|epcc|

THE UNIVERSITY OF EDINBURGH

Exercise: Fluids and Halos

April 21, 2014

1 Introduction and Aims

This exercise takes an example from one of the most common applications of HPC resources, Fluid Dynamics. This will look at how a fluid dynamics problem can be run on a system like ARCHER and how varying the number of processors and the problem size affect the performance of the code. This will require a set of configurations to be thought of, run and plotted as a graph.

This differs from the more straightforward task farm in that the problem requires more than source-worker-sink communications during the calculation.

This exercise aims to introduce:

- Grids
- Communications - Halos

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. It has many aspects and simulation and solving fluid dynamic problems requires large computational resources.

Fluid dynamics are 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. This method states that the value at any point in the grid is some combination of the neighbouring points.

Discretization is the process of approximating an infinite dimensional problem by a finite dimensional problem suitable for a computer. 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.

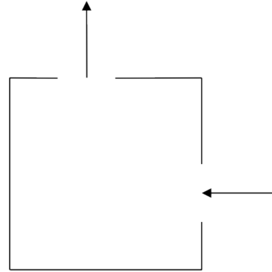

Figure 1: The Cavity

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:

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

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 where 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.

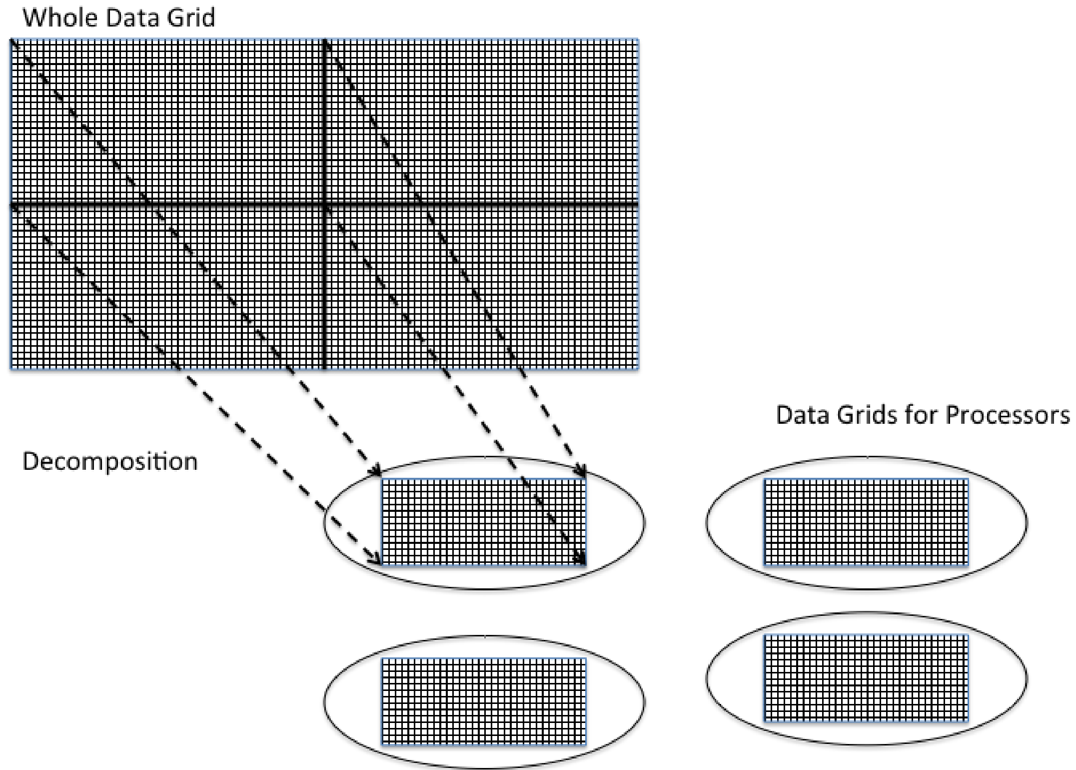

Figure 2: Breaking Up the Big Problem

2.3 An Algorithm

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

```
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
```

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 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 processors. This is usually known as **Decomposition**.

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

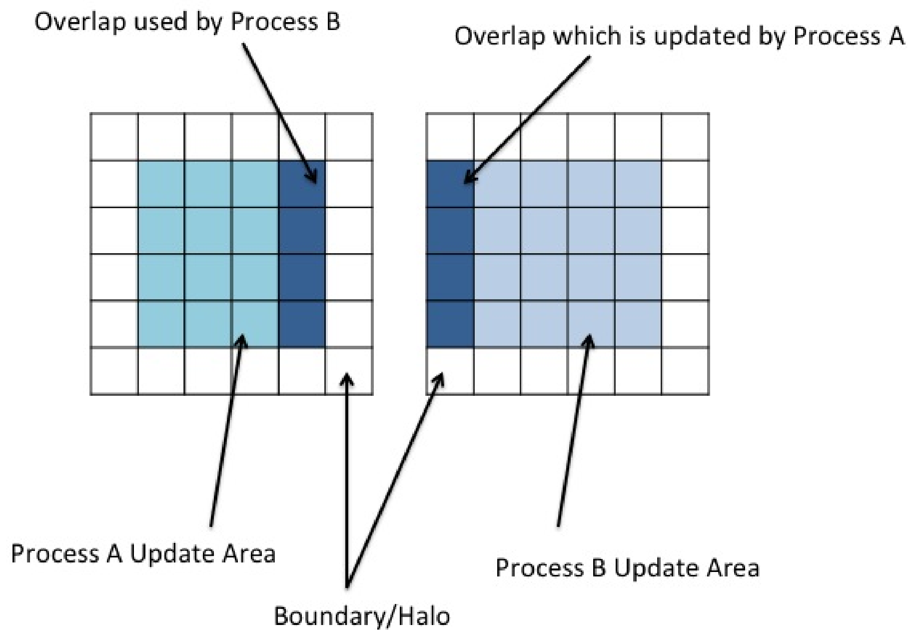

Figure 3: Halo: Process A and Process B

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

To counter this issue, each grid for the different processors has to have a boundary layer on its adjoining sides. This layer is not writable by the local process and is updated by another process which in turn will have a boundary updated by the local process. These layers are generally known as a halo. 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 processors affects the run-time for given problem sizes and evaluate what this means for speed up and efficiency.

3 Tasks

3.1 Compilation

A guest account ID and password should have been provided for ARCHER. In the event this is not the case, please contact a demonstrator.

3.1.1 Log into ARCHER fronted nodes

Open a command line Terminal and enter the following command, entering password when prompted:

```
local$ ssh guestXX@login.archer.ac.uk
Password:
```

3.1.2 Switch To Work

The exercise files are contained in an archive hosted on the EPCC web server. There is a command line utility called wget (Web GET) that allows you to download files from the web into your directories on ARCHER.

Change directory to make the /work filesystem on ARCHER. /work is a high performance parallel file system that can be accessed by both the frontend and compute nodes. All jobs on ARCHER should be run from the /work filesystem. Jobs run from /home will run much slower and cost more. Use the cd (Change Directory) command to move to user space on /work:

```
eslogin001:~> cd /work/y14/y14/guestXX
```

Use wget (on ARCHER) to get the exercise files archive from the EPCC webserver:

```
eslogin001:~> wget tinyurl.com/archer230414/cfd_ex.tar.gz
--2013-08-22 10:04:32--
http://www2.epcc.ed.ac.uk/~acollis/archer230414/cfd_ex.tar.gz
Resolving www.epcc.ed.ac.uk... 129.215.120.3
Connecting to www.epcc.ed.ac.uk|129.215.120.3|:80... connected.
HTTP request sent, awaiting response... 302 Moved Temporarily
Location:
http://www2.epcc.ed.ac.uk/~acollis/archer230414/cfd_ex.tar.gz [following]
```

```
--2013-08-22 10:04:32--
http://www2.epcc.ed.ac.uk/~acollis/archer230414/cfd_ex.tar.gz
Resolving www2.epcc.ed.ac.uk... 129.215.62.177
Connecting to www2.epcc.ed.ac.uk|129.215.62.177|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4503 (4.4K) [application/x-gzip]
Saving to: 'cfd_ex.tar.gz.1'

100%[=============================>] 4,503        --.-K/s    in 0s

2013-08-22 10:04:32 (376 MB/s) 'cfd_ex.tar.gz' saved [4503/4503]
```

Unpack the archive using the tar command:

```
eslogin001:~>  tar -xzvf cfd_ex.tar.gz
reggrid/
reggrid/cfd.f
reggrid/Makefile
```

To compile the exercise code, switch to the reggrid directory and issue the make command. Note exact command output may not match exactly.

```
eslogin001:~> cd reggrid
eslogin001:~> make
ftn -g -c cfd.f
ftn -g -o cfd cfd.o
```

After compilation a executable file will have been created called **cfd**.

3.2 Run the program

Use the script `cfd.pbs` provided, use emacs or your preferred editor, to modify the number of processes used from 1 to 12 after the aprun command:

```

#!/bin/bash --login

#PBS -l select=1
#PBS -l walltime=0:10:0
#PBS -A y14
#PBS -N cfd

# Change to directory that the job was submitted from
cd $PBS_O_WORKDIR
aprun -n  12 ./cfd arg1 arg2
```

You will also need to change arg1 and arg2:

- arg1 is the **scale factor** for the problem where a value of 1 will give a problem size of 32x32.
- arg2 is the **number of iterations** to be used in the calculation.
- -n flag is the number of distributed memory parallel tasks in the problem (which will be varied in the next part)

For example:

```
aprun -n  12 ./cfd 6 5000
```

Runs the program with a scale factor of 6 and 5000 iterations.

Try submitting the job using *qsub* with a command similar to:

```
eslogin001:~> qsub -q <resID> cfd.pbs
230853.sdb
```

The job submission system places the output from your job into two files: <job name>.o<jobID>and <job name>.e<jobID>. The *.o<jobID>file contains the output from your job and the *.e<jobID>file contains the errors from your job.

The output from the program should look like :

```
 scalefactor, number of iterations =
 6, 5000
 Running CFD on  192 x  192 grid using  12 processors

 Starting main loop ...
 ... finished

 Ran for  5000 iterations, 0.347290E-04 seconds per iteration
 with total time 0.173645

 Writing output file ...
  ... finished

 CFD completed
Application 7974730 resources: utime ~7s, stime ~1s, Rss ~5120,
inblocks ~23509, outblocks ~34228
```

These results give you the iteration time, total time inside the program and the time taken for the program to launch and complete. Another file output.ppm will have been written which when displayed should give images similar to Figure 4. The exact output depends on the configuration submitted in the job.

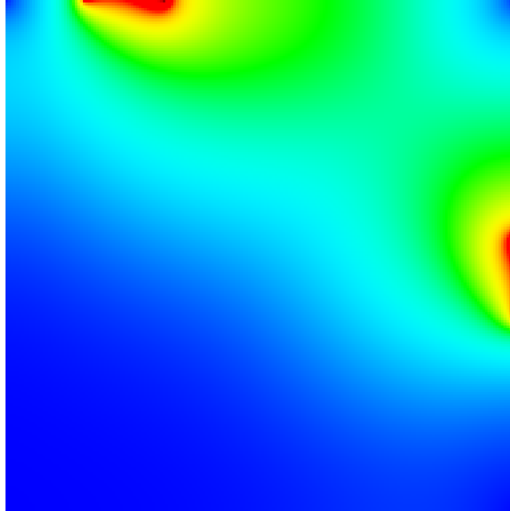

Figure 4: Output Image

3.3 Evaluation

While running jobs fast on big supercomputers is great, resources are finite. Particular available processors on the supercomputer and money to pay for the time using them. The next part of this exercise will be to determine what the best configuration for a group of problems size in the CFD code would be. This will be worked out using two measures, speed up and efficiency.

3.3.1 Speed Up

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 processor is T_1 and to run the code on N processors is T_N, the speedup S is found by:

$$S = \frac{T_1}{T_N} \tag{1}$$

This can be affected by many different factors, including the volume of communications to calculation. If the times are the same speedup is 1, there was no change.

3.3.2 Efficieny

Efficiency is how the use of resources (available processing power in this case) is being used. This can be bought of as the speed up per processor, this allows us to calculate the time a processor may spend idle or on redundant calculations. Efficiency E can be defined as the time to run N models on N processors to the time to run 1 model on 1 processor.

$$E = \frac{S}{N} = \frac{T_1}{NT_N} \tag{2}$$

3.3.3 Doing The Work

The two main evaluation points:

- How does the efficiency of the program vary as the number of processors is increased?
- Does this change as the problem size is varied?

To investigate the speedup and parallel efficiency the code should be run using the same problem size and number of iterations, but with varying numbers of processors. Calculate the speedup and efficiency (tables are provided overleaf for this) and plot a graph of the speedup against the number of processors. Is there any apparent pattern? It may be worth looking at Amdahl's law, see section 3.4.

Now choose a larger problem size and repeat the exercise. To increase the problem size, increase the scale factor. For example, setting **scale factor** to 2 will give a problem size of 64x64. What is the effect on the parallel scaling of the code?

Note that code performs `N` iterations of the Jacobi algorithm to find a solution (rather than checking for convergence.). As a result, when the final picture for large grids it may not look correct because more iterations are required. However, the timings obtained will still be valid. Use more iterations to investigate the problem more thoroughly (if time available).

3.4 Amdahl's Law

The speed-up achievable on a parallel computer can be limited by the portions of a code base which are inherently sequential and cannot be parallelised. This can be defined using Amdahl's law:

> *Let α be the fraction of operations in a computation that are sequential, where $0 \leq \alpha \leq 1$. The maximum speed-up achievable by P processors is limited as follows:*
>
> $$S(n, P) \leq \frac{1}{\alpha + (1-\alpha)/P} \leq \frac{1}{\alpha} \qquad (3)$$

For example, when 20% of the code has sequential nature, the maximum speed-up is limited by 5, independent of the number of processors used to run the code.

1. problem size (scalefactor) = ________________ iterations =________________

No of CPUs	Time	Speedup	Efficiency
1			
2			
4			
8			
16			

2. problem size (scalefactor) = ________________ iterations =________________

No of CPUs	Time	Speedup	Efficiency
1			
2			
4			
8			
16			

3. problem size (scalefactor) = ________________ iterations =________________

No of CPUs	Time	Speedup	Efficiency
1			
2			
4			
8			
16			

4. problem size (scalefactor) = ________________ iterations =________________

No of CPUs	Time	Speedup	Efficiency
1			
2			
4			
8			
16			