|epcc|

Computational Fluid Dynamics

February 27, 2018

1 Introduction and Aims

This exercise takes an example from one of the most common applications of HPC resources: Fluid Dynamics.

This exercise aims to introduce:

- Compiling and running a Fortran application
- Derived types

2 Fluid Dynamics

Fluid Dynamics is the study of the mechanics of fluid flow, liquids and gases in motion. This can encompass aero- and hydro- dynamics. It has wide ranging applications from vessel and structure design to weather and traffic 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.

2.2 A bit of Maths

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:

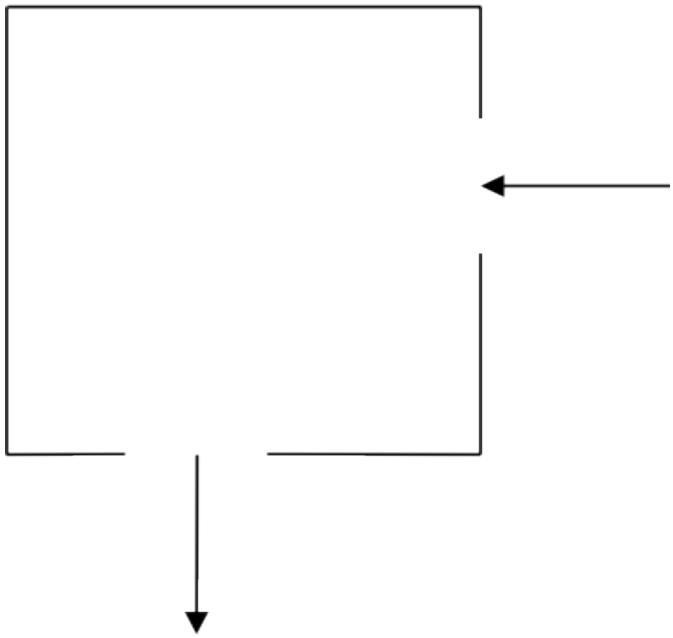

Figure 1: The Cavity

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

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 $\tilde{u}$. The x and y components of $\tilde{u}$ 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.

2.3 An Algorithm

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

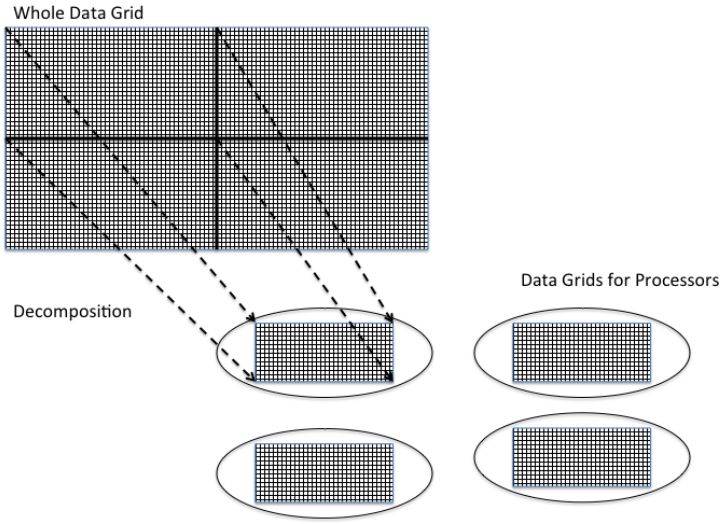

Figure 2: Breaking Up the Big Problem

```
Set the boundary values for Ψ and ũ
while (convergence= FALSE) do
   for each interior grid point do
      update value of 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.

2.4 Broken Up

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 to calculate each grid point, making it ideal for the regular domain decomposition approach. Figure 2 shows how a two dimension grid can be broken up into smaller grids for individual processes. This is usually known as **Decomposition**.

This process can hold for multiple other cases, where slices or sections of grids are sent to individual processes and the results can be collated at the end of a calculation cycle.

2.5 Halos

Splitting up the big grid into smaller grids introduces the need for interprocess communications. Looking at how each point is being calculated, how does the system deal with points on the edge of a grid? The

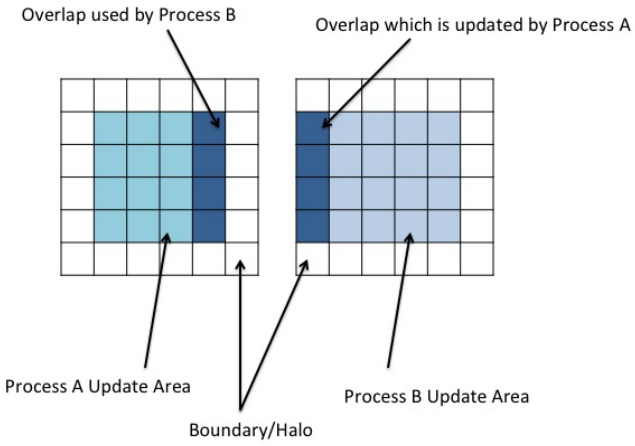

Figure 3: Halo: Process A and Process B

data a process needs has been shipped off to a different process.

To counter this issue, each grid for the different processes has to have a boundary layer on its adjoining sides. This layer is not updated by the local process: it is updated by another process which in turn will have a boundary updated by the local process. These 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 that if the grid is split into too many processes or the size of data transfers is very large, the communications can begin to dominate the runtime over actual processing work. Part of this exercise is to look at how the number of processes affects the run-time for given problem sizes and evaluate what this means for speed up and efficiency.

2.6 One-dimensional Domain decomposition for CFD example

For simplicity, we only decompose the problem in one dimension: the y-dimension. 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 the coding significantly easier. Each process only needs to communicate with a maximum of two neighbours, swapping halo data up and down.

3 Exercises

3.1 Compilation

Use `wget` to copy the file `cfd.tar.gz` from the ARCHER web pages ARCHER. Now unpack the file and compile the Fortran CFD code as follows: after compilation, an executable file will have been created called **cfd**.

```
guestXX@archer:~> tar -zxvf cfd.tar.gz
cfd/
cfd/Fortran
cfd/Fortran/cfd.plt
cfd/Fortran/Makefile
cfd/Fortran/cfd.f90

guestXX@archer:~> cd cfd/Fortran

guestXX@archer:~/cfd> make
ftn -g -c cfd.f90
ftn -g -o cfd cfd.o
```

3.2 Run the program

The program can be run like this: `./cfd 4 5000`. The arguments to the program have the following meaning:

- **4**: Use a scale factor of 4
- **5000**: Run for 5000 iterations.

The minimum problem size (scale factor = 1) is taken as a 32×32 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 128×128 grid. You can increase the number of iterations to ensure that the code does not run too fast, or decrease it so it is not too slow for large problem sizes. You can visualise results when the program has finished by using `gnuplot` as follows:

```
guestXX@archer:~> gnuplot -persist cfd.plt
```

which should produce a picture similar to Figure 4.

If the output picture looks strange (i.e the fluid is flowing down the right-hand edge then along the bottom, rather than through the middle of the cavity) 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.

When you run the program you will get output like this:

```
Scale factor =   4, number of iterations =   5000
Running CFD on  128 x  128 grid

Starting main loop ...

completed iteration  1000
completed iteration  2000
completed iteration  3000
completed iteration  4000
```

```
completed iteration   5000

... finished

Time for    5000 iterations was   0.1763      seconds
Each individual iteration took   0.3527E-04 seconds

Writing output file ...
 ... finished

CFD completed
```

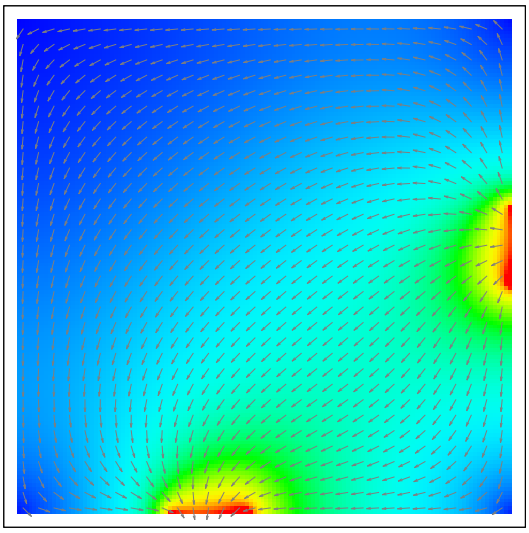

Figure 4: Output Image

4 Compiler Investigation

We will use the CFD example to investigate how different compilers and options affect performance.

4.1 Changing compilers on ARCHER

On ARCHER, 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.

```
guestXX@archer:~> module switch PrgEnv-cray PrgEnv-intel
guestXX@archer:~> make clean
guestXX@archer:~> make
```

Here. `make clean` ensures that all compiled code is removed which means that the new code will be built with the new compiler. The GNU compiler module is called `PrgEnv-gnu`.

4.2 Exercise 1

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?
- It is not really fair to compare compiler performance using default options: one compiler may simply have higher default settings than another. Using the option suggested in "Useful compiler options" at http://www.archer.ac.uk/documentation/user-guide/development.php, compare the best performance you can get with each compiler.

4.3 Exercise 2

- Replace the current data structures used for psi and psitmp with derived type(s). Does this have a performance impact on the code?
- The current solution does not split up the simulation domain into multiple blocks. Extend/alter your data type to enable it to store a portion of the grid, with a different type storing the whole grid (avoid replicating the data).
- Construct procedures that will copy data to halos on neighbouring blocks.
- Can you construct an operator that will perform the computation of the new psi value for a given block? Does this affect performance?
- Extend the program so that it can operate on data types that are integer or real.