Introduction to running C based MPI jobs on COGNAC.

Paul Bourke

November 2006

The following is a practical introduction to running parallel MPI jobs on COGNAC, the SGI Altix machine (160 Itanium2 cpus) based at the IVEC/ARRC node at TechPark. These notes are not intended to cover everything one may need to know about the machine, the queuing system or MPI. Rather it is intended to provide a complete operational example for first time users. It is assumed the reader is at least somewhat familiar with UNIX, ssh/sftp, and writing/compiling C programs.

Accompanying these notes is a tar.gz archive containing various files that will be used during this introduction. It is assumed that the reader has an account on COGNAC and knows how to connect and login using ssh. These details are available in documentation provided by IVEC to new users.

The supplied archive should contain at least the following files

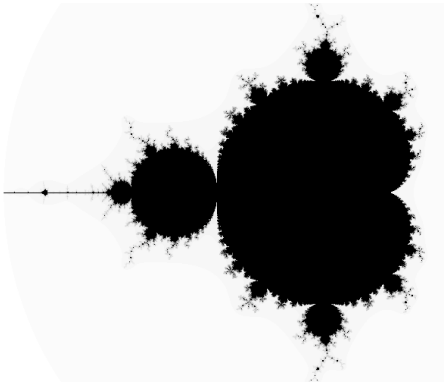
Makefile

buddha.h

buddha_mpi.c

buddha_single.c

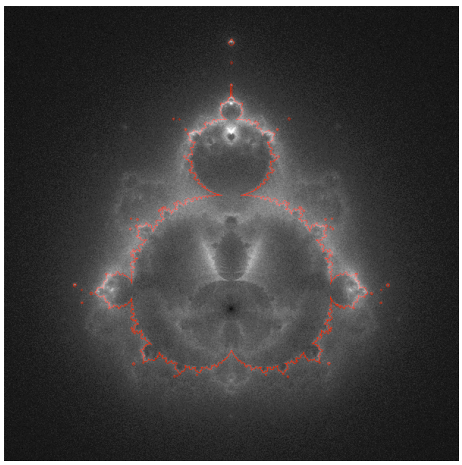
runme



Buddhabrot

Most people are familiar with the Mandelbrot image, perhaps the most explored and popularised of all fractal equations. The image itself is a visualisation of how the following complex valued series behaves

$$z_{n+1} = z_n^2 + z_c$$



Where z_c is determined from the pixel in question mapped from a region of the complex plane corresponding to the image. z_0 is the initial value of the series, normally just 0+i0. For any particular z_c the series does one of two things: it either tends to infinity in which case we say these points "escape" and are outside the Mandelbrot set, or it doesn't (it may decay to 0 or oscillate) and these points are said to be inside the Mandelbrot set and are usually coloured black.

The so called Buddhabrot is a more recent innovation. In this case for all the points that escape (tend to infinity) we cumulatively draw the path they take as a series of points on the complex image plane. The final image is a 2 dimensional histogram, the more populated parts of the histogram are those frequented often by the escaping series. The Buddhabrot takes significantly longer to create a well resolved image

than the Mandelbrot, and in a sense the true Buddhabrot image is only obtained after an infinite number of samples.

Exercise 1

Open buddha_short.c with a text editor and check that you understand roughly how it works. Very briefly: an image plane is created, starting points z_c for the Mandelbrot series are randomly chosen, the status of the series is determined, if the series escapes then the points along the escape path are cumulatively added to the image plane. Finally the histogram values at each pixel are scaled and mapped to a single byte (0 to 255) and the result written to a TGA image file that can be opened in GIMP (as one possibility).

Compile buddha_single.c on the local workstation with the following (it is totally self contained)

```
cc -o buddha_single buddha_single.c –lm
```

Run the program

```
buddha_single
```

and check the resulting image called buddha_single.tga. The image will be grainy, a higher quality image requires significantly more iterations. Instead of simply increasing the number of iterations and have it take longer on a single machine, let's achieve more iterations by running it simultaneously across a number of cpus.

Building the MPI code on COGNAC

To build the MPI version of the Buddhabrot program on COGNAC one needs to load the appropriate compiler module (refer to the IVEC documentation for more information on modules), for example

```
module load intel-cc
```

Modules are a mechanism by which software packages (possibly similar software from different suppliers) can be made available by mapping the appropriate shell environment variables. In the case of the C compler it allows the system to support compilers from multiple suppliers by changing include and library paths. To see what modules are available on the system use

```
module avail
```

To see what modules are currently loaded (eg: Intel C compler)

```
module list
```

To build the MPI version of Buddhabrot using the current Intel C compilers simply type

```
Make
```

See the supplied "Makefile" for more details.

Users are supplied with various disks on COGNAC, the main disk storage beside the home directory is a directory at "/disk". This provides users with significantly more space than their home area but it is not backed up.

Exercise 2

Have a look at the code buddha_mpi.c. This is not supposed to be a sophisticated MPI example, instead it is a rather straightforward MPI implementation using just blocking send and receive functions. Additionally this is an example of the very simplest type of parallel algorithm, since the final Buddhabrot is a histogram each process can independently calculate it's own histogram and the results combined into a final composite histogram (or image) at the end. Note that this program will take as long as the single processor version (aside from cpu performance differences) but the final number of iterations that contribute to the result is multiplied by the number of processors employed.

Running an MPI job on COGNAC

All jobs on COGNAC are run through a job manager called PBS (Portable Batch System), the command to submit our Buddhabrot job is

```
qsub runme
```

This does two things, it provides PBS with information it needs such as which queue to use, how much memory is required, how many processors to use, how much time is requested, and so on. Secondly it actually launches the job using mpirun. For more information on mpirun and the command line options see the man pages, there is some variation in how mpirun is used depending on the local environment … for example whether PBS is being used, whether one is running on a cluster, and so on.

Exercise 3

Look at the file "runme" and the man pages for qsub and determine what the various queue settings in the runme script (identified by lines starting with #PBS) do.

```
man qsub
```

Queues

The PBS queuing system can be configured to manage a number of queue types. For example it is very common for sites to have a queue designed primarily for short jobs and another for longer running jobs. The different queues are configured to give optimal use of the resources available while trying to cater for a range of job characteristics and user requirements.

For example, on COGNAC the main queue is called "normal" and is the one used in this exercise. There is another queue called "express", this queue is designed for code testing prior to long runs. As such, jobs submitted to this queue have a higher chance of being executed immediately but to discourage people from using it for other purposes it is charged at a higher rate and has limits placed on how long jobs may run for and how much memory they may request.

When a job is submitted to PBS it is allocated a unique identifier that can then be used by the various queue management tools. Useful queue commands are given below, for more information use the man pages, eg: man qstat

qstat: You may inspect the status of the queue with the PBS command "qstat". This shows all the current jobs, their state (eg: queued or running), which queue they are in, how long they have been running for, how much memory they are using, the job owner, and so on.

nqstat: To check on just the jobs you own then use "nqstat", or to see all the jobs organised by queue try "nqstat –a".

qdel This allows a user to remove one of the jobs they own from the queue, the job is identified by the identifier assigned by PBS.

Output

So where does output from the submitted job go, for example, the standard output and standard error streams that can result from UNIX commands? These are written to files that have the job name followed by either ".o" or ".e" which in turn are followed by the job identifier assigned by PBS. In the above example PBS was instructed to combine the standard error and standard output to the same file (see "#PBS -j oe" in the runme script). So after running the Buddhabrot example above there should exist a single file called "Buddhabrot.o------" (where the "-" symbols will be replaced by the job identifier). Have a look at this file (it is just a text file) and in particular the PBS statistics that give an indication of the resources used and for which you may be charged.

Unless instructed otherwise files created by the program will be written in the directory the program was launched from, as per usual UNIX conventions.

Exercise 4

Edit "runme" and change the number of cpus from 4 to 8, submit the job again. Optionally if the "normal" queue looks busier than the "express" queue, change the queue to "express". Transfer the .tga file it created to the local machine using sftp and open it using GIMP.

sftp username@cognac.ivec.org

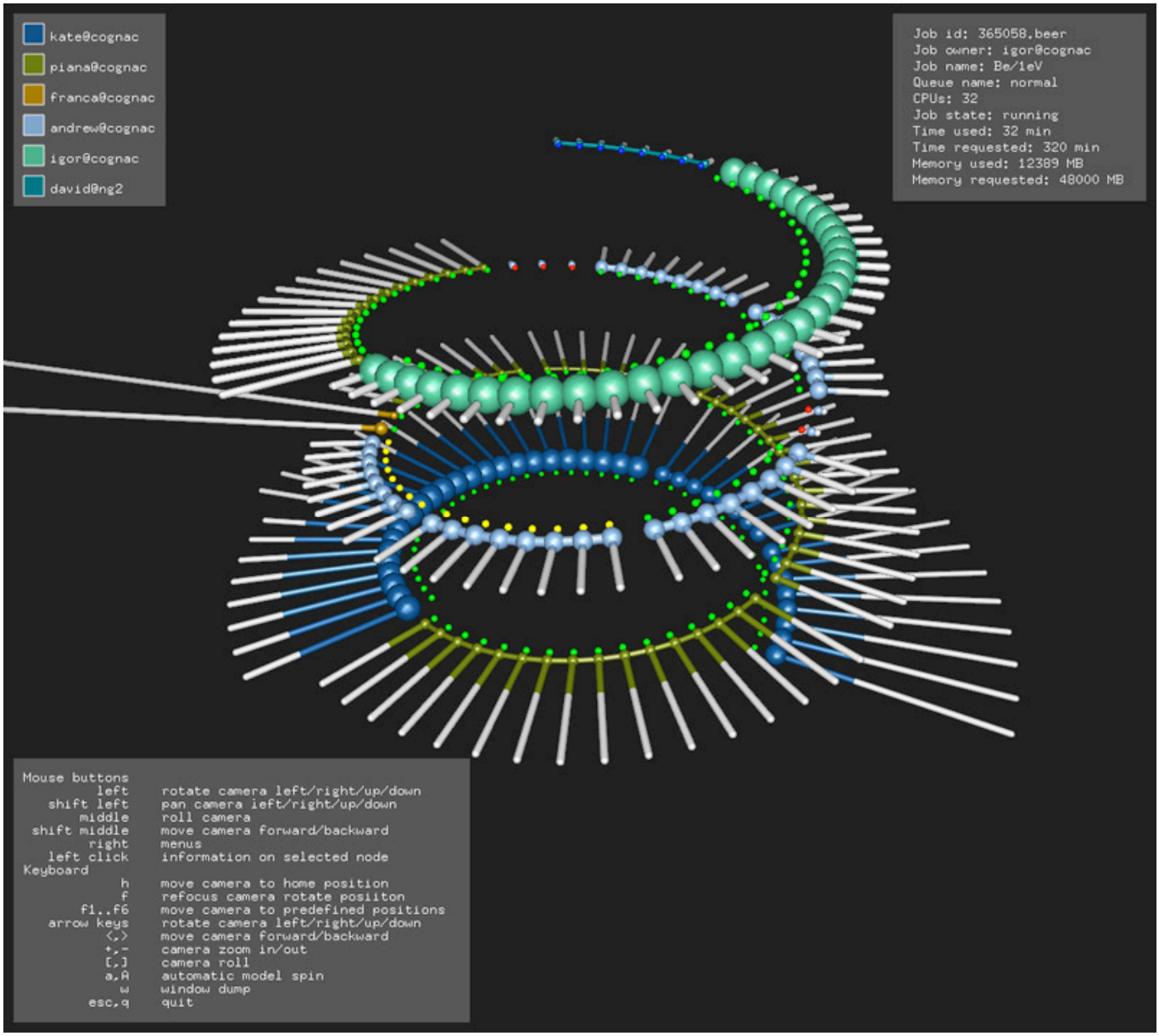
Create a higher quality version of the image by increasing TMAX by a factor of 10, recompile, submit, and check the image.

Additional notes

1. Jobs that exceed the time requested (walltime) will be terminated.
2. The maximum resources for each queue can be determined by the "ivec_limits" command.
3. Your usage quota is given with the "quotasu" command.
4. Jobs that exceed the memory requested will be terminated.

Graphical representation of the PBS queue

The following is a graphical representation of the jobs in the queue. Briefly, each sphere represents a process entry, MPI processes are joined together in groups. Each user is assigned a unique colour (ee top left) and their jobs are shaded accordingly. The radial arms have a length proportional to the requested wall time, the coloured part of the radial arms indicated the time currently used. The radius of the process spheres is proportional to the memory requested and the small spheres next to each process is coloured depending on the job status (running, queued, suspended). By clicking on any part of the spiral structure presents a panel (top right) indicating further details of the job in the queue, this fades away after a while until a new item is selected. The whole visualisation is interactive, the user changes the orientation to get the best view of the processes of interest.



Final smooth image

Rendered on 8 processors, each iterating 100000000 times.