Geometric decomposition of a serial scientific code

1 Introduction

In this practical we will be looking at parallelising an existing serial code using the geometric decomposition pattern (i.e. allocating different data elements to different processes) as the algorithm strategy, and SPMD as the implementation strategy. The existing code is in both ***serial.c*** and ***serial.F90***, depending which language you prefer, and solves Laplace's equation for diffusion. This is a partial differential equation, where the value at a specific point depends on the value of neighbouring points, which they themselves depend upon their neighbours for their values. In this case we are concerned with two dimensions, so the value at each point is determined by an average of the values held on the left, right, up and down neighbouring points. You can think of the problem as a pipe of length ***n*** with some concentration of pollutant at one end and a different concentration at the other. We want to find out the concentration of pollution all the way along the pipe and to do this are using Laplace's equation which solves the system to an equilibrium.

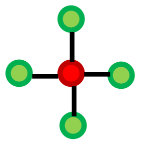
Figure 1

The actual solving follows an iterative method, where we start off with our prescribed concentrations at either end of the pipe (boundary conditions) and at each iteration the current values for the solution are updated to bring them closer to the final answer. At each iteration we calculate the relative residual, which basically tells us how far away from the answer the solution currently is, and common practice is to keep iterating until the residual is small enough to match a predetermined termination accuracy. The code you have been provided with uses a Jacobi iterative method, which is a simple (and inefficient) solver but the convergence (progress towards the final solution) is predictable, identical regardless of the number of processes, often linear and certainly good enough for our purposes (we will look at integrating different solvers in a later practical.)

2 Serial code

*Serial code, submission scripts and makefiles for C and Fortran versions are provided on Learn. You might find it easiest to download the **practical-one.zip** archive from Learn onto your local machine and then **scp** this onto Cirrus, just ask if you have any questions.*

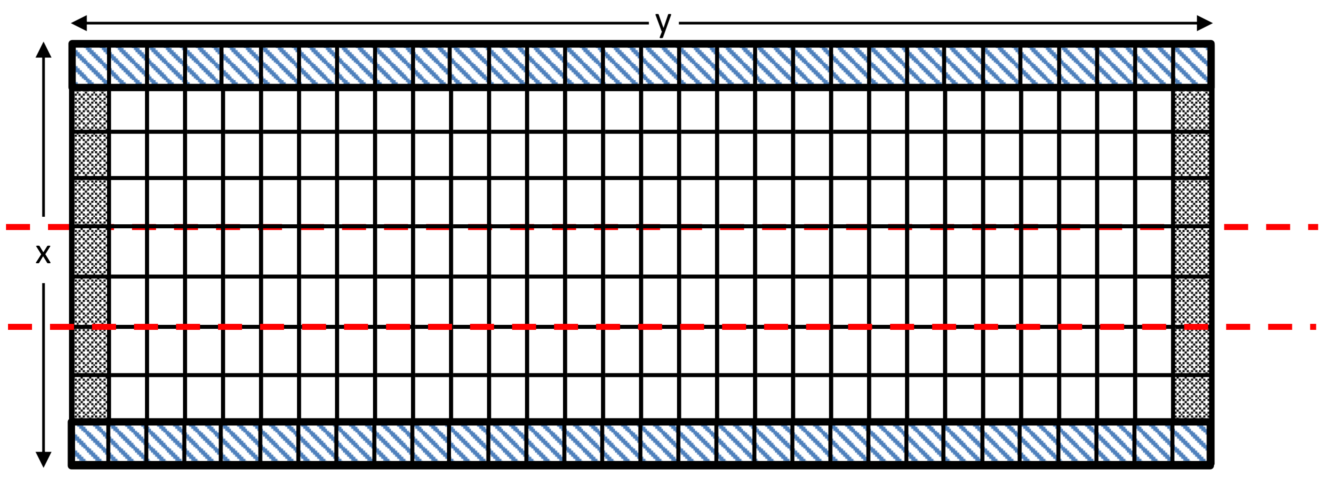

Figure 2: Domain in the original serial code, where the left and right hand boundary values are shaded

.

The first step is to compile the serial code (via the makefile, issue *make* in the directory) and run the executable. On Cirrus you will need to load the *mpt* and *intel-compilers-19* modules before building the code. A submission script is provided which will submit this on one Cirrus compute node running on 1 core (which you can modify to run your parallelised version in section 3) or alternatively just execute on the Cirrus login node/Informatics DICE node. On the first line of output (iteration 0) you will see that the relative residual is 1, which (should be) as inaccurate as the solution gets, as the iterations progress you will see the residual decrease and the smaller it is the more accurate the solution. The code will terminate once a residual threshold has been met or a maximum number of iterations performed. Figure 2 illustrates the domain and the left and right most values are the initial conditions and are hard coded to high pollution and low pollution (with the empty boxes *unknowns* and their values will be calculated as the code progresses.)

If you look in the submission script you will see there are four command line arguments provided to the code, the first argument is the pipe size in X (height), the second is the pipe size in Y (length), the third is the termination relative residual (3e-3 by default) and the fourth argument is the maximum number of iterations (0 means no maximum.) The code will terminate either when the relative residual reaches the target (third command line argument) or the maximum number of iterations has been reached, whichever is soonest.

If submitting to the back-end of Cirrus then you will need to provide your user's account ID to the submission command line, for instance ***sbatch --account=m230c_myuser subpractical.srun*** where ***m23oc_myuser*** is replaced with your own Cirrus account ID.

Hint: If it is taking some time to get your job through the queue, for instance if Cirrus is busy, then you can append ***–qos=short*** to you submission command (e.g. ***sbatch --account=m23oc_myuser –qos=short subpractical.srun***) which will submit to the short partition. Here jobs are limited to 2 nodes and 20 minutes runtime maximum, but this should be sufficient for the practical.

3 Geometric decomposition

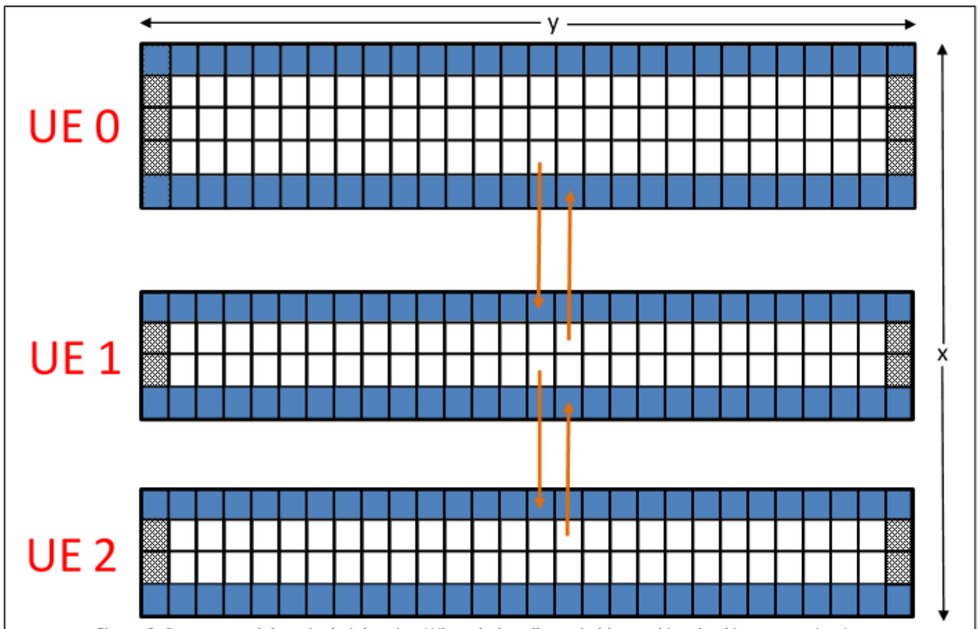

Figure 3: Decomposed domain, height wise. Where halo cells are in blue and involved in communications

In figure 2 you can see we have popped in some red dashed lines, this is where the domain can be split up across a number of UEs and figure 3 illustrates this in more detail. To keep it simple, whilst the problem is 2D, we are only going to split the problem up in 1D, in the X dimension. The reason we have chosen this X dimension (so effectively we split vertically i.e. the height of the pipe) rather than the Y dimension, along the pipe, is to keep the parallelism a bit simpler (as in the X dimension data is contiguous, so you don't need to pack values into buffers and un-pack them.) Therefore, you can think of splitting this problem up by cutting up the pipe into rows and assigning a number of rows of the pipe to a specific process. By looking at the code we can see that the calculation of each point depends on each neighbouring point (see figure 1), this is called the stencil and in this case the stencil is of size 1 (because we are concerned with only the closest neighbour in each direction.) Therefore, the majority of computation will be local, apart from the points on the boundary of the local data, which will require neighbouring values. The arrows of figure 3 illustrate data communication that occurs as part of a halo swap (which we will talk about in the geometric decomposition lecture), where the blue cells are halos. Note that in this exercise the top halo of UE0 and the bottom halo of UE2 remain unchanged, these are set to zero and can remain those values.

To keep figure 3 simple, we only show one arrow per communication (pairs of UE exchanging data, hence there are two arrows illustrated for each exchange.) But crucially we don't want to be issuing lots of small communications containing a single value (as many small communications is much more expensive than a single larger communication). Therefore, instead the technique of halo swapping is used where, in addition to its local data, each process also maintains a halo of neighbouring data (of stencil width) and these values are swapped at predefined intervals. The calculation then proceeds using these values until the next halo swap, in our code a halo swap is performed for every ***k*** iteration.

One of the nice properties of this problem is that when we parallelise it, the convergence of the algorithm should not change. Therefore, an easy way to check whether your code is still working correctly is that the overall number of iterations, termination relative residual and progress (i.e. relative residual at n iterations) should be the same regardless of the number of processors.

We will be using MPI for this practical, if you are not so familiar or a bit rusty with this then the course materials of a recent ARCHER course at https://github.com/EPCCed/archer2-MPI-2020-05-14 are a good reference. A good reference for the MPI API can be found at https://www.open-mpi.org/doc/v3.0/

Your task is to parallelise, in 1D, the serial code via geometric decomposition of the data using MPI for the halo swapping and other communications. This will comprise of a number of steps:

1. Determining the number of solution elements per UE, it will probably be useful to get the overall number of UEs (number of MPI processes) and then divide ***nx*** (the global size in X) by this number. You should also consider the situation where this doesn't divide evenly, for instance in figure 3 where we have 7 rows decomposed over three UEs. One way of handling this uneven decomposition is by adding an extra element to certain UEs - first divide ***nx*** by the number of UEs, store this as an integer ***local_nx***. Then have something like ***if (myrank < nx - local_nx * size) local_nx++;*** where ***myrank*** is the current UE rank, ***nx*** is the global size in X and ***size*** the overall number of UEs.
2. Allocate only the data size you need on a specific UEs. Remember that each UE will need two extra rows, the first row and last row being halos. You can set these to hold zero values.
3. Extend the two residue calculation routines to compute global sums by computing local sums for each UE (as the code currently does) and then doing an MPI allreduce with the ***MPI_SUM*** operator to find the global sum, before squaring rooting the result.
4. Implement the halo swap. For ease, the serial routine runs from 1 to ***nx*** rows and you probably want to keep this the same, in the zero'th row either the last non-halo row values from the up neighbour (rank-1) - or unchanged if this is UE 0 - and in the ***nx plus 1*** row either the first non-halo row values from the down neighbour or unchanged if this is the last UE. As a first step do the actual halo swapping at the start of an iteration (before calculating the

norm of the solution) and use the ***MPI_Sendrecv*** call for each halo swap communication (this is a blocking communication, we will look at optimising it in a minute!)

5. Compile (via *make*) and run it on Cirrus! You will need to edit the submission script, we suggest running over 1 node but all 36 cores.

Remember: The results from the parallel version should be identical to that of the serial code. So a good check is to run it and ensure that the relative residual displayed periodically matches the serial code. If not, then you have made an error somewhere!

Remember: You can test your parallelisation quickly on the Cirrus login node by executing ***mpirun -np n ./exec*** *(where* ***n*** *is the number of processes and* ***exec*** *is the name of the executable.) Whilst the code is set up to have four command line arguments, If you run without these arguments then a default (small problem size) is automatically chosen by the code. This can be useful to testing on the login node/DICE machine.*

4 Optimising the halo swap

The parallelisation that we have used is naïve, specifically the use of ***MPI_Sendrecv*** and whilst this is fairly simple to get going with, we should be able to speed up the execution of the code fairly considerably and easily by doing some simple optimisation steps.

1. Make a copy of your existing parallel code so you don't loose it
2. In this copy, replace the use of ***MPI_Sendrecv*** with that of non-blocking MPI communications. You will need to define an array of ***MPI_Request*** handles of size four (the communications that a UE will perform with its neighbouring UEs) and initially set each value to ***MPI_REQUEST_NULL***. For each halo swap replace the ***MPI_Sendrecv*** call with an ***Isend*** and ***Irecv*** call.
3. Once you have replaced the blocking call with non-blocking calls for both halo swaps, issue an ***MPI_Waitall*** for all the communication handles immediately after the halo swapping.
4. Issue *make* again to build this new version, edit the submission script to execute your new executable and run it. The code will provide an overall runtime on termination, how does using non-blocking communications impact the overall runtime?

5 Advanced exercise – further overlapping of communications

The non-blocking communications we have used up until this point is still fairly simple. For instance, immediately after issuing the halo swapping communication then the UE waits for these to complete. Also the reduction each iteration is still a blocking one. Based on this there is some further optimisation we can do to overlap the communications with computation to ameliorate the overhead of communication.

1. Make a copy of your non-blocking version and use this copy as a basis for the actions in this section.
2. First let's focus on the ***MPI_Allreduce*** issued on every iteration, replace this with a non-blocking version (***MPI_Iallreduce***) and think very carefully where you need to place the appropriate ***MPI_Wait***. Of course, you could place the wait immediately after the call, but with some code movement you can move this later on and perform computation whilst the reduction is in progress.
3. After halo swapping communication the computation starts from the first non-halo row and proceeds to the last non-halo row. Instead modify the loop to run from the second non-halo

row and proceed to the second to last non-halo row. Move the halo swapping ***MPI_Waitall*** call to after the loops and then duplicate the computation for the first non-halo row and last. The idea being that whilst halo swapping communication is in progress other, non dependent, calculations can be performed.

4. Build this new version of the code via *make* and run it on Cirrus. How does the runtime compare to the other versions?