Domain Decomposition

Computational Fluid Dynamics (CFD)

Contents

1. Aims

This exercise takes an example from one of the most common applications of HPC resources: Fluid Dynamics. We will look at how a simple fluid dynamics problem can be run on a system like ARCHER and how varying the number of processes it runs on and the problem size affect the performance of the code. This will require a set of simulations to be run and performance metrics to be recorded and plotted on a graph.

The CFD program differs from the more straightforward task farm in that the problem requires more than source-worker-sink communications. Here the workers are in regular communication throughout the calculation. This exercise aims to introduce:

- Grids
- Communications – Halos
- Performance metrics

2. Fluid Dynamics

Fluid Dynamics is the study of the mechanics of fluid flow, liquids and gases in motion. This encompasses aero- and hydrodynamics. It has wide ranging applications, from theoretical studies of flow to engineering problems such as vessel and structure design, and plays an important role in weather modelling. Simulating and solving fluid dynamic problems requires large computational resources.

Fluid dynamics is an example of continuous system which can be described by Partial Differential Equations. For a computer to simulate these systems, the equations must be discretised onto a grid. If this grid is regular, then a finite difference approach can be used. Using this method means that the value at any point in the grid is updated using some combination of the neighbouring points.

Discretisation is the process of approximating a continuous (i.e. infinite-dimensional) problem by a finite-dimensional problem suitable for a computer. This is often accomplished by putting the calculations into a grid or similar construct.

2.1. The problem

In this exercise the finite difference approach is used to determine the flow pattern of a fluid in a cavity. For simplicity, the liquid is assumed to have zero viscosity which implies that there can be no vortices (i.e. no whirlpools) in the flow. The cavity is a square box with an inlet on one side and an outlet on another as shown below in figure 1.

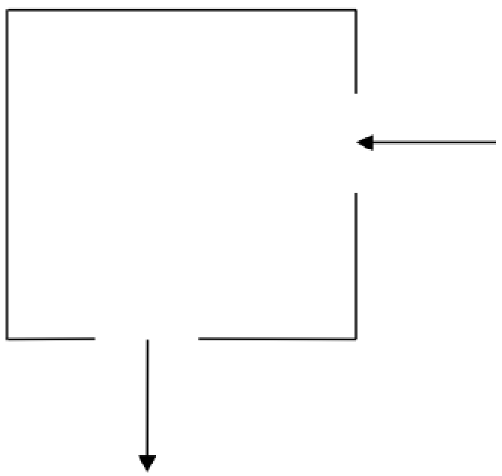

Figure 1 - The cavity

3. Mathematical formulation and the Jacobi algorithm solution

In two dimensions it is easiest to work with the stream function Ψ (see below for how this relates to the fluid velocity). For zero viscosity Ψ satisfies the following equation:

$$\nabla^2\Psi = \frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2} = 0$$

The finite difference version of this equation is:

$$\Psi_{i-1,j} + \Psi_{i+1,j} + \Psi_{i,j-1} + \Psi_{i,j+1} - 4\Psi_{i,j} = 0$$

With the boundary values fixed, the stream function can be calculated for each point in the grid by averaging the value at that point with its four nearest neighbours. The process continues until the algorithm converges on a solution which stays unchanged by the averaging process. This simple approach to solving a PDE is called the Jacobi Algorithm.

In order to obtain the flow pattern of the fluid in the cavity we want to compute the velocity field ũ. The x and y components of ũ are related to the stream function by:

$$u_x = \frac{\partial\Psi}{\partial y} = \frac{1}{2}(\Psi_{i,j+1} - \Psi_{i,j-1})$$

$$u_y = -\frac{\partial\Psi}{\partial x} = -\frac{1}{2}(\Psi_{i+1,j} - \Psi_{i-1,j})$$

This means that the velocity of the fluid at each grid point can also be calculated from the surrounding grid points.

4. Pseudocode for the Jacobi algorithm

The outline of the algorithm for calculating the velocities is as follows:

```
set the boundary values for Ψ
while (convergence == FALSE) do
    for each interior grid point do
        update Ψ by averaging with its 4 nearest neighbours
    end do
    check for convergence
end do

for each interior grid point do
    calculate u_x
    calculate u_y
end do
```

For simplicity, here we simply run the calculation for a fixed number of iterations; a real simulation would continue until some chosen accuracy was achieved.

4.1. Breaking up the problem to solve it in parallel

The calculation of the velocity of the fluid as it flows through the cavity proceeds in two stages:

- Calculate the stream function Ψ
- Use this to calculate the *x* and *y* components of the velocity

Both of these stages involve calculating the value at each grid point by combining it with the value of its four nearest neighbours. Thus the same amount of work is involved in calculating each grid point, making it ideal for the regular **domain decomposition** approach. Figure 2 shows how a two-dimensional grid can be broken up into smaller grids to be handled by individual processes.

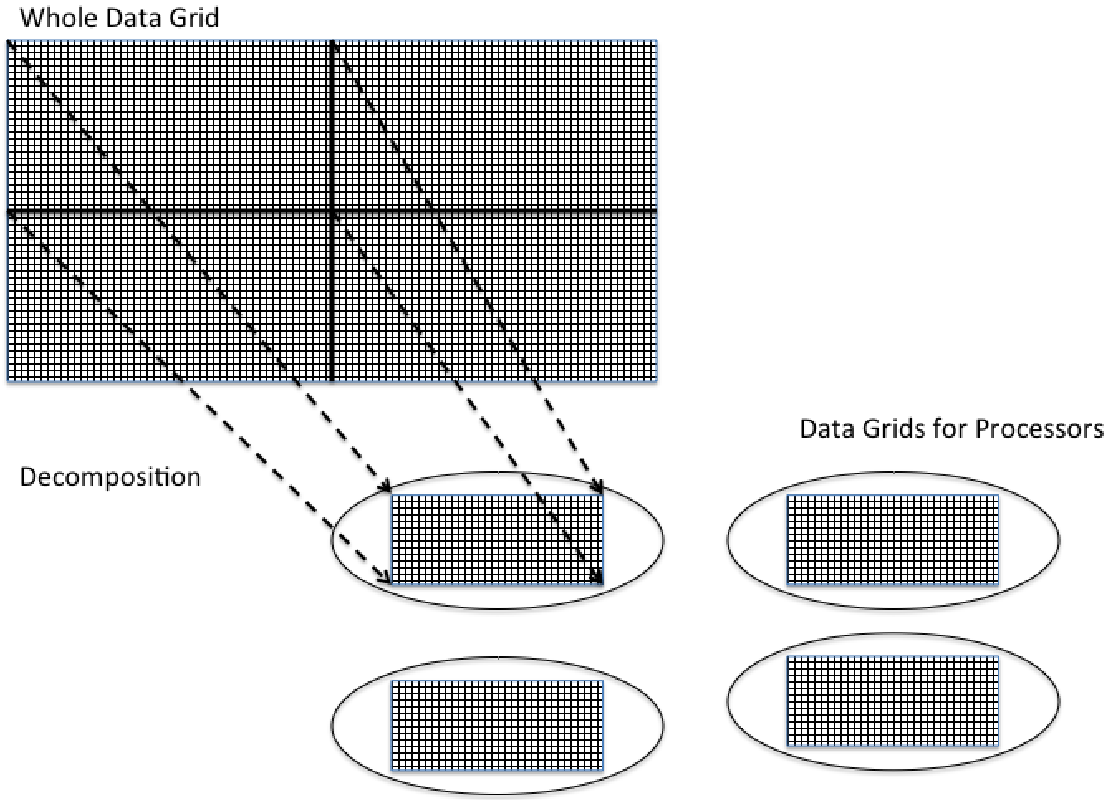

Figure 2 - Breaking up the problem through domain decomposition

This approach can be generalised to include cases where slices or irregular subsections of grids are sent to individual processes and the results are collated at the end of a calculation cycle.

4.2. Communicating through halo swaps

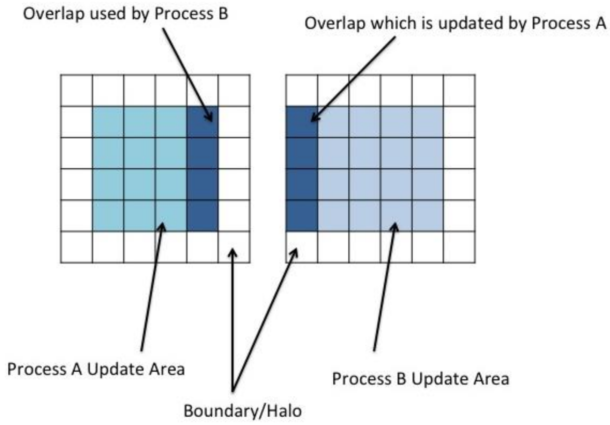

Figure 3 - Halo: Process A and Process B

Splitting the grid into smaller grids introduces a problem computing values at points on the edges of subgrids, as this requires the values of neighbouring points, which are part of a neighbouring subgrid whose values are being computed by a different process. This means that some form of communication between processes is needed.

One way to tackle this problem is to add a boundary layer to each edge of each subgrid that adjoins another subgrid. These boundary layers are not updated by the local process updating the interior points of the subgrid, but instead by the process working on the neighbouring subgrid. The boundary layer in the neighbouring subgrid is in turn updated by the local process. These boundary layers are generally known as *halos*. An example of this is show in Figure 3.

In order to keep the halos up to date, a halo swap must be carried out. When an element in process B which adjoins the boundary layer with process A is updated and process A has been updating, the halo must be swapped to ensure process B uses accurate data. This means that a communication between processes must take place in order to swap the boundary data. This halo swap introduces communications which, if the grid is split into too many processes or the size of data transfers is very large, can begin to dominate the runtime instead of actual processing work. Part of this exercise is to look at how the number of processes affects the runtime for given problem sizes, and evaluate what this means for speed up and efficiency.

4.3. One-dimensional domain decomposition for CFD example

In the parallel versions of the code provided for this exercise we have only decomposed the problem in one dimension, namely the y-dimension (Fortran) or x-dimension (C). This means that the problem is sliced up into a series of rectangular strips. Although for a real problem the domain would probably be split up in both dimensions as in Figure 2, splitting across a single dimension makes programming and understanding the code significantly easier. Each process only needs to communicate with a maximum of two neighbours, swapping halo data up and down (or left and right).

5. Exercises

5.1. Compilation

Use *wget* to copy the file *cfd.tar.gz* from the ARCHER web page for the course to your personal directory on */work/* on ARCHER, as for the previous exercises. Now unpack the file and compile the code provided for the CFD example as described below. We have provided serial versions (directories C-SER / F-SER) as well as parallel versions using the domain decomposition approach and MPI for halo swap communications as described above (directories C-MPI / F-MPI). After compilation, an executable file called **cfd** will have been created in each case.

You are free to choose to work with either the C or Fortran versions – below we use Fortran for illustration, but the procedure is the same for the C versions.

```
Input:
        tar -zxvf cfd.tar.gz
Output:
        cfd/
        cfd/F-MPI/
        cfd/F-SER/
        cfd/F-SER/boundary.f90
        cfd/F-SER/cfd.f90
        cfd/F-SER/cfd.pbs
        ...
```

```
Input:
        cd cfd/F-SER
        make
Output:
        ftn -g -c boundary.f90
        ftn -g -c jacobi.f90
        ftn -g -c cfdio.f90
        ftn -g -c cfd.f90
        ftn -g -o cfd boundary.o cfd.o cfdio.o jacobi.o
```

```
Input:
        cd cfd/F-MPI
        make
Output:
        ftn -g -c boundary.f90
        ftn -g -c jacobi.f90
        ftn -g -c cfdio.f90
        ftn -g -c cfd.f90
        ftn -g -o cfd boundary.o cfd.o cfdio.o jacobi.o
```

5.2. Run the program

As long as you do not run for a long time, e.g. for less than a minute, you can execute the serial executable on the login node directly from the command line as follows:

```
        ./cfd <scale> <numiter>
```

Where you should replace *<scale>* and *<numiter>* by integers with the following meaning:

- **<scale>**: a scale factor that is used to set the size of the grid (see below)
- **<numiter>**: how many iterations to run the Jacobi algorithm for.

The minimum problem size (scale factor = 1) is taken as a 32 x 32 grid. The actual problem size can be chosen by scaling this basic size, for example with a scale factor of 4 then it will use a 128 x 128 grid. After the executable has finished running the output on screen should look something like this (depending on the exact parameters):

```
./cfd 4 5000
 Scale factor =    4, iterations =    5000
 Irrotational flow
 Running CFD on  128 x  128 grid in serial

 Starting main loop ...

 completed iteration         1000
 completed iteration         2000
 completed iteration         3000
 completed iteration         4000
 completed iteration         5000

 ... finished

 After      5000 iterations, error is  0.1872E-03
 Time for   5000 iterations was 0.4869     seconds
 Each individual iteration took  0.9737E-04 seconds

 Writing output file ...
  ... finished
 CFD completed
```

The code also produces some graphical output in the form of a gnuplot file *cfd.plt*. Provided you have logged in to ARCHER with X11 forwarding turned on (e.g. *ssh -X* from the command line) you can view this as follows:

```
        gnuplot -persist cfd.plt
```

which should produce a picture similar to Figure 4.

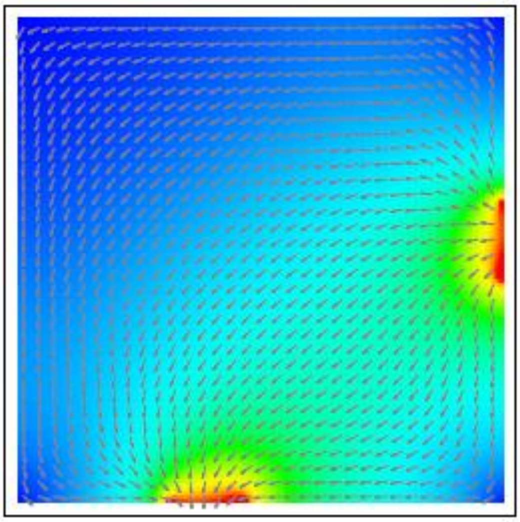

Figure 4 - Output image

If the fluid is flowing down the right-hand edge then along the bottom, rather than through the middle of the cavity, then this is an indication that the Jacobi algorithm has not yet converged. Convergence requires more iterations on larger problem sizes.

5.3. Running in parallel

Use emacs or your preferred editor to look at the *cfd.pbs* batch script inside the *F-MPI* directory:

```
#!/bin/bash --login

#PBS -l select=1
#PBS -l walltime=00:05:00
#PBS -A y14
#PBS -N cfd

#Change to directory that the job was submitted from
cd $PBS_O_WORKDIR

aprun -n 4 ./cfd 4 5000
```

The argument to aprun has the following meaning:

- **-n 4**: run the code on 4 processes;

The arguments to cfd are the same as in the serial case, with the scale factor now setting the size of the overall grid (which will be decomposed into smaller subgrids). Varying the number of processes and scale factor allows us to investigate Amdahl's and Gustafson's laws. The number of iterations is not particularly important as we are interested in the time per iteration. You can increase the number of iterations to ensure that the code does not run too fast on large numbers of processes, or decrease it so it is not too slow for large problem sizes. Three things to note:

- For more than 24 processes (the size of a single node) then you will need to select multiple nodes, e.g for 96 processes then specify: #PBS -l select=4
- The code assumes the problem size decomposes exactly onto the process grid. If this is not the case (e.g. scale factor = 2 with 7 processes, since 7 is not a divisor of 64) it will complain and exit.
- Again if the output picture looks strange then you may not have used a sufficient number of iterations to converge to the solution. This is not a problem in terms of the performance figures, but it is worth running with more iterations just to check that the code is functioning correctly

Once you have run the job via the PBS batch system, the output file should look something like this (depending on the exact parameters):

```
Scale factor =    4, iterations =    5000
Irrotational flow
Running CFD on  128 x  128 grid using 4 process(es)

Starting main loop ...

completed iteration         1000
completed iteration         2000
completed iteration         3000
completed iteration         4000
completed iteration         5000

... finished

After      5000 iterations, error is  0.1872E-03
Time for   5000 iterations was  0.1296     seconds
Each individual iteration took  0.2592E-04 seconds

Writing output file ...
 ... finished
CFD completed
```

5.4. Performance Evaluation

The next part of this exercise will be to determine what the best configuration for a group of problems sizes in the CFD code would be. This will be worked out using two measures: speed-up and efficiency.

5.4.1. Speedup

The speedup of a parallel code is how much faster the parallel version runs compared to a non-parallel version. Taking the time to run the code on 1 process is T_1 and to run the code on P processes is T_P , the speed-up S is found by:

$$S = \frac{T_1}{T_p}$$

5.4.2. Efficiency

Efficiency is how the well the resources (available processing power in this case) are being used. This can be thought of as the speed-up (or slow-down) per process. Efficiency E can be defined as:

$$E = \frac{S}{P} = \frac{T_1}{PT_p}$$

where $E = 1.0$ means 100% efficiency, i.e. perfect scaling.

5.4.3. Doing the work

The two main evaluation points:

- How do the speed-up and efficiency vary as the number of processes is increased?
- Does this change as the problem size is varied?

To investigate the speed-up and parallel efficiency the code should be run using the same problem size but with varying numbers of processes. Calculate the speed-up and efficiency (tables are provided overleaf for this) and plot a graph of the speed-up against the number of processes. Is there any apparent pattern, e.g. does it follow Amdahl's law?

Now choose a different problem size and repeat the exercise. To increase the problem size, increase the scale factor; to decrease the size, decrease the scale factor. For example, setting **scale factor** = 2 will give a problem size of 64x64; **scale factor** = 6 gives a size of 192x192.

What is the effect of problem size the parallel scaling of the code?

Note that, for large numbers of processes, you may have to increase the number of iterations so that the code runs for a reasonable length of time -- it is difficult to interpret timings quantitatively if a program runs too quickly, e.g. for much less than a second. The time per iteration, however, should be independent of the number of iterations as the internal timing excludes all the IO overheads.

You can place multiple consecutive aprun commands in a single PBS script, i.e. you can perform runs on many different process counts using a single qsub command.

1) Problem size (scalefactor) -________ Iterations = _______

No. of processes	**Time per iteration**	**Speedup**	**Efficiency**
1			
2			
4			
8			
16			
32			
64			
128			

2) Problem size (scalefactor) -________ Iterations = _______

No. of processes	**Time per iteration**	**Speedup**	**Efficiency**
1			
2			
4			
8			
16			
32			
64			
128			

3) Problem size (scalefactor) -________ Iterations = _______

No. of processes	Time per iteration	Speedup	Efficiency
1			
2			
4			
8			
16			
32			
64			
128			

4) Problem size (scalefactor) -________ Iterations = _______

No. of processes	Time per iteration	Speedup	Efficiency
1			
2			
4			
8			
16			
32			
64			
128			

5) Problem size (scalefactor) -________ Iterations = _______

No. of processes	Time per iteration	Speedup	Efficiency
1			
2			
4			
8			
16			
32			
64			
128			

6) Problem size (scalefactor) -________ Iterations = _______

No. of processes	Time per iteration	Speedup	Efficiency
1			
2			
4			
8			
16			
32			
64			
128			

6. Compiler Investigation

We will now investigate how different compilers and options affect performance.

6.1. Changing compilers on ARCHER

On ARCHER, the C compiler is always called *cc* and the Fortran compiler is always called *ftn*. However, what compiler this actually points to is determined by what *module* you have loaded. For example, to switch from the default (Cray) compiler to the Intel compiler.

```
module switch PrgEnv-cray PrgEnv-intel
make clean
make
```

Here *make clean* ensures all compiled code is removed so the new code will be built with the new compiler. This also removes output files velocity.dat, colourmap.dat and cfd.plt, so to keep these you should rename them. The GNU compiler module is called *PrgEnv-gnu*.

6.2. Exercises

Here are a number of suggestions:

- By default, the code is built with the *-g* debugging option. Edit the Makefile to remove this and recompile - what is the effect on performance?
- What is the difference between the performance of the code using the three different compilers (Cray, Intel and GNU) with no compiler options?
- Is the performance of the C and Fortran versions significantly different?
- It is not really fair to compare compiler performance using default optimisation options: one compiler may simply have higher default settings than another. Using the options suggested in ``Useful compiler options'', http://www.archer.ac.uk/documentation/user-guide/development.php, compare the best performance you can get with each compiler.

- The code can actually simulate fluids with a finite viscosity which gives more realistic simulations which include features such as whirlpools. You can pass a third parameter to the program, the Reynolds Number *Re*, which here is effectively the input velocity of the fluid. For example, *aprun -n 4 ./cfd 4 5000 1.8* sets *Re = 1.8* (note that the code becomes unstable for values of *Re > 3.7*). This increases the computation required so the code will be slower, although the actual amount of computation is independent of the value of *Re*.

  Investigate the parallel scaling behaviour of the code for some particular value of *Re*. Is it any different from the previous (zero viscosity) case?

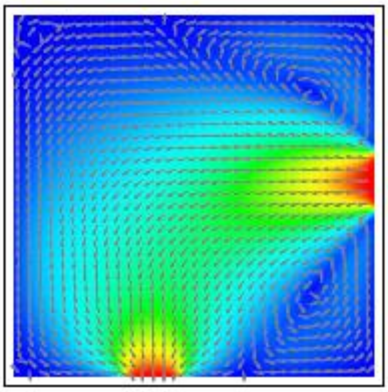

Figure 5 - Simulation with *Re = 1.8*