15-418, Spring 2008
Tutorial on Using `cobalt` at NCSA and `rachel` at PSC

1 About the Machines

We will be using two parallel machines that support OpenMP this semester: `cobalt` at the National Center for Supercomputing Applications (NCSA), and `rachel` at the Pittsburgh Supercomputing Center (PSC).

`cobalt` is a collection of SGI Altix machines, which are built using Intel Itanium 2 processors. There is a smaller cluster (`co-login1`) for interactive use: you will be logging onto these machines to compile and test your code. Once your code is ready for a timing measurement, you will submit it to a batch queue to run on the 512-processor SGI Altix machine (`co-compute1`). There is a wealth of information about `cobalt` and how to use it at the following web site:

`http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/SGIAltix/`

`rachel` is an HP AlphaServer comprised of 64 Alpha EV7 processors. Details about `rachel` can be found here:

`http://www.psc.edu/machines/marvel/rachel.html`

2 Connecting

2.1 Logging In

You should use `ssh` to connect to the front-end machines at both NCSA and PSC. In particular, type the following to log onto the interactive cobalt machine (`co-login1`) at NCSA:

```
> ssh yourUserName@cobalt.ncsa.uiuc.edu
```

Type the following to log into rachel:

```
> ssh yourUserName@rachel.psc.edu
```

2.2 Transferring files via `scp`

Unfortunately, neither `cobalt` nor `rachel` support AFS. You will have a local directory on each of these machines. In order to transfer files to and from `cobalt` and `rachel`, we suggest that you use `scp` (the secure copy command). For example, to copy a file from your filesystem to `rachel`, you can do the following:

```
> scp yourLocalFile yourUserName@rachel.psc.edu:yourRemoteDirectory/yourRemoteFile
```

If you are a Windows user, you might want to use `WinSCP` (available at `http://winscp.net`), which has a GUI interface and a command to automatically synchronize local and remote directories.

3 Compiling your programs

Because the OpenMP standard is supported on both `cobalt` and `rachel`, you should not need to modify your source code to run on either platform. Since these machines were designed by different vendors and have different processors, compilers, etc., you do need to compile your code with different compilers and different compiler flags. You can find several trivial OpenMP C code examples in *ASSTDIR*`/examples`, along with a `Makefile` for each machine: i.e. *ASSTDIR*`/examples/Makefile.ncsa` and *ASSTDIR*`/examples/Makefile.psc`. In addition, there is a sample `Makefile` you can use for your project in *ASSTDIR*`/code/Makefile` that supports both platforms.

3.1 Compiling on `cobalt`

To compile OpenMP programs on `cobalt`, use `icc` and specify `-openmp` to the compiler. For example, to compile a program `hello.c` with optimization, type:

```
> icc -openmp -O hello.c -o hello
```

3.2 Compiling on `rachel`

To compile OpenMP programs on `rachel`, use `cc` and specify `-omp` to the compiler. Because the frontend node of `rachel` have different processors (EV6.7) than the compute node (EV7). You need to tell the compiler to optimize for the compute processors by specifying `-O -fast -tune ev7 -arch ev7`. For example, to compile a program `hello.c` with optimization, type:

```
> cc -omp -O -fast -tune ev7 -arch7 hello.c -o hello
```

4 Running your programs

For both `cobalt` and `rachel`, you can run your program on the frontend nodes (i.e. the ones that you login to) by simply typing the program name at the command line directly (i.e. the way that you would normally run a program on a UNIX-based system.) This is a very useful way to debug your program, and to get some rough performance numbers.

To get a proper performance measurement on either machine, however, you need to submit your program to the batch queueing systems. When your program reaches the head of the batch queue, it will run with the machine (or your portion of the machine) to itself. It is important to note that these machines are sometimes busy and it can take a while for jobs to reach the front of the queue. This is especially true for larger jobs requiring many processors. For that reason it is probably in your best interests to allow some extra time for running your final experiments.

4.1 Submitting batch jobs

Both `cobalt` and `rachel` use the Portable Batch Scheduler (PBS) system to control access to compute processors for batch jobs. Therefore you will need to compose job scripts in order to submit your jobs. You may need slight different scripts for the two different systems. Please see the following web pages for details:

```
http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/SGIAltix/Doc/Jobs.html
http://www.psc.edu/machines/marvel/rachel.html#batch
```

Job scripts are regular shell scripts with additional PBS directives at the top, immediately after the specification of the shell to use. To see an example, search for the sample script that follows *"A sample job script for an OpenMP program is"* in the `rachel` documentation. Notice the PBS directives at the top of the script.

The first directive – `#PBS -l walltime=5:00:00` – indicates to PBS that you expect your job to take five hours to run. It is important to provide a reasonable estimate for the runtime of your job. This time is a hard limit on how long your job will be permitted to run. Once the time elapses, your job will be killed if it has not already terminated. This makes it undesireable to under estimate how long your job will take to run. On the other hand, shorter jobs are given priority so grossly over-estimating the run time required by your job will result in slower service. It is, therefore, important to produce reasonable estimates for the runtime of your jobs. Experience running on the front end machines can help you to do this.

The second directive – `#PBS -l nodes=1:ppn=8` – requests that the job be run on 8 processors. Note that the number of nodes should always be 1. You may have noticed that there is no explicit memory size specification in this job script. This is because *Rachel* always associates 3.7 GB memory with each processor. Therefore, when the job script requests 8 processors, 3.7GB * 8 = 29.6GB memory is allocated for the job implicitly.

The third directive – `#PBS -j oe` – tells PBS to combine stdout and stderr which typically makes it easier to debug your program. When a job is run via PBS, the output is automatically redirected to a file uniquely associated with the job. That file will appear in the directory the job is submitted in.

After you have a job script, make sure it marked executable

```
> chmod 755 yourscript.job
```

and then it can be submitted to PBS with the `qsub` command

```
> qsub yourscript.job
```

You may also find the `qstat` and `qdel` commands useful for monitoring your jobs and removing jobs from the queue, respectively. To learn more about these commands, look at their respective man pages. You can also learn more about PBS by looking at the PBS man page:

```
> man pbs
```

5 Debugging

Both `cobalt` and `rachel` have tools available for debugging. Both of these include a debugger, although they don't have the same one. `gdb`, which you should be familiar with from 213, is available on `cobalt` while `rachel` provides access to `dbx`, which you may not be familiar with. They provide simular functionality, but the commands in each have different names. Man pages for these are available on their respective systems, however if these are inadequate, we are happy to help you learn how to use these tools.