid data is then received from the other processor one line at a time, and placed into the appropriate part of `global_grid_spin`. The other MPI tasks first send `grid_domain_start` to rank 0, and then, one line at a time, they send `grid_spin` to rank 0. You should insert the required calls to `MPI_Send` and `MPI_Recv` to achieve this. Make sure the messages from each rank cannot be confused.

You should also modify `comms_get_global_mag`. The magnetisation of the system is just the spin (σ_i) averaged over all points on the grid. Each MPI task already computes the magnetisation within its own part of the grid as `local_mag`. You should use an appropriate MPI collective communication routine to average each value of `local_mag` into `global_mag` on MPI rank zero, who prints out this value in the main program.

Run the resulting program on four processors (`procs=4`). You should now get a complete image. However you will note that all boundaries between subdomains, are the same colour. Save a copy of this image for your report and explain its context.

Halo swapping

The reason for this is simple. Each MC update of a spin requires information from the four neighbouring spins. Hence spins on the boundary of each processor's sub-grid cannot be updated correctly without some communication between neighbouring MPI tasks. These are being updated using a 'halo' of grid points surrounding each sub-grid (the array `grid_halo`) which is currently set to $\sigma_i = 1$ (yellow) everywhere. This prevents the correct evolution of the system across boundaries.

To fix this, you will need to complete the function/subroutine `comms_halo_swaps`. For each direction in turn (left, right, down, up) the current processor should populate an array `sendbuf` with the line of data from `grid_spin` corresponding to the boundary elements in that direction. This should be sent to the neighbouring MPI task in that direction, identified from the array `my_rank_neighbours` you populated earlier. Boundary information from neighbours should be received into `recvbuf` and then copied into the appropriate part of `grid_halo`. Note that `grid_halo` is a two-dimensional array, and you must be careful to fill the appropriate section of it in each case.

Once you've completed the routine, run it on four processors on `orac` via the `PX425` reservation again. You should now see coloured regions spanning the boundaries between processors in the resulting images. Save a copy of this image for your report. Your code should now be a fully functional domain decomposed parallel simulation! Briefly discuss what you have had to do to achieve this in your report.

Timings and report

As well as the information collected from your program about processor grid coordinates and neighbours, your report should include timings from your program. For these benchmark tests there is no need to print images, so you should comment or remove lines 217 to 223 in `rfim.c`.

I want you to generate timings on $P = 1$, 4, 9, 16 and 25 processors on `orac` (in the `PX425` reservation) using grid sizes `Ngrid` = 480, 600, 720, and 840. For each grid size, you should compute the speedup ψ as a function of P using

$$\psi\,(N) = \frac{T(1)}{T(P)} \tag{3}$$

where $T(P)$ is the time taken to run with P MPI tasks, and $T(1)$ is the time taken with 1 MPI task, such that $\psi(1) = 1.0$ exactly.

Your report should comment on these timings. What are the trends with `Ngrid` and P? Do you notice anything unexpected about the timings (you might not). If so, can you explain this?

Deadline and Submission

You have two weeks in which to complete this assignment. You should submit the final version of your file `comms.c`, your report, and the final output `.png` image from a simulation with `Ngrid=480` on $P = 9$ processors. Rename this to `image.png`. Compress these into a zip file with the same name as your SCRTP user code, for example like this

```
zip phuxyz.zip comms.c image.png report.pdf
```

where you use your SCRTP username instead of `phuxyz`. This should compress all versions of your code plus your report into a single .zip file. Then upload the zip file to the Moodle page for Assignment 4.