User Tutorial

SciNet HPC Consortium — Compute Canada — July 3, 2014

1 Introduction

SciNet is a consortium for High-Performance Computing made up of researchers at the University of Toronto and its associated hospitals. It is part of Compute/Calcul Canada, as one of seven consortia in Canada providing HPC resources to their own academic researchers, other users in Canada and international collaborators.

SciNet runs a unix-type environment. Users not familiar with such an environment, please read appendix A.

Any qualified researcher at a Canadian university can get a SciNet account through this two-step process:

- Register for a Compute Canada Database (CCDB) account at ccdb.computecanada.org
- Non-faculty need a sponsor (supervisors CCRI number), who has to have a SciNet account already.
- Login and apply for a SciNet account (click Apply beside SciNet on the "Consortium Accounts" page)

For groups who need more than the default amount of resources, the PI must apply for it through the competitively awarded account allocation process once a year, in the fall. Without such an allocation, a user may still use up to 32 nodes (256 cores) of the General Purpose Cluster at a time at low priority.

The SciNet wiki, wiki.scinethpc.ca , contains a wealth of information about using SciNet's systems and announcements for users, and also shows the current system status. Other forms of support:

- Users are kept up-to-date with development and changes on the SciNet systems through a monthly email.
- Monthly SciNet User Group lunch meetings including one or more TechTalks.
- Classes, such as the Intro to SciNet, 1-day courses on parallel I/O, (parallel) programming, and a full-term graduate course on scientific computing. The courses web site is support.scinet.utoronto.ca/courses
- Past lecture slide and some videos can be found on the Tutorials and Manuals page.
- Usage reports are available on the SciNet portal portal.scinet.utoronto.ca .
- If you have problems, questions, or requests, and you couldn't find the answer in the wiki, send an e-mail to support@scinet.utoronto.ca. The SciNet team can help you with wide range problems such as more efficient setup of your runs, parallelizing or optimizing your code, and using and installing libraries.

SciNet has currently two main production clusters available for users, and two smaller ones:

1.1 General Purpose Cluster (GPC)

- 3864 nodes with 8 cores each (two 2.66GHz quad-core Intel Xeon 5500 Intel processors)
- HyperThreading lets you run 16 threads per node efficiently.
- 16GB RAM per node
- Running CentOS 6.2 (a linux distribution derived from Red Hat).
- Interconnected by InfiniBand: non-blocking DDR on 1/4 of the nodes, 5:1 blocking QDR on the rest
- 328 TFlops $\longrightarrow$ #16 on the June 2009 *TOP500* list of supercomputer sites (#1 in Canada)
- Moab/Torque schedules by node with a maximum wall clock time of 48 hours.

1.2 Tightly Coupled System (TCS)

- 104 nodes with 32 cores (16 dual-core 4.7GHz POWER6 processors).
- Simultaneous MultiThreading allows two tasks to be very efficiently bound to each core.
- 128GB RAM per node
- Running AIX 5.3L operating system.
- Interconnected by full non-blocking DDR InfiniBand
- 62 TFlops $\longrightarrow$ #80 on the June 2009 *TOP500* list of supercomputer sites
- Moab/LoadLeveler schedules by node. The maximum wall clock time is 48 hours.
- Access to this highly specialized machine is not enabled by default. For access, email us explaining the nature of your work. Your application should scale well to 64 processes/threads to run on this system.

1.3 Accelerator Research Cluster (ARC)

- Eight GPU devel nodes and four NVIDIA Tesla M2070. Per node:
  - 2 quad-core Intel Xeon X5550 2.67GHz
  - 48 GB RAM
  - 2 GPUs with CUDA capability 2.0 (Fermi) each with 448 CUDA cores @ 1.15GHz and 6 GB of RAM.
- Interconnected by DDR InfiniBand
- 16.48 TFlops from the GPUs in single precision
- 8.24 TFlops from the GPUs in double precision
- Running CentOS 6.0 linux.
- Moab/Torque schedules jobs similarly as on the GPC, with a maximum wall clock time of 48 hours.
- Access disabled by default. Email us if you want access.

1.4 Power 7 System (P7)

- Five nodes with four 8-core 3.3 GHz Power7 processors
- 128 GB RAM per node
- Simultaneous MultiThreading allows four tasks to be very efficiently bound to each core.
- DDR Infiniband interconnect
- 4.2 TFlops theoretical.
- Running Red Hat Enterprise Linux 6.0.
- LoadLeveler schedules by node. The maximum wall clock time is 48 hours.
- Accessable to TCS users

1.5 Storage space and data management

- 1790 1TB SATA disk drives, for a total of 1.4 PB of storage
- Two DCS9900 couplets, each delivering 4-5GB/s read/write access to the drives
- Single *GPFS* file system on the TCS, GPC and ARC.
- I/O shares the same InfiniBand network used for parallel jobs on the clusters.
- HPSS: a tape-backed storage expansion solution, only for users with a substantial storage allocation.

The storage at SciNet is divided over different file systems:

file system	quota	block-size	time-limit	backup	devel	comp
/home	10GB	256kB	indefinite	yes	read/write	read-only
/scratch	20TB or 1M files	4MB	3 months	no	read/write	read/write
/project	by allocation	4MB	indefinite	yes	read/write	read/write
/archive (on HPSS)	by allocation	-	indefinite	no	on HPSS only	

Of these four, /home and /scratch are the two most important ones, while the access to the latter two is restricted to users with an appropriate storage allocation.

Every SciNet user gets a 10GB directory on /home (called /home/G/GROUP/USER, where GROUP is your group name, G is the first (lower case) letter of the group name and USER is your user name). For example, user rzon in the group scinet has his user directory under /home/s/scinet/rzon. The home directory is regularly backed up. Do not keep many small files on the system. They waste quite a bit of space. On home, with a block size of 256kB, you can at most have 40960 files no matter how small they are, so you would run out of disk quota quite rapidly with many small files.

On the compute nodes of the GPC /home is mounted read-only; thus GPC jobs can read files in /home but cannot write to files there. /home is a good place to put code, input files for runs, and anything else that needs to be kept to reproduce runs. In addition, every SciNet user gets a directory in /scratch (/scratch/G/GROUP/USER). Note: the environment variables $HOME and $SCRATCH contain the location of your home and scratch directory, respectively.

In $SCRATCH, up to 20TB could be stored — although there is not enough room for each user to do this! In addition, there is a limit of one million files for $SCRATCH. Because $HOME is read-only on compute nodes, $SCRATCH is where jobs would normally write their output. *Note that there are NO backups of scratch.* Furthermore, scratch is purged routinely. The current policy is that files which have not been accessed for over three months will be deleted, with users getting a two-week notice on what files are to be deleted.

File transfers to and from SciNet

To transfer less than 10GB to SciNets, you can use the login nodes. The login nodes are visible from outside SciNet, which means that you can transfer data to and from your own machine to SciNet using scp or rsync starting from SciNet or from your own machine. The login node has a cpu time out of 5 minutes, which means that even if you tried to transfer more than 10GB, you would probably not succeed.

Large transfers of data (more than 10GB) to or from SciNet are best done from the datamover1 or datamover2 node. From any of the interactive SciNet nodes, one can ssh to datamover1 or datamover2. These machines have the fastest network connection to the outside world (by a factor of 10; a 10Gb/s link as vs 1Gb/s). Datamover2 is sometimes under heavy load for sysadmin purposes, but datamover1 is for user traffic only.

Transfers must be originated from the datamovers; that is, one can not copy files from the outside world directly to or from a datamover node; one has to log in to that datamover node and copy the data to or from the outside network. Your local machine must be reachable from the outside, either by its name or its IP address. If you are behind a firewall or a (wireless) router, this may not be possible. You may need to ask your system administrator to allow datamover to ssh to your machine.

I/O on SciNet systems

The compute nodes do not contain hard drives, so there is no local disk available to use during your computation. The available disk space, i.e., the home and scratch directories, are all part of the GPFS file system which runs over the network. GPFS is a high-performance file system which provides rapid reads and writes to large data sets in parallel from many nodes. As a consequence of this design, however, ***it performs quite poorly at accessing data sets which consist of many, small files.*** Furthermore, the file system is a shared resource. Creating many small files or opening and closing files with small reads, and similar inefficient I/O practices hurt your job's performance, and are felt by other users too.

Because of this file system setup, you may well find that you have to reconsider the I/O strategy of your program. The following points are very important to bear in mind when designing your I/O strategy

- Do not read and write lots of small amounts of data to disk. Reading data in from one 4MB file can be enormously faster than from 100 40KB files.
- Unless you have very little output, make sure to write your data in binary.
- Having each process in an MPI run write to a file of its own is not a scalable I/O solution. A directory gets locked by the first process accessing it, so the other processes have to wait for it. Not only has the code just become considerably less parallel, chances are the file system will have a time-out while waiting for your other processes, leading your program to crash mysteriously. Consider using MPI-IO (part of the MPI-2 standard), NetCDF or HDF5, which allow files to be opened simultaneously by different processes. You could also use dedicated process for I/O to which all other processes send their data, and which subsequently writes this data to a single file.
- If you must read and write a lot to disk, consider using the ramdisk. On the GPC, you can use up to 11GB of a compute node's ram like a local disk. This *will* reduce how much memory is available for your program. The ramdisk can be accessed using /dev/shm. Anything written to this location that you want to preserve must be copied back to the scratch file system as /dev/shm is wiped after each job.

1.6 Acknowledging SciNet

In publications based on results from SciNet computations, please use the following acknowledgment:

> *Computations were performed on the <systemname> supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.*

where you replace <systemname> by GPC or TCS. Also please cite the SciNet datacentre paper:

> Chris Loken *et al.*, *SciNet: Lessons Learned from Building a Power-efficient Top-20 System and Data Centre*, J. Phys.: Conf. Ser. **256**, 012026 (2010).

In any talks you give, please feel free to use the SciNet logo, and images of GPC, TCS, and the data centre. These can be found on the wiki page Acknowledging SciNet .

We are very interested in keeping track of SciNet-powered publications! We track these for our own interest, but such publications are also useful evidence of scientific merit for future resource allocations as well. Please email details of any such publications, along with PDF preprints, to support@scinet.utoronto.ca.

2 Using the GPC

Using SciNet's resources is significantly different from using a desktop machine. The rest of this document is will guide you through the process of using the GPC first (as most users will make use of that system), while details of the other systems are given afterwards.

A reference sheet for the GPC can be found in Appendix B at the end of this document.

Computing on SciNet's clusters is done through a batch system. In its simplest form, it is a four stage process:

1. Login with ssh to the login nodes and transfer files.
   These login nodes are gateways, you do not run or compile on them!
2. Ssh to one of the development nodes gpc01-04, where you load modules, compile your code and write a script for the batch job.
3. Move the script, input data, etc. to the scratch disk, *as you cannot write to your home directory from the compute nodes*. Submit the job to a queuing system.
4. After the scheduler has run the job on the compute nodes (this can take some time), and the job is completed, deal with the output of the run.

2.1 Login

Access to the SciNet systems is via secure shell (`ssh`) only. Ssh to the gateway `login.scinet.utoronto.ca`:

```
$ ssh -X -l <username> login.scinet.utoronto.ca
```

The `-X` flag is there to set up X forwarding so that you could run Xwindows applications, such as graphical editors and debuggers. If the `-X` flag does not work for you try the less secure `-Y` flag.
The login nodes are a front end to the data centre, and are not part of the GPC. For anything but small file transfer and viewing your files, the next step is to ssh in to one of the devel nodes gpc01,...,gpc04, e.g.

```
$ ssh -X gpc03
```

These develop nodes have the same architecture as the compute nodes, but with more memory.

- The SciNet firewall monitors for too many connections, and will shut down access (including previously connections) from your IP address if more than four connection attempts are made within a few minutes. In that case, you will be locked out of the system for an hour. Be patient in attempting new logins!
- More about ssh and logging in from Windows can be found on the wiki page Ssh .

2.2 Software modules

Most software and libraries on the GPC have to be loaded using the `module` command. This allows us to keep multiple versions for different users, and it allows users to easily switch between versions. The module system sets up environment variables (PATH, LD_LIBRARY_PATH, etc.).

Basic usage of the `module` command is as follows

`module load <module-name>`	to use particular software
`module unload <module-name>`	to stop using particular software
`module switch <module1> <module2>`	to unload module1 and load module2
`module purge`	to remove all currently loaded modules
`module avail`	to list available software packages (+ all versions)
`module list`	to list currently loaded modules in your shell

Although it may seem convenient, you should not load modules in the file `.bashrc` in your home directory, as this file is read by any script that your run (including job scripts and the mpi wrapper scripts mpicc, mpif90, etc.), and 'module load' commands in this file can lead to surprising and hard to debug errors. It is better to load modules explicitly on the command line when compiling and testing, and then load them explicitly in your job scripts.

Many modules are available in several versions (e.g. `intel/12` and `intel/12.1.3`). When you load a module with its short name (the part before the slash '/',e.g., `intel`), you get the most recent and recommended version of that library or piece of software. In general, you probably want to use the short module name, especially since we may upgrade to a new version and deprecate the old one. By using the short module name, you ensure that your existing `module load` commands still work. However, for reproducibility of your runs, record the full names of loaded modules.

Library modules define the environment variables pointing to the location of library files, include files and the base directory for use Makefiles. The names of the library, include and base variables are as follows:

```
SCINET_[shortmodulename]_LIB
SCINET_[shortmodulename]_INC
SCINET_[shortmodulename]_BASE
```

That means that to compile code that uses that package you add the following flags to the command line

```
-I${SCINET_[shortmodulename]_INC}
```

while to the link command, you have to add

```
-L${SCINET_[shortmodulename]_LIB}
```

before the necessary link flags (-l...).

- On July 3, 2014, the module list for the GPC contained:
  `intel, gcc, intelmpi, openmpi, nano, emacs, xemacs, autoconf, cmake, git, scons, svn, ddt, ddd, gdb, mpe, openspeedshop, scalasca, valgrind, padb, grace, gnuplot, vmd, ferret, ncl, ROOT, paraview, pgplot, ImageMagick,netcdf, parallel-netcdf, ncview, nco, udunits, hdf4, hdf5, encfs, gamess, nwchem, gromacs, cpmd,`

`blast, amber, gdal, meep, mpb, R, petsc, boost, gsl, fftw, intel, extras, clog, gnu-parallel, guile, java, python, ruby, octave, gotoblas, erlang, antlr, ndiff, nedit, automake, cdo, upc, inteltools, cmor, ipm, cxxlibraries, Xlibraries, dcap, xml2, yt`

- Mathematical libraries supporting things like BLAS and FFT are part of modules as well: The Intel's Math Kernel Library (MKL) is part of the intel module and the goto-blas modules, and there are separate fftw modules (although mkl supports this as well).
- Other commercial packages (MatLab, Gaussian, IDL,...) are **not** available for licensing reasons. But Octave, a highly MatLab-compatible open source alternative, is available as a module.
- A current list of available software is maintained on the wiki page Software and Libraries .

2.3 Compiling

The GPC has compilers for C, C++, Fortran (up to 2003 with some 2008 features), Co-array Fortran, and Java. We will focus here on the most commonly used languages: C, C++, and Fortran.

It is recommended that you compile with the Intel compilers, which are `icc`, `icpc`, and `ifort` for C, C++, and Fortran. These compilers are available with the module `intel` (i.e., say `module load intel` on the command line and in your job script). If you really need the GNU compilers, recent versions of the GNU compiler collection are available as modules, with `gcc,g++,gfortran` for C, C++, and Fortran. The ol' g77 is not supported, but both `ifort` and `gfortran` are able to compile Fortran 77 code.

Optimize your code for the GPC machine using at least the following compilation flags `-O3 -xhost`, e.g.

```
$ ifort -O3 -xhost example.f   example
$ icc   -O3 -xhost example.c   example
$ icpc  -O3 -xhost example.cpp example
```

(the equivalent flags for GNU compilers are `-O3 -march=native`).

Compiling OpenMP code

To compile programs using shared memory parallel programming using OpenMP, add `-openmp` to the compilation and linking commands, e.g.

```
$ ifort -openmp -O3 -xhost omp_example.f   -o omp_example
$ icc   -openmp -O3 -xhost omp_example.c   -o omp_example
$ icpc  -openmp -O3 -xhost omp_example.cpp -o omp_example
```

Compiling MPI code

Currently, the GPC has following MPI implementations installed:

1. Open MPI, in module `openmpi` (default version: 1.4.4)
2. Intel MPI, in module `intelmpi` (default versions: 4.0.3)

You can choose which one to use with the module system, but you are recommended to stick to OpenMPI unless you have a good reason not to. Switching between MPI implementations is not always obvious.

Once an mpi module is loaded, MPI code can be compiled using `mpif77/mpif90/mpicc/mpicxx`, e.g.,

```
$ mpif77 -O3 -xhost mpi_example.f   -o mpi_example
$ mpif90 -O3 -xhost mpi_example.f90 -o mpi_example
```

```
$ mpicc  -O3 -xhost mpi_example.c   -o mpi_example
$ mpicxx -O3 -xhost mpi_example.cpp -o mpi_example
```

These commands are wrapper (bash) scripts around the compilers which include the appropriate flags to use MPI libraries.

Hybrid MPI/OpenMP applications are compiled with same commands, but with openmp flags, e.g.

```
$ mpif77 -openmp -O3 -xhost hybrid_example.f   -o hybrid_example
$ mpif90 -openmp -O3 -xhost hybrid_example.f90 -o hybrid_example
$ mpicc  -openmp -O3 -xhost hybrid_example.c   -o hybrid_example
$ mpicxx -openmp -O3 -xhost hybrid_example.cpp -o hybrid_example
```

For hybrid OpenMP/MPI code using Intel MPI, add the compilation flag `-mt_mpi` for full thread-safety.

2.4 Testing and debugging

Apart from compilation, the devel nodes may also be used for short, small scale test runs (on the order of a few minutes), although there is also a specialized queue for that (see next section). It is important to test your job's requirements and scaling behaviour before submitting a large scale computations to the queuing system. Because the devel nodes are used by "everyone" who needs to use the GPC, be considerate.

To run a short test of a serial (i.e., non-parallel) program, simply type from a devel node

```
$ ./<executable> [arguments]
```

Serial *production* jobs must be bunched together to use all 8 cores; see below.

To run a short 4-thread OpenMP run on the GPC, type

```
$ OMP_NUM_THREADS=4 ./<executable> [arguments]
```

To run a short 4-process MPI run on a single node, type

```
$ mpirun -np 4 ./<executable> [arguments]
```

For debugging, we highly recommend DDT, Allinea's graphical parallel debugger. It is available in the module ddt. DDT can handle serial, openmp, mpi, as well as gpu code (useful on the ARC system). To enable debugging in your code, you have to compile it with the flags `-g`, and you probably want to dial down the optimization level to `-O1` or even `-O0` (no optimization). After loading the ddt module, simply start ddt with `ddt`, and follow the graphical interface's questions.

Larger, multi-node debugging debugging should be done on the dedicated debug nodes which can be accessed through the debug queue. The queuing system will be explained in more detail below. As a preliminary example, to start a thiry-minute ddt debugging session on three nodes for a 24 process mpi program, you can do the following from a gpc devel node:

```
$ qsub -l nodes=3:ppn=8,walltime=30:00 -X -I -q debug
qsub:  waiting for job <jobid>.gpc-sched to start
...wait until you get a prompt...
qsub:  job <jobid>.gpc-sched ready
```

```
--- --- --- --- --- --- --- --- --- ---
Begin PBS Prologue <datestamp>
Job ID:        <jobid>.gpc-sched
Username:      <username>
Group:         <groupname>
Nodes:         <node1> <node2> <node3>
End PBS Prologue <datestamp>
--- --- --- --- --- --- --- --- --- ---
$ module load <your-libraries>
$ module load Xlibraries
$ module load ddt
$ ddt
```
...follow the ddt menus on screen...

Note: the GNU debugger (`gdb`), the Intel debugger (`idbc/idb`) and a graphical debugger called ddd, are available on the GPC as well.

2.5 Running jobs though the Moab queuing system

To run a job on the compute nodes, it must be submitted to a queue. The queuing system used on the GPC is based around the Moab Workload Manager, with Torque (PBS) as the back-end resource manager. The queuing system will send the jobs to the compute nodes. It schedules by nodes, so you cannot request e.g. a two-core job. It is the user's responsibility to make sure that the node is used efficiently, i.e., all cores on a node are kept busy.

Job submission starts with a script that specifies what executable to run, from which directory to run it, on how many nodes, with how many threads, and for how long. A job script can be submitted to a queue with

```
$ qsub script.sh
```

This submits the job to the 'batch' queue and assigns the job a `jobid`. There are two other queues on the GPC, 'largemem' and 'debug', whose use will be explained below.

Once the job is incorporated into the queue (which can take a minute), you can use:

```
$ showq
```

to show the all jobs in the queue. To just see your jobs, type

```
$ showq -u <username>
```

and to see only your running jobs, you can use

```
$ showq -r -u <username>
```

There are also job-specific commands such as `showstart <jobid>`, `checkjob <jobid>`, `canceljob <jobid>`, to estimate when a job will start, to check the status of your job, and to cancel a job (we recommend not using the torque commands qdel, etc.).

Jobs scripts serve a dual purpose:

1. they specify what resources your job needs, and for how long;
2. and they contain the command to be executed (on the first node).

The first purpose is accomplished without interfering with the second by using special command lines starting with `#PBS` , typically at the top of the script. After the `#PBS`, one can specify resource options. Only one is mandatory:

- `-l`: specifies requested nodes and time, e.g.
  ```
  -l nodes=1:ppn=8,walltime=1:00:00
  -l nodes=2:ppn=8,walltime=1:00:00
  ```
  The ":ppn=8" part is mandatory as well, since scheduling goes by 8-core node.
  To make your jobs start faster, reduce the requested time (`walltime`) to be closer to the estimated run time (perhaps adding about 10 percent to be sure). Shorter jobs are scheduled sooner than longer ones.

Other resource options are

- `-N` gives your job a name (so it's easily identified in the queue).
- `-q`: specifies the queue, e.g.
  ```
  -q largemem
  -q debug
  ```
  This is not necessary for the regular 'batch' queue, which is the default.
- `-o`: change the default name of the file to contain any output to standard output from your job.
- `-e`: change the default name of the file to contain any output to standard error from your job.
  Note: the output and error files are not available until the job is finished.

After the resource option, one specifies the commands to be run, just as in a regular shell script. Your default shell used for the command line as well as for scripts is bash, but it is possible to use csh as well for job scripts. The shell that should execute a script is specified in the very first line of the script, and should read either `#!/bin/bash` or `#!/bin/csh`.
Runs are to be executed from $SCRATCH, because your $HOME is read-only on the compute nodes. The easiest way to ensure that your runs starts from a directory that you have write access to, is to have copy all the files necessary for your run to a work directory under $SCRATCH, to invoke the `qsub` command from that directory, and to have as the first line of your job script:

```
cd $PBS_O_WORKDIR
```

If you don't, the scheduler will start your job in $HOME. The environment variable `PBS_O_WORKDIR` is set by the scheduler to the submission directory.

Here is a simple example of a job script for an openmp application to run with 16 threads:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
#PBS -l walltime=1:00:00
#PBS -N simple-openmp-job
module load intel
cd $PBS_O_WORKDIR
./openmp_example
```

The reason this job uses 16 threads although the node has 8 cores, is that the cpus on the GPC nodes have HyperThreading enabled. HyperThreading allows efficient switching between tasks, and makes it seem to the operating system like there are 16 logical cpus rather than 8 on each node. Using this requires no changes to the code, only running 16 rather than 8 tasks on the node. By default, OpenMP applications will use all

logical cpus, i.e., 16 threads, unless the environment variable `OMP_NUM_THREADS` is set do a different value. E.g., to disable HyperThreading for OpenMP application, use `export OMP_NUM_THREADS=8`. In contrast, MPI application always need to be told explicitly how many processes to use. Thus, to use HyperThreading for a single node mpi job, it is enough to set `-np 16` instead of `-np 8` (hybrid mpi/openmp applications require a bit more care, see example 2.5.3 below) .

Monitoring jobs

Once submitted, `checkjob <jobid>` can give you information on a job (after a short delay). It will tell you if your job is *runnning*, *idle*, or *blocked*.

Blocked jobs are usually just jobs that the scheduler will not consider yet because the user of the group that the user belongs to has reached their limit on the number of jobs or the number of cores to be used simultaneously. Once running jobs from that user or group finish, these jobs will be moved from the *blocked* list to the *idle* list.

Idle in this context means that your job will be considered to run by the scheduler, based on a mechanism called *fair-share*. In this mechanism, the main factors for when a job will run are the priority of your group (based on the group's allocation and previous usage) and how long the job has been in the idle queue. In addition, the scheduler performs backfilling, which means that if, to schedule a large multinode job, it has to keep some nodes unused for a while, it will put jobs lower on the *idle* list on these reserved nodes that fit within that time and node slot. This may seem like jumping the queue, but it maximizes utilization of the GPC without delaying the start time of other queued jobs. Keep in mind, however, that start times of jobs in the queue can change if a user with a higher priority submits a job. Barring that, an estimate of the start time of an idle job can be obtained using

```
$ showstart <jobid>
```

Once the job is running, you can still use `checkjob`, but you can also ssh to any of the nodes on which it is running (listed by `checkjob <jobid>`). This allows you to monitor the progress and behaviour of your job in more detail, using e.g. the `top` command. It also allows you to check the output and error messages from your job on the fly. They are located in the directory `/var/spool/torque/spool`, which exists only on the (first) node assigned to your job.

After a job has finished, these files with standard output and error are copied to the submission directory. By default the files are named `<jobname>.o<jobid>` and `<jobname>.e<jobid>`, respectively. In addition to any output that your application write to standard output, the .o file contains PBS information such as how long your job took and how much memory it required. The .e file contains any error messages. If something goes wrong with a job of yours, inspect these files carefully for hints on what went wrong.

Large memory jobs

There are 84 GPC nodes with 32 GB of memory instead of 16GB, which are of the same architecture as the regular GPC compute and devel nodes. To request these, you use the regular batch queue, but add a flag "m32g" to the node request, i.e.

```
#PBS -q batch
#PBS -l nodes=1:ppn=8:m32g
```

In addition, there are two GPC nodes with 128GB ram and 16 cores, intended for one-off data analysis or visualization that may require such resources. Because these have a different architecture than the rest of the GPC nodes, they have there own queue, and you have to compile code for these nodes on one of the GPC

devel nodes without the `-xHost` compilation flag.

To request a large memory nodes for a job, specify

```
#PBS -q largemem
#PBS -l nodes=1:ppn=16
```

in your job script (although ppn=8 will be accepted too).

Interactive jobs

Most PBS parameters in `#PBS` lines can also be given as parameters to qsub, but it is advisable to keep them in the job script so you have a record on how you submitted your job. The exception is when you request an interactive job, which is accomplished by giving the `-I` flag to qsub, e.g.

```
$ qsub -l nodes=1:ppn=8,walltime=1:00:00 -X -q debug -I
```

In addition to a interactive job, this requests the debug queue instead of the regular batch queue. This gives access to a small number of reserved debug nodes. These nodes are the same as the usual compute nodes, but jobs have a higher turnover in the debug queue than in the regular queue — i.e., they start sooner — because only short jobs are allowed. The debug queue is ideal for short multinode tests and for debugging. Finally, the `-X` flags in the above qsub command requests that X is forwarded, which is essential if you are going to use ddt to debug. Note that this will only work if you gave the `-X` (or `-Y`) flag to each ssh command (i.e., at a minimum from your machine to login.scinet.utoronto.ca, and from the login node to gpc01..gpc04).

The interactive flag `-I` works in combination with the `largemem` and `batch` queues as well. However, its use with the `batch` queue is discouraged, as it may take a very long time for you to get a prompt.

Queue limits

queue	min.time	max.time	max jobs	max cores
batch	15m	48h	32, 1000 w/allocation	256, 8000 w/allocation
debug		2h/30m	1	walltime dependent, between 16 and 64
largemem	15m	48h	1	16 (32 threads)

Serial jobs on the GPC

SciNet is a parallel computing resource, and our priority will always be parallel jobs. Having said that, if you can make efficient use of the resources using serial jobs and get good science done, that's acceptable too. There is however no queue for serial jobs, so if you have serial jobs, you will have to bunch them together to use the full power of a node (Moab schedules by node).

The GPC nodes each have 8 processing cores, and making efficient use of these nodes means using all eight cores. As a result, we'd like users to run multiples of 8 jobs at a time. The easiest way to do this is to bunch the jobs in groups of 8 that will take roughly the same amount of time.

It is important to group the programs by how long they will take. If one job takes 2 hours and the rest running on the same node only take 1, then for one hour 7 of the 8 cores on the GPC node are wasted; they are sitting idle but are unavailable for other users, and the utilization of this node is only 56 percent.

You should have a reasonable idea of how much memory the jobs require. The GPC compute nodes have about 14GB in total available to user jobs running on the 8 cores. So the jobs have to be bunched in ways that will fit into 14GB. If that's not possible, one could in principle run fewer jobs so that they do fit.

Another highly recommended method is using GNU parallel, which can do the load balancing for you. See the wiki page User Serial .

2.5.1 Example job script: OpenMP job (without hyperthreading)

```
#!/bin/bash
#PBS -l nodes=1:ppn=8,walltime=6:00:00
#PBS -N openmp-test
module load intel
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=8
./openmp_example
```

2.5.2 Example job script: MPI job on two nodes

```
#!/bin/bash
#PBS -l nodes=2:ppn=8,walltime=6:00:00
#PBS -N mpi-test
module load intel openmpi
cd $PBS_O_WORKDIR
mpirun -np 16 ./mpi_example
```

- When using hyperthreading, add the parameter `--mca mpi_yield_when_idle 1` to mpirun.
- For MPI code using Intel MPI with hyperthreading, add `-genv I_MPI_SPIN_COUNT 1` to mpirun.

2.5.3 Example job script: hybrid MPI/OpenMP job

```
#!/bin/bash
#PBS -l nodes=3:ppn=8,walltime=6:00:00
#PBS -N hybrid-test
module load intel openmpi
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=4
mpirun --bynode -np 6 ./hybrid_example
```

- The `--bynode` option is essential; without it, MPI processes bunch together in eights on each node.
- For Intel MPI, that option needs to be replaced by `-ppn 2`.
- In addition, for hybrid OpenMP/MPI code using Intel MPI, make sure you have `export I_MPI_PIN_DOMAIN=omp` in .bashrc or in the job script.

2.5.4 Example job script: bunch of serial runs

```
#!/bin/bash
#PBS -l nodes=1:ppn=8,walltime=6:00:00
#PBS -N serialx8-test
module load intel
cd $PBS_O_WORKDIR
(cd jobdir1; ./dojob1) &
(cd jobdir2; ./dojob2) &
(cd jobdir3; ./dojob3) &
(cd jobdir4; ./dojob4) &
```

```
(cd jobdir5; ./dojob5) &
(cd jobdir6; ./dojob6) &
(cd jobdir7; ./dojob7) &
(cd jobdir8; ./dojob8) &
wait #  Without this, the job will terminate immediately, killing the 8 runs you just started
```

Note: Another highly recommended method is using GNU parallel, which can do the load balancing for you. See the wiki page User Serial .

3 Using the Other SciNet Systems

3.1 Login

As the login nodes are a front end to the data centre, and are not part of any two compute cluster, For anything but small file transfer and viewing your files, you next login to the TCS, ARC or P7 through their development nodes (tcs01 or tcs02 for TCS, arc01 for ARC, and p701 for P7, respectively).

3.2 Software modules

As on the GPC, most software and libraries have to be loaded using the `module` command.

- On July 3, 2014, the module list for the ARC contained:
  `intel, gcc, intelmpi, openmpi, cuda, nano, emacs, xemacs, autoconf, cmake, git, scons, svn, ddt, ddd, gdb, mpe, openspeedshop, scalasca, valgrind, padb, grace, gnuplot, vmd, ferret, ncl, ROOT, paraview, pgplot, ImageMagick,netcdf, parallel-netcdf, ncview, nco, udunits, hdf4, hdf5, encfs, gamess, nwchem, gromacs, cpmd, blast, amber, gdal, meep, mpb, R, petsc, boost, gsl, fftw, intel, extras, clog, gnu-parallel, guile, java, python, ruby, octave, gotoblas, erlang, antlr, ndiff, nedit, automake, cdo, upc, inteltools, cmor, ipm, cxxlibraries, Xlibraries, dcap, xml2, yt`
- The module list for the TCS contains :
  `upc, xlf, vacpp, mpe, scalasca, hdf4, hdf5, extras, netcdf, parallel-netcdf, nco, gsl, antlr, ncl, ddt, fftw, ipm`
- The IBM compilers are standard available on the TCS and do not require a module to be loaded, although newer versions may be installed as modules.
- Math software supporting things like BLAS and FFT is either standard available, or part of a module: on the ARC (as on the GPC), there is the Intel's Math Kernel Library (MKL) which is part of the intel module and the goto-blas modules, while on the TCS, IBM's ESSL high performance math library is standard available.
- A current list of available software is maintained on the wiki page Software and Libraries .

3.3 Compiling

TCS compilers

The TCS has compilers for C, C++, Fortran (up to 2003), UPC, and Java. We will focus here on the most commonly used languages: C, C++, and Fortran.

The compilers are `xlc,xlC,xlf` for C, C++, and Fortran compilations. For OpenMP or other threaded applications, one has to use 're-entrant-safe' versions `xlc_r,xlC_r,xlf_r`. For MPI applications, `mpcc,mpCC,mpxlf` are the appropriate wrappers. Hybrid MPI/OpenMP applications require `mpcc_r,mpCC_r,mpxlf_r`.

We strongly suggest the compilation flags

```
-O3 -q64 -qhot -qarch=pwr6 -qtune=pwr6
```

For OpenMP programs, you should add

```
-qsmp=omp
```

In the link command, we suggest using

```
-q64 -bdatapsize:64k -bstackpsize:64k
```

supplemented by

```
-qsmp=omp
```

for OpenMP programs.

To use the full C++ bindings of MPI (those in the MPI namespace) with the IBM c++ compilers, add `-cpp` to the compilation line. If you're linking several c++ object files, add `-bh:5` to the link line.

P7 compilers

Compilation for the P7 should be done with the IBM compilers on the devel node p701. The compilers are `xlc,xlC,xlf` for C, C++, and Fortran compilations and become accessible by loading the modules vacpp and xlf, respectively. For OpenMP or other threaded applications, one has to use 're-entrant-safe' versions `xlc_r,xlC_r,xlf_r`. For MPI applications, `mpcc,mpCC,mpxlf` are the appropriate wrappers. Hybrid MPI/OpenMP applications require `mpcc_r,mpCC_r,mpxlf_r`. We suggest the compilation flags

```
-O3 -q64 -qhot -qarch=pwr7 -qtune=pwr7
```

For OpenMP programs, add, for with compilation and linking,

```
-qsmp=omp
```

ARC compilers

To compile cuda code for runs on the ARC, you log in from `login.scinet.utoronto.ca` to the devel node:

```
$ ssh -X arc01
```

The ARC has the same compilers for C, C++, Fortran as the GPC, and in addition has the PGI and NVIDIA cuda compilers for GPGPU computing. To use the cuda compilers, you have to load a cuda module. The current default version of cuda is 4.1, but modules for cuda 3.2, 4.0, 4.2, 5.0 and 5.5, are installed as well. The cuda c/c++ compiler is called nvcc. To optimize your code for the ARC architecture (and access all their cuda capabilities), use at least the following compilation flags

```
-O3 -arch=sm_20
```

To use the PGI compiler, you have to

```
$ module load gcc/4.4.6 pgi
```

The compilers are pgfortran, pgcc and pgcpp. These compilers support CUDA Fortran and OpenACC using the `-acc -ta=nvidia -Mcuda=4.0` options.

3.4 Testing and debugging

TCS testing and debugging

Short test runs are allowed on devel nodes if they only don't use much memory and only use a few cores.

To run a short 8-thread OpenMP test run on tcs02:

```
$ OMP_NUM_THREADS=8 ./<executable> [arguments]
```

To run a short 16-process MPI test run on tcs02:

```
$ mpiexec -n 16 ./<executable> [arguments] -hostfile <hostfile>
```

- `<hostfile>` should contain as many of the line `tcs-f11n05` or `tcs-f11n05` (depending on whether you're on tcs01 or tcs02) as you want processes in the MPI run.
- Furthermore, the file `.rhosts` in your home directory has to contain a line with `tcs-f11n06`.

The standard debugger on the TCS is called `dbx`. The DDT debugger is available in the module ddt.

P7 testing and debugging

This works as on the TCS, but with different hostfile with up to 128 lines containing `p07n01`.

ARC testing and debugging

Short test runs are allowed on the devel node arc01. For GPU applications, simply run the executable. If you use MPI and/or OpenMP as well, follow the instructions for the GPC. Note that because arc01 is a shared resource, the GPUs may be busy, and so you may experience a lag in you programs starting up.

The NVIDIA debugger for cuda programs is cuda-gdb. The DDT debugger is available in the module ddt.

3.5 Running your jobs

As for the GPC, to run a job on the compute nodes you must submit it to a queue. You can submit jobs from the devel nodes in the form of a script that specifies what executable to run, from which directory to run it, on how many nodes, with how many threads, and for how long. The queuing system used on the TCS and P7 is LoadLeveler, while Torque is used on the ARC. The queuing system will send the jobs to the compute nodes. It schedules by node, so you cannot request e.g. a two-core job. It is the user's responsibility to make sure that the node is used efficiently, i.e., all cores on a node are kept busy.

Examples of job scripts are given below. You can use these example scripts as starting points for your own.

Note that it is best to run from the scratch directory, because your home directory is read-only on the compute nodes. Since the scratch directory is not backed up, copy essential results to $HOME after the run.

TCS queue

For the TCS, there is only one queue:

queue	time(hrs)	max jobs	max cores
verylong	48	2/25	64/800 (128/1600 threads)

Submitting is done from tcs01 or tcs02 with

```
$ llsubmit <script>
```

and `llq` shows the queue.

As for the GPC, the job script is a shell script that serves two purposes:

1. It specifies what the requirements of the job in special comment lines starting with `#@`.
2. Once the required nodes are available (i.e., your job made it through the queue), the scheduler runs the script on the first node of the set of nodes. To run on multiple nodes, the script has to use `poe`. It is also possible to give the command to run as one of the requirement options.

There are a lot of possible settings in a loadleveler script. Instead of writing your own from script, it is more practical to take one of the examples of job scripts given below and adapt it to suit your needs.

- The POWER6 processors have a facility called Simultaneous MultiThreading which allows two tasks to be very efficiently bound to each core. Using this requires no changes to the code, only running 64 rather than 32 tasks on the node. For OpenMP application, see if setting `OMP_NUM_THREADS` and `THRDS_PER_TASK` to a number larger than 32 makes your job run faster. For MPI, increase `tasks_per_node`>32.
- Once your job is in the queue, you can use `llq` to show the queue, and job-specific commands such as `llcancel, llhold, ...`
- *Do not run serial jobs on the TCS!* The GPC can do that, of course, in bunches of 8.
- To make your jobs start sooner, reduce the `wall_clock_limit`)to be closer to the estimated run time (perhaps adding about 10 % to be sure). Shorter jobs are scheduled sooner than longer ones.

P7 queue

The P7 queue is similar to the P6 queue. For differences, see the wiki page on the P7 Linux Cluster .

ARC queue

There is only one queue for the ARC:

arc	min.time	max.time	max cores	max gpus
batch	15m	48h	32	8

This queue is integrated into the gpc queuing system.

You submit to the queue from arc01 or a gpc devel node with

```
$ qsub [options] <script> -q arc
```

where you will replace `<script>` with the file name of the submission script. Common options are:

- `-l`: specifies requested nodes and time, e.g.
  `-l nodes=1:ppn=8:gpus=2,walltime=6:00:00`
  The nodes option is mandatory, and has to contain the "ppn=8" part, since scheduling goes by node, and each node has 8 cores!
  Note that the "gpus" setting is per node, and nodes have 2 gpus.
- It is presently probably best to request a full node.
- `-I` specifies that you want an interactive session; a script is not needed in that case.

Once the job is incorporated into the queue, you can see what's queued with

```
$ showq -w class=arc
```

and use job-specific commands such as `canceljob`.

3.5.1 Example job script: OpenMP job on the TCS

```
#Specifies the name of the shell to use for the job
#@ shell = /usr/bin/ksh
#@ job_name = <some-descriptive-name>
#@ job_type = parallel
#@ class    = verylong
#@ environment = copy_all; memory_affinity=mcm; mp_sync_qp=yes; \
#           mp_rfifo_size=16777216; mp_shm_attach_thresh=500000; \
#           mp_euidevelop=min; mp_use_bulk_xfer=yes; \
#           mp_rdma_mtu=4k; mp_bulk_min_msg_size=64k; mp_rc_max_qp=8192; \
#           psalloc=early; nodisclaim=true
#@ node            = 1
#@ tasks_per_node  = 1
#@ node_usage      = not_shared
#@ output          = $(job_name).$(jobid).out
#@ error           = $(job_name).$(jobid).err
#@ wall_clock_limit= 04:00:00
#@ queue
export target_cpu_range=-1
cd /scratch/<username>/<some-directory>
## To allocate as close to the cpu running the task as possible:
export MEMORY_AFFINITY=MCM
## next variable is for OpenMP
export OMP_NUM_THREADS=32
## next variable is for ccsm_launch
export THRDS_PER_TASK=32
## ccsm_launch is a "hybrid program launcher" for MPI/OpenMP programs
poe ccsm_launch ./example
```

3.5.2 Example job script: MPI job on the TCS

```
#LoadLeveler submission script for SciNet TCS: MPI job
#@ job_name        = <some-descriptive-name>
#@ initialdir      = /scratch/<username>/<some-directory>
#@ executable      = example
#@ arguments       =
#@ tasks_per_node  = 64
#@ node            = 2
#@ wall_clock_limit= 12:00:00
#@ output          = $(job_name).$(jobid).out
#@ error           = $(job_name).$(jobid).err
#@ notification    = complete
#@ notify_user     = <user@example.com>
#Don't change anything below here unless you know exactly
#why you are changing it.
#@ job_type        = parallel
#@ class           = verylong
```

```
#@ node_usage      = not_shared
#@ rset = rset_mcm_affinity
#@ mcm_affinity_options = mcm_distribute mcm_mem_req mcm_sni_none
#@ cpus_per_core=2
#@ task_affinity=cpu(1)
#@ environment = COPY_ALL; MEMORY_AFFINITY=MCM; MP_SYNC_QP=YES; \
#            MP_RFIFO_SIZE=16777216; MP_SHM_ATTACH_THRESH=500000; \
#            MP_EUIDEVELOP=min; MP_USE_BULK_XFER=yes; \
#            MP_RDMA_MTU=4K; MP_BULK_MIN_MSG_SIZE=64k; MP_RC_MAX_QP=8192; \
#            PSALLOC=early; NODISCLAIM=true
# Submit the job
#@ queue
```

3.5.3 Example job script: hybrid MPI/OpenMP job on the TCS

To run on 3 nodes, each with 2 MPI processes that have 32 threads, create a file `poe.cmdfile` containing

```
ccsm_launch ./example
ccsm_launch ./example
ccsm_launch ./example
ccsm_launch ./example
ccsm_launch ./example
ccsm_launch ./example
```

and create a script along the following lines

```
#@ shell = /usr/bin/ksh
#@ job_name = <some-descriptive-name>
#@ job_type = parallel
#@ class    = verylong
#@ environment = COPY_ALL; memory_affinity=mcm; mp_sync_qp=yes; \
#               mp_rfifo_size=16777216; mp_shm_attach_thresh=500000; \
#               mp_euidevelop=min; mp_use_bulk_xfer=yes; \
#               mp_rdma_mtu=4k; mp_bulk_min_msg_size=64k; mp_rc_max_qp=8192; \
#               psalloc=early; nodisclaim=true
#@ task_geometry   = {(0,1)(2,3)(4,5)}
#@ node_usage      = not_shared
#@ output          = $(job_name).$(jobid).out
#@ error           = $(job_name).$(jobid).err
#@ wall_clock_limit= 04:00:00
#@ core_limit      = 0
#@ queue
export target_cpu_range=-1
cd /scratch/<username>/<some-directory>
export MEMORY_AFFINITY=MCM
export THRDS_PER_TASK=32:32:32:32:32:32
export OMP_NUM_THREADS=32
poe -cmdfile poe.cmdfile
wait
```

3.5.4 Example job script: GPU job on the ARC

```
#!/bin/bash
# Torque submission script for SciNet ARC
#PBS -l nodes=1:ppn=8:gpus=2
#PBS -l walltime=6:00:00
#PBS -N gpu-test
module load gcc cuda/5.5
cd $PBS_O_WORKDIR
./example
```

A Brief Introduction to the Unix Command Line

As SciNet systems run a Unix-like environment, you need to know the basics of the Unix command line. With many good Unix tutorials on-line, we will only give some of the most commonly used features.

Unix prompt

The Unix command line is actually a program called a shell. The shell shows a prompt, something like:

```
user@scinet01:/home/g/group/user$ _
```

At the prompt you can type your input, followed by enter. The shell then proceeds to execute your commands.

For brevity, in examples the prompt is abbreviated to `$`.

There are different Unix shells (on SciNet the default is `bash`) but their basic commands are the same.

Files and directories

Files are organized in a directory tree. A file is thus specified as `/<path>/<file>`. The directory separating character is the slash `/`. For nested directories, paths are sequences of directories separated by slashes.

There is a root folder `/` which contains all other folders. Different file systems (hard disks) are mounted somewhere in this global tree. There are no separate trees for different devices.

In a shell, there is always a current directory. This solves the impracticality of having to specify the full path for each file or directory. The current directory is by default displayed in the prompt. You can refer to files in the current directory simply by using their name. You can specify files in another directory by using absolute (as before) or relative paths. For example, if the current directory is `/home/g/group/user`, the file `a` in the directory `/home/g/group/user/z` can be referred to as `z/a`. The special directories `.` and `..` refer to the current directory and the parent directory, respectively.

Home directory

Each user has a home directory, often called `/home/<user>`, where `<user>` is replaced by your user name. On scinet, this directory's location is group based (`/home/<first letter of group>/<group>/<user>`). By default, files in this directory can be seen only by users in the same group. You cannot write to other users' home directory, nor can you read home directories of other groups, unless these have changed the default permissions. The home directory can be referred to using the single character shorthand `~`. On SciNet, you have an additional directory at your disposal, called `/scratch/g/group/<user>`

Commands

Commands typed on the command line are either built-in to the shell or external, in which case the are a file somewhere in the file system. Unless you specify the full path of an external command, the shell has to go look for the corresponding file. The directories that it looks for are stored as a list separated by a colon (`:`) in a variable called PATH. In bash, you can append a directory to this as follows:

```
$ export PATH="$PATH:<newpath>"
```

Common commands

command	function
ls	list the content of the given or of the current directory.
cat	concatenate the contents of files given as arguments (writes to screen)
cd	change the current directory to the one given as an argument
cp	copy a file to another file
man	show the help page for the command given as an argument
mkdir	create a new directory
more	display the file given as an argument, page-by-page
mv	move a file to another file or directory
pwd	show the current directory
rm	delete a file (no undo or trash!)
rmdir	delete a directory
vi	edit a file (there are alternatives, e.g., nano or emacs)
exit	exit the shell

Hidden files and directories

A file or directory of which the name starts with a period (`.`) is hidden, i.e., it will not show up in an `ls` (unless you give the option `-a`). Hidden files and directories are typically used for settings.

Variables

Above we already saw an example of a shell variable, namely, PATH. In the bash shell, variables can have any name and are assigned a value using 'equals', e.g., `MYVAR=15`. To use a variable, you type `$MYVAR`. To make sure commands can use this variable, you have to `export` it, i.e., `export MYVAR`, if it is already set, or `export MYVAR=15` to set it and export it in one command.

Scripts

One can put a sequence of shell commands in a text file and execute it using its name as a command (or by `source <file>`). This is useful for automating frequently typed commands, and is called a shell script. Job scripts as used by the scheduler are a special kind of shell script. Another special script is the hidden file `~/.bashrc` which is executed each time a shell is started.

SciNet

GPC Quick Start Guide

Logging In

SciNet allows login only via ssh, a secure protocol. Log into the login machines at the data centre, and then into the development nodes, where you do all your work. Batch computing jobs are run on the compute nodes.

For Linux/MacOS users

From a terminal window,

```
$ ssh -Y [USER]@login.scinet.utoronto.ca
scinet01-$ ssh -Y gpc01 (or gpc02, gpc03, gpc04.)
```

The first command logs you into the login nodes (replace `[USER]` with your username), the second logs you into one of the four development nodes. `-Y` allows Xwindows programs to pop up windows on your local machine.

For Windows users

For ssh we suggest:

The cygwin environment (`http://cygwin.com`), a linux-like environment. Be sure to install X11 and OpenSSH, and you can then (after launching the X11 client) run the commands listed above; or

MobaXterm (`mobaxterm.mobatek.net`), a tabbed ssh client.

Machine Details

Each GPC node has 8 processors, ∼14GB of free memory, and supports up to 16 threads or processes. Jobs are allocated entire nodes and must make full use of each.

Modules

Software is accessed by loading `modules` which place the package in your environment.

`module avail`	List packages.
`module load [pkg]`	Use default version of [pkg].
`module load [pkg]/[v.]`	Use version [v.] of [pkg].
`module unload [pkg]`	Remove [pkg] from path.
`module purge`	Remove all packages from path.

Common modules:

`module load gcc intel`	gcc, intel compilers.
`module load openmpi`	OpenMPI
`module load intelmpi`	Intel MPI (recommended)

Editors

Text-based editors are more responsive over a network connection than graphical editors, but both are available.

`vi filename`	vi editor.
`gvim filename`	vi editor (graphical)
`module load emacs`	
`emacs filename`	emacs editor (text).
`emacs -x filename`	emacs editor (graphical)
`module load nano`	
`nano filename`	Simple nano editor (text).

Disk

`/home/[USER]`	10 GB, backed up.
`/scratch/[USER]`	Large capacity; not backed up; purged every 3 months.
`module load extras`	Shows user, group disk usage
`diskUsage`	on /home, /scratch.

All SciNet nodes see the same filesystems. /home can only be *read* from on the compute nodes; batch jobs must be run from /scratch. The shared disk system is optimized for high bandwidth large reads and writes. Using many small files, or doing many small inputs and outputs, is inefficient and slows down the file system for all users.

Copying Files

`scp` copies files via the secure ssh protocol. Small (few GB) files may be copied to or from the login nodes. Eg, from your local machine, to copy files *from* SciNet,

```
scp -C [USER]@login.scinet.utoronto:~/[PathToFile]
  [LocalPathToNewFile]
```

copies [PathToMyfile] to the local directory. To copy a file *to* SciNet:

```
scp -C [myfile]
  [USER]@login.scinet.utoronto:~/[PathToNewFile]
```

Large files must be sent through datamover nodes; see the SciNet wiki for details.

Running Jobs

It is ok to run short (few minute), small-memory tests on the development nodes. Others must be run on the compute nodes via the queues, from the /scratch directory.

Debug queue

A small number of compute nodes are set aside for a debug queue, allowing short jobs (under 2 hours) to run quickly. To get a single debug node for an hour to run interactively,

```
qsub -I -l nodes=1:ppn=8,walltime=1:00:00 -q debug
```

and one can run as if one were logged into the devel nodes. One can also run short debug nodes in batch mode.

Batch queue

The usual usage of SciNet is to build and compile your code on /home, then copy the executable and data files to a directory on /scratch, write a script which descibes how to run the job, and submit it to the queue. When resources are free, your job runs to completion. Jobs in the batch queue may run no longer than 48 hours per session. Sample scripts follow.

Sample batch script - MPI

```
#!/bin/bash
#PBS -l nodes=2:ppn=8          Request 2 nodes
#PBS -l walltime=1:00:00       ..for 1 hour.
#PBS -N test                   Job name
cd $PBS_O_WORKDIR              cd to submission dir.
mpirun -np 16 [prog]           Run program w/ 16 tasks.
```

Sample batch script - OpenMP

```
#!/bin/bash
#PBS -l nodes=1:ppn=8          Request 1 node
#PBS -l walltime=1:00:00       ..for 1 hour.
#PBS -N test                   Job name
cd $PBS_O_WORKDIR              cd to submission dir.
export OMP_NUM_THREADS=8       Run with 8 OpenMP threads
[prog] > job.out               Run, save output in job.out.
```

Sample batch script - Serial Jobs

It is also possible to run batches of 8 serial jobs on a node to make sure the node is fully utilized. If all tasks take roughly the same amount of time:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8          Request 1 node
#PBS -l walltime=1:00:00       ..for 1 hour.
#PBS -N serialx8               Job name
cd $PBS_O_WORKDIR              cd to submission dir.
(cd jobdir1; ./dojob1) &       Start task 1
(cd jobdir2; ./dojob2) &
...
(cd jobdir8; ./dojob8) &
wait                           Wait for all to finish
```

For more complicated cases, see the wiki.

Queue Commands

`qsub [script]`	Submit job to batch queue
`qsub [script] -q debug`	Submit job to debug queue
`qstat`	Show your queued jobs
`showq --noblock`	Show all jobs
`checkjob [jobid]`	Details of your job [jobid]
`showstart [jobid]`	*Estimate* start time
`canceljob [jobid]`	Cancel your job [jobid]

Ramdisk

Some of a node's memory may be used as a "ramdisk", a very fast filesystem visible only on-node. If your job uses little memory but does many small disk inputs/outputs, using ramdisk can significantly speed your job. To use: Copy your inputs to /dev/shm; cd to /dev/shm and run your job; then copy outputs from /dev/shm to /scratch.

Other Resources

`http://wiki.scinet.utoronto.ca`	Documentation
`support@scinet.utoronto.ca`	Email us for help