

Introduction to High Performance Computing Tutorial

Version 1.1

8th May 2007.

Introduction to High Performance Computing - Tutorial

ICHEC - Introduction to HPC

1 Overview

This tutorial provides an overview of how to use the ICHEC computing resources. It is intended as a "getting started" document for new users or for those who want to know "in a nutshell" what computing at ICHEC is all about from a practical user's perspective. It is also intended as the first presentation in a hands-on workshop that covers in great detail parallel programming on the ICHEC systems.

Level/Prerequisites:

Some basic knowledge of Unix skills are required. Some knowledge of high-end computing systems, particularly parallel computing, would be useful, but certainly not required. This tutorial might be considered a prerequisite for using ICHEC systems and other tutorials intended to follow.

Learning Outcomes:

After the practical session users should be able to:

- Login to the ICHEC Walton cluster
- Compile code on Walton and create a makefile
- Utilize the available libraries and optimization flags
- Submit a PBS batch job to the queue
- Utilize the ICHEC Taskfarm
- Calculate speed-up using a serial and MPI code to evaluate pi
- Debug code using the distributed debugging tool (DDT)

2 Logging in to the ICHEC systems

You will be running the examples on the ICHEC distributed memory cluster – Walton. Once you have successfully logged in on the local system you can **ssh** to the Walton cluster. Temporary ICHEC accounts have been set up for the purpose of this tutorial, which run from course00-course24. From the command line you can log in as follows:

```
ssh -X username@walton.ichec.ie
```

For the purpose of this tutorial the course files, packaged as Intro.tar, are located in ~course00/HPC.

To copy the file to your home directory:

```
cp ~course00/Intro.tar ./
```

To untar the file:

```
tar -xvf Intro.tar
```

This should produce three directories 3_Exercises, 4_Exercises and 5_Exercises corresponding to the following sections. To change to the 3_Exercises directory:

```
cd 3_Exercises
```

You are now ready to commence the practical exercises.

3 Compiling/Optimizing Codes

In this section you will work through several examples, beginning with the very simplest compile and link. Then you will move on to other topics such as optimization flags, timers, libraries and Makefiles.

Before compiling the following codes please ensure that you have loaded the Portland compiler module using the command:

```
module load portland
```

3.1 Exercise: Compile and run simple serial programs

C Example:

This example uses the sample file *cprog.c*. A copy of this file is located in the 3_Exercises/3.1/C directory. The following commands, compile (using the Portland C compiler) and load the program; the executable is named *cprog*.

```
pgcc -o cprog cprog.c
```

To run the program (the *./* tells the loader to look for *cprog* in your current working directory; otherwise you may see "file not found" messages)

```
./cprog
```

This code requires that input values be read in. Try the following values:

```
12.5 6.2
```

You should see output that looks like this:
The sum of 12.500000 and 6.200000 is 18.700000

Fortran Example:

For this example, use the sample Fortran file *fprog.f* in the 3_Exercises/3.1/Fortran directory. If you look at the file, you will see that it is a very simple Fortran program which adds two numbers. Issue the following command to try the example.

```
pgf90 -o fprog fprog.f
```

This compiles (using the Portland Fortran compiler) and links your program; the executable is named *fprog*. If there were any syntax errors in your source file or problems during the link, they would be displayed. In this case, both the compile and link steps should be fine, so you should be able to run the program:

```
./fprog
```

This command runs the program (the `./` characters tell the loader to look for the *fprog* file in your current working directory; without these, you might get a "Command not found" message)

Then the program should compute and print the answer:

```
VALUE OF C = 18.70
```

Edit the program to change the values of A and B. Run the program several times with different values. You will need to recompile and link.

3.2 Use of Optimisation Flags and Timers

The use of appropriate optimization switches at compile time can lead to significant improvements in execution time. Fortran and C/C++ compilers have a number of similar optimization switches. There are also a large number of mathematical libraries available on the Walton cluster. You should get everything you are looking for by compiling with the Portland compilers (pgf90, pgcc and pgCC) and linking against acml (AMD Core Math Library) which contains, among other features, optimized versions of blas, lapack, blacs and scalapack for AMD processors.

The CPU time can be measured in different ways. The simplest is to use the program (or built-in shell command) `time`:

```
time app
```

The output from `time` when running the program `app` may look as follows:

```
real 0m4.314s
user 0m3.950s
sys 0m0.020s
```

user -- the amount of CPU time used by the user's program
sys (or *system*) -- the amount of CPU time used by the system in support of the user's program
cpu -- the total CPU time, i.e., user + sys
wall -- the wall clock time, i.e., elapsed real time

Typically the *cpu* time and the *wall* clock time are the same, unless there are other user processes running or there is significant system usage as in excessive disk usage from I/O operations or swapping/paging.

Timers for C codes:

The `clock()` function returns clock ticks. The `gettimeofday()` function provides all clock time and has a resolution of microseconds. As with Fortran codes, `MPI_Wtime()` may also be used with C/C++ codes to determine the time since a particular date.

Timers for Fortran codes:

The Fortran routine `system_clock()` offers good portability. MPI also defines a timer, `MPI_Wtime()` which returns a double precision number of seconds, representing elapsed wall-clock time since some time in the past. You can also use the Fortran 95 `cpu_time` intrinsic function. This is probably the best timing routine to use.

Exercise 3.2: Optimization Flags, Timers and Libraries

Matrix multiplication is a basic building block in many scientific computations; and since it is an $O(n^3)$ algorithm, these codes often spend a lot of their time in matrix multiplication. The most naive code to multiply matrices is short, sweet, simple, and very slow:

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to m
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end
  end
end
```

In this problem, we will look at a number of optimization flags, timers and ways to hand tune the code to improve performance. First, we need a "baseline" of performance for a simple program before we start tuning the code.

1. This section uses **mma.f** (Fortran version in 3_Exercises/3.2/Fortran) or **mma.c** (C version in 3_Exercises/3.2/C). Look over the code, so that you understand what it does.
2. Use the compiler, with no options except the -o (should you want to pick a name for the executable). Run the code and note the execution time for this naïve matrix-matrix multiply.

```
pgf90 -o mma mma.f (or pgcc -o mma mma.c)
time ./mma
```

Use the basic *time* command to compare execution times using the following optimization flags, -O2, -O3. Additionally, recompile the code using the ACML library using

```
-L /opt/packages/pgi-compat/acml/pgi64/lib/ -lacml
```

Optimization Flags	Time taken (s)
No flags	
-O2	
-O3	
-lacml	

What are the results of using the various optimization flags? Use the **man** pages to find out what these flags are doing. Check the sample output carefully: It may not be the same as your first run.

Do you see any improvement in the execution time using the various libraries/compiler switches?

The following steps 3-5 should only be attempted by more advanced users:

3. Repeat the above timing measurements, having implemented your own timer in your C/Fortran code. Try the `clock()` function for the C code and the `cpu_time()` function for the Fortran code.
4. Next we experiment with "hand tuning" the code. First, we tinker with the structure of the loops.
 - a. **Loop reordering:** FORTRAN programmers only need to complete this step. Create another copy of `mma.f` and switch the I and J loops. Compile with `-O2`, run and time. Evidently it is important to minimize the memory stride in the `C(I,J)` matrix.
 - b. **Loop unrolling:** Now unroll the *middle* loop to depth 4. The object is to stride by 4 over the middle loop, while creating 4 temporary

variables to use as accumulators in the innermost loop. In addition, it will be advantageous to create a temporary variable to hold the matrix variable which needs to be used in each of the 4 assignments in the innermost loop. Retime your edited version with the same optimization flags. How differently did optimization flags work on a properly coded program? Which made a bigger difference?

Optimization Flags	Part 3 Time taken (s)	Part 4 Time taken (s)
No flags		
-O2		
-O3		
-lacml		

3.3 Use of Makefiles

A Makefile is a configuration file used by the *make* utility defining the location of source files, how they will be compiled and linked to create the executable program. *make* is tremendously helpful in keeping executables synchronized with changes in source code.

Sample makefiles can be found in 3_Exercises/3.3 directory.

Exercise 3.3: Compilation utilizing Makefiles

Makefile for C Code

A copy of the sample files should be in 3_Exercises directory/3.3/C. This example uses the sample makefile to compile main.c, file1.c, file2.c and file3.c.

To compile:

```
make -f Makefile
```

What is the result of this command?

Makefile for Fortran Code

This example uses the sample makefile to compile *myprogram*. A copy of the necessary files are located in the 3_Exercises/3.3/Fortran directory.

To compile:

```
make -f Makefile
```

What is the outcome of using this command?

3.4 Summary & Conclusions

The morals of the story:

- Do not reinvent the wheel. Standard operations have been optimized by the experts, and you probably won't be able to outdo the software that is found in the numerical libraries.
- Even if you are doing something nonstandard, a good precompiler knows the basic tricks pretty well. For the most part, you'll save yourself the work by preprocessing your code.

4 Running MPI Code on Walton

MPICH is a freely available, portable implementation of [MPI](#), the Standard for Message Passing libraries. To use MPICH and the Portland compilers on the Walton cluster:

```
module load mpich
```

To find out more information about the available modules:

```
module avail (lists the available modules)  
module list (lists the modules you are using at the moment)
```

Or to unload a particular module:

```
module unload mpich (removes that module)
```

ICHEC also has MPICH2 installed on the Walton cluster. MPICH2 is an all-new implementation of MPI, designed to support research into high-performance implementations of MPI-1 and MPI-2 functionality. In addition to the features in MPICH, MPICH2 includes support for one-side communication, dynamic processes, intercommunicator collective operations, and expanded MPI-IO functionality. To use MPICH2:

```
module load mpich2
```

would mean that using `mpicc` to compile your code uses MPICH2 with `gcc`.

So, once a user loads one of the modules, `mpicc` or `mpif77`, will compile their code using the appropriate tools and libraries.

4.1 Exercise: Compile and run a simple MPI program

1. Load the mpich2 module
2. Choose whichever language (Fortran or C) you prefer and look at the program, either hello.c or hello.f in the 4_Exercises/4.1 directory.
3. Load the desired module (ensure you have no other gcc4 module loaded).
4. Compile the program using:

```
Make -f Makefile
```

5. Execute the program.

```
./hellompi
```

```
Hello World! I am process 0 of 1
```

To try to utilise compute resources in a fair and efficient manner, all compute jobs should be run through the batch queueing system which shall now be discussed.

4.2 Running batch jobs on the ICHEC systems

By specifying how many processors you need and for how long, the system can mix and match resource timeslots with jobs from multiple users. All production runs on the cluster must be submitted to the batch queueing system. Every portable batch system (PBS) job must be submitted through a job command file. This contains a number of PBS keyword statements which specify the various resources needed by the job and describes the job to be submitted.

Exercise: Basic PBS scripts for Hello World Example

Examine the components of the sample pbs scripts for the hello world mpi examples (a sample script can be found at 4_Exercises/4.1 directory)

Submit your script to the batch queue using the following command:

```
qsub filename
```

what are the results of the following commands?

```
qstat  
qstat -Q  
qstat -a  
showq
```

Try submitting jobs requesting 8, 16, 32 processors. What are the basic requirements for a pbs script? Become familiar with the various queues available on Walton.

4.3 Exercise: Running the ICHEC Taskfarm

Task farming is a convenient technique to achieve high throughput on large parallel computers. Using this technique you will gain the ability to execute concurrently a (large) number of serial tasks (or series of sub-tasks). The most efficient way to use this technique within a batch environment (as implemented on walton) is to control the tasks using a master/worker paradigm, where one MPI process (known as "the master") will distribute the tasks to other MPI processes ("the workers"). As soon as a worker process has completed the task, which has been allocated to it, it will request a new task from the master process, and so on until all tasks have been executed. ICHEC has provided such a "taskfarm" module which is a MPI wrapper implementing a task farm.

To use this taskfarm:

- The number of tasks (or series of sub-tasks) to be executed must be equal to or greater than the number of CPUs which you are requesting in your batch job
- The tasks (or series of sub-tasks) to be executed within the task farm should preferably have no dependencies on each other
- Ensure the **taskfarm module** is loaded

Specifying the tasks

Tasks (or series of sub-tasks) must be specified in an ASCII file. You must specify a single "task" per line, but each "task" may consist of a series of sub-tasks separated by a semicolon. For instance:

```
[jcdesplat@magma 0020]$ cat tasks.001
cd R01; pwd; date; ../a.out 1; date
cd R02; pwd; date; ../a.out 3; date
cd R03; pwd; date; ../a.out 4; date
cd R04; pwd; date; ../a.out 1; date
cd R05; pwd; date; ../a.out 4; date
```

Submitting the taskfarm through a pbs script for Walton:

```
#!/bin/bash
#PBS -N myjob
#PBS -l nodes=2:ppn=2
#PBS -l walltime=72:00:00
#PBS -V
echo Working directory is $PBS_O_WORKDIR
cd $PBS_O_WORKDIR;mpiexec taskfarm tasks.001
```

What is the result of running the taskfarm in .../4_Exercises/4.3?

4.4 Exercise: Running the PI code

In parallel computing **speedup** refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. It is defined by the following formula: where:

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

- p is the number of processors
- T_1 is the execution time of the sequential algorithm
- T_p is the execution time of the parallel algorithm with p processors.

Change to directory 4_Exercises/4.4. There are sample MPI codes (C or Fortran) to determine p . You will have to add timing routines to your code to help measure the time spent in computation. Run the executable on, 8, 16, 32 processors (Use $\sim 10,000,000$ intervals in your test runs). What is the time taken? Determine the speed-up achieved.

No. of Processors	C		Fortran	
	Time taken (s)	Speed-Up	Time taken (s)	Speed-Up
1				
8				
16				
32				

5 Using Debuggers

Debugging applications is an essential part of the cycle of the development and improvement of codes in computational science. Debuggers provide the means to follow the execution of programs. The Distributed Debugging Tool (DDT) is a graphical debugger specifically designed for debugging parallel codes written using MPI or OpenMP that is available on Walton.

In this session, you will use the example program provided with DDT (5_Exercises/C or 5_Exercises/Fortran) to gain an overview of DDT and its features. After completing this section you should be able to start a program, control your processes, examine data, set breakpoints and learn how to find and recover from errors in your program.

5.1 Exercise : Starting a session

Please review the ICHEC web content (from the ICHEC website regarding starting a DDT session on Walton).

Do not forget to also load mpich2.

```
module load mpich2
```

To compile the relevant piece of code:

F77:

```
make -f fmake.makefile
```

C:

```
make -f simple.makefile
```

(Use the sample DDT Submission script provided in the code directory)

Note: For the following walkthrough please use the C code.

DDT will present a dialog box; you should ensure that the right number of processors are selected as well as the correct debugger interface for your compilers and the correct MPI implementation. Select "run" and DDT will connect to your processes, load your source files and take you into the main DDT window, ready to begin your debugging.

Checking program status:

After loading the example program into DDT a new main DDT window pops up. The source files that were found are shown and the current file, that from which MPI_Init is called will be loaded and shown. At the top of the screen is a collection of coloured numbered boxes, and three lines: "All", "Root" and "Workers". These are process groups. Each numbered box represents a process. A green process is running and a red process is stopped.

Controlling your processes:

Process groups are used to control most of the operations you perform on your processes. There are process operation buttons found in the toolbar just below the process groups: *Play*, *Pause*, *Step In*, *Step over* and *Step out*.

Advancing Processes Forward by a line

Select the "All" process group and then press the "step over" button. The red line in your code will advance. In the examples, press step over again – until the processes are at FUNC1() (line 128 in the C Code).

Stepping into a Function

If we wish to see what happens inside FUNC1(), press the 'Step into' button, and the code window leaps to the relevant part of the source code. The processes have now moved into FUNC1(). Do this again to enter FUNC2().

Stepping out of a function

You should now be seeing the uninteresting definition of FUNC2(), so to return to FUNC1(), click the 'step out' button. The processes will now continue until they reach the end of FUNC2() and then return to FUNC1().

Following If Branches

Probably the most important part of FUNC1() is the If statement and we would like to watch what happens in there; which path is taken. By stepping over we can watch what happens as the program proceeds. Use the 'Step over' button to do this until you leave FUNC1().

Pausing processes at a breakpoint

We would like to see what is happening during an MPI operation – where process 0 is receiving data from all of the other processes. Select the process group "Workers" and then click on the line containing MPI_Send; (line 54 in F77 code, line 146 in the C Code). Click the right mouse button and choose "Add breakpoint", this will add a breakpoint for the currently selected group. Now select the "All" process group and press play to resume execution. After a short period of computation, you will see the process 0 is green, still running and processes 1,2 and 3 have reached the breakpoint.

Manually pausing processes

Process 0 is still running, but we can guess that it will be waiting to receive data and is not actually "working". By selecting either the "Root" or the "All" group if neither is selected and then pressing 'Pause', process 0 can now be stopped.

Select process 0 and you will see the stack, located to the right side of the code. This can be viewed by clicking the mouse over the stack display window, which will currently be displaying the top of the stack.

If you double click on process 0 the code window will jump to the point in the code where process 0 has paused and will bring the stack frame up to the correct stack to view the currently highlighted line in the code and its associated variables.

All of the process operation buttons, and the setting of breakpoints work on the currently selected process group. The ability to configure and save your own groups is a very powerful feature making it much easier to debug your programs using DDT.

Viewing Data

Current Lines

By clicking on any of the paused processes, we can examine the data on the current line of code. If we choose any of the processes that are about to execute a send procedure we can view the message that will be sent to process 0. First select process 1 (using a double click) and then click on the "Current Line(s)" tab, to the right of the window.

The current line(s) panel shows the variables that used on the current line of code for the currently selected process. In this case we can see the "message" variable, and the dest and tag variables. We can see that the dest and tag variables are integers with a value of 0. We can also see from looking at the "message" variable that it is a scalar character variable. You can see more than the currently selected line by dragging the mouse around the lines of interest in the code window.

Local Data

The variables known as "locals", those in scope in your current routine, are displayed in this tab. Some of the information here is dependent on your compiler information, so the precise list of variables displayed may differ from machine to machine.

Keeping An Expression In View

It would be useful to keep a particular eye on the values in "ierr", in order to know if our mpi calls are throwing errors or not; we'd like to watch this as we proceed. We can do this by putting expressions into the evaluate window. This is found below the current line(s)/locals window.

You can drag "ierr" from the locals window into the evaluate window using mouse drop and drop techniques, or you can type it in directly by right clicking in the evaluate window and selecting "Add expression".

We will now make processes 1, 2 and 3 send the data. Press `Step Over` whilst the "Workers" group is selected. Now examine the value of ierr for each of the 3 processes and it should still be 0. On some MPIs the buffering may not be sufficient to let all three communications occur without first setting process 0 running, you will be able to use the process operation buttons talked about in the previous section to find this out if your processes do not stop and instead are running but waiting on I/O.

Exercise Fortran Code: Finding And Fixing An Error

The "hello" program was designed to break. If the program is started with exactly 8 processes it will abort with an error. Lets try to find and fix this error. We can restart "hello" by selecting `Session` and then the `Session Control` option from the tool bar. Then click on `End Session`. In the session control dialog, change the number of processes to 8. Now commence the program as before. When DDT is ready, press play to start everything going. Process 5 will immediately stop, and may print output to stderr (depending on your MPI implementation and other system settings). Doubleclick on process 5 and look at its stack trace – you can clearly see that a call to MPI_Send resulted in MPI_Error being called. Click on the stack and choose the frame containing the `main` function – for gnucompiled codes this may actually be called main__. You'll know you have the right one when the current line variables appear below. Looking at these it is now obvious to see that you are trying to call a MPI_Send without first initializing the variable "dest". In this way it is possible to find errors and faults in your code and locate where they are and what the problem is.

Exercise C Code: Finding a Segmentation Fault

The 'hello' program was designed to crash: if the argument crash is supplied, it will cause a segmentation violation.

We can restart "hello" by selecting "Session" and then the 'Session Control' option from the tool bar. Now click on "End Session". We will now modify the parameters and start debugging again. Click on 'Advanced' to display extra functionality. The advanced start up dialog allows you to specify arguments to your program. In the session dialog, type the word "crash" in the arguments box. Now commence the program as before (click on 'Run'). When DDT is ready, press play to start everything going. One of the processes will bring up a SIGSEGV message. If you clear this message by clicking 'OK', the process that has failed will be selected by default and you can examine exactly where it was when the segmentation violation occurred. It is now straightforward to see that as argv[] is 0, then dereferencing this has thrown the error. You can go back, edit the program, remove the error and recompile. You have now mastered the basic operations of DDT for debugging programs.

6 Further ICHEC Information and Support

The website www.ichec.ie provides

- Information on Training Courses
- User support information
- Hardware and software details
- General information about ICHEC

The Helpdesk:

The Helpdesk (<https://www.ichec.ie/helpdesk/>) is the main entry point to ICHECs' support teams for registered users. Here you can get help in using the service, find out more about ICHEC or send us your comments. If you are not a registered user please send any comments to systems@ichec.ie.