SDSC Gordon User Guide

Technical Summary

Gordon is a dedicated XSEDE cluster designed by Appro and SDSC consisting of 1024 compute nodes and 64 I/O nodes. Each compute node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. The I/O nodes each contain two 6-core 2.67 GHz Intel X5650 (Westmere) processors, 48 GB of DDR3-1333 memory, and sixteen 300 GB Intel 710 solid state drives. The network topology is a 4x4x4 3D torus with adjacent switches connected by three 4x QDR InfiniBand links (120 Gbit/s). Compute nodes (16 per switch) and I/O nodes (1 per switch) are connected to the switches by 4x QDR (40 Gbit/s). The theoretical peak performance of Gordon is 341 TFlop/s.

Transition to Comet

Please read this important information for current users of the SDSC Gordon cluster. Gordon is scheduled to be decommissioned at the end of March, 2017. To minimize the impact of this change and allow you to continue computing without interruption, SDSC will transition Gordon users to Comet. More details are provided in the Transitioning from Gordon section of the SDSC Comet User Guide.

Table 1.1 Technical Summary

SYSTEM COMPONENT	CONFIGURATION
Intel EM64T Xeon E5 Compute Nodes	
Sockets	2
Cores	16
Clock speed	2.6 GHz
Flop speed	333 Gflop/s
Memory capacity	64 GB
Memory bandwidth	85 GB/s
STREAM Triad bandwidth	60 GB/s
I/O Nodes	
Sockets	2
Cores	12
Clock speed	2.67 GHz
Memory capacity	48 GB
Memory bandwidth	64 GB/s
Flash memory	4.8 TB
Full System	
Total compute nodes	1024
Total compute cores	16384
Peak performance	341 Tflop/s
Total memory	64 TB
Total memory bandwidth	87 TB/s
Total flash memory	300 TB
QDR InfiniBand Interconnect	

Topology	3D Torus
Link bandwidth	8 GB/s (bidirectional)
Peak bisection bandwidth	TB/s (bidirectional)
MPI latency	1.3 Âµs
DISK I/O Subsystem	
File Systems	NFS, Lustre
Storage capacity (usable)	150 TB: Dec 2010 2 PB: June 2011 4 PB: July 2012
I/O bandwidth	GB/s

Gordon supports the XSEDE core software stack, which includes remote login, remote computation, data movement, science workflow support, and science gateway support toolkits.

Table 1.2 Systems Software Environment

SOFTWARE FUNCTION	DESCRIPTION
Cluster Management	Rocks
Operating System	CentOS
File Systems	NFS, Lustre
Scheduler and Resource Manager	Catalina, TORQUE
XSEDE Software	CTSS
User Environment	Modules
Compilers	Intel and PGI Fortran, C, C++
Message Passing	Intel MPI, MVAPICH, Open MPI
Debugger	DDT
Performance	IPM, mpiP, PAPI, TAU

System Access

As an XSEDE computing resource, Gordon is accessible to XSEDE users who are given time on the system. In order to get an account, users will need to submit a proposal through the XSEDE User Portal.

Interested users may contact SDSC User Support for assistance with applying for time on Gordon.

Logging in to Gordon

Gordon supports several access methods:

- Single Sign On through the XSEDE User Portal
- Command-line using SDSC username and XSEDE User Portal Password
- Command-line using SDSC username and SDSC password

To login to Gordon from the command line, use the hostname:

```
gordon.sdsc.xsede.org
```

The following are examples of Secure Shell (ssh) commands that may be used to log in to Gordon:

```
ssh sdscusername@gordon.sdsc.xsede.org
ssh -l sdscusername gordon.sdsc.xsede.org
```

Notes and hints

When you login to `gordon.sdsc.xsede.org`, you will be assigned one of the four login nodes: `gordon-ln[1-4].sdsc.xsede.org`. These nodes are identical in both architecture and software environment. Users should normally login to `gordon.sdsc.xsede.org`, but may access one of the four nodes directly if they see poor performance.

Secure shell users should feel free to append their public RSA key to their ~/.ssh/authorized_keys file to enable access from authorized hosts without having to enter their password.

Do not use the login nodes for computationally intensive processes. These nodes are meant for compilation, file editing, simple data analysis, and other tasks that use minimal compute resources. All computationally demanding jobs should be submitted and run through the batch queuing system.

Modules

The Environment Modules package provides for dynamic modification of your shell environment. Module commands set, change, or delete environment variables, typically in support of a particular application. They also let the user choose between different versions of the same software or different combinations of related codes.

For example, if the Intel module and mvapich2_ib module are loaded and the user compiles with mpif90, the generated code is compiled with the Intel Fortran 90 compiler and linked with the mvapich2_ib MPI libraries.

Several modules that determine the default Gordon environment are loaded at login time. These include the MPVPICH implementation of the MPI library and the Intel compilers. We strongly suggest that you use this combination whenever possible to get the best performance.

Useful Module Commands

Table 3.1 Common module commands and their descriptions

COMMAND	DESCRIPTION
module list	List the modules that are currently loaded
module avail	List the modules that are available
module display <module name>	Show the environment variables used by <module name> and how they are affected
module unload <module name>	Remove <module name> from the environment
module load <module name>	Load <module name> into the environment
module swap <module one> <module two>	Replace <module one> with <module two> in the environment

Loading and unloading modules

You must remove some modules before loading others. Some modules depend on others, so they may be loaded or unloaded as a consequence of another module command. For example, if intel and mvapich are both loaded, running the command module unload intel will automatically unload mvapich. Subsequently issuing the module load intel command does not automatically reload mvapich.

If you find yourself regularly using a set of module commands, you may want to add these to your configuration files (.bashrc for bash users, .cshrc for C shell users). Complete documentation is available in the module(1) and modulefile(4) manpages.

Module: command not found

The error message "`module: command not found`" is sometimes encountered when switching from one shell to another or attempting to run the module command from within a shell script or batch job. The reason that the module command may not be inherited as expected is that it is defined as a function for your login shell. If you encounter this error execute the following from the command line (interactive shells) or add to your shell script (including Torque batch scripts)

```
$ source /etc/profile.d/modules.sh
```

Software Packages

See also the SDSC Gordon User Guide.

Sample scripts using some common applications and libraries are provided in /home/diag/opt/sdsc/scripts. If you have any questions regarding setting up run scripts for your application, please email help@xsede.org.

Table 4.1 Gordon Software Packages

SOFTWARE PACKAGE	COMPILER SUITES	PARALLEL INTERFACE
AMBER: Assisted Model Building with Energy Refinement	intel	mvapich2_ib
APBS: Adaptive Poisson-Boltzmann Solver	intel	mvapich2_ib
Car-Parrinello 2000 (CP2K)	intel	mvapich2_ib
DDT		
FFTW: Fastest Fourier Transform in the West	intel,pgi,gnu	mvapich2_ib
GAMESS: General Atomic Molecular Electronic Structure System	intel	native: sockets, ip over ib vsmp: scalemp mpich2
GAUSSIAN	pgi	Single node, shared memory
GROMACS: GROningen MAchine for Chemical Simulations	intel	mvapich2_ib
HDF4/HDF5: Hierarchical Data Format	intel,pgi,gnu	mvapich2_ib for hdf5
Lammps:Large-scale Atomic/Molecular Massively Parallel Simulator.	intel	mvapich2_ib
NAMD: NAnoscale Molecular Dynamics	intel	mvapich2_ib
NCO: netCDF Operators	intel,pgi,gnu	none
netCDF: Network Common Data Format	Intel,pgi,gnu	none
Python modules (scipy etc)	gnu:ipython,nose,pytz intel:matplotlib,numpy,scipy,pyfits,	None
P3DFFT: Parallel 3-D Fast Fourier Transforms	intel,pgi,gnu	mvapich2_ib
Singularity: User Defined Images	none	none
VisIt Visualization Package	intel	openmpi

Software Package Descriptions

AMBER

AMBER is package of molecular simulation programs including SANDER (Simulated Annealing with NMR-Derived Energy Restraints) and a modified version PMEME (Particle Mesh Ewald Molecular Dynamics) that is faster and more scalable.

APBS

APBS evaluates the electrostatic properties of solvated biomolecular systems.

APBS documentation

CP2K

CP2K is a program to perform simulations of molecular systems. It provides a general framework for different methods such as Density Functional Theory (DFT) using a mixed Gaussian and plane waves approach (GPW) and classical pair and many-body potentials.

CP2K documentation

DDT

DDT is a debugging tool for scalar, multithreaded and parallel applications.

DDT Debugging Guide from TACC

FFTW

FFTW is a library for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data.

FFTW documentation

GAMESS

GAMESS is a program for ab initio quantum chemistry. GAMESS can compute SCF wavefunctions, and correlation corrections to these wavefunctions as well as Density Functional Theory.

GAMESS documentation, examples, etc.

Gaussian 09

Gaussian 09 provides state-of-the-art capabilities for electronic structure modeling.

Gaussian 09 User's Reference

GROMACS

GROMACS is a versatile molecular dynamics package, primarily designed for biochemical molecules like proteins, lipids and nucleic acids.

GROMACS Online Manual

HDF

HDF is a collection of utilities, applications and libraries for manipulating, viewing, and analyzing data in HDF format.

HDF 5 Resources

LAMMPS

LAMMPS is a classical molecular dynamics simulation code.

LAMMPS User Manual

NAMD

NAMD is a parallel, object-oriented molecular dynamics code designed for high-performance simulation of large biomolecular systems.

NAMD User's Guide

NCO

NCO operates on netCDF input files (e.g. derive new data, average, print, hyperslab, manipulate metadata) and outputs results to screen or text, binary, or netCDF file formats.

NCO documentation on SourceForge

netCDF

netCDF is a set of libraries that support the creation, access, and sharing of array-oriented scientific data using machine-independent data formats.

netCDF documentation on UCAR's Unidata Program Center

Python Modules (scipy etc.)

The Python modules under /opt/scipy consist of: node, numpy, scipy, matplotlib, pyfits, ipython and pytz.

Video tutorial from a TACC workshop on Python

Python videos from Khan Academy

The HPC University Python resources

P3DFFT

Parallel Three-Dimensional Fast Fourier Transforms is a library for large-scale computer simulations on parallel platforms. 3D FFT is an important algorithm for simulations in a wide range of fields, including studies of turbulence, climatology, astrophysics and material science.

Read the User Guide on GitHub.

Singularity: User Defined Images

Singularity is a platform to support users that have different environmental needs than what is provided by the resource or service provider. While the high level perspective of other container solutions seems to fill this niche very well, the current implementations are focused on network service virtualization rather than application level virtualization focused on the HPC space. Because of this, Singularity leverages a workflow and security model that makes it a very reasonable candidate for shared or multi-tenant HPC resources like Comet without requiring any modifications to the scheduler or system architecture. Additionally, all typical HPC functions can be leveraged within a Singularity container (e.g. InfiniBand, high performance file systems, GPUs, etc.). While Singularity supports MPI running in a hybrid model where you invoke MPI outside the container and it runs the MPI programs inside the container, we have not yet tested this.

Examples for various modes of usage are available in /home/diag/opt/scripts/Singularity. Please email help@xsede.org (reference Gordon as the machine, and SDSC as the site) if you have any further questions about usage and configuration. Details on the Signularity project are available at http://singularity.lbl.gov/#home.

VisIt Visualization Package

The VisIt visualization package supports remote submission of parallel jobs and includes a Python interface that provides bindings to all of its plots and operators so they may be controlled by scripting.

Getting Started With VisIt tutorial

Accounts

The show_accounts command lists the accounts that you are authorized to use, together with a summary of the used and remaining time.

```
[sinkovit@dash-login ~]$ show_accounts
ID name project used available
----------------------------------------
sinkovit uic168 0 30000
sinkovit sds122 14700 505995
```

To charge your job to one of these projects replace << project >> with one from the list and put this PBS directive in your job script:

```
#PBS -A << project >>
```

Many users will have access to multiple accounts (e.g. an allocation for a research project and a separate allocation for classroom or educational use). On some systems a default account is assumed, but please get in the habit of explicitly setting an account for all batch jobs. Awards are normally made for a specific purpose and should not be used for other projects.

Adding users to an account

Project PIs and co-PIs can add or remove users from an account. To do this, login to your XSEDE portal account and go to the add user page.

Charging

The charge unit for all SDSC machines, including Gordon, is the SU (service unit) and corresponds to the use of one compute core for one hour. Keep in mind that your charges are based on the resources that are tied up by your job and don't necessarily reflect how the resources are used.

Unlike some of SDSC's other major compute resources, Gordon does not provide for shared use of a compute node. A serial job that requests a compute node for one hour will be charged 16 SUs (16 cores x 1 hour), regardless of how the processors-per-node parameter is set in the batch script.

Compiling

Gordon provides the Intel, Portland Group (PGI), and GNU compilers along with multiple MPI implementations (MVAPICH2, MPICH2, OpenMPI). Most applications will achieve the best performance on Gordon using the Intel compilers and MVAPICH2 and the majority of libraries installed on Gordon have been built using this combination. Although other compilers and MPI implementations are available, we suggest using these only for compatibility purposes.

Using the Intel compilers (Default/Suggested)

The Intel compilers and the MVAPICH2 MPI implementation will be loaded by default. For best performance, compile with the `-fast` flag. This will ensure that the AVX instruction set will be used. If you have modified your environment, you can reload by executing the following commands at the Linux prompt or placing in your startup file (~/.cshrc or ~/.bashrc)

```
module purge
module load intel mvapich2_ib
```

Table 5.1 Intel Compiler Commands

	SERIAL	MPI	OPENMP	MPI+OPENMP
FORTRAN	ifort	mpif90	ifort -openmp	mpif90 -openmp
C	icc	mpicc	icc -openmp	mpicc -openmp
C++	icpc	mpicxx	icpc -openmp	mpicxx -openmp

Note for vSMP users on Gordon: MPI applications intended for use on the large memory vSMP nodes should use MPICH2 instead of MVAPICH2. This version of MPICH2 has been specially tuned for optimal vSMP message passing performance.

Using the PGI compilers

The PGI compilers can be loaded by executing the following commands at the Linux prompt or placing in your startup file (~/.cshrc or ~/.bashrc)

```
module purge
module load pgi mvapich2_ib
```

Table 5.2 PGI Compiler Commands

	SERIAL	MPI	OPENMP	MPI+OPENMP
FORTRAN	pgf90	mpif90	pgf90 -mp	mpif90 -mp
C	pgcc	mpicc	pgcc -mp	mpicc -mp
C++	pgCC	mpicxx	pgCC -mp	mpicxx -mp

Using the GNU compilers

The GNU compilers can be loaded by executing the following commands at the Linux prompt or placing in your startup files (~/.cshrc or ~/.bashrc)

```
module purge
module load gnu openmpi
```

Table 5.3 GNU Compiler Commands

	SERIAL	MPI	OPENMP	MPI+OPENMP
FORTRAN	gfortran	mpif90	gfortran -fopenmp	mpif90 -fopenmp
C	gcc	mpicc	gcc -fopenmp	mpicc -fopenmp
C++	g++	mpicxx	g++ -fopenmp	mpicxx -fopenmp

Notes and hints

The mpif90, mpicc, and mpicxx commands are actually wrappers that call the appropriate serial compilers and load the correct MPI libraries. While the same names are used for the Intel, PGI and GNU compilers, keep in mind that these are completely independent scripts.

If you use the PGI or GNU compilers or switch between compilers for different applications, make sure that you load the appropriate modules before running your executables.

When building OpenMP applications and moving between different compilers, one of the most common errors is to use the wrong flag to enable handling of OpenMP directives. Note that Intel, PGI, and GNU compilers use the `-openmp`, `-mp`, and `-fopenmp` flags, respectively.

Explicitly set the optimization level in your makefiles or compilation scripts. Most well written codes can safely use the highest optimization level (`-O3`), but many compilers set lower default levels (e.g. GNU compilers use the default `-O0`,

which turns off all optimizations).

Turn off debugging, profiling, and bounds checking when building executables intended for production runs as these can seriously impact performance. These options are all disabled by default. The flag used for bounds checking is compiler dependent, but the debugging (`-g`) and profiling (`-pg`) flags tend to be the same for all major compilers.

The large memory vSMP nodes on the other hand can be shared by multiple jobs. Be sure to request as many cores as your job needs, but be aware that if you request all 128 cores within a vSMP node your job will be charged at the rate of 128 SUs per hour.

Running Jobs

Gordon uses the TORQUE Resource Manager, together with the Catalina Scheduler, to manage user jobs. If you're familiar with PBS, note that TORQUE is based on the original PBS project and shares most of its syntax and user interface. Whether you run in batch mode or interactively, you will access the compute nodes using the `qsub` command as described below. Remember that computationally intensive jobs should be run only on the compute nodes and not the login nodes.

Gordon Queues

Gordon has two queues available:

Table 6.1 Gordon queues

QUEUE NAME	MAX WALLTIME	MAX NODES	COMMENTS
normal	48 hours	64	Used for exclusive access to regular (non-vsmp) compute nodes
vsmp	48 hours	1	Used for shared access to vsmp node

Submitting jobs

A job can be submitted using the `qsub` command, with the job parameters either specified on the command line or in a batch script. Except for simple interactive jobs, most users will find it more convenient to use batch scripts.

```
$ qsub options
$ qsub batch_script
```

Batch script basics

TORQUE batch scripts consist of two main sections. The top section specifies the job parameters (e.g. resources being requested, notification details) while the bottom section contains user commands. A sample job script that can serve as a starting point for most users is shown below. Content that should be modified as necessary is between angled brackets (the brackets are not part of the code).

```
#!/bin/bash
#PBS -q normal
#PBS -l nodes=<2>:ppn=<16>:native
#PBS -l walltime=<1:00:00>
#PBS -N <jobname>
#PBS -o <my.out>
#PBS -e <my.err>
#PBS -A <abc100>
#PBS -M <email_address>
#PBS -m abe
#PBS -V
# Start of user commands - comments start with a hash sign (#)
cd /oasis/scratch/$USER/temp_project
<usercommands>
```

The first line indicates that the file is a bash script and can therefore contain any legal bash code. The lines starting with '#PBS' are special comments that are interpreted by TORQUE and must appear before any user commands.

The second line states that we want to run in the queue named "normal". Lines three and four define the resources being requested: 2 nodes with 16 processors per node, for one hour (1:00:00). The next three lines (5-7) are not essential, but using them will make it easier for you to monitor your job and keep track of your output. In this case, the job will be

appear as 'jobname' in the queue; stdout and stderr will be directed to 'my.out' and 'my.err', respectively. The next line specifies that the usage should be charged to account abc123. Lines 9 and 10 control email notification: notices should be sent to 'email_address@domain.edu' when the job aborts (a), begins (b), or ends (e).

Finally, "`#PBS -V`" specifies that your current environment variables should be exported to the job. For example, if the path to your executable is found in your PATH variable and your script contains the line "`#PBS -V`", then the path will also be known to the batch job.

The statement '`cd /oasis/scratch/$USER/temp_project`' changes the working directory to the directory where the job was submitted. This should always be done unless you provide full paths to all executables and input files. The remainder of the script is normally used to run your application.

Interactive jobs

The only difference between running batch and interactive jobs is that the latter requires the `-I` flag. The following command shows how to get interactive use of one node for 30 minutes:

```
qsub -I -q normal -l nodes=2:ppn=16:native:flash,walltime=30:00 -A abc123
```

Monitoring and deleting jobs

Use the `qstat` command to monitor your jobs and `qdel` to delete a job. Some useful options are described below.

Running MPI jobs - regular compute nodes

MPI jobs are run using the `mpirun_rsh` command. When your job starts, TORQUE will create a PBS_NODEFILE listing the nodes that had been assigned to your job, with each node name replicated *ppn* times. Typically *ppn* will be set equal to the number of physical cores on a node (16 for Gordon) and the number of MPI processes will be set equal to nodes x *ppn*. Relevant lines from a batch script are shown below.

```
#PBS -l nodes=2:ppn=16:native
cd /oasis/scratch/$USER/temp_project
mpirun_rsh -np 32 -hostfile $PBS_NODEFILE mpi_app [command line args]
```

Sometimes you may want to run fewer than 16 MPI processes per node; for instance, if the per-process memory footprint is too large (> 4 GB on Gordon) to execute one process per core or you are running a hybrid application that had been developed using MPI and OpenMP. In these cases you will want to set *ppn* to reflect the reduced number of processes per node.

```
# Running 16 MPI processes across two nodes
#PBS -l nodes=2:ppn=8:native
cd /oasis/scratch/$USER/temp_project
mpirun_rsh -np 16 -hostfile $PBS_NODEFILE mpi_app [command line args]
```

If *ppn* had been set to 16 in the previous example, the PBS_NODEFILE would have contained 16 replicates of the first node followed by 16 replicates of the second node. As a result all MPI processes would run on the first node leaving the second node idle.

Running OpenMP jobs - regular compute nodes

For an OpenMP application, set and export the number of threads. Relevant lines from a batch script are shown below.

```
#PBS -l nodes=1:ppn=16:native
cd /oasis/scratch/$USER/temp_project
export OMP_NUMTHREADS=16
openmp_app [command line args]
```

Running hybrid (MPI + OpenMP) jobs - regular compute nodes

For hybrid parallel applications (MPI+OpenMP), use `mpirun_rsh` to launch the job and specify both the number of MPI processes and the number of threads per process. To pass environment variables through `mpirun_rsh`, list the key/value pairs before the executable name. The number of MPI processes should equal nodes x *ppn*, just as in a regular MPI job. Ideally the product of the MPI process count and the threads per process should equal the number of physical cores on the nodes being used.

Hybrid jobs will typically use one MPI process per node with the threads per node equal to the number of physical cores per node, but as the examples below show this is not required.

```
# 2 MPI processes x 16 threads/node = 2 nodes x 16 cores/node = 32
#PBS -l nodes=2:ppn=1:native
cd /oasis/scratch/$USER/temp_project
mpirun_rsh -np 2 -hostfile $PBS_NODEFILE OMP_NUMTHREADS=16 executable [command line args]
```

```
# 8 MPI processes x 4 threads/node = 2 nodes x 16 cores/node = 32
#PBS -l nodes=2:ppn=4:native
cd /oasis/scratch/$USER/temp_project
mpirun_rsh -np 8 -hostfile $PBS_NODEFILE OMP_NUMTHREADS=4 executable [command line args]
```

Note and hints

Try to provide a realistic upper estimate for the wall time required by your job. This will often improve your turnaround time, especially on a heavily loaded machine. Do not rely on the default values for the wall time since they tend to be quite long, vary across queues and machines, and are subject to change.

Storage

Scratch File System

Gordon users have access to a 1.6 PB high performance scratch file system that is intended expressly for the running of user applications. In order to ensure the highest levels of performance, capacity, and stability the scratch files system is subject to routine purge. We therefore strongly encourage users to monitor their usage and delete files as soon as possible, as this allows the space to be used more flexibly by all users. Users are reminded that this is scratch space and is not backed up; files that must be retained should be moved to alternate locations.

The Gordon scratch file system is configured as follows:

```
/oasis/scratch/$USER/$PBS_JOBID
```

This directory is created at the start of a job and users should use this for all intermediate job files. Executables and other input data can be copied here from the user's home directory. Upon completion, files that are to be retained should be copied back to their home directories, or to their home institution. Files in `/oasis/scratch/$USER/$PBS_JOBID` are subject to routine purge without notice.

```
/oasis/scratch/$USER/temp_project
```

This temporary project directory is being provided on an interim basis for the next several weeks while an additional project storage area is deployed to Gordon. The future project space will be approximately 400 TB and is intended to give users a convenient location for staging executables, input files, and other data needed for routine job execution. It is not intended for long term data storage. Once the project storage is deployed, users will be responsible for moving their data from `/oasis/scratch/$USER/temp_project` to this new project area. In the meantime, files in `/oasis/scratch/$USER/temp_project` will be purged as needed, with a minimum of 5 days notice being provided to users.

vSMP

Most applications will be run on the regular compute nodes, but in certain instances you will want to use the large-memory vSMP nodes:

- Serial or threaded applications requiring more than 64 GB of memory
- MPI applications requiring more than 64 GB per process
- Applications that can take advantage of a RAM file system (ramfs)

Before reading this section, we suggest that you first become familiar with the user guide sections that cover compiling and running jobs.

Compilation instructions are the same for the regular compute nodes and the vSMP nodes, although ScaleMP's specially tuned version of MPICH2 should be used when building MPI applications.

Jobs are submitted to the vSMP nodes using TORQUE and, with the exception of specifying a different queue name, all of the instructions given in the user guide sections describing the batch submission system still apply. In the remainder of this section, we describe the additional steps needed to make effective use of the vSMP nodes. Note that we do not provide complete scripts below, but rather just the critical content needed to run under vSMP.

Memory and core usage

Since multiple user jobs can be run simultaneously on a single vSMP node, there has to be a mechanism in place to ensure that jobs do not use more than their fair share of the memory. In addition, jobs should run on distinct sets of physical nodes to prevent contention for resources. Our approach is to implement the following policies:

- Cores are requested in units of 16
- Memory is allocated in proportion to number of cores requested

Each physical node has 64 GB of memory, but after accounting for the vSMP overhead the amount available for user jobs is closer to 60 GB. A serial job requiring 120 GB of memory should request 32 cores.

```
#PBS -l nodes=1:ppn=32:vsmp
```

OpenMP jobs

The example below shows a partial batch script for an OpenMP job that will use 16 threads. Note that the queue name is set to vsmp.

```
#PBS -q vsmp
#PBS -l nodes=1:ppn=16:vsmp
export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so
export LD_PRELOAD=/usr/lib/libhoard.so:$LD_PRELOAD
export PATH=/opt/ScaleMP/numabind/bin:$PATH
export OMP_NUM_THREADS=16
export KMP_AFFINITY=compact,verbose,0,`numabind --offset=16`
./openmp-app
```

The first export statement preloads the vSMP libraries needed for an application to run on the vSMP nodes. The second export statement is not strictly required, but is suggested since using the Hoard library can lead to improved performance particularly when multiple threads participate in dynamic memory management. The third export statement prepends the PATH variable with the location of the numabind command, while the fourth sets the number of OpenMP threads.

The final export statement requires a more in depth explanation and contains the "magic" for running on a vSMP node. Setting the `KMP_AFFINITY` variable determines how the threads will be mapped to compute cores. Note that it is used only for OpenMP jobs and will not affect the behavior of pThreads codes.

The assignment maps the threads compactly to cores, as opposed to spreading out across cores, provides verbose output (will appear in the job's stderr file), and starts the mapping with the first core in the set. The numabind statement looks for an optimal set of 16 contiguous cores, where optimal typically means that the cores span the smallest number of physical nodes and have the lightest load. Enclosing within the back ticks causes the numabind output to appear in place as the final argument in the definition.

For most users, this `KMP_AFFINITY` assignment will be adequate, but be aware that there are many options for controlling the placement of OpenMP threads. For example, to run on 16 cores evenly spaced and alternating over 32 contiguous cores, set the *ppn* and offset values to 32, the number of threads to 16, and the affinity type to scatter. Relevant lines are shown below.

```
#PBS -l nodes=1:ppn=32:vsmp
export OMP_NUM_THREADS=16
export KMP_AFFINITY=scatter,verbose,0,`numabind --offset=32`
```

pThreads jobs

As mentioned in the previous section, setting the `KMP_AFFINITY` variable has no effect on pThreads jobs. Instead you will need to create an additional configuration file with all of the contents appearing on a single line. It is absolutely critical that the executable name assigned to the pattern is the same as that used in the script.

```
$ cat config

(output shown on two lines for clarity)
```

```
name=myconfig pattern=pthreads-app verbose=0 process_allocation=multi
task_affinity=cpu rule=RULE-procgroup.so flags=ignore_idle
```

The `sleep` statement in the following batch script ensures that enough time has elapsed to create all of the threads before they are bound to cores by numabind.

```
#PBS -q vsmp
#PBS -l nodes=1:ppn=16:vsmp
export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so
export LD_PRELOAD=/usr/lib/libhoard.so:$LD_PRELOAD
export PATH=/opt/ScaleMP/numabind/bin:$PATH
./pthreads-app &
sleep 10
numabind --config config
```

Serial jobs

Serial job submission is very similar to that for OpenMP jobs, except that the taskset command is used together with numabind to bind the process to a core. Even though a single core will be used for the computations, remember to request a sufficient number of cores to obtain the memory required (~60 GB per 16 cores).

```
#PBS -q vsmp
#PBS -l nodes=1:ppn=16:vsmp
export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so
export PATH=/opt/ScaleMP/numabind/bin:$PATH
taskset -c `numabind --offset=1` ./scalar-app
```

MPI jobs

If you require less than 64 GB of memory per MPI process, we suggest that you run on the regular compute nodes. While ScaleMP does provide a specially tuned version of the MPICH2 library designed to achieve optimal performance under vSMP, the vSMP foundation software does add a small amount of unavoidable communications latency (applies not just to vSMP, but any hypervisor layer).

Before compiling your MPI executable to run under vSMP, unload the mvapich module and replace with mpich2

```
$ module swap mvapich2_ib mpich2_ib
```

To run an MPI job, where the number of processes equals the number of compute cores requested, use the following options. Note that `VSMP_PLACEMENT` is set to PACKED, indicating that the MPI processes will be mapped to a set of contiguous compute cores.

```
#PBS -q vsmp
#PBS -l nodes=1:ppn=32:vsmp
export PATH=/opt/ScaleMP/mpich2/1.3.2/bin:$PATH
export VSMP_PLACEMENT=PACKED
export VSMP_VERBOSE=YES
export VSMP_MEM_PIN=YES
vsmputil --unpinall
time mpirun -np 32 ./mpitest
```

The situation is slightly more complicated if the number of MPI processes is smaller than the number of requested cores. Typically you will want to spread the MPI processes evenly across the requested cores (e.g. when running 2 MPI processes across 64 cores, the processes will be mapped to cores 1 and 33). In the example below the placement has been changed from "PACKED" to "SPREAD^2^32" The SPREAD keyword indicates that the processes should be spread across the cores and "2^32" is a user defined topology of 2 nodes with 32 cores per node.

```
#PBS -q vsmp
#PBS -l nodes=1:ppn=64:vsmp
export PATH=/opt/ScaleMP/mpich2/1.3.2/bin:$PATH
export VSMP_PLACEMENT=SPREAD^2^32
export VSMP_VERBOSE=YES
export VSMP_MEM_PIN=YES
vsmputil --unpinall
mpirun -np 2 ./mpitest
```

Note, hints, and additional resources

We have only touched on the basics required for job submission. For additional information see the following external resources:

Thread Affinity Interface (KMP_AFFINITY)
The Hoard Memory Allocator
vSMP Foundation numabind

Torque in Depth

This section takes a closer look at the some of the more useful TORQUE commands. Basic information on job submission can be found in the User Guide section Running Jobs. For a comprehensive treatment, please see the official TORQUE documentation and the man pages for `qstat`, `qmgr`, `qalter`, and `pbsnodes`. We also describe several commands that are specific to the Catalina scheduler and not part of the TORQUE distribution.

Listing jobs (`qstat -a`)

Running the `qstat` command without any arguments shows the status of all batch jobs. The `-a` flag is suggested since this provides additional information such as the number of nodes being used and the required time. The following output from `qstat -a` has been edited slightly for clarity.

```
$ qstat -a
                                                  Req'd  Req'd   Elap
Job ID   Username Queue    Jobname NDS  Memory Time S Time
-------- -------- ------  ------- --- ------ ----- - -----
1.gordon user1    normal  g09       1    1gb 26:00 R 23:07
2.gordon user2    normal  task1    32    --  48:00 R 32:15
3.gordon user2    normal  task2     2    1gb 24:00 H   --
4.gordon user3    normal  exp17     1    --  12:00 C 01:32
5.gordon user3    normal  stats1    8    --  12:00 Q   --
6.gordon user3    normal  stats2    8    --  12:00 E 15:27
```

The output is mostly self-explanatory, but a few points are worth mentioning. The Job ID listed in the first column will be needed if you want to alter, delete, or obtain more information about a job. On SDSC resources, only the numeric portion of the Job ID is needed. The queue, number of nodes, wall time, and required memory specified in your batch script are also listed. For jobs that have started running, the elapsed time is shown. The column labeled "S" lists the job status.

R = running
Q = queued
H = held
C = completed after having run
E = exiting after having run

Jobs can be put into a held state for a number of reasons including job dependencies (e.g. task2 cannot start until task1 completes) or a user exceeding the allowed number of jobs that can be in a queued state.

On a busy system, the `qstat` output can get to be quite long. To limit the output to just your own jobs, use the `-u` username option

```
$ qstat -a -u user2
Req'd Req'd Elap
Job ID Username Queue Jobname NDS Memory Time S Time
-------- -------- ------ ------- --- ------ ----- - -----
2.gordon user2 normal task1 32 -- 48:00 R 32:15
3.gordon user2 normal task2 2 1gb 24:00 H --
```

Detailed information for a job (`qstat -f <jobid>`)

Running `qstat -f <jobid>` provides the full status for a job. In addition to the basic information listed by `qstat -a`, this includes the job's start time, compute nodes being used, CPU and memory usage, and account being charged.

Nodes allocated to a job (`qstat -n <jobid>`)

To see the list of nodes allocated to a job, use `qstat -n`. Note that this output doesn't reflect actual usage, but rather the resources that had been requested. Knowing where your job is running is valuable information since you'll be able to access those nodes for the duration of your job to monitor processes, threads, and resource utilization.

Altering job properties (`qalter`)

The `qalter` command can be used to modify the properties of a job. Note that the modifiable attributes will depend on the job state (e.g. number of nodes requested cannot be changed after a job starts running). See the `qalter` man page for more details.

```
$ qstat -a 8
Req'd Req'd Elap
Job ID Username Queue Jobname NDS Memory Time S Time
-------- -------- ------ ------- --- ------ ----- - -----
8.gordon user4 normal task1 32 -- 10:00 R 6:15

$ qalter -l walltime=9:00 8
$ qstat -a 8
Req'd Req'd Elap
Job ID Username Queue Jobname NDS Memory Time S Time
-------- -------- ------ ------- --- ------ ----- - -----
8.gordon user2 normal task1 32 -- 09:00 R 6:15
```

Obtaining queue properties (`qstat -q`)

Queue properties, including the walltime and nodes limits, can be obtained using `qstat -q`.

```
$ qstat -q

server: gordon-fe2.local



Queue              Memory CPU Time Walltime Node   Run Que Lm   State
---------------- ------ -------- -------- ----   --- --- --   -----
normal               --       --     48:00:00     64  57    6 --    E R
vsmp                 --       --     48:00:00      1   2    1 --    E R
                                                   ----- -----
                                                      59     7
```

More detailed information in an alternative format is displayed using the `qmgr` command.

```
$ qmgr -c 'list queue normal'
Queue normal
queue_type = Execution
total_jobs = 63
state_count = Transit:0 Queued:6 Held:0 Waiting:0 Running:57 Exiting:0
resources_max.nodect = 64
resources_max.walltime = 48:00:00
mtime = Wed Feb 29 14:25:41 2012
resources_assigned.mem = 0b
resources_assigned.nodect = 442
enabled = True
started = True
```

Node attributes (`pbsnodes -a`)

The full set of attributes for the nodes can be listed using `pbsnodes -a`. To limit output to a single node, provide the node name.

Node states (`pbsnodes -l`)

Running `pbsnodes -l` lists nodes that are unavailable to the batch systems (e.g. have a status of "down", "offline", or "unknown"). To see the status of all nodes, use `pbsnodes -l` all. Note that nodes with a "free" status are not necessarily available to run jobs. This status applies both to nodes that are idle and to nodes that are running jobs using a *ppn* value smaller than the number of physical cores.

Monitoring Your Job

In this section, we describe some standard tools that you can use to monitor your batch jobs. We suggest that you at least familiarize yourself with the section of the user guide that deals with running jobs to get a deeper understanding of the batch queuing system before starting this section.

Figuring out where your is job running

Using the `qstat -n` jobid command, you'll be able to see a list of nodes allocated to your job. Note that this output doesn't reflect actual usage, but rather the resources that had been requested. Knowing where your job is running is valuable information since you'll be able to access those nodes for the duration of your job to monitor processes, threads, and resource utilization.

The output from `qstat -n` shown below lists the nodes that have been allocated for job 362256. Note that we have 16 replicates of the node gcn-5-22, each followed by a number corresponding to a compute core on the node.

```
[sinkovit@gordon-ln1 ~]$ qstat -n 362256
gcn-5-22/15+gcn-5-22/14+gcn-5-22/13+gcn-5-22/12
+gcn-5-22/11+gcn-5-22/10+gcn-5-22/9+gcn-5-22/8
+gcn-5-22/7+gcn-5-22/6+gcn-5-22/5+gcn-5-22/4
+gcn-5-22/3+gcn-5-22/2+gcn-5-22/1+gcn-5-22/1
```

Connecting to a compute node

Under normal circumstances, users cannot login directly to the compute nodes. This is done to prevent users from bypassing the batch scheduler and possibly interfering with other jobs that may be running on the nodes.

```
$ ssh gcn-5-22
Connection closed by 10.1.254.187
```

Access is controlled by the contents of the /etc/security/access.conf file, which for idle nodes denies access to everyone except root and other administrative users.

```
$ cat /etc/security/access.conf
-:ALL EXCEPT root diag :ALL
```

The situation is different though once a batch job transitions from the queued state to the run state. A special script known as the prologue makes a number of changes to the compute environment, including altering the access.conf file so that the owner of the job running on the node can login.

```
$ cat /etc/security/access.conf
-:ALL EXCEPT root diag username:ALL
```

Once the job terminates, a corresponding script known as the epilogue undoes the changes made by the prologue. If you happen to be logged in to a compute node when your job terminates, your session will be terminated.

Static view of threads and processes with ps

Once you login into a compute node, the ps command can provide information about current processes. This is useful if you want to confirm that you are running the expected number of MPI processes, that all MPI processes are in the run state, and that processes are consuming relatively equal amounts of CPU time. The ps command has a large number of options, but two that are suggested are `-l` (long output) and `-u` user (restrict output to processes owned by username or user ID).

In the following example, we see that the user is running 16 MPI processes, named pmemd.MPI in the CMD column, on the compute node gcn-5-22. The TIME column shows that the processes are all using very nearly the same amount of CPU time and the S (state) column shows that these are all in the run (R) state. Not all applications will exhibit such ideal load balancing and processes may sometimes go into a sleep state (D or S) while performing I/O or waiting for an event to complete.

```
[diag@gcn-5-22 ~]$ ps -lu username
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
5 S 503412 23686 23684 0 76 0 - 23296 stext ? 00:00:00 sshd
0 S 503412 23689 23686 0 78 0 - 16473 wait ? 00:00:00 bash
0 S 503412 23732 23689 0 75 0 - 6912 stext ? 00:00:00 mpispawn
0 R 503412 23735 23732 99 85 0 - 297520 stext ? 12:40:50 pmemd.MPI
0 R 503412 23736 23732 99 85 0 - 213440 stext ? 12:40:55 pmemd.MPI
0 R 503412 23737 23732 99 85 0 - 223276 stext ? 12:40:54 pmemd.MPI
0 R 503412 23738 23732 99 85 0 - 255149 stext ? 12:40:55 pmemd.MPI
[ --- 12 additional lines very similar to previous line not shown --- ]
```

The ps command with the `m` (not `-m`) option gives thread-level information, with the summed CPU usage for the threads in each process followed by per-thread usage. In the examples below, we first run ps to see the five MPI processes, then use the m option to see the six threads per process.

```
[diag@gcn-4-28 ~]$ ps -lu cipres # Processes only
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
0 R 503187 22355 22354 99 85 0 - 245402 stext ? 06:05:46 raxmlHPC
0 R 503187 22356 22354 99 85 0 - 246114 stext ? 06:05:47 raxmlHPC
0 R 503187 22357 22354 99 85 0 - 245325 stext ? 06:05:47 raxmlHPC
0 R 503187 22358 22354 99 85 0 - 222089 stext ? 06:05:47 raxmlHPC
0 R 503187 22359 22354 99 85 0 - 245630 stext ? 06:05:47 raxmlHPC

[diag@gcn-4-28 ~]$ ps m -lu cipres # Processes and threads
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
0 - 503187 22355 22354 99 - - - 245402 - ? 402:38 raxmlHPC
0 R 503187 - - 99 85 0 - - stext - 67:06 -
1 R 503187 - - 99 85 0 - - stext - 67:06 -
1 R 503187 - - 99 85 0 - - stext - 67:06 -
1 R 503187 - - 99 85 0 - - stext - 67:06 -
1 R 503187 - - 99 85 0 - - stext - 67:06 -
1 R 503187 - - 99 85 0 - - stext - 67:06 -
[ --- similar output for other processes/threads not shown --- ]
```

Dynamic view of threads and processes with top

While ps gives a static view of the processes, top can be used to provide a dynamic view. By default the output of top is updated every three seconds and lists processes in order of decreasing CPU utilization. Another advantage of top is that it displays the per-process memory usage thereby providing a very easy way to confirm that your job is not exceeding the available physical memory on a node.

In the first example below, the CPU usage is around 600% for all five processes, indicating that each process had spawned multiple threads. We can confirm this by enabling top to display thread-level information with the H option. (This option can either be specified on the top command line or entered after top is already running)

```
[diag@gcn-4-28 ~]$ top # Processes only
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
22356 cipres 25 0 961m 251m 2316 R 600.0 0.4 481:32.53 raxmlHPC
22359 cipres 25 0 959m 252m 2312 R 600.0 0.4 481:31.90 raxmlHPC
22355 cipres 25 0 958m 256m 7448 R 599.6 0.4 481:31.55 raxmlHPC
22357 cipres 25 0 958m 249m 2300 R 599.6 0.4 481:32.17 raxmlHPC
22358 cipres 25 0 867m 249m 2312 R 599.6 0.4 481:32.42 raxmlHPC
[ --- Additional lines not shown ---]

[diag@gcn-4-28 ~]$ top H # Processes and threads
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
22355 cipres 25 0 958m 256m 7448 R 100.1 0.4 82:35.68 raxmlHPC
22387 cipres 25 0 958m 256m 7448 R 100.1 0.4 82:35.70 raxmlHPC
22388 cipres 25 0 958m 256m 7448 R 100.1 0.4 82:35.93 raxmlHPC
22375 cipres 25 0 961m 252m 2316 R 100.1 0.4 82:36.03 raxmlHPC
22377 cipres 25 0 961m 252m 2316 R 100.1 0.4 82:36.04 raxmlHPC
[ --- output for other 9 threads and other lines not shown ---]
```

Additional Resources

We have not covered all of the options and capabilities of `qstat`, `ps`, and `top`. For more information consult the man pages for these commands.

Gordon ION - Dedicated I/O nodes

A limited number of Gordon's 64 flash-based I/O nodes are available as dedicated resources. The PI and designated users of a dedicated I/O node have exclusive use of the resource at any time for the duration of the award without having to go through the batch scheduler. Any data stored on the nodes is persistent (but not backed up) and can only be removed by the users of the project. These nodes are referred to as the "Gordon ION" and must be requested separately from the regular compute cluster.

Each I/O node contains two hex-core 2.66 GHz Intel Westmere processors, 48 GB of DDR3-1333 memory and sixteen 300 GB Intel 710 series solid-state drives. The drives will typically be configured as a single RAID 0 device with 4.4 TB of usable space. However, this default configuration can be changed based on the needs of the project and in collaboration with SDSC staff. Each of the I/O nodes can mount both the NFS home directories and Data Oasis, SDSC's 4 PB Lustre-based parallel file system, via two 10 GbE connections. The aggregate sequential bandwidth for the 16 SSDs within an I/O node was measured to be 4.3 GB/s and 3.4 GB/s for reads and writes, respectively. The corresponding random performance is 600 KIOPS for reads and 37 KIOPS for writes.

Recommended Use

Gordon ION awards are intended for projects that can specifically benefit from persistent, dedicated access to the flash storage. Applications that only need temporary access to the flash storage (e.g. fast scratch space, staging of data sets that will be accessed multiple times within a batch job) can do so via a regular Gordon compute cluster award.

Gordon ION is particularly well suited to database and data-mining applications where high levels of I/O concurrency exist, or where the I/O is dominated by random access data patterns. The resource should be of interest to those who, for example, want to provide a high-performance query engine for scientific or other community databases. Consequently, Gordon ION allocations are for long-term, exclusive, and dedicated use of the awardee.

SDSC staff can help you configure the I/O node based on your requirements. This includes the installation of relational databases (MySQL, PostgreSQL, DB2) and Apache web services.

Allocations

Dedicated Gordon I/O nodes are only available through the XSEDE startup allocations process. Normally a single I/O node will be awarded, but two I/O nodes can be awarded for exceptionally well-justified requests. Projects that also have accompanying heavy computational requirements can ask for up to 16 compute nodes for each I/O node.

Successful allocation requests must describe how you will make use of the I/O nodes. This should include relevant benchmarks on spinning disks, with projections of how the applications will scale when using flash drives. Additionally, the request should include a strong justification for why these should be provided as a dedicated resource-for example, providing long-term access to data for a large community via a Science Gateway

Gordon's flash-based I/O nodes are a unique resource in XSEDE and we encourage you to contact SDSC staff to discuss the use of this resource before you submit your allocation request. SDSC applications staff will be able to help you understand if this resource is a good match for your project, and subsequently, provide assistance in getting started with your project.

Last updated: February 24, 2017