

# Compiling and Running Codes on the HPCx System

April 12, 2006

## 1 Introduction

The aim of this tutorial is to introduce you to the HPCx system.

By the end of this session, you will have:

- Logged on to the HPCx service.
- Compiled serial and parallel code on the system.
- Used the batch system to execute sequential and parallel code.

## 2 Logging on to the HPCx Service

From the terminals, you may log on to the front-end of the HPCx system using ssh.

```
$ ssh -l userid login.hpcx.ac.uk
```

```
##### W A R N I N G #####
This is a private computer facility. Access for any reason must be
specifically authorised by the owner. Unless you are so authorised,
your continued access and any other use may expose you to criminal
and/or civil proceedings.
#####
```

```
userid@login.hpcx.ac.uk's password:
```

```
HPCx - UK National Supercomputer Service.          00CD7EAF4C00
etc.
```

Please note that you will be logged out of the machine after 5 hours of idle time.

The HPCx machines run AIX, the IBM version of Unix. The default login shell is the Korn shell (ksh).

## 3 Resource Usage

Accounting on HPCx is administered using projects. Every user of the machine should have access to at least one project, containing an allocation of CPU resources. To see what projects you have access to, and how much time is left within these, you may use the `budgets` command. Try this command now:

```
$ budgets
z004: 1000.3254 AU 416:48:8
```

## 4 Obtaining the source code

If you have logged on via a course account (courseXX) the example source can be obtained as follows:

```
bash-2.05a$ cp /hpcx/home/z004/z004/course00/HPCxUser.tar.gz .
```

Unzip the archive with

```
bash-2.05a$ gunzip HPCxUser.tar.gz
```

Unpack the tar file:

```
bash-2.05a$ tar xvf HPCxUser.tar
x HPCxUser
x HPCxUser/C
x HPCxUser/C/Makefile.MPI.c, 370 bytes, 1 media blocks.
x HPCxUser/C/Makefile.SER.c, 361 bytes, 1 media blocks.
x HPCxUser/C/cio.c, 2720 bytes, 6 media blocks.
x HPCxUser/C/edge384x256.dat, 499721 bytes, 977 media blocks.
x HPCxUser/C/image.c, 2698 bytes, 6 media blocks.
x HPCxUser/C/image.ll, 298 bytes, 1 media blocks.
x HPCxUser/C/imagempi.c, 3804 bytes, 8 media blocks.
x HPCxUser/C/imagempi.ll, 401 bytes, 1 media blocks.
x HPCxUser/F
x HPCxUser/F/Makefile.MPI.f90, 396 bytes, 1 media blocks.
x HPCxUser/F/Makefile.SER.f90, 386 bytes, 1 media blocks.
x HPCxUser/F/edge384x256.dat, 499721 bytes, 977 media blocks.
x HPCxUser/F/fio.f90, 1824 bytes, 4 media blocks.
x HPCxUser/F/image.f90, 2931 bytes, 6 media blocks.
x HPCxUser/F/image.ll, 298 bytes, 1 media blocks.
x HPCxUser/F/imagempi.f90, 4320 bytes, 9 media blocks.
x HPCxUser/F/imagempi.ll, 401 bytes, 1 media blocks.
```

The source code can also be downloaded from:

<http://www.hpcx.ac.uk/support/training/Intro/HPCxUser.tar.gz>. After downloading the file you will need to unzip and untar the file as described above and then change to the HPCxUser directory.

Within the HPCxUser directory there are two sub-directories, C and F, containing codes written in ANSI C and Fortran 90 respectively. Please choose the language you are most familiar with. To simplify the compilation processes, we have created a set of example Makefiles. These are simply modified versions of the standard HPCx templates which can be downloaded from the HPCx web site: <http://www.hpcx.ac.uk/support/FAQ/template.tar>

## 5 Compiling the serial code

The serial versions of the code are called `image.c` and `image.f90`, and there are additional files `cio.c` and `fio.f90` containing some input/output routines. The programs can be compiled using the makefiles `Makefile.SER.c` and `Makefile.SER.f90`.

Trying running `make` to build the application.

For C:

```
$ make -f Makefile.SER.c
      xlc -c image.c
      xlc -c cio.c
```

```
    xlc -lm -o image image.o cio.o
Target "all" is up to date.
```

For Fortran 90:

```
$ make -f Makefile.SER.f90
    xlf90 -qsuffix=f=f90 -c image.f90
** image    === End of Compilation 1 ===
** jacobistep  === End of Compilation 2 ===
** residue    === End of Compilation 3 ===
** new2old    === End of Compilation 4 ===
1501-510  Compilation successful for file image.f90.
    xlf90 -qsuffix=f=f90 -c fio.f90
** datread    === End of Compilation 1 ===
** pgmwrite   === End of Compilation 2 ===
1501-510  Compilation successful for file fio.f90.
    xlf90 -qsuffix=f=f90 -o image image.o fio.o
Target "all" is up to date.
```

Alternatively, if you make a copy of the appropriate makefile called `Makefile` then you simply have to type `make`.

## 6 Running the code on the HPCx System

The code is a simple image processing code that takes an  $M \times N$  data file as input and produces an  $M \times N$  output image in PGM (Portable Grey Map) format. For this particular exercise we set  $M = 384$  and  $N = 256$ , and the input file is called `edge384x256.dat`. The output file will be called `image384x256.pgm`. The sizes are set as constants near the top of the main program file. For more details on the program see the Appendix.

The compilation has produced an executable called `image`. This is a serial code and you can run it in the normal way, eg for the C version:

```
$ ./image
Processing 384 x 256 image in serial
Tolerance = 0.002000, maximum iterations = 5000
```

```
Reading <edge384x256.dat>
```

```
Iteration 200, delta = 0.0122231
Iteration 400, delta = 0.00791338
Iteration 600, delta = 0.00625514
Iteration 800, delta = 0.00532517
Iteration 1000, delta = 0.00468775
Iteration 1200, delta = 0.00420363
Iteration 1400, delta = 0.00381493
Iteration 1600, delta = 0.00349218
Iteration 1800, delta = 0.00321812
Iteration 2000, delta = 0.0029816
Iteration 2200, delta = 0.00277492
Iteration 2400, delta = 0.00259253
Iteration 2600, delta = 0.00243024
Iteration 2800, delta = 0.00228481
Iteration 3000, delta = 0.0021537
Iteration 3200, delta = 0.00203485
```

```
Writing <image384x256.pgm>
```

```
After 3263 iterations delta = 0.00199968
```

```
Computation time    = 17.3658 seconds  
Time per iteration  = 0.00532203 seconds
```

Running in this way executes the program on a single CPU of the front-end.

You can visualise the output by transferring it from HPCx to the Manchester system and typing

```
userid@spinel$ xv image384x256.pgm
```

## 6.1 Running in batch

Some of the back-end HPCx processors are dedicated to serial jobs. To run programs on these processors you must submit a Loadleveler script. You are provided with such a script called `image.ll`. You can submit this script using `llsubmit`

```
$ llsubmit image.ll  
llsubmit: Processed command file through Submit Filter:  
  "/hpcx/home/loadl/newfilter.pl".  
llsubmit: The job "14f42.739" has been submitted.
```

You should have already been introduced to the commands in this file in earlier lectures, and these are also summarised in the README file that accompanies the standard templates. However of particular importance are the options:

- `job_type = serial`. This is not a parallel job, so only needs one CPU. Please note, the serial queue may not be enabled for course accounts. If this is the case you will need to run the job on the parallel queue using a single processor. Check with your demonstrator for details.
- `node_usage = shared`. As you are only using a single CPU in an LPAR you do not require exclusive access to the node.
- `wall_clock_limit = 00:10:00`. Specifies a wall clock limit of 10 minutes for the job.
- `account_no`. This specifies which project account to charge the job to. You can determine which accounts you have access to with the
- `budgets` command (see earlier). For the course accounts this will be `z004`, but if you are using your own account then you should change this line accordingly.
- `##@ network.MPI = csss,shared,US` if you run on less than 16 processors then you should remove or comment out this line. To comment out this command place a `#` in front of the command, e.g `##@ network.MPI = csss,shared,US`

Any screen output will be saved to a file `jobname.schedd_host.jobid` where `schedd_host` is the name of the lpar which scheduled the job and `jobid` is a job identifier number. For the job above, the output will appear in `image.14f42.739.out`; there also a file with the suffix `.err` containing any output sent to standard error.

The `llq` command displays information on the current status of your job in the batch system. You can also view this information using `xloadl`, which is an X-Windows GUI.

For example:

```
$ llq  
Id                Owner           Submitted      ST PRI Class           Running On
```

```

-----
12f42.739.0      zhiwei      1/23 13:49 R  50  par8_12      14f36
11f41.1048.2    dlrojo      1/23 21:12 R  50  par32_12     11f17
11f41.1050.0    oberg       1/24 06:09 R  50  par16_12     12f13
13f42.747.0    xadey       1/24 08:57 R  50  par16_12     14f09
14f41.762.0     jd          1/24 09:01 R  50  par2_12      14f41
11f41.1052.0    zhang       1/24 10:09 R  50  par16_12     14f24
11f42.851.0     limcdap     1/24 11:11 R  50  par1_12      11f39
...

```

28 job step(s) in queue, 15 waiting, 0 pending, 13 running, ...

Or to view this information using `xloadl`, simply type `xloadl` at the command prompt. Job status information is displayed in the first dialogue box.

Now have a look at the log file and error file from your job and check that they contain what you expect.

## 7 Compiler optimisation - optional

There are a number of compilation flags which can be used to increase the performance of your code. The most important compilation switch is `-O`. The `-O` flag provides a series of optimisation levels, from simple low level optimisation `-O2` to extensive optimisation `-O5`. Other useful compilation switches include `-qarch=pwr4` and `-qtune=pwr4`, which target the compilation to a particular processor and memory system.

The example code you have been running has been compiled with none of these options. Try compiling, running and timing this code with various combinations of these compiler flags, and see how this influences the performance.

Compiler Flag	Computation Time	Correct Answer (Y/N)?
none		
-O3		
-O3 -qarch=pwr4		
-O3 -qtune=pwr4		
-O3 -qarch=pwr4 -qtune=pwr4		
-O4		
-O5		

To use these options, you should change the line in the makefile beginning:

```
CFLAGS= or FFLAGS=
```

to something like

```
CFLAGS= -O3 or FFLAGS = -O3
```

It is always important to verify that program still works when you increase the optimisation level. Subtle bugs in your code may only become apparent when the compiler does aggressive optimisation; for example it may assume strict compliance to the language standard when your code actually does something a slightly devious!

## 8 Parallel Jobs

Parallel versions of the code are available in `imagempi.c` and `imagempi.f90`. You are supplied with an MPI makefile, `eg` to compile the C version:

```
$ make -f Makefile.MPI.c clean
      rm -f imagempi.o cio.o imagempi core
$ make -f Makefile.MPI.c
      mpcc_r -c imagempi.c
      mpcc_r -c cio.c
      mpcc_r -lm -o imagempi imagempi.o cio.o
Target "all" is up to date.
```

## 8.1 Parallel Execution

Unlike serial jobs, parallel jobs cannot be run on the front-end and must be submitted to Loadleveler. You should submit the file `imagempi.ll` in order to execute the program in parallel on the back-end nodes.

Some important lines in this parallel Loadleveler file are:

- `job_type = parallel`. This informs LoadLeveler that this is a parallel job which requires scheduling on multiple processors.
- `cpus = 4`. Specify the number of CPUs you require.
- `node_usage = shared`. Normally this parameter should be set to `not_shared` to provide exclusive access to the processors. When using the course accounts shared usage is allowed to prevent slow turnaround for jobs.

Note that LPAR's are allocated exclusively to one user at a time. As a result, you are charged for 16 CPUs per LPAR regardless of how many tasks you actually run. We recommend that you use multiples of 16, e.g. 16, 32, 48, 64, etc. If you select a number of CPUs different from the above, you will be charged for the next full multiple of 16. For example, if you request 59 CPUs, you will be charged for  $4 \times 16 = 64$  CPUs. This is because your jobs is granted exclusive access to the nodes (LPAR's) that it occupies.

Submit the job to Loadleveler and look at the output file. You should see a substantial increase in speed due to the parallelisation.

## 9 Conclusions

The aim of the practical was to introduce you to the HPCx service. You should now be able to compile and execute parallel applications on the machine. If you have time, feel free to experiment with your own codes during the practical sessions.



Figure 1: Result of simple edge detection

## 10 Appendix: The Image Processing Code

You will be given a data file that represents the *output* from a very simple edge-detection algorithm applied to a greyscale image of size  $M \times N$ .

The edge pixel values are constructed from the image using

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4 image_{i,j}$$

If an image pixel has the same value as its four surrounding neighbours (ie no edge) then the value of  $edge_{i,j}$  will be zero. If the pixel is very different from its four neighbours (ie a possible edge) then  $edge_{i,j}$  will be large in magnitude. We will always consider  $i$  and  $j$  to lie in the range  $1, 2, \dots, M$  and  $1, 2, \dots, N$  respectively. Pixels that lie outside this range (eg  $image_{i,0}$  or  $image_{M+1,j}$ ) are simply considered to be set to zero.

Many more sophisticated methods for edge detection exist, but this is a nice simple approach. See Figure 1 for an example of how this works in practice.

The exercise is actually to do the reverse operation and construct the initial image given the edges. This is a slightly artificial thing to do, and is only possible given the very simple approach used to detect the edges in the first place. However, it turns out that the reverse calculation is iterative, requiring many successive operations each very similar to the edge detection calculation itself. The fact that calculating the image from the edges requires a large amount of computation, including many boundary swaps in the parallel code, makes it a much more suitable program than edge detection itself for the purposes of timing and parallel scaling studies.

It turns out that the full image can be reconstructed from the edges by *repeated* operations of the form

$$new_{i,j} = \frac{1}{4}(old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j})$$

where *old* and *new* are the image values at the beginning and end of each iteration. We will take the initial value for the image array (the value of *old* at the start of the first iteration) to be equal to *edge*.

The simplest way to set pixels off the edge of the array to zero is to declare all arrays in your program with explicit halos, ie `float old[M+2][N+2]` in C or `REAL old(0:M+1,0:N+1)` in Fortran (similarly for *new* and *edge*), and set the halo values to zero.

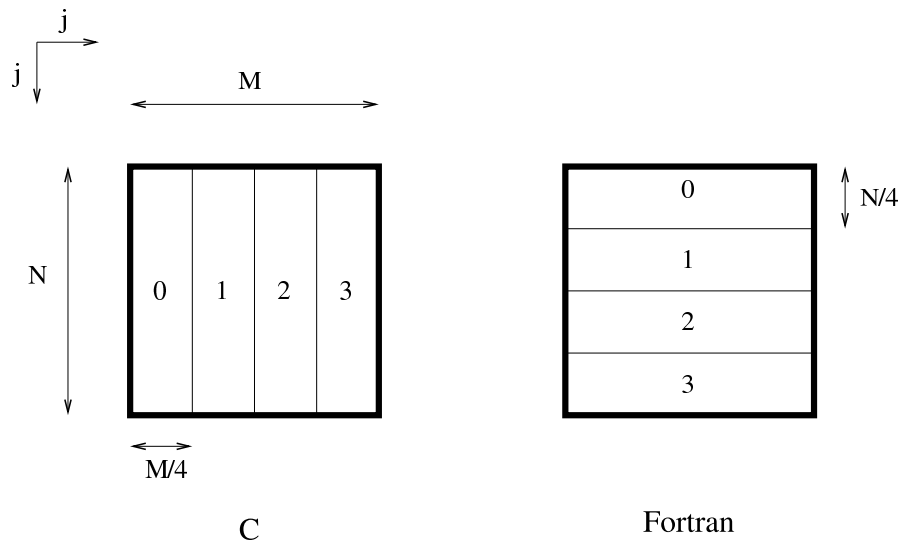


Figure 2: Decomposition strategies for 4 processes

As an aside, this inverse operation is also very similar to a large number of real scientific HPC calculations that solve partial differential equations using iterative algorithms such as Jacobi or Gauss-Seidel.

If you want to view the input data (ie the edges identified by the simple edge-detection algorithm) then you can run the program for zero iterations, in which case the output file will be identical to the input.

## 10.1 Parallelisation

The communications involves sending and receiving entire rows or columns of halo data (depending on whether you are using C or Fortran) every iteration. The process is as follows - it may be helpful to look at the appropriate decomposition in Figure 2.

For C:

- send the  $N$  array elements (`old[MP][j]`;  $j = 1, N$ ) to  $rank + 1$
- receive  $N$  array elements from  $rank - 1$  into (`old[0][j]`;  $j = 1, N$ )
- send the  $N$  array elements (`old[1][j]`;  $j = 1, N$ ) to  $rank - 1$
- receive  $N$  array elements from  $rank + 1$  into (`old[MP+1][j]`;  $j = 1, N$ )

For Fortran:

- send the  $M$  array elements (`old[i][NP]`;  $i = 1, M$ ) to  $rank + 1$
- receive  $M$  array elements from  $rank - 1$  into (`old[i][0]`;  $i = 1, M$ )
- send the  $M$  array elements (`old[i][1]`;  $i = 1, M$ ) to  $rank - 1$
- receive  $M$  array elements from  $rank + 1$  into (`old[i][NP+1]`;  $i = 1, M$ )

The IO is parallelised by reading in on the master process and scattering the data to the others. When computing the `delta`, the change in the image at each iteration, it is also necessary to perform a global sum.