Sharpen Exercise: Using HPC resources and running parallel applications

Contents

1 Aims **2**

2 Introduction **2**

3 Instructions **3**
- 3.1 Log into ARCHER frontend nodes and run commands 3
- 3.2 Download and extract the exercise files 3
- 3.3 Compile the source code to produce an executable file 4
- 3.4 Running a job 5
  - 3.4.1 Write the PBS job script 6
  - 3.4.2 Submit the script to the PBS job submission system . 7
  - 3.4.3 Monitoring/deleting your batch job 7
  - 3.4.4 Finding the output 8
- 3.5 Submit an interactive job and wait for it to start 9
- 3.6 Run the parallel executable on a compute node 10
- 3.7 Viewing the images 11
- 3.8 Additional Exercises 11

4 Appendix **13**
- 4.1 Detailed Login Instructions 13
  - 4.1.1 Procedure for Mac and Linux users 13
  - 4.1.2 Procedure for Windows users 13
- 4.2 Running commands 13
- 4.3 Using the Emacs text editor 14
- 4.4 Useful commands for examining files 15

1 Aims

The aim of this exercise is to get you used to logging into the ARCHER resource, using the command line and an editor to manipulate files, and using the batch submission system.

In this exercise we will be using the successor to the HECToR HPC resource: ARCHER. ARCHER is a Cray XC30 system with a total of 72,192 cores. Although the overall software environment should be familiar to existing users of the HECToR Cray XE6 system, there are several important differences between ARCHER and HECToR. For example, ARCHER uses 12-core 2.7GHz Intel E5-2697 v2 (Ivy Bridge) series processors as opposed to HECToR's 16-core 2.3GHz AMD Opteron (Interlagos) processors. The network is the new Cray Aries interconnect, and ARCHER now supports the Intel compiler suite.

You can find more details on ARCHER and how to use it in the User Guide at:

- http://www.archer.ac.uk/documentation/user-guide/

2 Introduction

In this exercise you will run a simple MPI parallel program to sharpen the provided image.

Using your provided guest account, you will:

1. log onto the ARCHER frontend nodes;
2. copy the source code from a central location to your account;
3. unpack the source code archive;
4. compile the source code to produce an executable file;
5. submit a parallel job using the PBS batch system;
6. run the parallel executable on a compute node using a varying number of processors and examine the performance difference.
7. submit an interactive job.

Demonstrators will be on hand to help you as required. Please do ask questions if you do not understand anything in the instructions - this is what the demonstrators are here for.

3 Instructions

3.1 Log into ARCHER frontend nodes and run commands

You should have been given a guest account ID – referred to generically here as `guestXX` and password. (If you have not, please contact a demonstrator.)

These credentials can be used to access ARCHER by connecting to

```
ssh -X guestXX@login.archer.ac.uk
```

with the SSH client of your choice (`-X` ensures that graphics are routed back to your desktop). Once you have successfully logged in you will be presented with an interactive command prompt.

For more detailed instructions on connecting to ARCHER, or on how to run commands, please see the Appendix.

3.2 Download and extract the exercise files

Firstly, change directory to make sure you are on the "/work" filesystem on ARCHER.

```
guestXX@archer:~> cd /work/y14/y14/guestXX/
```

/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.** Unlike HECToR, where running from /home would result in jobs being much slower and costing more, ARCHER compute nodes cannot access the /home filesystem at all. Any jobs attempting to use /home will fail with an error.

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

```
guestXX@archer:~> wget tinyurl.com/archer100214/sharpen.tar.gz
--2013-11-24 13:55:18--  http://tinyurl.com/archer100214/sharpen.tar.gz
Resolving tinyurl.com... 195.66.135.249, 195.66.135.241, 195.66.135.248
Connecting to tinyurl.com|195.66.135.249|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www2.epcc.ed.ac.uk/~adrianj/archer100214/sharpen.tar.gz [following]
--2013-11-24 13:55:18--  http://www2.epcc.ed.ac.uk/~adrianj/archer100214/sharpen.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: 1921882 (1.8M) [application/x-gzip]
Saving to: ‘sharpen.tar.gz’

100%[======================================>] 1,921,882   --.-K/s   in 0.02s

2013-11-24 13:55:18 (86.9 MB/s) - ‘sharpen.tar.gz’ saved [1921882/1921882]
```

To unpack the archive:

```
guestXX@archer:~> tar -xzvf sharpen.tar.gz
sharpen/
sharpen/C/
sharpen/C-MPI/
.
--snip--
.
sharpen/C/sharpen.c
sharpen/C/sharpen.h
sharpen/C/sharpen.pbs
```

This program takes a fuzzy image and uses a simple algorithm to sharpen the image. A very basic parallel version of the algorithm has been implemented which we will use in this exercise. There are a number of versions of the sharpen program available:

C-MPI Parallel C version using MPI

F-MPI Parallel Fortran version using MPI

C-HYB Parallel C version using a hybrid of MPI and OpenMP

F-HYB Parallel Fortran version using a hybrid of MPI and OpenMP

3.3 Compile the source code to produce an executable file

We will compile the C/MPI parallel version of the code for our example. Move to the *C-MPI* subdirectory and build the program:

```
guestXX@archer:~> cd sharpen/C-MPI
guestXX@archer:~> ls
Makefile  dosharpen.c  fuzzy.pgm   location.h  sharpen.h
cio.c     filter.c     location.c  sharpen.c   sharpen.pbs
```

```
guestXX@archer:~> make
cc -g -c sharpen.c
cc -g -c dosharpen.c
cc -g -c filter.c
cc -g -c cio.c
cc -g -c location.c
cc  -o sharpen sharpen.o dosharpen.o filter.o cio.o location.o
```

This should produce an executable file called *sharpen* which we will run on the ARCHER compute nodes. (Note: this executable will not work on the ARCHER frontend nodes as it requires MPI which is dependent on being run on compute nodes.)

For the Fortran MPI version, the process is much the same as above:

```
guestXX@archer:~> cd sharpen/F-MPI
guestXX@archer:~> ls
Makefile       filter.f90  fuzzy.pgm   sharpen.f90
dosharpen.f90  fio.f90     location.c  sharpen.pbs
guestXX@archer:~> make
ftn -g -c sharpen.f90
ftn -g -c dosharpen.f90
ftn -g -c filter.f90
ftn -g -c fio.f90
cc -g -c location.c
ftn  -o sharpen sharpen.o dosharpen.o filter.o fio.o location.o
```

As before, this should produce a *sharpen* executable.

Don't worry about the C file - here it is just providing an easy method for printing out the program's CPU bindings at run time.

3.4 Running a job

As with HECToR and other HPC systems, use of the compute nodes on ARCHER is mediated by the PBS job submission system. This is used to ensure that all users get access to their fair share of resources, to make sure that the machine is as efficiently used as possible and to allow users to run jobs without having to be physically logged in.

Whilst it is possible to run interactive jobs (jobs where you log directly into the backend nodes on ARCHER and run your executable there) on ARCHER, and they are useful for debugging and development, they are not

ideal for running long and/or large numbers of production jobs as you need to be physically interacting with the system to use them.

The solution to this, and the method that users generally use to run jobs on systems like ARCHER, is to run in *batch* mode. In this case you put the commands you wish to run in a file (called a job script) and the system executes the commands in sequence for you with no need for you to be interacting.

3.4.1 Write the PBS job script

Make sure you are logged onto ARCHER and not in an interactive job session. Using the editor of your choice, open a new file. For example:

```
guestXX@archer:~> emacs sharpen_batch.pbs
```

If you are unfamiliar with using a terminal for text editing, detailed instructions on the use of emacs, as well as general commands for examining files, are included in the Appendix.

Add the following lines to the file:

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

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

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

aprun -n 4 ./sharpen
```

The first line specifies which *shell* to use to interpret the commands we include in the script. Here we use the Bourne Again SHell (bash) which is the default on most modern systems. The –login option tells the shell to behave as if it was an interactive shell.

If you are an experienced user of HECToR, you may have expected to see "mppwidth" as an argument to "-l". This PBS directive, as well as "mppnppn", are not available on ARCHER and have been subsumed by the "select" option. This option is used as *-l select=[nodes]* to request the total number of compute nodes required for your job (1 in the example above). In the

simplest case (when using a single physical core per MPI process) you can get this number by dividing your total number of MPI processes by 24 (the number of physical cores per compute node).

The #PBS lines provide options to the job submission system where "-l select" specifies that we want to reserve 1 compute node for our job - the minimum job size on ARCHER is 1 node (24 cores); the "-l walltime=00:05:00" sets the maximum job length to 5 minutes; "-A y14" sets the budget to charge the job to "y14"; "-N sharpen" sets the job name to "sharpen". You may notice that these resemble the options you specified on the command line when using *qsub* to submit an interactive job earlier. This would be correct, these options can be specified either on the command line to *qsub* or via a job submission script. If you are submitting batch jobs it is often more convenient to add the options to the job script like this rather than type them on the command line every time you submit a job.

The remaining lines are the commands to be executed in the job. Here we have a comment beginning with "#", a directory change to $PBS_O_WORKDIR (an environment variable that specifies the directory the job was submitted from) and the aprun command (this command tells the system to run the jobs on the compute nodes rather than the frontend nodes).

3.4.2 Submit the script to the PBS job submission system

Simply use the qsub command with the reservation ID (i.e. (replace <resID> in the command below with the reservation ID provided by the trainer) and the job submission script name:

```
guestXX@archer:~> qsub -q <resID> sharpen_batch.pbs
58306.sdb
```

The jobID returned from the *qsub* command is used as part of the names of the output files discussed below and also when you want to delete the job (for example, you have submitted the job by mistake).

3.4.3 Monitoring/deleting your batch job

The PBS command *qstat* can be used to examine the batch queues and see if your job is queued, running or complete. *qstat* on its own will list all the jobs on ARCHER (usually hundreds) so you can use the "-u $USER" option to only show your jobs:

```
guestXX@archer:~> qstat -u $USER

sdb:
                                                            Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
58306.sdb       guest01  standard sharpen       --    1  24    --  00:15 Q   --
```

if you do not see your job, it usually means that it has completed.

If you want to delete a job, you can use the *qdel* command with the jobID. For example:

```
guestXX@archer:~> qdel 58306.sdb
```

3.4.4 Finding the output

The job submission system places the output from your job into two files: <job name>.o<jobID> and <job name>.e<jobID> (note that the files are only produced on completion of the job). The *.o<jobID> file contains the output from your job *.e<jobID> contains the errors.

```
guestXX@archer:~> cat sharpen.o58306

Image sharpening code running on 4 processor(s)

Input file is: fuzzy.pgm
Image size is 564 x 770

Using a filter of size 17 x 17

Reading image file: fuzzy.pgm
... done

Starting calculation ...
Process  0 is on cpu  0 on node nid01133
Process  1 is on cpu  1 on node nid01133
Process  3 is on cpu  3 on node nid01133
Process  2 is on cpu  2 on node nid01133
... finished

Writing output file: sharpened.pgm
```

```
... done

Calculation time was 1.552566 seconds
Overall run time was 1.614773 seconds
Application 742518 resources: utime ~6s, stime ~0s, Rss ~24192, inblocks ~13696, outblo
```

3.5 Submit an interactive job and wait for it to start

As well as using the batch system to schedule and run your jobs, it is also possible to submit an *interactive job*. An interactive job allows us to run executables on the ARCHER compute nodes directly from the command line using the *aprun* command as we did in the batch script. This mode of use is extremely useful for debugging and code development as it allows you to get instant feed back on your executable on the compute nodes rather than having to wait for the end of the job (as is the case for non-interactive or *batch* jobs). It has the disadvantage that you have to be physically logged into the machine to issue the commands.

Submit an interactive job using the command (replace <resID> with the reservation ID provided by the trainer):

```
guestXX@archer:~> qsub -IV -l select=1,walltime=3:0:0 -A y14 -q <resID>
qsub: waiting for job 57939.sdb to start
```

(Note that there is no space between the select and walltime options above as they are both arguments to the "-l" option.) The meanings of the various options are:

-I Interactive job

-V Make the job environment match my current session

-l select=1 Reserve 1 node (24 cores) for this job.

-l walltime=3:0:0 Set a maximum wallclock time of 3 hours for this job

-A y14 Charge the job to the y14 budget

-q <resID> Submit the job in the specified reservation

The job may take a minute to start, when it starts you will be returned to a command prompt in your home directory.

Once you have finished running executables on the compute nodes you can exit the interactive job using the *exit* command (do not do this right now):

```
guestXX@mom3:~> exit
logout

qsub: job 57939.sdb completed
```

If you type exit by mistake you can simply resubmit the job using the qsub command above again.

3.6 Run the parallel executable on a compute node

Firstly, change to the directory where you compiled the code. For example:

```
guestXX@mom3:~> cd /work/y14/y14/guestXX/sharpen/C-MPI
guestXX@mom3:/work/y14/y14/guestXX/sharpen/C-MPI> ls
Makefile  dosharpen.c  filter.o    location.h  sharpen.c  sharpen.pbs
cio.c     dosharpen.o  fuzzy.pgm   location.o  sharpen.h
cio.o     filter.c     location.c  sharpen     sharpen.o
```

Use the *aprun* command to run the sharpen executable using 4 parallel tasks - the '-n' option to *aprun* specifies how many parallel tasks to use:

```
guestXX@mom3:/work/y14/y14/guestXX/sharpen/C-MPI> aprun -n 4 ./sharpen

Image sharpening code running on 4 processor(s)

Input file is: fuzzy.pgm
Image size is 564 x 770

Using a filter of size 17 x 17

Reading image file: fuzzy.pgm
... done

Starting calculation ...
Process  0 is on cpu  0 on node nid02206
Process  1 is on cpu  1 on node nid02206
Process  2 is on cpu  2 on node nid02206
```

```
Process  3 is on cpu  3 on node nid02206
... finished

Writing output file: sharpened.pgm

... done

Calculation time was 1.541406 seconds
Overall run time was 1.602274 seconds
Application 738180 resources: utime ~6s, stime ~0s, Rss ~24192, inblocks ~13693, outblo
```

If you try to run the executable using *aprun* outwith an interactive job then you will see an error that looks like:

```
guestXX@archer:~> aprun -n 4 ./sharpen
apsched: request exceeds max nodes, alloc
```

3.7 Viewing the images

To see the effect of the sharpening algorithm, you can view the images using the `eog` Eye of Gnome program, e.g.

```
guestXX@archer:~> eog fuzzy.pgm
guestXX@archer:~> eog sharpened.pgm
```

Type "q" in the image window to close the program.

3.8 Additional Exercises

Now you have successfully run a simple parallel job on ARCHER, here are some suggestions for additional exercises.

1. The sharpen program comprises some purely serial parts (all IO is performed by rank 0) and some purely parallel parts (the sharpening algorithm operates independently on each pixel). Measure the performance on different numbers of processes. Does the computation time decrease linearly? Does the total time follow Amdahl's law as expected?

2. Compile the program using different compilers (you will need to type `make clean` then `make` to force the program to be rebuilt). Do you see any differences in performance?

3. Use the `-N` option to `aprun` to control the number of processes on each node of ARCHER, and set it to a value less than 24. Look at the log file to see how the processes are allocated to the cores on different nodes – is it as you expected? We will see how to control the placement of processes more precisely in subsequent lectures.

4. Run the sharpen program on HECToR and compare the performance to ARCHER. How does the performance compare on a core-to-core basis, and on a node-to-node basis?

4 Appendix

4.1 Detailed Login Instructions

4.1.1 Procedure for Mac and Linux users

Open a command line *Terminal* and enter the following command:

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

you should be prompted to enter your password.

4.1.2 Procedure for Windows users

Windows does not generally have SSH installed by default so some extra work is required. You need to download and install a SSH client application - PuTTY is a good choice:

- http://www.chiark.greenend.org.uk/s̃gtatham/putty/

When you start PuTTY you should be able to enter the ARCHER login address (login.archer.ac.uk). When you connect you will be prompted for your user ID and password.

4.2 Running commands

You can list the directories and files available by using the *ls* (LiSt) command:

```
guestXX@archer:~> ls
bin  work
```

You can modify the behaviour of commands by adding options. Options are usually letters or words preceded by '-' or '–'. For example, to see more details of the files and directories available you can add the '-l' (l for long) option to *ls*:

```
guestXX@archer:~> ls -l
total 8
drwxr-sr-x 2 user z01 4096 Nov 13 14:47 bin
drwxr-sr-x 2 user z01 4096 Nov 13 14:47 work
```

If you want a description of a particular command and the options available you can access this using the *man* (MANual) command. For example, to show more information on *ls*:

```
guestXX@archer:~> man ls
Man: find all matching manual pages
 * ls (1)
   ls (1p)
Man: What manual page do you want?
Man:
```

In the manual, use the spacebar to move down a page, 'u' to move up, and 'q' to quit and exit back to the command line.

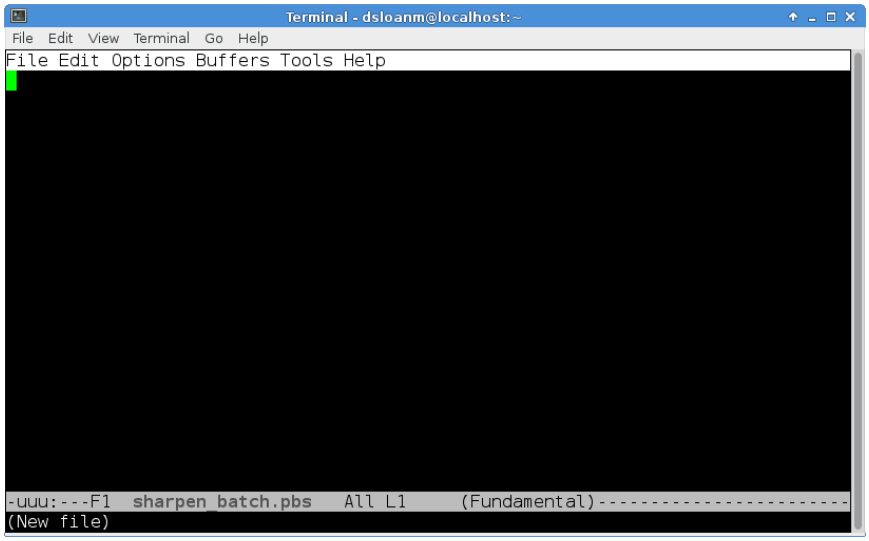
4.3 Using the Emacs text editor

As you do not have access to a windowing environment when using ARCHER, Emacs will be used in *in-terminal* mode. In this mode you can edit the file as usual but you must use keyboard shortcuts to run operations such as "save file" (remember, there are no menus that can be accessed using a mouse).

Start Emacs with the *emacs* command and the name of the file you wish to create. For example:

```
guestXX@archer:~> emacs sharpen_batch.pbs
```

The terminal will change to show that you are now inside the Emacs text editor:



Typing will insert text as you would expect and backspace will delete text. You use special key sequences (involving the Ctrl and Alt buttons) to save files, exit Emacs and so on.

Files can be saved using the sequence "Ctrl-x Ctrl-s" (usually abbreviated in Emacs documentation to "C-x C-s"). You should see the following briefly appear in the line at the bottom of the window (the minibuffer in Emacs-speak):

```
Wrote ./sharpen_batch.pbs
```

To exit Emacs and return to the command line use the sequence "C-x C-c". If you have changes in the file that have not yet been saved Emacs will prompt you (in the minibuffer) to ask if you want to save the changes or not.

Although you could edit files on your local machine using whichever windowed text editor you prefer it is useful to know enough to use an in-terminal editor as there will be times where you want to perform a quick edit that does not justify the hassle of editing and re-uploading.

4.4 Useful commands for examining files

There are a couple of commands that are useful for displaying the contents of plain text files on the command line that you can use to examine the contents of a file without having to open in in Emacs (if you want to edit a file then you will need to use Emacs). The commands are *cat* and *less*. *cat*

simply prints the contents of the file to the terminal window and returns to the command line. For example:

```
guestXX@archer:~> cat sharpen_batch.pbs
aprun -n 4 ./sharpen
```

This is fine for small files where the text fits in a single terminal window. For longer files you can use the *less* command. *less* gives you the ability to scroll up and down in the specified file. For example:

```
guestXX@archer:~> less sharpen.c
```

Once in *less* you can use the spacebar to scroll down and 'u' to scroll up. When you have finished examining the file you can use 'q' to exit *less* and return to the command line.

|epcc|

THE UNIVERSITY OF EDINBURGH

Computational Fluid Dynamics & MPI performance on ARCHER

February 7, 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.

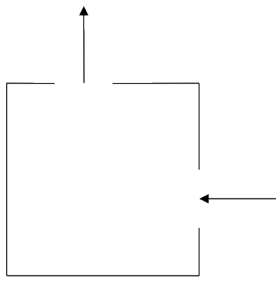


Figure 1: The Cavity

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.

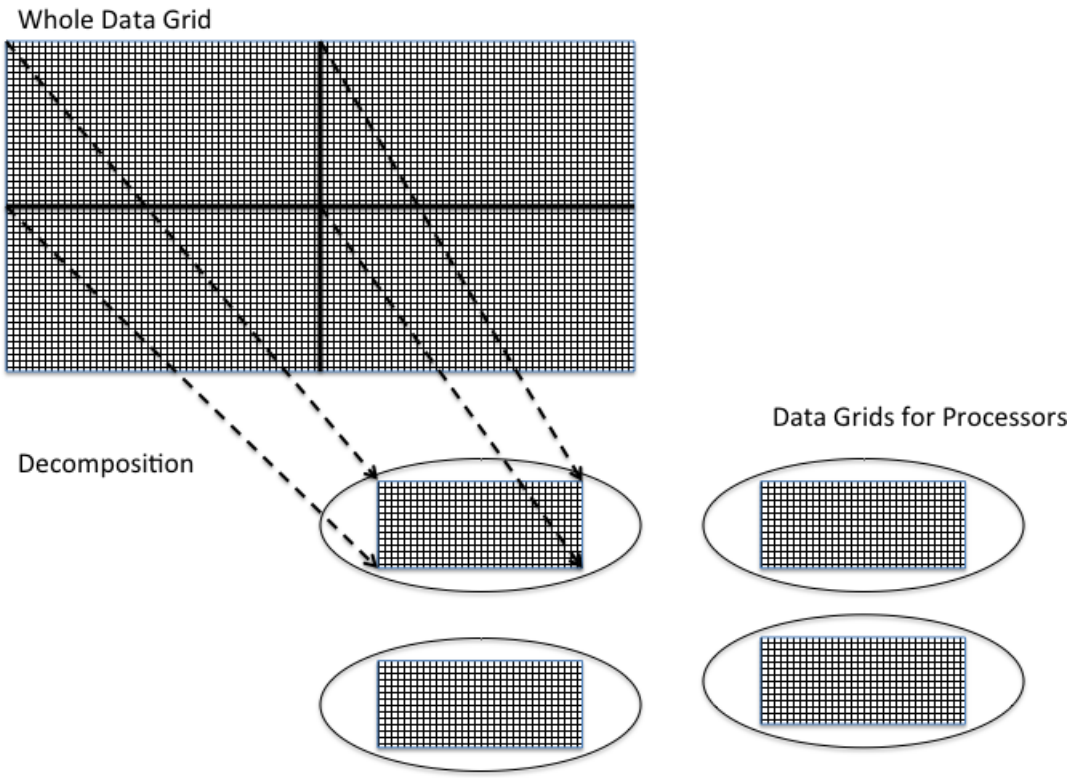


Figure 2: Breaking Up the Big Problem

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.

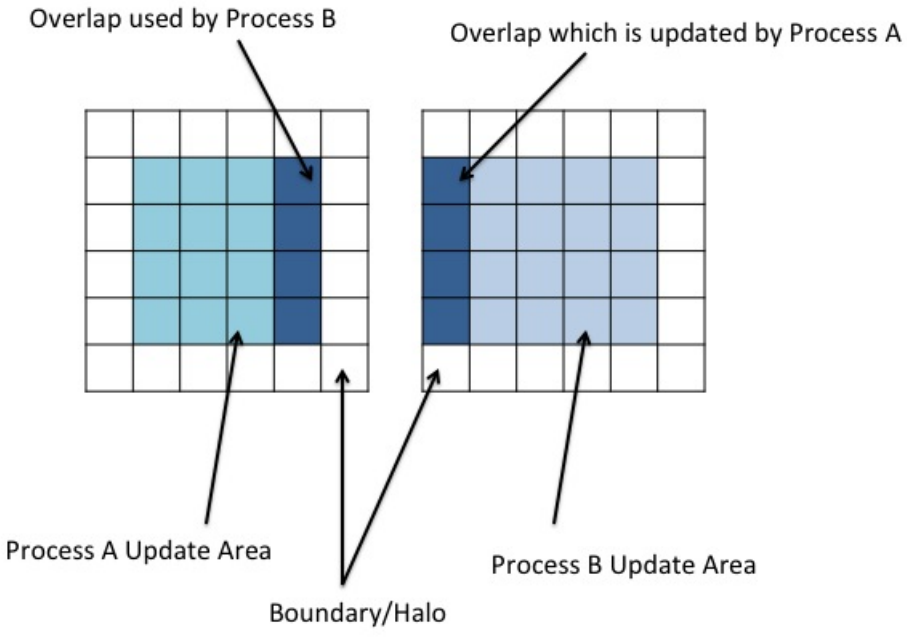


Figure 3: Halo: Process A and Process B

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/archer100214/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
```

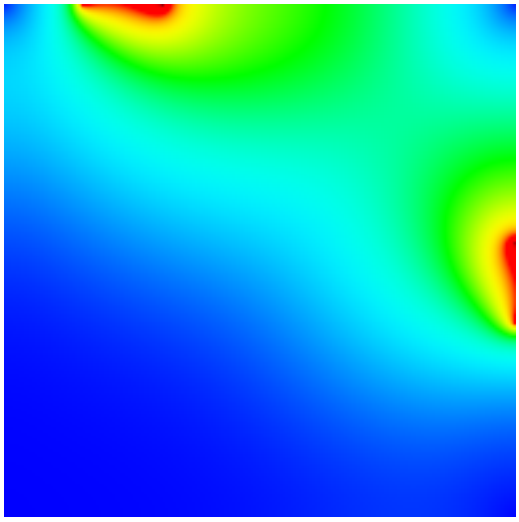


Figure 4: Output Image

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/archer100214/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/archer100214/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			

|epcc|

Debugging Practical

Introduction

This practical aims to give you hands on experience using the DDT debugger on ARCHER. 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/archer100214/Debug.tar.gz
```

Unpack the archive using the `tar` command:

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

This should produce a directory called `Debug`, which contains a ddt sub-directory. In that sub-directories are C and FORTRAN version of a broken MPI code that we will use DDT to debug and fix. Choose one of the source code directories and move into it. Compile the code using the `compile` script as follows:

```
archer:~> ./compile
```

You will need to modify the compile script to include the `-g` flag to ensure that the debugger can properly link the executable to the source code when debugging.

The code will work with all the compilers on ARCHER.

DDT debugging

The recommended method for using DDT on ARCHER is to install the remote DDT client on your machines and use that to connect to ARCHER and run your jobs. Follow the instructions on the ARCHER website, in the Best Practice Guide (`http://www.archer.ac.uk/documentation/best-practice-guide/debug.php#sec-7.4`) to install and setup the DDT remote client. Because we are using a queue reservation for the training sessions we need to use a slightly different *Submission template file* for the setup, you should use `/home/y07/y07/cse/allinea/templates/archer_phase1_reservation.qtf` instead of the nromal `/home/y07/y07/cse/allinea/templates/archer_phase1.qtf` (this will allow use to specify the reservation ID as the queue on ARCHER).

Once you have installed the client and set it up to connect to ARCHER and submit jobs you should choose the **Run** option from the DDT client and configure it to run the executable you

have compiled. Remember that you will need to provide a budget code for the run (for these accounts the code is y14) and because we are using a reservation you will need to specify this as the queue. Also, this application has been built to require 8 processes, so you will need to set the `Number of processes` and `Processes per Node` to 8.

Now you can click *Submit* and the job will be submitted to the batch system. You will then get a screen showing the state of the job in the queue. Once the job runs `ddt` will open showing the source code and ready to run your program.

To run the program click the play icon (the little green triangle on the top left of the window). Run the program and identify where there is a problem (this MPI program has a fault which causes it not to work properly) and see if you can see what the problem is. Once the program is running you can pause it (using the yellow pause button next to the play button) to see where the program is. When you think you've identified the problem you can quit `ddt` by selecting *End session* from the File menu. `ddt` remembers your setup so you can close and open `ddt` without having to re-configure the program for debugging.

Try setting breakpoints in `ddt` and running the program. Experiment with some of the other `ddt` functionality. For a full guide of what `ddt` can do see the user guide at `http://www.allinea.com/products/support`.

Additional Exercises

As well as using DDT you could try using the `STAT` debugging functionality on ARCHER to identify the deadlock problem with the code you have been given. Have a look at slide 20 of the L07-Tools lecture at `http://www.tinyurl.com/archer100214/`, follow the method outlined for using STAT and see if it identifies where the problem is for you.

Introduction to ARCHER and Cray MPI: MPI Exercises

1 Introduction

The purpose of this exercise is to investigate the performance of MPI on an ARCHER node, and between nodes across the Aries interconnect. We will use the standard Intel MPI Benchmark codes, with one small modification so that each rank prints out which node and core it is running on.

2 Compiling the code

Download `IMB.tar.gz` from `tinyurl.com/archer100214/` and unpack: `tar -xvfz IMB.tar.gz`. Go to the directory `IMB/imb/3.2.4/src/` and compile as follows:

```
guestXX@archer:~> make clean
guestXX@archer:~> make -f make_archer
touch exe_io *.c; rm -rf exe_ext exe_mpi1
make -f Makefile_archer.base IO CPP=MPIIO
make[1]: Entering directory '/fs4/y14/y14/guest23/IMB/imb/3.2.4/src'
cc -I/opt/cray/mpt/6.1.1/gni/mpich2-cray/81/include  -DMPIIO -O3 -c IMB.c
```

3 Running the code

Go to `IMB/results_archer/`. In here are a number of PBS scripts to run the benchmarks. You will need to copy the three IMB executables to this directory before running, i.e.

```
guestXX@archer:~> cp ../imb/3.2.4/src/IMB-* .
```

There are a number of tests, e.g. `N0C0_N0C23.pbs` runs the MPI tests between cores 0 and 23 on the same node; `N0C0_N0C1.pbs` runs between two cores on different nodes. Although IMB runs a whole set of MPI tests, here we are mainly interested in the very first PingPong test.

- How do the latency and bandwidth vary depending on the location of the two MPI processes?
- Is there a difference when nodes are on the same blade (i.e. are in the same block of 4 node id's)?
- How do the results compare to HECToR (see `results_hector/`)?
- How do they vary with `MPICH_GNI_MAX_EAGER_MSG_SIZE` and `MPICH_GNI_MAX_VSHORT_MSG_SIZE`?

3.1 Extra Exercise: Allreduce

Change the scripts so that IMB runs on a large number of processes.

- Note the performance of `MPI_Allreduce` for small message sizes.
- Does performance improve by turning on hardware acceleration as described in the lectures?