|epcc|

THE UNIVERSITY OF EDINBURGH

Computational Fluid Dynamics & MPI performance on ARCHER

September 1, 2014

1 Introduction and Aims

In this exercise we will investigate an example of one of the most common application areas that make use of HPC resources: fluid dynamics. We will look at a distributed-memory implementation of an algorithm that computes the fluid flow pattern for a simple cavity geometry. We will run this code on ARCHER and investigate the default performance using different compilers, the effect of suggested optimisations, hyperthreading, and process placement. If you also have an account on HECToR you will be encouraged to compare the performance of this code on ARCHER versus on HECToR. Finally you may also like to investigate the parallel scaling behaviour of the code.

If you are familiar with this type of problem you may want to go directly to section 3, run the accompanying code, and investigate the MPI performance on ARCHER. However this exercise also introduces:

- Representation of a partial differential equation (PDE) on a grid
- Domain decomposition of this grid
- A numerical algorithm for solving the PDE for a given geometry and boundary values
- Halo swapping (a model for distributed-memory communication)
- An MPI implementation of the above

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.

In fluid dynamics we typically describe continuous systems by means of partial differential equations. For a computer to simulate these systems these 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.

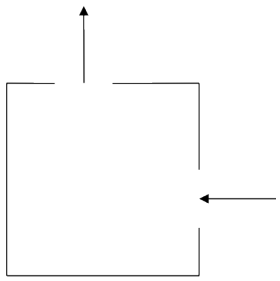

Figure 1: The Cavity

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.

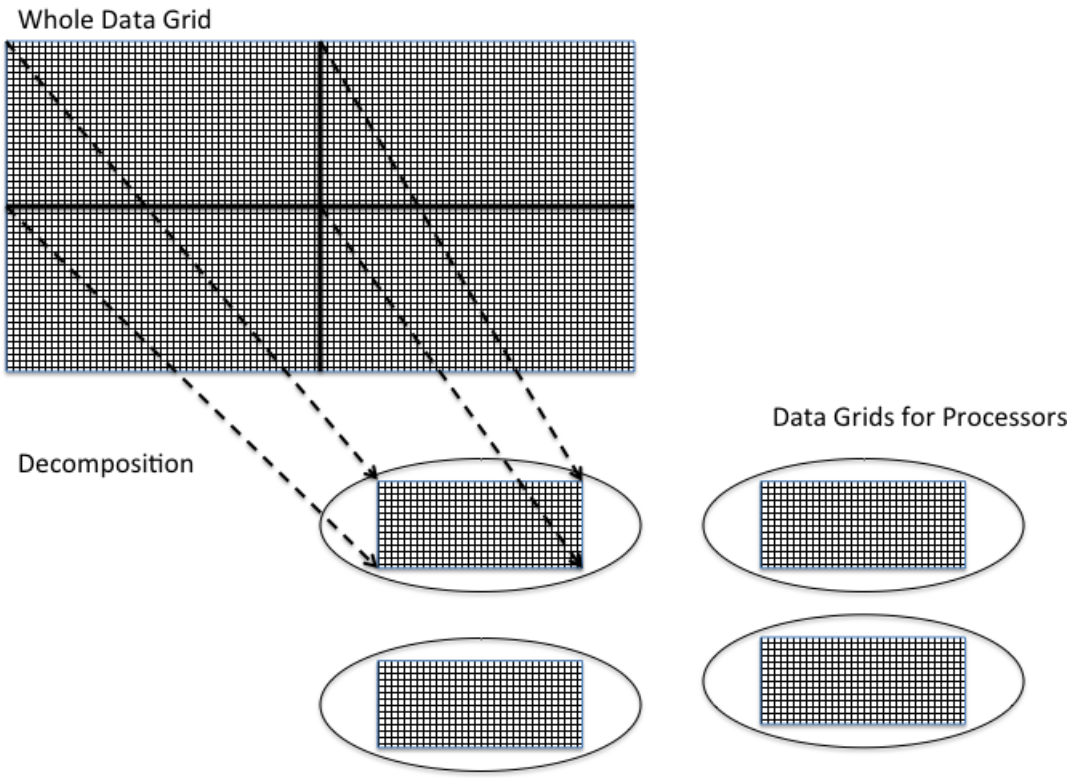

Figure 2: Breaking Up the Big Problem

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.

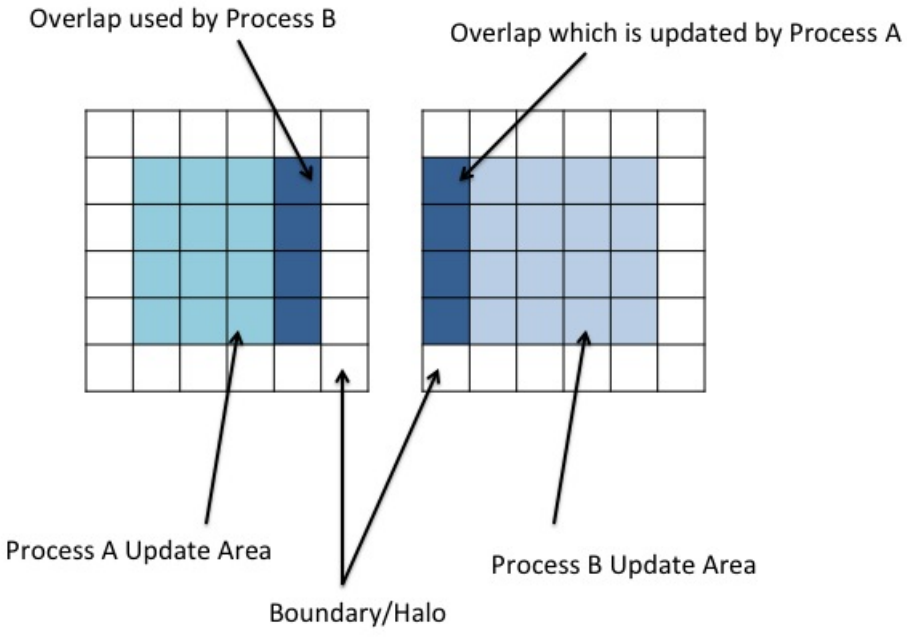

Figure 3: Halo: Process A and Process B

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 Investigating MPI performance

The goal of this exercise is to use the CFD code as a simple but realistic test case to investigate the MPI performance of ARCHER. For now we will look at performance on a single node, in a later exercise we may explore inter-node performance. Concretely, we want to:

- Investigate the relative performance of the code compiled using the Cray, GNU and Intel compilers
- Investigate the effect of suggested compiler optimisations on the default performance
- Look at the effect of explicit utilisation of hyperthreads
- Investigate the effect on performance of process placement

3.1 Compilation

Log on to ARCHER using the guest account ID and password provided. Change to your work directory in the /work filesystem. Use wget to obtain the archive file for the exercise:

```
archer:~> wget http://tinyurl.com/archer030914/Exercises/cfd.tar.gz
```

Unpack the archive using the `tar` command:

```
archer:~> tar -xzvf cfd.tar.gz
```

To compile the exercise code, enter the `cfd/F-MPI` directory and issue the make command. The Makefile has been constructed so that which compiler is used depends on which PrgEnv module you have loaded,

and the name of the resulting binary (cfd_GNU, cfd_CRAY or cfd_INTEL) reflects this fact. If you wish to compile with a different compiler simply load the relevant `PrgEnv` module and run `make` again. The Makefile can also easily be modified to experiment with optimisation flags for the different compilers.

3.2 Running the code

In order to run the code you can write and submit your own job script, edit the existing cfd-ARCHER.pbs script, or run the code in an interactive session. In each case cfd takes two input argument:

- arg1 is a **scale factor** controlling the size of the system, where a value of 1 will give a system of dimensions 48x48.
- arg2 is the **number of iterations** of the Jacobi algorithm to be used in the calculation (the more iterations the more accurate the answer but keep in mind that we are more interested in performance than in getting an accurate answer).
- -n flag is the number of distributed-memory parallel tasks in the problem, i.e. the number of MPI processes

To submit the code using the existing cfd-ARCHER.pbs script simply run the following command:

```
archer:~> qsub -q <resID> cfd-ARCHER.pbs
```

Replacing <resID> with the reservation ID that you have been provided with.

In order to run the code in an interactive job we first request the interactive job using

```
archer:~> qsub -I -l select=1,walltime=02:00:00 -A y14 -q <resID>
```

If, once the job has been granted, we want to run the GNU-compiled version of the code with 24 MPI processes, scale factor 10 and 10000 iterations, we request placement of the runtime as follows:

```
archer:~> aprun -n 24 cfd_GNU 10 10000
```

The output from the program should look like:

```
scalefactor, number of iterations =
          10        10000
 Running CFD on  480 x  480 grid using  24 processors

 Starting main loop ...
 ... finished

 Ran for 10000 iterations, 0.135117E-03 seconds per iteration
                                        with total time  1.35117

 Writing output file ...
  ... finished

 CFD completed
```

```
Application 839950 resources: utime ~33s, stime ~1s,
Rss ~3768, inblocks ~5802, outblocks ~14048
```

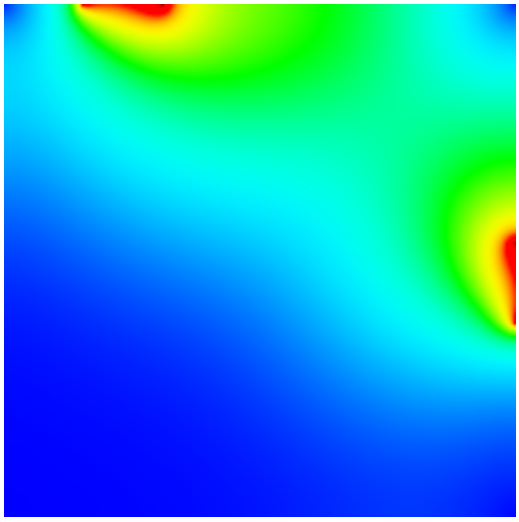

Figure 4: Output Image

These results give you the iteration time, total time inside the program and the time taken for the program to launch and complete. We are interested in the "total time" as this measures the time spent executing the Jacobi algorithm computation (i.e. excluding I/O). Although writing the output file can be a considerable factor in overall performance for larger system sizes, we are interested for now purely in the MPI performance of the computation involving the halo swaps.

A file called output.ppm will have been written which, when displayed using `evince` with X-forwarding enabled over your terminal sessions, should give an image similar to Figure 4. The exact output depends on the size of the problem and the number of iterations.

3.3 Investigation

You are now encouraged to first of all compare the performance of the CFD code compiled using the Cray, GNU, and Intel compilers with their default optimisation settings, i.e. without specifying any optimisation flags. You may like to verify whether the difference you observe holds for different system sizes (by changing the scaling factor) and for different number of MPI processes.

Once you have an initial impression of which compiler(s) seems to be producing less performant code you might like to try using some of the optimisation suggestions in

```
http://tinyurl.com/archer030914/Slides/L03-Compiling.pdf
```

in particular the optimisation flags on page 29.

Next you might like to investigate the effect of explicitly asking aprun to assign MPI processes to individual hyperthreads. In principle hyperthreading promises improved performance so you will find it interesting to

see if explicitly assigning them to be used makes a difference. This can be done with the `-j2` aprun option (see e.g. http://tinyurl.com/archer030914/Slides/L05-Processors.pdf for more information).

It should also be informative to experiment with process placement. In particular, you could try spreading processes between two different NUMA regions, e.g. by running 12 MPI processes on an ARCHER node as follows:

```
aprun -n 12 -d 2 cfd_CRAY 10 10000
```

and compare to the default performance, ie without the -d option (the default is -d 1). You may also like to try the `-cc numa_node` aprun option.

In order to see how MPI performance on ARCHER differs from that on HECToR with which you are likely to be familiar you could compare the performance of the CFD code on ARCHER with that on HECToR compiled with GNU, Cray, and PGI, for the same problem parameters and number of processes.

3.4 Performance measures

If you want to systematically characterise the scaling performance of the CFD code with the number of MPI processes, different compilers, process placements, etc., you could focus on two measures, `speed up` and `efficiency` (defined below).

3.4.1 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.

3.4.2 Efficiency

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.

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

3.4.3 Investigating parallel scaling performance

- 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 3.5.

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 96x96. 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).

3.5 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:*
>
> $$S(n, P) \leq \frac{1}{\alpha + (1 - \alpha)/P} \leq \frac{1}{\alpha} \tag{3}$$

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			