Using the Linux/Athlon Cluster at OSC

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

OSC

Table of Contents

Hardware Overview

- Hardware introduction
- File server node configuration
- Login node configuration
- Compute node configuration
- I/O node configuration
- Processor and memory performance
- System area network
- External network connectivity

Hardware Introduction

The OSC Linux/IA32 cluster consists of the following:

- File server nodes (currently 1, soon 2) for NFS and PBS/Maui service.
- Login nodes (2) for interactive use, compiling, testing, etc.
- Compute nodes (128) used by jobs.
- I/O nodes (16) to support the PVFS parallel file system.
- A high-speed system area network (SAN) for inter-node communication.
- External network access.

Current File Server Node Configuration

- Quad Intel Pentium III Xeon processors running at 550 MHz with 512kB of L2 cache.
- 6 GB RAM.
- Dual UW SCSI controllers supporting 72 GB of SCSI disk (mirrored system disk, `/usr/local` for cluster-wide software).
- Dual Gigabit Ethernet interfaces.
- HIPPI interface for fast access to the OSC mass storage server (*mss.osc.edu*).

Login Node Configuration

- Dual AMD Athlon MP processors running at 1.533 GHz with 256kB of L2 cache.
- 4 GB RAM.
- ATA100 controller supporting 74 GB of IDE disk (swap, local `/tmp`).
- Gigabit Ethernet interface.
- Fast Ethernet interface.

Compute Node Configuration

- Dual AMD Athlon MP processors running at 1.4 GHz with 256kB of L2 cache.
- 2 GB RAM.
- ATA100 controller supporting 74 GB of IDE disk (swap, local `/tmp`).
- Myrinet interface (except on 16 serial-only nodes).
- Fast Ethernet interface.

I/O Node Configuration

- Dual Intel Pentium III processors running at 933 MHz with .256kB L2 cache.
- 1 GB RAM.
- ATA100 RAID controller supporting 596 GB of IDE disks in RAID 5.
- Myrinet interface (coming soon).
- Gigabit Ethernet interface.
- Fast Ethernet interface.

Processor Performance

All of the compute nodes in the OSC cluster use the AMD Athlon MP processor with a 1.4 GHz clock:

- x86 instruction decoder in front of a RISC-style super-scalar execution core with out-of-order execution.
- 64 kB L1 instruction cache, 64 kB L1 data cache, and 256 kB unified L2 cache.
- 9 execution units: 3 integer units, 3 load/store units, and 3 FP/MMX/SSE units.
- **2.8 GFLOPs peak, ~680 MFLOPs on Linpack 100x100**.

Memory Performance

All of the nodes in the OSC cluster use 133MHz DDR SDRAM memory:

- 64-bit wide data path.
- Double pumped (two transactions per clock cycle).
- 6 ns latency.
- **2.1 GB/s peak, ~735 MB/s on stream_d memory copy.**

System Area Network

Each node has a Myrinet 2000 interface:

- Switched 2 Gbit/s bidirectional network.
- Private to the cluster -- no outside traffic.
- MPI runs over Myrinet using Myricom’s GM message passing library -- no overhead from TCP/IP stack.

External Network Connectivity

- All nodes mount the filesystems containing the users' home directories from the OSC Mass Storage System, `mss.osc.edu`.
- Interactive logins using the `telnet`, `rlogin`, `rsh`, and `ssh` protocols from anywhere on the Internet are handled by the front end node, `oscbw.osc.edu`. The `ssh` protocol is preferred because it does not send clear-text passwords and uses encryption.
- Documentation can be found on the OSC Technical Information Web Server, `http://oscinfo.osc.edu/`.

The Linux Operating System

- What is Linux?
- Linux features
- Why use Linux in a cluster environment?
- Processes and threads in Linux

What is Linux?

- Linux is a freely redistributable, open source operating system developed by a programmers from all over the world.
- It is based on many of the ideas espoused by UNIX and its variants (UNICOS, IRIX, Solaris, AIX, HP/UX, etc.), but it is not directly based on UNIX code.
- It implements a superset of the POSIX and Open Group Single UNIX specifications for system interfaces.

Features of Linux

- Freely distributable with full source code.
- Runs on a variety of platforms (Intel IA32 and IA64, DEC Alpha, MIPS, Sun SPARC, several embedded processors).
- Multi-threaded, fully preemptive multitasking.
- Implements most of the POSIX and Open Group Single UNIX system APIs.
- Protocol and source compatibility with most other UNIX-like operating systems.

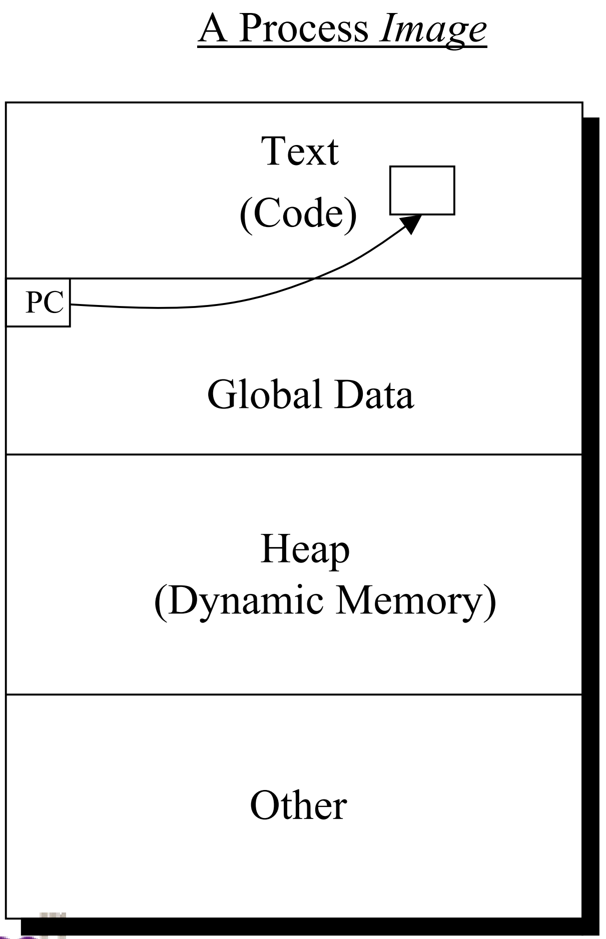
Why Use Linux in a Cluster Environment?

- Widely available.
- Inexpensive.
- Easily modified and customized.
- Compatible with most existing cluster software (MPI, batch systems, numerical libraries, etc.).
- Performs as well or better than other operating systems on the same hardware for many technical computing applications.

Processes, Threads and Load Sharing in Linux

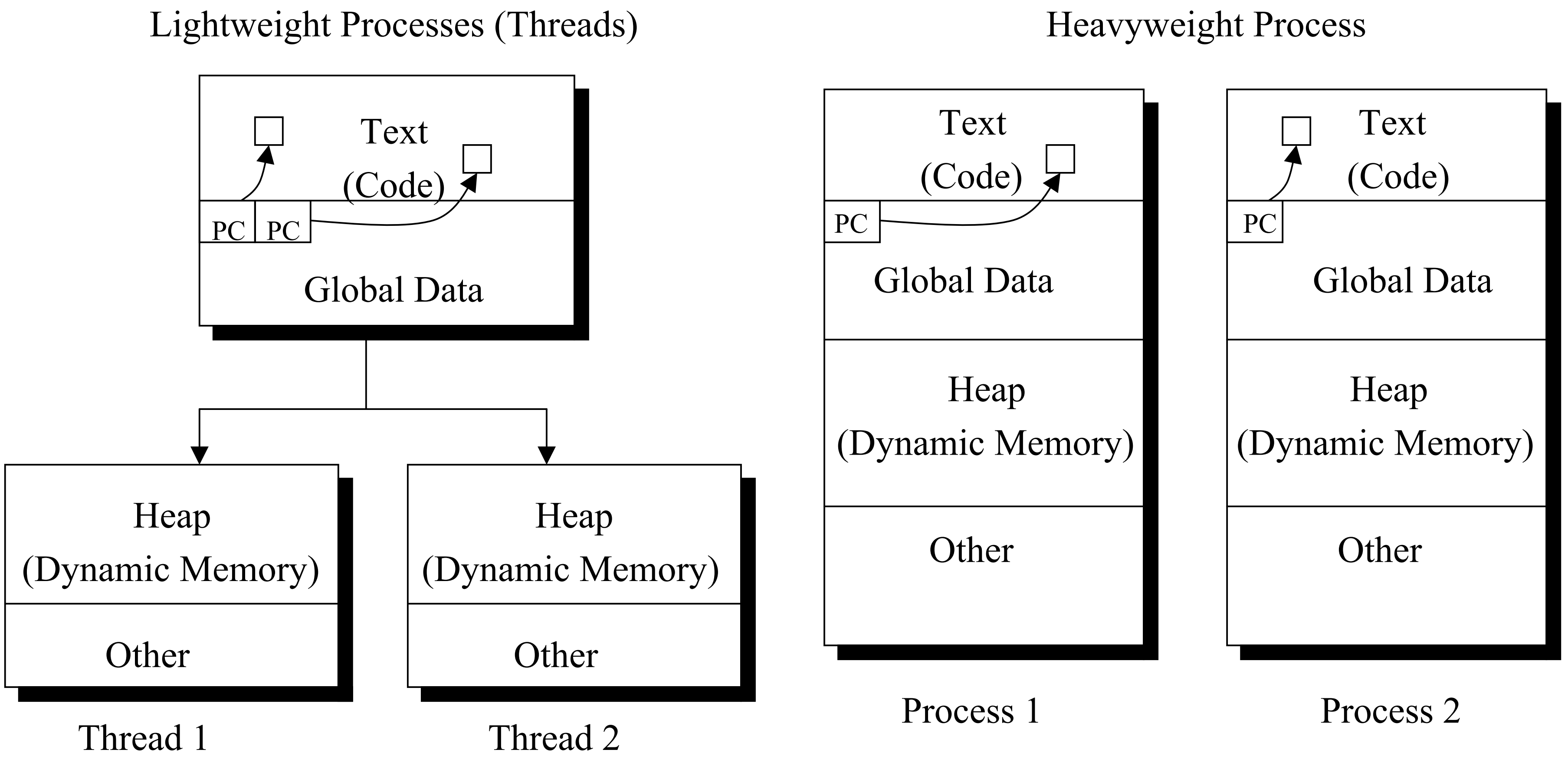
- The basic block of scheduling in UNIX has historically been the *process*.
- Recent UNIXes have also added the concept of multiple *threads* of execution within a single process.
- Linux supports both processes and threads.
- Linux's internal scheduler will also try to load-balance running processes and threads, so that they will be given full use of a processor so long as there are as many or fewer active processes/threads as there are processors.

Process - Definition



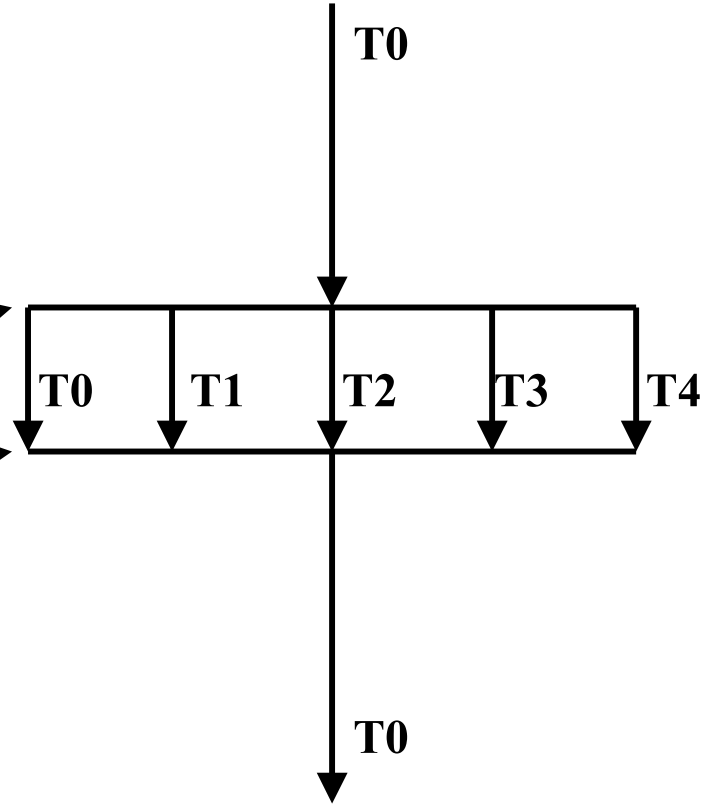
- A process is a running program.
- Elements of a process:
  - Memory (*text*, *data*)
  - Register contents
  - *Program Counter* (PC)
  - Process status
- Each process has a unique *process id*.
- Keep concepts of *process* and *processor* separate.

Types of Processes

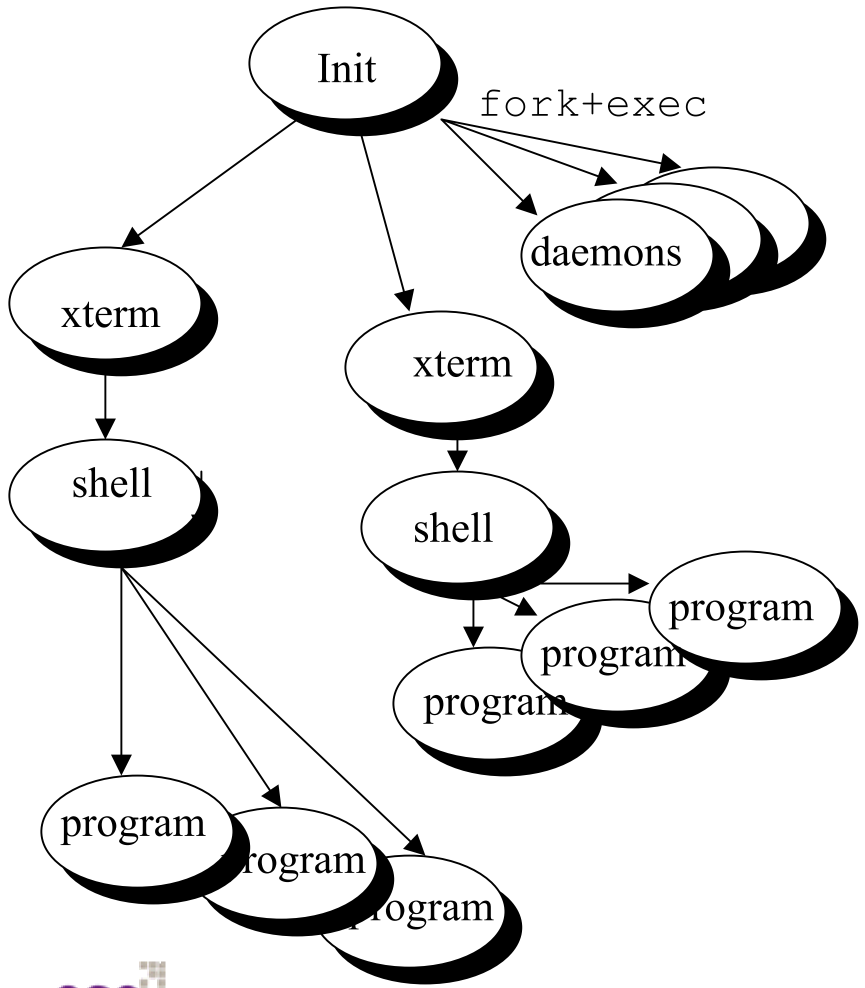


Lightweight Process - An Example

```
#
# Pseudocode for lightweight thread example
#

static data(10000)

InitializeData

SpawnThreads
JoinThreads

Output

Done
```



Process Creation



- Processes exist in a hierarchy
- The `init` process is at the top of the process hierarchy. (has process id (*pid*) of 1.
- All processes other than init are created by a *parent* process and are considered *child* processes.
- Heavyweight processes are created by a call to `fork(2)`. (Typically involves call to `exec(2)`)
- Lightweight processes are created by a call to `clone(2)` and are form a *process group*.
- Lightweight processes have the same process name.

Process States

- Processes can be in one of several states:

  S Sleeping (blocked), waiting for a resource.

  R Running (actually doing work).

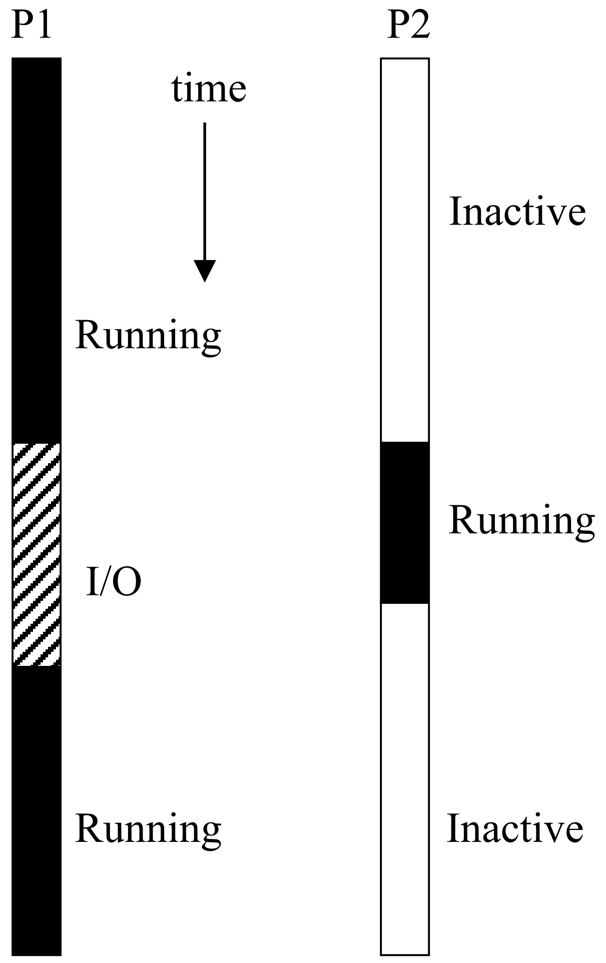
  Z Terminated and parent not waiting

  T Stopped.

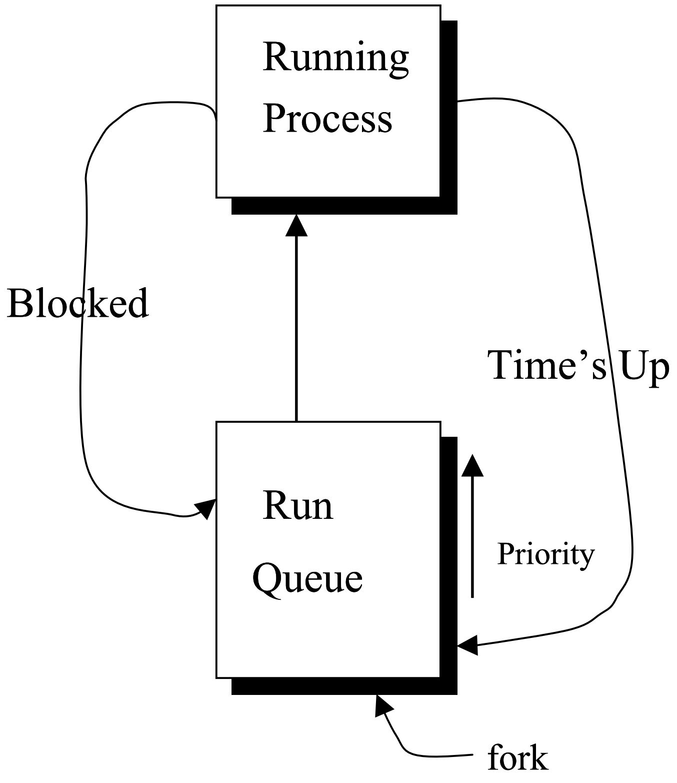
  I In intermediate state of creation.

  X Waiting for memory.

- Sometimes processes *spin wait* or *busy wait*. They eat CPU without doing anything useful.
- Processes can be *switched out* to allow a higher-priority process to run, or while waiting for something to happen like I/O.



Load Sharing - Time Slicing



- The front end node of the cluster is a **shared resource**.
- Any number of people can be using the system at any given time.
- Processes are scheduled for efficient and equitable use of CPU resources.
- The scheduler (part of the OS kernel) handles the running of processes using *run queues* and process *priorities*.
- No scheduler is perfect.
- All processes can run only for a specified *time slice* before giving up control to another process. (30 millisec default)

Priority and `nice`ness

- Every process has its own priority
- Priority is simply a number between 0 and 254. The higher the number, the lower the priority.
- As a process runs, its priority gets worse (i.e., the number gets larger). Priority is periodically updated by the kernel.
- Processes also have a "niceness" associated with them. It is represented by a number between -20 and 19. (0 is the default).
- By increasing the niceness value for a process, the priority of the process is effectively made worse. Syntax follows:

```
    /bin/nice -increment <pid>
```

- Can only increase niceness unless you are superuser
- May want to use if you don't need quick turnaround.

User Environment Management

- Accessing the cluster
- Modules
- Text editing
- System status
- File management
- 3rd party applications

Accessing the Cluster

- To access the login node of the cluster (*oscbw.osc.edu*), use `ssh`:

```
ssh oscbw.osc.edu -l myuserid
```

- `ssh` sends your commands over an encrypted stream, so your passwords and so forth can’t be sniffed over the network.

Remote X Display from the Cluster

- You can run applications which use the X Window System on the front end node and have them displayed on your remote workstation or PC.
- Using `ssh`, you should be able to display X applications remotely with no further work; `ssh` does all the necessary steps itself.

More on X Display from the Cluster

- While you can run virtually any X client program on the login node displayed to your remote workstation, the OSC systems staff would prefer that you use this **only** for programs which can't be run any other way.
- In particular, running remotely displayed `xterm` or `rxvt` sessions chews up lots of I/O bandwidth and doesn't really doesn't gain you anything over `ssh`.
- Remote X display in interactive batch jobs (something we'll discuss later with respect to debugging MPI programs) is also supported in `ssh` sessions.

Modules

- The “modules” interface is a way to allow multiple versions of software to coexist.
- They allow you to add or remove software from your environment without having to manually modify environment variables.
- This is a “Cray-ism” which OSC has adopted for all of our HPC systems.

Using modules

- You can get a list of modules you currently have loaded by running `module list`:

```
troy@oscbw:/home/troy> module list
Currently Loaded Modulefiles:
        1) pbs_2_2_0
        2) pgi_3_1
        3) modules_0_2
        4) mpich_gm
```

- To get a list of all available modules, run `module avail`:

```
troy@oscbw:/home/troy> module avail
---------- /usr/local/lanl-modules-0.2/modules ----------
hdf -> hdf_4_1_2
pbs -> pbs_2_1_13
…list continues...
```

Using Modules (con't)

- To add a software module to your environment, run `module load <modulename>`:

```
troy@oscbw:/home/troy> module load scms
troy@oscbw:/home/troy> which scms
/usr/local/scms/bin/scms
troy@oscbw:/home/troy> module list
Currently Loaded Modulefiles:
          …scms…
```

- To remove a software package from your environment, run `module unload <modulename>`:

```
troy@oscbw:/home/troy> module unload scms
troy@oscbw:/home/troy> which scms
scms: Command not found.
troy@oscbw:/home/troy> module list
Currently Loaded Modulefiles:
          …no scms…
```

Modules and the UNIX Shell

- Modules work by modifying environment variables like $PATH and $MANPATH within your shell
- Because of this, you should **NOT** explicitly set $PATH in your `.profile` or `.cshrc`; instead, you should append directories to the existing $PATH:

  `setenv PATH $HOME/bin:$PATH` (for csh users)

  `export PATH=$HOME/bin:$PATH` (for ksh users)
- Also, if you use a mixture of `csh` and `ksh` (for instance, you use `csh` interactively but write batch scripts in `ksh`), you should add the following to your `.profile` and `.cshrc`:

```
# .profile modules init
. $MODULESHOME/init/ksh
# .cshrc modules init
source $MODULESHOME/init/csh
```

Text Editing

- As with virtually all Unix systems, the front end node has the `vi` editor installed:

```
troy@oscbw:/home/troy> which vi
/usr/bin/vi
```

- The popular `emacs` and `xemacs` editors are also available, as well as `jed` (an `emacs`-like editor which uses much less memory):

```
troy@oscbw:/home/troy> which emacs
/usr/bin/emacs
troy@oscbw:/home/troy> which xemacs
/usr/local/bin/xemacs
troy@oscbw:/home/troy> which jed
/usr/bin/jed
```

System Status

- Linux supplies a number of tools for examining what the front end node is running:

```
uptime
w
ps
top
```

- In addition, there are commands for examining the PBS batch queue state on the rest of the cluster and the OSC accounting information:

`qstat` (more on this later)

`OSCusage`

`uptime` and `w`

- The `uptime` command prints out how long the system has been up, along with the number of users currently logged in and the load average (the number of processes/threads actively running) for the last minute, five minutes, and fifteen minutes:

```
troy@oscbw:/home/troy> uptime
12:22pm  up 1 day, 23:29,  6 users,  load average: 0.03, 0.01, 0.01
```

- The `w` command gives the same information as `uptime`, but also lists all the users currently logged on the system:

```
troy@oscbw:/home/troy> w
12:22pm  up 1 day, 23:29,  6 users,  load average: 0.03, 0.01, 0.01
USER     TTY      FROM             LOGIN@   IDLE   JCPU  PCPU  WHAT
djohnson ttyp0    neptune.osc.edu  Wed12pm 21:44m  0.26s 0.26s csh
troy     ttyp3    paladin.osc.edu  Tue 3pm 11:21   0.23s 0.23s csh
cls022   ttyp4    scifac209-215.dh 11:32am 11:34   0.36s 0.36s csh
…[list truncated]...
```

Monitoring Processes with `ps` and `top`

`ps (1)`

```
 USER          PID  %CPU  %MEM   SIZE    RSS  TTY  STAT  START    TIME  COMMAND
bin            210   0.0   0.0    764    212  ?   S     Jul 27    0:00  portmap
cls022        5645   0.0   0.1   1572   1060  p4  S      11:32    0:00  -csh
cls022        6446   3.0   0.0   1308    820  p4  S      12:30    0:00  vi imid1.job
daemon         257   0.0   0.0    784    108  ?   S     Jul 27    0:00  /usr/sbin/atd
```

- Use `ps` to see the current state of a process.
- `ps aux` will show the long listing for every process on the system.
- Use `top` to see CPU usage for processes.
- `kill` and `killall` send signals to processes:

```
kill -KILL <pid>
killall -KILL <progname>
```

Sample `top` output

```
 12:42pm  up 1 day, 23:48,  6 users,  load average: 0.99, 0.86, 0.47
56 processes: 54 sleeping, 2 running, 0 zombie, 0 stopped
CPU states: 100.3% user,  1.1% system,  0.0% nice,  0.0% idle
Mem:  971344K av, 336224K used, 635120K free,  21288K shrd, 200392K buff
Swap: 1025392K av,   2484K used, 1022908K free                  63844K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
 6597 cls022    14   0 29324  28M  2636 R       0 99.6  3.0   9:21 l502.exe
 6605 troy       2   0   824  824   628 R       0  1.3  0.0   0:00 top
 6064 root       0   0  1168 1168   640 S       0  0.3  0.1   0:00 sshd
 3406 root       0   0  1276 1276   868 S       0  0.1  0.1   2:54 mapper
    1 root       0   0   156  124   100 S       0  0.0  0.0   0:10 init
    2 root       0   0     0    0     0 SW      0  0.0  0.0   0:04 kflushd
    3 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 kpiod
    4 root       0   0     0    0     0 SW      0  0.0  0.0   0:01 kswapd
    5 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 md_thread
   30 root       0   0    84   48    36 S       0  0.0  0.0   0:00 kerneld
```

OSCusage

- `OSCusage` is an interface to OSC’s local accounting database.
- It lets you see the RU usage for your project on a specified date or range of dates.
- Example:

```
troy@oscbw:/home/troy> OSCusage
                      OSC Usage Statistics for Academic Grant: PZS390
                                 07/31/99  To  07/31/99
    (1 mRU = 0.001 Rsrc Units)                    BALANCE (mRU): -
   1567207.52
    USER        DATE                  mRU's USED STATUS
    work020     07/31 to 07/31              0.00 Normal Modified
    work019     07/31 to 07/31              0.00 Normal Modified
```

…[Output truncated]...

OSCusage (con't)

- By default, you'll see output for everyone on your project on the previous day. To see only your own statistics, use the -q option:

```
troy@oscbw:/home/troy> OSCusage -q
                 OSC Usage Statistics for Academic Grant: PZS390
                             07/31/99  To  07/31/99
    (1 mRU = 0.001 Rsrc Units)                BALANCE (mRU): -
   1567207.52

    USER        DATE                 mRU's USED STATUS
    troy        07/31 to 07/31            11.70 Normal Active
    Period Total for Username:            11.70
```

`OSCusage` (con't.)

- To see a date or range of dates, specify the start and end dates in MM/DD/YY form; the ending date is needed only if you want more than one day.
- The `-v` (verbose) flag will give you more details on how much was charged for CPU usage on each of OSC's machines as well as for disk usage on the mass storage server.

File Management

- File management on OSC's cluster is largely automatic -- the mass storage server automatically takes care of moving files between disk and tape.
- However, since you do get charged a small amount for the total amount of storage you're using, you may want to compress large unused files using either the `compress` or `gzip` commands. `gzip` tends to do a better compression job, but it also isn't yet universally available.
- The standard UNIX `ftp` command can be used to transfer files between the mass storage server (*mss.osc.edu*, **NOT** the front end node!) and your local workstation or PC. Since all the cluster nodes mount home directories from `mss`, any files you transfer to your home directory on `mss` will also be accessible from the cluster.
- The secure remote copy command `scp` is also available.

3rd Party Applications

- Computational Chemistry
  - Amber
  - Columbus
  - Gaussian 98
  - MPQC
  - NWChem
- Structural Analysis
  - HyperMesh solver
  - LS-Dyna3D
- Miscellaneous
  - Gnuplot

Program Development Tools and Libraries

- GNU compilers
- Portland Group compilers
- MPI compiler wrappers
- Libraries
- Debuggers
- Performance analysis tools

GNU Compilers

Virtually every Linux system includes the GNU compiler suite from the Free Software Foundation. This a freely available open source compiler system including support for:

- C (`gcc`)
- C++ (`g++`)
- Fortran 77 (`g77`)

While these are quite good compilers in terms of standards conformance, they do not generate as fast code as other compilers and lack support for parallelization.

GNU Compilers: Common Options

- `-c` (compile only; do not link)
- `-DMACRO[=value]` (defines preprocessor macro `MACRO` with optional `value`; default value is 1)
- `-g` (generate symbols for debugging; disables optimization)
- `-I/dir/name` (add `/dir/name` to the list of directories to be searched for `#include`d files)
- `-lname` (add library `libname.{a|so}` to the list of libraries to be linked -- order is important!)
- `-L/dir/name` (add `/dir/name` to the list of directories to be searched for library files)
- `-o outfile` (name resulting output file `outfile`; default is `a.out`)
- `-UMACRO` (removes definition of `MACRO` from preprocessor)

GNU Compilers: Common Options (con't.)

- `-O0` (no optimization; default)
- `-O1` (light optimization)
- `-O2` (moderate optimization)
- `-O3` (heavy optimization; may cause slight numerical differences)
- `-fexpensive-optimizations` (enables minor but expensive optimizations)
- `-finline-functions` (enables function inlining)
- `-fschedule-insns` (enables instruction scheduling and reordering)
- `-funroll-loops` (enables loop unrolling optimizations)

GNU Compilers: C/C++ Options

- `-ansi` (enforces ANSI C/C++ compliance; default; opposite of `–traditional`)
- `-pedantic` (increases strictness of language compliance)
- `-traditional` (enforces K&Rv1 C or pre-ANSI C++ compliance; opposite of `-ansi`)
- `-Wall` (enables all common warnings)

Recommened flags: `-O2 -funroll-loops -Wall -ansi -pedantic`

GNU Compilers: F77 Options

- `-ffree-form` (allows Fortran 90 style free form source)
- `-ff90` (allows some Fortran 90 constructs)
- `-finit-local-zero` (initializes all local variables to zero)
- `-malign-double` (causes word alignment of `DOUBLE PRECISION` variables)
- `-pedantic` (issues warnings on non-standard code)
- `-Wall` (enables all common warnings)
- `-Wsurprising` (issues warnings on code which may be interpreted different ways on different systems)

Recommended flags: `-O2 -funroll-loops -malign-double -Wall -pedantic`

Portland Group Compilers

Because of the performance of code generated by the GNU compilers, the OSC cluster also has the Portland Group's cluster development kit installed. This is a complete development including support for:

- C (`pgcc`)
- C++ (`pgCC`)
- Fortran 77 (`pgf77`)
- Fortran 90 (`pgf90`)
- High Performance Fortran (`pghpf`)

The Portland Group compiler suite also includes a debugger and a profiler, which we will discuss later. Complete manuals can be found on the Web at http://oscinfo.osc.edu/software/Portland/.

Portland Group Compilers: Common Options

Most of these are identical to their counterparts in the GNU compilers

- `-c` (compile only; do not link)
- `-DMACRO[=value]` (defines preprocessor macro `MACRO` with optional `value`; default value is 1)
- `-g` (generate symbols for debugging; disables optimization)
- `-I/dir/name` (add `/dir/name` to the list of directories to be searched for `#include`d files)
- `-lname` (add library `libname.{a|so}` to the list of libraries to be linked)
- `-L/dir/name` (add `/dir/name` to the list of directories to be searched for library files)
- `-o outfile` (name resulting output file `outfile`; default is `a.out`)
- `-UMACRO` (removes definition of `MACRO` from preprocessor)

Portland Group Compilers: Common Options (con't.)

- `-fast` ("best" optimization for machine; equivalent to `-O2 -Mnoframe -Munroll` on IA32 systems)
- `-O0` (no optimization)
- `-O1` (light optimization; default)
- `-O2` (heavy optimization)
- `-Mnoframe` (eliminates stack pointer operations)
- `-Munroll` (enables loop unrolling)
- `-Mvect=assoc` (enables vector optimizations)
- `-Mvect=cachesize:524288` (sets assumed L2 cache size to 512 kB)
- `-Mconcur` (enables automatic parallelization)
- `-tp p6` (enables generation of code for the Intel P6 [Pentium Pro/II/III] instruction set)

Portland Group Compilers: C Options

- `-B` (allows C++ style comments)
- `-mp` (enables support for OpenMP and SGI-style PCF pragmas for parallelization)
- `-Xa` (enforces strict ANSI C compliance)
- `-Xc` (enforces loose ANSI C compliance)
- `-Xs` (enforces strict K&Rv1 C compliance)
- `-Xt` (enforces loose K&Rv1 C compliance)

Recommended flags: `-Xa -fast -tp p6 -Mvect=assoc -Mvect=cachesize:524288`

Portland Group Compilers: C++ Options

- `-A` (enforces strict ANSI C++ compliance)
- `--exceptions` (enables ANSI C++ exceptions)
- `-mp` (enables support for OpenMP and SGI-style PCF pragmas for parallelization)
- `--prelink-objects` (enables support for template libraries within template libraries)
- `-tall` (forces all templates to be instantiated)
- `-tlocal` (forces template instantiations to be local)
- `-tnone` (forces no templates to be instantiated)
- `-tused` (instantiates only those templates used)

Recommended flags: `-A -fast -tp p6 -Mvect=assoc`

```
   -Mvect=cachesize:524288 --prelink-objects
```

Portland Group Compilers: F77/F90 Options

- `-byteswapio` (uses byte-swapping unformatted I/O compatible with Sun and SGI systems)
- `-i4` (assumes 4-byte `INTEGER`s; default)
- `-i8` (assumes 8-byte `INTEGER`s)
- `-module /dir/name` (adds `/dir/name` to the list of directories searched for F90 modules)
- `-mp` (enables support for OpenMP and SGI-style PCF directives for parallelization)
- `-r4` (assumes 4-byte `REAL`s; default)
- `-r8` (assumes 8-byte `REAL`s)
- `-Mcray=pointer` (forces compatibility with Cray CF77 pointer semantics)

Recommended flags:
```
-fast -tp p6 -Mvect=assoc
   -Mvect=cachesize:524288
```

Portland Group Compilers: HPF Options

All of the Fortran 77/90 options, plus the following:

- `-Mautopar` (attempts to automatically parallelize loops)
- `-Mcmf` (enables compatibility with CM Fortran)
- `-Mf90` (compile using Fortran 90 rather than HPF semantics)
- `-Mhpf2` (enables compatibility with revision 2 of the HPF standard)
- `-Mmpi` (uses MPI as the HPF communication mechanism)
- `-Mrpm` (uses PGI's RPM library as the HPF communication mechanism; default)
- `-Msmp` (uses shared memory as the HPF communication mechanism)
- `-Mvampir` (instruments the code to generate Vampir trace files)

Recommended flags: `-fast -tp p6 -Mvect=assoc -Mvect=cachesize:524288 -Mmpi`

MPI Compiler Wrappers

The MPICH/GM implementation of MPI uses a set of compiler scripts to keep users from having to remember how to set include and library paths for the their MPI compiles. These scripts call the system compilers to do the actual compilation. The scripts support the following languages:

- C (`mpicc` -- wrapper for `pgcc`)
- C++ (`mpiCC` -- wrapper for `pgCC`)
- Fortran 77 (`mpif77` -- wrapper for `pgf77`)
- Fortran 90 (`mpif90` -- wrapper for `pgf90`)

These compiler scripts accept the same arguments as the compiler they wrap, i.e. `mpicc` accepts the same arguments as `pgcc`, `mpif77` accepts the same arguments as `pgf77`, etc.

MPI Compiler Wrappers (con’t.)

The MPI compiler wrappers also accept a few command line arguments of their own:

- `-mpilog` (generates MPE log files compatible with the `jumpshot` MPI profiler)
- `-mpitrace` (prints trace messages on entry and exit to all MPI routines)

When the MPI Compiler Wrappers Break

- Occasionally, a program's build process will make make assumptions about quoting around the arguments for compilers that will not work with the MPI compiler wrappers (which are after all only shell scripts). In these cases, you should use the Portland Group compilers directly and use the following environment variables:

- Compile with
  - `$MPI_CFLAGS` (C)
  - `$MPI_CXXFLAGS` (C++)
  - `$MPI_FFLAGS` (F77)
  - `$MPI_F90FLAGS` (F90)

- Link with `$MPI_LIBS`

When the MPI Compiler Wrappers Break: Examples

```
troy@oscbw:/home/troy/Beowulf/mpi-c> pgcc -fast \
   $MPI_CFLAGS -DVERSION=′ ″v0.2 build 1/21/00″ ′ -c \
   vbcast.c

troy@oscbw:/home/troy/Beowulf/mpi-f77> pgf77 -fast \
   $MPI_FFLAGS -DSIZE=128 cp3.F -o cp3-128 $MPI_LIBS
```

Libraries

The OSC cluster has several Fortran numerical libraries installed which can be used in conjunction with the Portland Group compilers:

- BLAS (link with `-lblas`)
- LAPACK (link with `-llapack -lblas`)
- ScaLAPACK (compile with `mpif77` or `mpif90`, link with `${SCALAPACK} ${PBLAS} ${FBLACS}`)
- A public domain version of Cray's `libsci` FFT routines (link with `-L/usr/local/lib -lsci`)
- FFTw (`module load fftw` to use, compile with `$FFTW_FFLAGS`, link with `$FFTW_LIBS`)
- NAG F77 (`module load NAG_F77` to use, link with `$NAGF77LIB`)
- PETSc (`module load petsc` to use, look at the examples' Makefiles in `$PETSC_ROOT/examples` for how to build programs which use it)

Libraries (con't)

The OSC cluster also has some C numerical libraries:

- NAG (`module load NAG_C` to use, compile with `$NAGCINCLUDE`, link with `$NAGCLIB`)
- FFTw (`module load fftw` to use, compile with `$FFTW_CFLAGS`, link with `$FFTW_LIBS`)
- PETSc (`module load petsc` to use, look at the examples' Makefiles in `$PETSC_ROOT/examples` for how to build programs which use it)
- ScaLAPACK (compile with `mpicc` or `mpiCC`, link with `${SCALAPACK} ${PBLAS} ${CBLACS}`)

Libraries (con't)

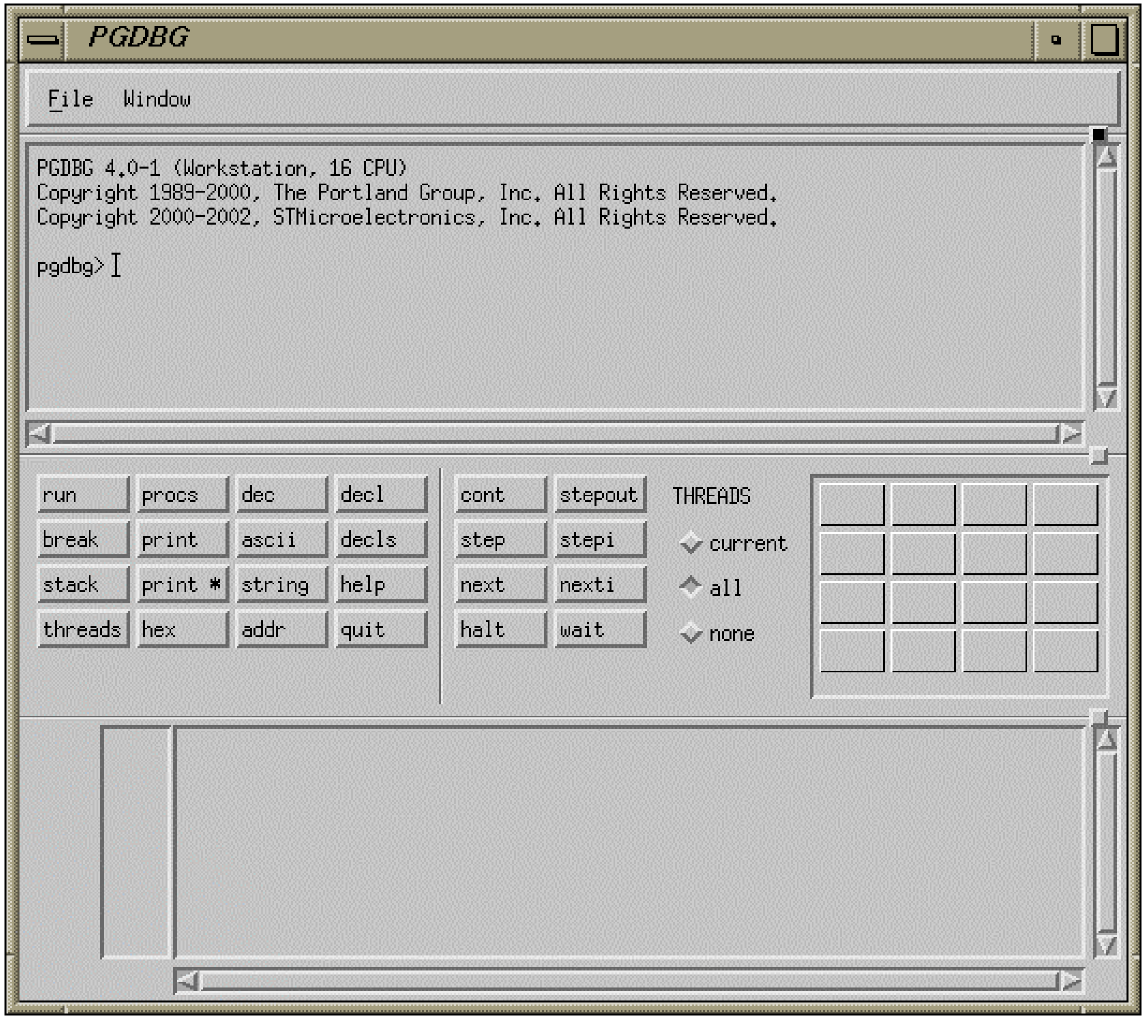
The OSC cluster also has several I/O libraries installed for writing files in platform independent formats:

- HDF (`module load hdf` to use, compile with `$HDF_INCLUDE`, link with `$HDF_LIBS`)
- HDF5 (`module load hdf5` to use, compile with `$HDF5_INCLUDE`, link with `$HDF5_LIBS`)
- NetCDF (link with `-lnetcdf` for C or Fortran, or `-lnetcdf_c++` for C++)

Debuggers

- Almost all Linux systems include the `gdb` command line symbolic debugger.
- In addition, the Portland Group compilers include a debugger, `pgdbg`.
- OSC has also purchased a license for the `totalview` parallel debugger.

Debuggers: `pgdbg`



```
PGDBG 4.0-1 (Workstation, 16 CPU)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2002, STMicroelectronics, Inc. All Rights Reserved.

pgdbg>
```

Debuggers: `totalview` Within a Batch Job

- The `totalview` debugger is designed to be run on parallel programs using MPI, OpenMP, or pthreads.

- However, the OSC cluster is set up such that MPI jobs using the Myrinet network can only be run through the PBS batch system.

- Luckily, PBS allows for interactive jobs, which can be used to run interactive programs (like a debugger) within the framework of a batch job.

Debuggers: `totalview` Example

```
troy@oscbw:/home/troy/Beowulf/multilevel> more \
   totalview.pbs
#PBS -l nodes=2:ppn=4
#PBS -l cput=3600
#PBS -j oe
#PBS -N totalview
#PBS -S /bin/ksh
#PBS -V
```

Notice that there is no script part of this job. That is because this is intended to be submitted as an interactive job.

Debuggers: `totalview` Example (con't)

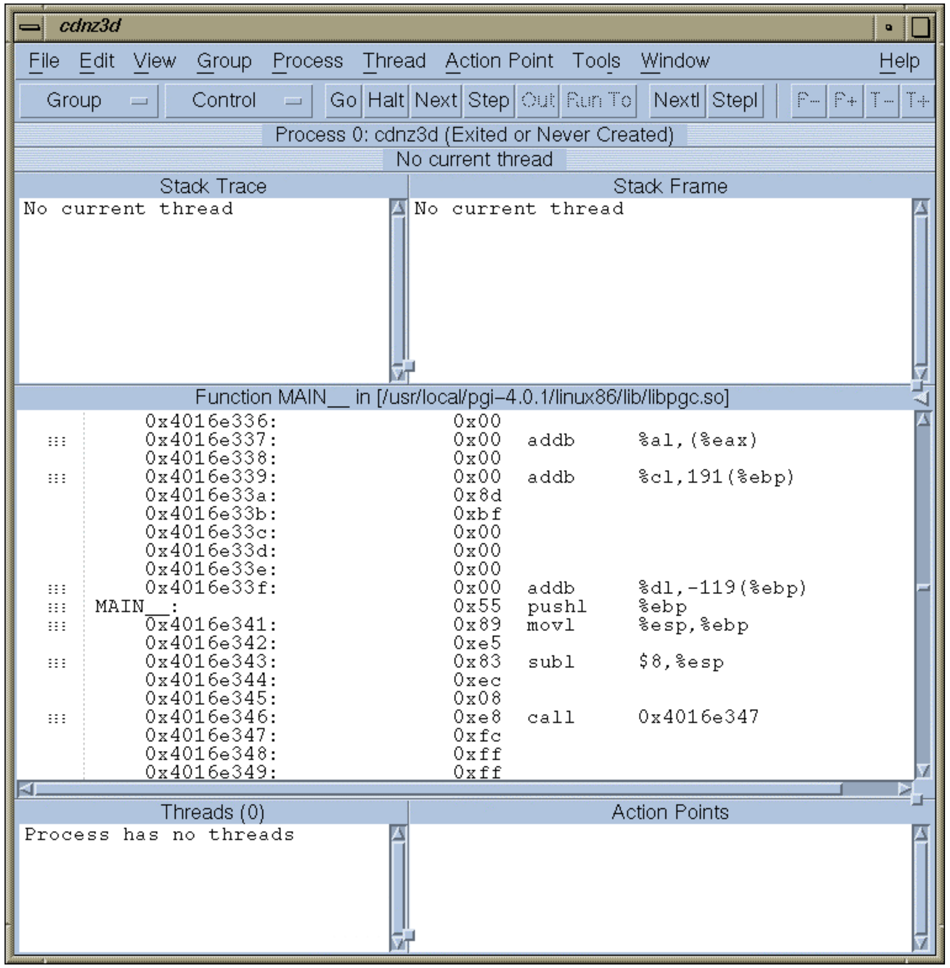
```
troy@oscbw:/home/troy/Beowulf/multilevel> qsub -I \
   totalview.pbs
qsub: waiting for job 6176.oscbw.osc.edu to start
qsub: job 6176.oscbw.osc.edu ready
```

At this point, you now have an interactive shell on one of the compute nodes which has all the limits of your batch request. You now can run `mpiexec` with the `-tv` option to invoke `totalview` on your MPI program.

```
[node02.cluster.osc.edu]$ cd Beowulf/multilevel
[node02.cluster.osc.edu]$ mpiexec -tv laplace-mpi
```

Debuggers: `totalview`

- Within `totalview`, you can set breakpoints and examine variables on a per-process basis.


cdnz3d
File Edit View Group Process Thread Action Point Tools Window Help
Group Control Go Halt Next Step Out Run To NextI StepI P- P+ T- T+
Process 0: cdnz3d (Exited or Never Created)
No current thread
Stack Trace
Stack Frame
Function MAIN__ in [/usr/local/pgi-4.0.1/linux86/lib/libpgc.so]
Threads (0)
Action Points

```
No current thread
```

```
No current thread
```

```
      0x4016e336:          0x00
:::   0x4016e337:          0x00   addb     %al,(%eax)
      0x4016e338:          0x00
:::   0x4016e339:          0x00   addb     %cl,191(%ebp)
      0x4016e33a:          0x8d
      0x4016e33b:          0xbf
      0x4016e33c:          0x00
      0x4016e33d:          0x00
      0x4016e33e:          0x00
:::   0x4016e33f:          0x00   addb     %dl,-119(%ebp)
::: MAIN__:                0x55   pushl    %ebp
:::   0x4016e341:          0x89   movl     %esp,%ebp
      0x4016e342:          0xe5
:::   0x4016e343:          0x83   subl     $8,%esp
      0x4016e344:          0xec
      0x4016e345:          0x08
:::   0x4016e346:          0xe8   call     0x4016e347
      0x4016e347:          0xfc
      0x4016e348:          0xff
      0x4016e349:          0xff
```

```
Process has no threads
```

Performance Analysis Tools

Performance analysis tools on Linux systems are currently a little more primitive than on their supercomputer cousins. However, Linux has support for the following:

- Timing
- Profiling
- Hardware performance counters

Performance Analysis Tools: Timing

- The easiest way to time a program running on a single node is with the `/usr/bin/time` command:

```
troy@oscbw:/home/troy/Beowulf/j3> /usr/bin/time j3
5415.03user 13.75system 1:30:29elapsed 99%CPU
   (0avgtext+0avgdata 0maxresident)k 0inputs+0outputs
   (255major+509333minor)pagefaults 0swaps
```

- Note that you should hardcode the path, as some shells have a built-in `time` command which is less informative.
  - `/usr/bin/time` will give results for
    - user time (CPU time spent running your program)
    - system time (CPU time spent by your program in system calls)
    - elapsed time (wallclock)
    - % CPU
    - memory, pagefault, and swap statistics
    - I/O statistics

Performance Analysis Tools: Timing (con't.)

You can also manually add calls to timing routines in your code:

- C/C++
  - Wallclock: `time`(2), `difftime`(3), `getrusage`(2)
  - CPU: `times`(2)
- Fortran 77/90
  - Wallclock: `SYSTEM_CLOCK`(3)
  - CPU: `DTIME`(3), `ETIME`(3),
- MPI (C/C++/Fortran)
  - Wallclock: `MPI_Wtime`(3)

Performance Analysis Tools: Profiling

Profiling is a method by which you can determine in which routines your code spends the most time. This usually requires support from the compiler as well as an analysis tool. The OSC cluster has two such tools:

- `gprof`
- `pgprof`

In addition, the OSC cluster also supports the `jumpshot` utility for profiling MPI codes.

Profiling: `gprof`

`gprof` is the GNU profiler. To use it, you need to do the following:

- Compile and link your code with the GNU compilers (`gcc`, `egcs`, `g++`, `g77`) using the `-pg` option flag.
- Run your code as usual. A file called `gmon.out` will be created containing the profile data for that run.
- Run `gprof progname gmon.out` to analyze the profile data.

Profiling: `gprof` Example

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> make
g77 -O2 -pg  -c cdnz3d.f -o cdnz3d.o
g77 -O2 -pg  -c sdbdax.f -o sdbdax.o
g77 -O2 -pg -o cdnz3d cdnz3d.o sdbdax.o

troy@oscbw:/home/troy/Beowulf/cdnz3d> ./cdnz3d
```

(…`gmon.out` created…)

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> gprof cdnz3d \
   gmon.out | more
```

Profiling: `gprof` Example (con't.)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 24.67    942.76   942.76  4100500     0.00     0.00  lxi_
 23.51   1841.45   898.69  4100500     0.00     0.00  leta_
 20.10   2609.66   768.21  4100500     0.00     0.00  damping_
 12.64   3092.90   483.24  4100500     0.00     0.00  lzeta_
 11.55   3534.28   441.38  4100500     0.00     0.00  sum_
  4.12   3691.73   157.45      250     0.63    14.83  page_
  2.91   3802.84   111.11      250     0.44     0.44  tmstep_
  0.41   3818.62    15.78      500     0.03     0.03  bc_
  0.03   3819.59     0.97                             pow_dd
```

(…output continues…)

Profiling: `pgprof`

`pgprof` is the profiler from the Portland Group compiler suite; it is somewhat more powerful than `gprof`. To use it, you need to do the following:

- Compile and link your code with the Portland Group compilers (`pgcc`, `pgCC`, `pgf77`, `pgf90`, `pghpf`) using the `-Mprof=func` or `-Mprof=lines` options depending whether you want function-level or line-level profiling.

- Run your code as usual. A file called `pgprof.out` will be created containing the profile data for that run.

- Run `pgprof pgprof.out` to analyze the profile data.

Profiling: `pgprof` Example

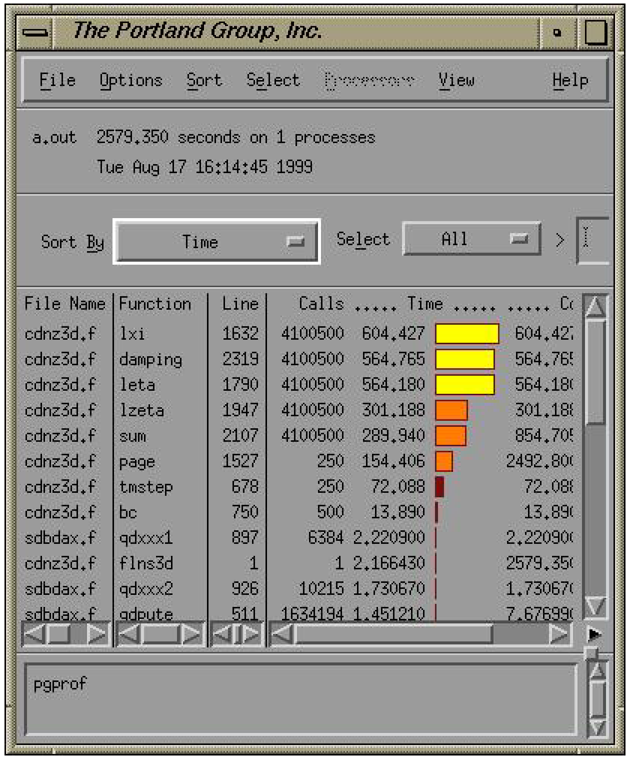
```
troy@oscbw:/home/troy/Beowulf/cdnz3d> make
pgf77 -fast -tp p6 -Mvect=assoc,cachesize:524288 -Mprof=func \
   -c cdnz3d.f -o cdnz3d.o
pgf77 -fast -tp p6 -Mvect=assoc,cachesize:524288 -Mprof=func \
   -c sdbdax.f -o sdbdax.o
pgf77 -fast -tp p6 -Mvect=assoc,cachesize:524288 -Mprof=func \
   -o cdnz3d cdnz3d.o sdbdax.o
Linking:

troy@oscbw:/home/troy/Beowulf/cdnz3d> ./cdnz3d
```

(…`pgprof.out` created…)

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> pgprof pgprof.out
```

Profiling: `pgprof` Example (con't.)

- If you have a working X display, you will get a graphical window like the following:



```
a.out   2579.350 seconds on 1 processes
        Tue Aug 17 16:14:45 1999

File Name  Function   Line     Calls ..... Time ..... ..... Cc
cdnz3d.f   lxi        1632   4100500   604.427 🟨    604.42
cdnz3d.f   damping    2319   4100500   564.765 🟨    564.76
cdnz3d.f   leta       1790   4100500   564.180 🟨    564.18
cdnz3d.f   lzeta      1947   4100500   301.188 🟧    301.18
cdnz3d.f   sum        2107   4100500   289.940 🟧    854.70
cdnz3d.f   page       1527       250   154.406 🟧   2492.80
cdnz3d.f   tmstep      678       250    72.088 🟥     72.08
cdnz3d.f   bc          750       500    13.890 |      13.89
sdbdax.f   qdxxx1      897      6384  2.220900 |    2.22090
cdnz3d.f   flns3d        1         1  2.166430 |    2579.35
sdbdax.f   qdxxx2      926     10215  1.730670 |    1.73067
sdbdax.f   qdpute      511   1634194  1.451210 |    7.67699

pgprof
```

Profiling: `pgprof` Example (con't.)

- If you do not have a working X display, you will get a command line interface like the following:

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> pgprof pgprof.out
Loading....
   Datafile   : pgprof.out
   Processes : 1
pgprof> print
Time/                           Function
     Calls  Call(%)  Time(%)  Cost(%)   Name:
------------------------------------------------------------------------
   4100500     0.00    23.43       23   lxi                  (cdnz3d.f:1632)
   4100500     0.00    21.90       22   damping              (cdnz3d.f:2319)
   4100500     0.00    21.87       22   leta                 (cdnz3d.f:1790)
   4100500     0.00    11.68       12   lzeta                (cdnz3d.f:1947)
   4100500     0.00    11.24       33   sum                  (cdnz3d.f:2107)
       250     0.02     5.99       97   page                 (cdnz3d.f:1527)
       250     0.01     2.79        3   tmstep               (cdnz3d.f:678)
pgprof> quit
```

Profiling: `jumpshot`

`jumpshot` is a Java based profiling tool that comes with the MPICH implementation of MPI. It allows you to profile all calls to MPI routines. To use `jumpshot`, you need to do the following:

- Compile your MPI code using one of the MPI compiler wrappers (`mpicc`, `mpiCC`, `mpif77`, `mpif90`) using the `-mpilog` option, and link using `-lmpe`.
- Run your MPI code as usual. A `.clog` file will be created (i.e. if your executable is named `progname`, a log file called `progname.clog` will be created).
- Run `jumpshot` on the `.clog` file (eg. `jumpshot progname.clog`)

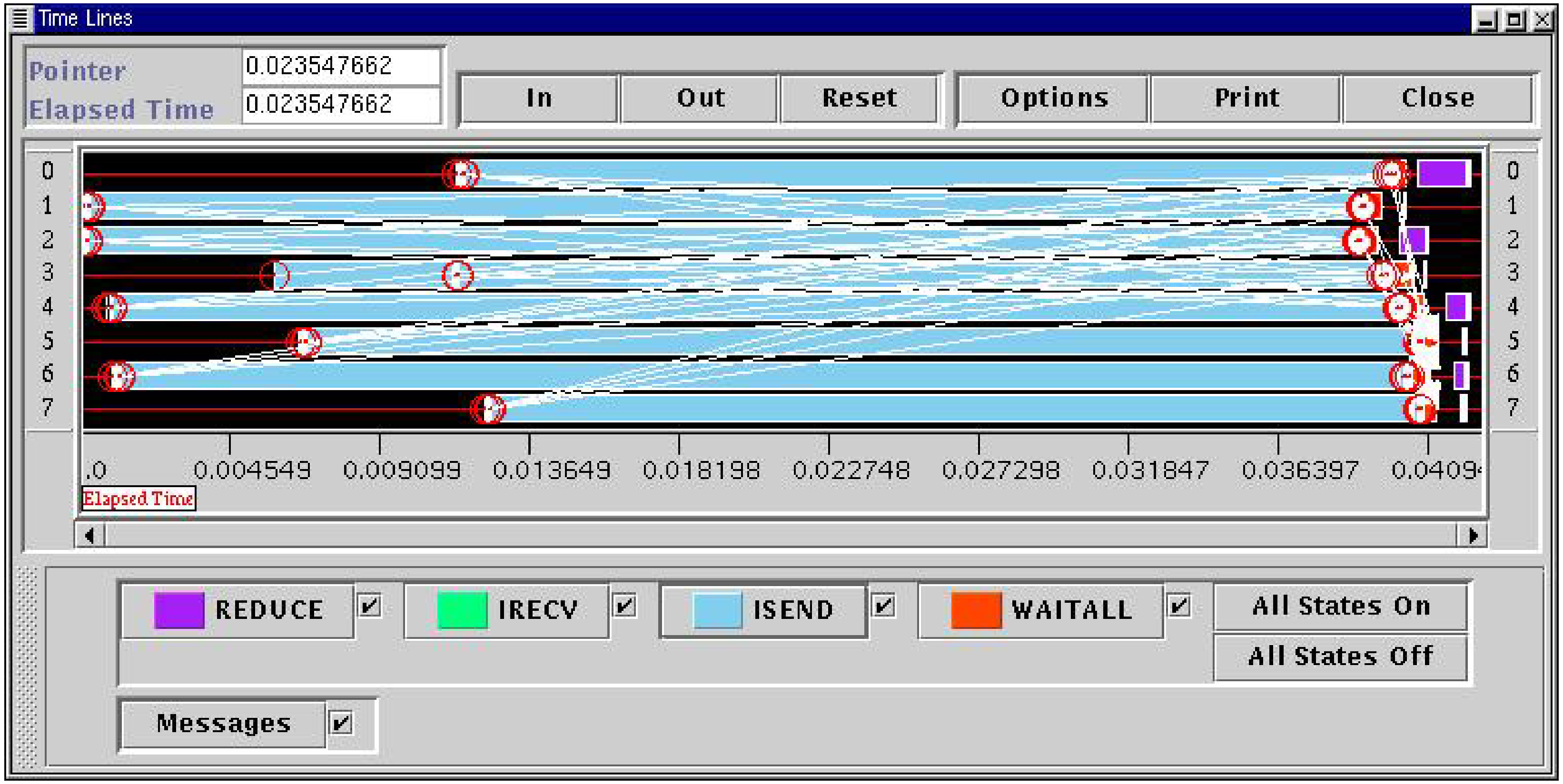
Profiling: `jumpshot` Example

```
troy@oscbw:/home/troy/Beowulf/mpi-c> more jumpshot.pbs
#PBS -l nodes=2:ppn=4
#PBS -N jumpshot
#PBS -j oe
cd $HOME/Beowulf/mpi-c
mpicc -mpilog nblock2.c -o nblock2 -lmpe
mpiexec ./nblock2

troy@oscbw:/home/troy/Beowulf/mpi-c> qsub jumpshot.pbs
```

(…`nblock2.clog` created…)

```
troy@oscbw:/home/troy/Beowulf/mpi-c> jumpshot nblock2.clog
```

Profiling: `jumpshot` Example (con't)



Hardware Performance Counters: `lperfex`

- OSC has developed a utility called `lperfex` to access the hardware performance counters built into newer Intel and AMD processors.
- `lperfex` functions much like the `time` command, in that it is run on other programs. However, it also gives the ability to count and report on hardware events. The default events if none are specified are floating point operations and L2 cache line loads.
- No special compilation is required, and `lperfex` can be used within batch jobs and with MPI programs (eg. `mpiexec lperfex -e 13 -e 14 -y ./a.out`). However, it currently does not work with multithreaded programs, such as those using OpenMP or pthreads.

Hardware Performance Counters: `lperfex` Example

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> lperfex -e \
 P6_FLOPS -y ./cdnz3d
838.239990 seconds of CPU time elapsed and 0.000000 MB of memory on
   oscbw.cluster.osc.edu

Event #                 Event                 Events Counted
-------                 -----                 --------------
   41   Floating point operations retired         3042728032

Statistics:
-----------
MFLOPS                             65.694389
```

Hardware Performance Counters: Some Countable Events

- 2: K7_DATA_CACHE_ACCESSES
- 3: K7_DATA_CACHE_MISSES
- 4: K7_DATA_CACHE_REFILLS
- 5: K7_DATA_CACHE_REFILLS_FROM_SYSTEM
- 6: K7_DATA_CACHE_WRITEBACKS
- 14: K7_INTERNAL_CACHE_LINE_INVALIDATES
- 15: K7_CYCLES_PROCESSOR_IS_RUNNING
- 16: K7_L2_REQUESTS
- 28: K7_RETIRED_INSTRUCTIONS
- 29: K7_RETIRED_OPS

Batch Processing with OpenPBS and Maui Scheduler

- Why a batch system?
- PBS basics
- Determining job requirements
- Creating a job script
- Submitting a job
- Monitoring a job
- Deleting a job
- Job output
- SMP jobs
- Parallel jobs
- Maui Scheduler
- Tips and Tricks

Why Batch?

- Interactive resource limits (10 min. CPU time, 32MB memory on the front end node -- use the `limit` command to check this).
- Improves overall system efficiency by weighing user requirements against system load.
- Makes sure all users can get equal access to resources by enforcing a scheduling policy.
- **<u>Only way to access compute nodes!</u>**

Introduction to PBS

- PBS is short for “Portable Batch System”; it is an open source batch queuing system.
- It is an outgrowth/extension of the NQS batch queuing system from the NAS project at NASA Ames Research Center.
- PBS is available for virtually anything that is UNIX-like, including Linux, the BSDs, UNICOS, IRIX, Solaris, AIX, HP/UX, and Digital UNIX.

How PBS Handles a Job

- User determines resource requirements for a job and writes a batch script.
- User submits job to PBS with the `qsub` command.
- PBS places the job into a queue based on its resource requests and runs the job when those resources become available.
- The job runs until it either completes or exceeds one of its resource request limits.
- PBS copies the job’s output into the directory from which the job was submitted and optionally notified the user via email that the job has ended.

Determining Job Requirements

- For single CPU jobs, PBS needs to know at least two resource requirements:
  - CPU time
  - memory

- For multiprocessor parallel jobs, PBS also needs to know how many nodes/CPUs are required.

- Other things to consider:
  - Job name?
  - Working in `/tmp` or `$TMPDIR`?
  - Where to put standard output and error output?
  - Should the system email when the job completes?

Determining Job Requirements (con't)

- Memory requirements can be estimated using the `size` command:

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> size -A -d cdnz3d
cdnz3d  :
…several lines deleted...
Total      32745200
```

- The output of `size` is in bytes, so this program above requires about 32 MB. Note that the `size` command does not take dynamic memory into account.

- CPU time requirements can be determined by running short jobs interactively; however, this requires you to understand how CPU time scales with the size of your problem.

- The number of CPUs used is up to you, but you are limited to the number physically available (currently 128).

PBS Job Scripts

- An PBS job script is just a regular shell script with some comments (the ones starting with `#PBS`) which are meaningful to PBS. These comments are used to specify properties of the job.

- PBS job scripts always start in your home directory, `$HOME`. If you need to work in another directory, your job script will need to `cd` to there.

- Every PBS job has a unique temporary directory, `$TMPDIR`. This in on each compute node's local disk array and thus is much faster than your home directory, which is mounted over the network from the mass storage server. For best I/O performance, you should try to copy all the files you need into `$TMPDIR`, do your work there, and then copy any files you want to keep back to your home directory.

PBS Job Scripts (con't)

- Useful PBS options:
  - `-e errfile` (redirect standard error to `errfile`)
  - `-I` (run as an interactive job)
  - `-j oe` (combine standard output and standard error)
  - `-l cput=N` (request N seconds of CPU time; N can also be in hh:mm:ss form)
  - `-l mem=N[KMG][BW]` (request `N` {kilo|mega|giga}{bytes|words} of memory)
  - `-l nodes=N:ppn=M` (request N nodes with M processors per node)
  - `-m e` (mail the user when the job completes)
  - `-m a` (mail the user if the job aborts)
  - `-o outfile` (redirect standard output to `outfile`)
  - `-N jobname` (name the job `jobname`)
  - `-S shell` (use `shell` instead of your login shell to interpret the batch script; must include a complete path)
  - `-V` (job inherits the full environment of the current shell, including `$DISPLAY`)

A First Batch Script

- Here is a simple batch job:

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> more cdnz3d.pbs
#PBS -l cput=40:00:00
#PBS -l mem=36MB
#PBS -l nodes=1:ppn=1
#PBS -N cdnz3d
#PBS -j oe
#PBS -S /bin/ksh
cd $HOME/Beowulf/cdnz3d
cp *.dat cdnz3d.in cdnz3dxyz.bin $TMPDIR
cd $TMPDIR
/usr/bin/time ./cdnz3d > cdnz3d.hist
cp cdnz3d.out cdnz3dq.bin $HOME/Beowulf/cdnz3d
```

This job asks for one CPU on one node, 36MB of memory, and 40 hours of CPU time. Its name is "cdnz3d".

Submitting a Job

- To submit a job to PBS, use the `qsub` command:

```
troy@oscbw:/home/troy/Beowulf/cdnz3d> qsub cdnz3d.pbs
102.oscbw.cluster.osc.edu
```

- `qsub` can take all of the options show for job scripts on the command line as well; specifying these on the command line overrides settings in the job script.

- Notice that `qsub` prints a request number (102 in the case shown above). This number is important for finding the output files generated by this run as well as for killing the job if necessary.

Monitoring a Job

- The status of all the jobs running on the OSC cluster can be shown with the `qstat` command:

```
troy@oscbw:/home/troy> qstat -a
oscbw.cluster.osc.edu:
                                                            Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
80.oscbw.clust osu2376  serial   h1.com       1207   1   1   64mb 16:40 R 01:05
86.oscbw.clust troy     serial   cdnz3d        776   1   1   36mb 40:00 R 00:00
93.oscbw.clust cls022   serial   NAME          787   1   1  128mb 11:06 R 00:00
101.oscbw.clus cls022   SMP      imid1        6542   1   2   64mw 10:00 R 00:40
```

`qstat` Output Fields

- `Job Id` (request number)
- `Username` (userid)
- `Queue` (queue the job is in)
- `Jobname` (name of the job)
- `SessId` (job identifier)
- `NDS` (number of nodes requested)
- `TSK` (number of CPUs per node requested)
- `Req'd Memory` (memory requested [if waiting] or used [if running])
- `Req'd Time` (CPU time requested)
- `S` (status)
  - `R` (running)
  - `Q` (queued and waiting)
- `Elap Time` (time the job has been running)

Monitoring a Job (con't)

- The stdout and stderr streams from a job can be viewed while a job is running using the `qpeek` command:
  - `qpeek -c request_number` shows all of stdout (c == "cat")
  - `qpeek -h -# request_number` shows the first # lines of stdout
  - `qpeek -t -# request_number` shows the last # lines of stdout
  - `qpeek -f -# request_number` shows the last # lines of stdout and listens for more output until interrupted (Ctrl-C)
  - Adding the `-e` flag causes qpeek to look at stderr rather than stdout.
  - If the `-#` flag is omitted with the `-h`, `-t`, or `-f` options, 10 lines are shown

- To get a listing of all the currently running processes associated with a job, use the `qps` command:
  - `qps request_number`

Killing a Job

- If, for whatever reason, you need to delete a queued job or kill a running job, use the `qdel` command.
- Usage: `qdel request_number`

Job Output

- When an PBS job ends, it writes out two files in the directory from which it was submitted:
  - `<jobname>.e<request_number>` (stanard error)
  - `<jobname>.o<request_number>` (standard output)

- These two files can be combined using the `-j oe` option, or directed to set file names using the `-e errfile` and `-o outfile` options.

- These are in addition to any files generated by the programs run in your job.

SMP Jobs

So far, the job scripts we've seen have been serial, uniprocessor jobs. The following is an example of a job that used more than one processor on a single node:

```
troy@oscbw:/home/troy/Beowulf/omp> more smp.pbs
#PBS -N smp
#PBS -j oe
#PBS -S /bin/ksh
#PBS -l nodes=1:ppn=4
#PBS -l mem=128mb
#PBS -l cput=0:01:00
cd $HOME/Beowulf/omp
export OMP_NUM_THREADS=4
/usr/bin/time ./matmul-omp
```

More on SMP and Serial Jobs

- The only real difference between a uniprocessor job and an SMP job (at least from PBS's point of view) is the `-l nodes=1:ppn=4` limit in the SMP job. This tells PBS to allow the job to run four processes (or threads) concurrently on one node.

- If you simply request a number of nodes (eg. `-l nodes=1`), PBS will assume that you want one processor per node.

Parallel Jobs

Both serial and SMP jobs run on only 1 node. However, most MPI programs should be run on more than 1 node. Here is an example of how to do that:

```
troy@oscbw:/home/troy/Beowulf/mpi-c> more nblock.pbs
#PBS -N nblock
#PBS -j oe
#PBS -l nodes=4:ppn=4
#PBS -l cput=1:00:00
#PBS -l mem=256MB
cd ~/Beowulf/mpi-c
mpiexec ./nblock
```

More on Parallel Jobs

- The `mpiexec` command is used to run MPI jobs over the Myrinet in PBS. It figures out from PBS on which nodes a job is supposed to run and starts it running on only those nodes.
- You can use (and we encourage you to use!) more than one processor per node in parallel jobs. To use four processors per node on `N` nodes, add a `-l nodes=N:ppn=4` limit to your job.
- If you need to run a shell command on all of the nodes assigned to your job (eg. copying a data file into or out of `$TMPDIR`), use `pbsdcp`:

```
cd $HOME/Beowulf/mpi-c
# scatter executable and input
pbsdcp -s mpiprog input.dat $TMPDIR
cd $TMPDIR
mpiexec ./mpiprog
# gather output files
pbsdcp -g "output.*" $HOME/Beowulf/mpi-c
```

Maui Scheduler

- The OSC cluster systems use the Maui scheduler rather than one of the several schedulers that comes with PBS, because Maui has a number of features that the PBS schedulers do not:
  - Advance reservations
  - Fair-share scheduling
  - Quality of service levels
- Maui also comes with its own set of tools for checking on job state:
  - `showq` (lists currently running and queued jobs)
  - `showstart` (estimates start time of a job)
  - `showbf` (describes resources currently available for backfill scheduling)

Maui Scheduler: `showq`

```
[oscbw]$ showq
ACTIVE JOBS--------------------
           JOBNAME USERNAME      STATE  PROC   REMAINING            STARTTIME
             …
             87706  osu2779    Running    16    10:57:11  Thu May  2 09:13:08
             87710  osu2778    Running    16    11:03:27  Thu May  2 09:19:24
             87712  osu2778    Running    16    11:05:12  Thu May  2 09:21:09
             88620     troy    Running     4  1:00:33:20  Fri May  3 13:49:17
             87063  osu2779    Running     8  1:03:10:00  Wed May  1 09:25:57
             …
    25 Active Jobs     133 of  142 Processors Active (93.66%)
                        67 of   71 Nodes Active      (94.37%)

IDLE JOBS----------------------
           JOBNAME USERNAME      STATE  PROC     WCLIMIT            QUEUETIME
             88519   utl170       Idle     1  3:01:01:00  Fri May  3 11:06:01
1 Idle Job

BLOCKED JOBS----------------
           JOBNAME USERNAME      STATE  PROC     WCLIMIT            QUEUETIME
             88449  osu2779       Idle    16  1:16:00:00  Fri May  3 08:52:47
Total Jobs: 27   Active Jobs: 25   Idle Jobs: 1   Blocked Jobs: 1
```

Terminology of `showq` Listings

- Active jobs: Jobs which are currently running. Active jobs are listed in order of expected completion, soonest first.
- Idle jobs: Jobs which are not running but can run once sufficient resources become available. Idle jobs are listed in priority order, for highest to lowest. The highest priority idle job has a reservation to run as soon as sufficient resources become available; all other idle jobs are candidates for backfill.
- Blocked jobs: Jobs which are not running because they are held or exceed a job limit policy. Blocked jobs will not run until they move into the idle jobs list.

Maui Scheduler: `showstart`

```
[oscbw]$ showstart 88519
job 88519 requires 1 proc for 3:01:01:00
Earliest start is in       7:02:27:31 on Fri May 10 17:00:00
Earliest Completion is in 10:03:28:31 on Mon May 13 18:01:00
Best Partition: DEFAULT
```

- Note that `showstart` only gives the scheduler's best estimate based on the current queue state, which can (and will) change as jobs are submitted, run, and exit.
- Advance reservations (such as scheduled system downtime) can also influence job start time.

Maui Scheduler: showbf

```
[oscbw]$ showbf
backfill window (user: 'troy' group: 'G-0541' partition: ALL)
Fri May  3 14:41:14

 23 procs available for    2:17:18:46
```

- `showbf` can give a rough idea of how many processors are immediately available, and how long they're available for.

- As with `showstart`, the output from `showbf` is only an estimate.

Tips and Tricks

- The following `csh` aliases are handy for checking what the PBS load on the Beowulf cluster is like:

```
alias myjobs 'qstat -a | grep `whoami`'
alias rjobs 'qstat -r | grep "R[0-9 ]” | grep -v IDENT'
alias nrjobs 'rjobs | wc -l'
alias qjobs 'qstat -i | grep "Q[a-z ]” | grep -v TSK'
alias nqjobs 'qjobs | wc -l'
alias njobs 'echo `nrjobs` running \+ `nqjobs` queued’
```

- There is also a graphical utility called `xpbs` which you can use to construct, submit, and track PBS jobs.

SMP Programming with OpenMP

- What’s OpenMP?
- A simple OpenMP program
- Compiling OpenMP programs
- Running OpenMP programs
- OpenMP and the Portland Group compilers

What's OpenMP?

OpenMP is a *de facto* standard for portable, directive-based threaded parallel programming for SMP systems. It consists of:

- A set of compiler directives.
- A library of support functions.

OpenMP is supported by a number of vendors' compilers, including Cray, SGI, IBM, HP/Compaq, the Portland Group, and Kuck and Associates.

A Simple OpenMP Program

```
troy@oscbw:/home/troy/Beowulf/omp> more omphw.f90
PROGRAM omphw
IMPLICIT NONE
INTEGER,EXTERNAL :: omp_get_thread_num

!$OMP PARALLEL
WRITE (*,*) 'Hello from thread ',omp_get_thread_num()
!$OMP END PARALLEL

END PROGRAM omphw
```

Compiling OpenMP Programs

- To compile a program with OpenMP support, you need to use one of the Portland Group compilers and add the `-mp` option:

```
troy@oscbw:/home/troy/Beowulf/omp> pgf90 -mp omphw.f90 \ -
   o omphw
```

- This is in addition to any optimization flags and so forth, of course.
- If you have an OpenMP program that uses one of the library calls (like `omp_get_thread_num()` in the previous example) and compile without the `-mp` flag, the compiler will complain that it can’t link in the OpenMP library.

Running OpenMP Programs

To run an OpenMP program, you first need to tell the program how many threads to use. This can be done in two ways:

- Hardcoded into the program source using the `omp_set_num_threads()` function.
- Set at run-time using the `OMP_NUM_THREADS` environment variable.

Once the number of threads is set, you can run your program like any other:

```
troy@oscbw:/home/troy/Beowulf/omp> setenv \
  OMP_NUM_THREADS 2
troy@oscbw:/home/troy/Beowulf/omp> ./a.out
 Hello from thread 0
 Hello from thread 1
```

Running OpenMP Programs in Batch

To run an OpenMP program in batch, make sure to request multiple processors and set `OMP_NUM_THREADS` in your batch job:

```
troy@oscbw:/home/troy/Beowulf/omp> more omphw.pbs
#PBS -N omphw
#PBS -j oe
#PBS -S /bin/ksh
#PBS -l nodes=1:ppn=2
#PBS -l mem=128mb
#PBS -l vmem=128mb
#PBS -l cput=1:00

cd $HOME/Beowulf/omp
export OMP_NUM_THREADS=2
/usr/bin/time ./omphw
```

OpenMP and the Portland Group Compilers

There are a few known limitations in the Portland Group compilers' OpenMP support under Linux:

- There are a few directive clauses (`schedule(dynamic)` and `schedule(guided)`) which are currently ignored.
- Nested parallelism (i.e. a `!$omp parallel do` inside another `!$omp parallel do`) is not supported.

None of these are particularly egregious problems.

Parallel Programming with MPI

- What’s MPI?
- A simple MPI program
- Compiling MPI programs
- Running MPI programs

What’s MPI?

MPI is the *de facto* standard for portable message passing parallel programming on distributed memory systems. It consists of:

- A message passing library
- A run-time environment (`mpiexec` and compiler wrappers)

MPI is supported by all of the major parallel machine manufacturers (Cray, HP/Compaq, IBM, SGI, Sun), and there are several third-party implementations for various hardware and software platforms. The OSC cluster uses MPICH/ch_gm, which is a version of the MPICH reference implementation of MPI that runs over Myrinet; this supports all of the MPI-1.1 standard as well as the MPI-IO part of the MPI-2 standard.

A Simple MPI Program

```
troy@oscbw:/home/troy/Beowulf/mpi> more mpihw.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   int rank,size;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   printf("Hello from node %d of %d\n",rank,size);
   MPI_Finalize();
   return(0);
}
```

Compiling MPI Programs

- To compile an MPI program, you need to compile with the MPI compiler wrappers (`mpicc`, `mpiCC`, `mpif77`, and `mpif90`):

```
troy@oscbw:/home/troy/Beowulf/mpi> mpicc -o mpihw \
  mpihw.c
```

- The MPI compiler wrappers accept the same arguments as the compilers they wrap as far as optimization and so forth.

Running MPI Programs

- To manage contention for the Myrinet SAN, OSC asks that MPI programs be run in batch only.
- OSC provides a program called `mpiexec` which automatically determines from PBS on which nodes an MPI program is allowed to run and starts the program running:

```
troy@oscbw:/home/troy/Beowulf/mpi> more mpihw.pbs
#PBS -N mpihw
#PBS -j oe
#PBS -l nodes=4:ppn=1
#PBS -l mem=50MB
#PBS -l cput=1:00

cd $HOME/Beowulf/mpi
mpiexec ./mpihw
```

Running MPI Program (con't.)

```
troy@oscbw:/home/troy/Beowulf/mpi> qsub mpihw.pbs
245.oscbw.cluster.osc.edu
```

…Wait for job to complete…

```
troy@oscbw:/home/troy/Beowulf/mpi> more mpihw.o245
Hello from node 0 of 4
Hello from node 1 of 4
Hello from node 2 of 4
Hello from node 3 of 4
```

Multilevel Parallel Programming

- A collision between OpenMP and MPI
- A simple multilevel parallel program
- Compiling multilevel parallel programs
- Running multilevel parallel programs
- When does multilevel parallel make sense?
- Multilevel parallelism tips and tricks

A Collision Between OpenMP and MPI

- Because of the architecture of the OSC cluster (i.e. a cluster of SMP systems), it is possible to write programs which take advantage of both the message passing and shared memory environments simultaneously.
- In this type of approach, MPI message passing is used for coarse-grained parallelism between nodes, and OpenMP compiler directives are used for fine-grained parallelism within a node.
- This approach allows you to take maximum advantage of the compute power of the nodes, because there is less contention between multiple processes for the Myrinet interface cards.
- Codes written in this fashion are also ready for use on **extremely** large systems such as the DOE ASCI Blue Mountain (6000+ CPUs) and ASCI White (8000+ CPUs) supercomputers.

A Simple Multilevel Parallel Program

```
troy@oscbw:/home/troy/Beowulf/multilevel> more mlhw.f90
PROGRAM mlhw
INCLUDE 'mpif.h'
INTEGER :: ierr,rank,size,tnum
INTEGER,EXTERNAL :: omp_get_thread_num
CALL MPI_Init(ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD,size,ierr)
CALL omp_set_num_threads(4)
!$OMP PARALLEL PRIVATE(tnum)
tnum=omp_get_thread_num()
!$OMP CRITICAL
WRITE (*,*) 'Hello from thread ',tnum,' on node ',rank,' of ',size
!$OMP END CRITICAL
!$OMP BARRIER
!$OMP END PARALLEL
CALL MPI_Finalize()
END PROGRAM mlhw
```

Compiling Multilevel Parallel Programs

- You need to compile multilevel parallel programs with the MPI compiler wrappers (`mpicc`, `mpif77`, `mpif90`, etc.).
- However, you also need to use the `-mp` option to enable OpenMP support:

```
troy@oscbw:/home/troy/Beowulf/multilevel> mpif90 -fast \ -
  mp -o mlhw mlhw.f90
Linking:
```

- This should also work using the PCF directives rather than OpenMP; however, OpenMP is the preferred method.

Running Multilevel Parallel Programs

- As with MPI programs, multilevel parallel programs must be run in batch.
- Also, setting the `OMP_NUM_THREADS` environment variable does not work in multilevel parallel programs; you **must** use the `omp_set_num_threads()` library call in your program instead.

Running Multilevel Parallel Programs in Batch

```
troy@oscbw:/home/troy/Beowulf/multilevel> more mlhw.pbs
#PBS -l nodes=2:ppn=4
#PBS -j oe
#PBS -N mlhw

cd $HOME/Beowulf/multilevel
mpiexec -pernode ./mlhw

troy@oscbw:/home/troy/Beowulf/multilevel> qsub mlhw.pbs
247.oscbw.cluster.osc.edu
```

Running Multilevel Parallel Programs in Batch (con’t.)

```
troy@oscbw:/home/troy/Beowulf/multilevel> more mlhw.o247
 Hello from thread 0 on node 0 of 2
 Hello from thread 0 on node 1 of 2
 Hello from thread 1 on node 1 of 2
 Hello from thread 1 on node 0 of 2
 Hello from thread 2 on node 0 of 2
 Hello from thread 2 on node 1 of 2
 Hello from thread 3 on node 1 of 2
 Hello from thread 3 on node 0 of 2
```

When Does Multilevel Parallel Make Sense?

Obviously, multilevel parallel programming is not easy, because you need to know both OpenMP and MPI. However, the following types of applications may benefit from multilevel parallel approaches:

- Existing MPI applications which have vector-style nested loop computations.
- Existing OpenMP applications which are amenable to domain decomposition with MPI.
- Applications with multiple interacting grid zones which can be treated “mostly” independently (eg. multiblock CFD solvers).

Multilevel Parallelism Tips and Tricks

- Make sure all MPI calls are outside of any OpenMP parallel regions; otherwise each thread will try to call the MPI routine, possibly resulting in deadlock. (In theory, wrapping MPI calls in `master` or `single` directives should also work; however, in practice this seems to have problems.)
- You should only invoke as many MPI processes as there are Myrinet interface cards available to your job (currently 2 per node). You can use the `-n N` option to force `mpiexec` to start `N` MPI processes rather than the default of 1 MPI process per requested processor. You can also use the `-pernode` option to force `mpiexec` to run 1 MPI process per node.

PVFS Parallel File System

- OSC Parallel I/O Cluster
- Accessing PVFS
- Examples
  - Serial jobs
  - Parallel jobs
- When to Use PVFS
- Caveats

OSC Parallel I/O Cluster

- A relatively new service is the parallel I/O cluster:
  - 16 I/O nodes, each with
    - 2 Pentium III processors running at 933MHz
    - 1 GB RAM
    - 3ware IDE RAID controller
    - 8 80-GB disks in RAID 5 (~520 GB usable space)
    - Gigabit and 100Mbit Ethernet interfaces
  - PVFS software from Clemson University and Argonne National Lab
    - Equivalent of RAID 0 (striping) across the I/O nodes
    - ~8 TB of usable space, mounted on `/pvfs`
    - Large block sizes (64kB by default, settable on a per-file basis at the time of file creation)
    - Accessible in two ways
      - Linux file system driver for standard UNIX file semantics
      - MPI-IO for high performance parallel I/O

Accessing the PVFS Parallel File System

- To access the PVFS file system from a batch job, you'll need to tell the batch system you intend to use it by adding a `pvfs` attribute to your job's `nodes` request:

```
#PBS -l nodes=4:ppn=2:pvfs
```

- In a batch job which requests PVFS, there will be an environment variable `$PFSDIR` -- this is similar to `$TMPDIR` in that it is a directory that only exists for the duration of the job, but it resides on PVFS and is accessible by all the nodes in your job (as opposed to `$TMPDIR` which is private to each node).

Example: Serial Job Using PVFS

```
[oscbw]$ cat bigfile.pbs
#PBS -N bigfile
#PBS -j oe
#PBS -l nodes=1:ppn=2:pvfs
#PBS -l walltime=10:00:00
cd myscience
cp input.dat $PFSDIR
cd $PFSDIR
$HOME/myscience/bigfileapp
cp output.dat $HOME/myscience
```

Example: Parallel Job Using PVFS

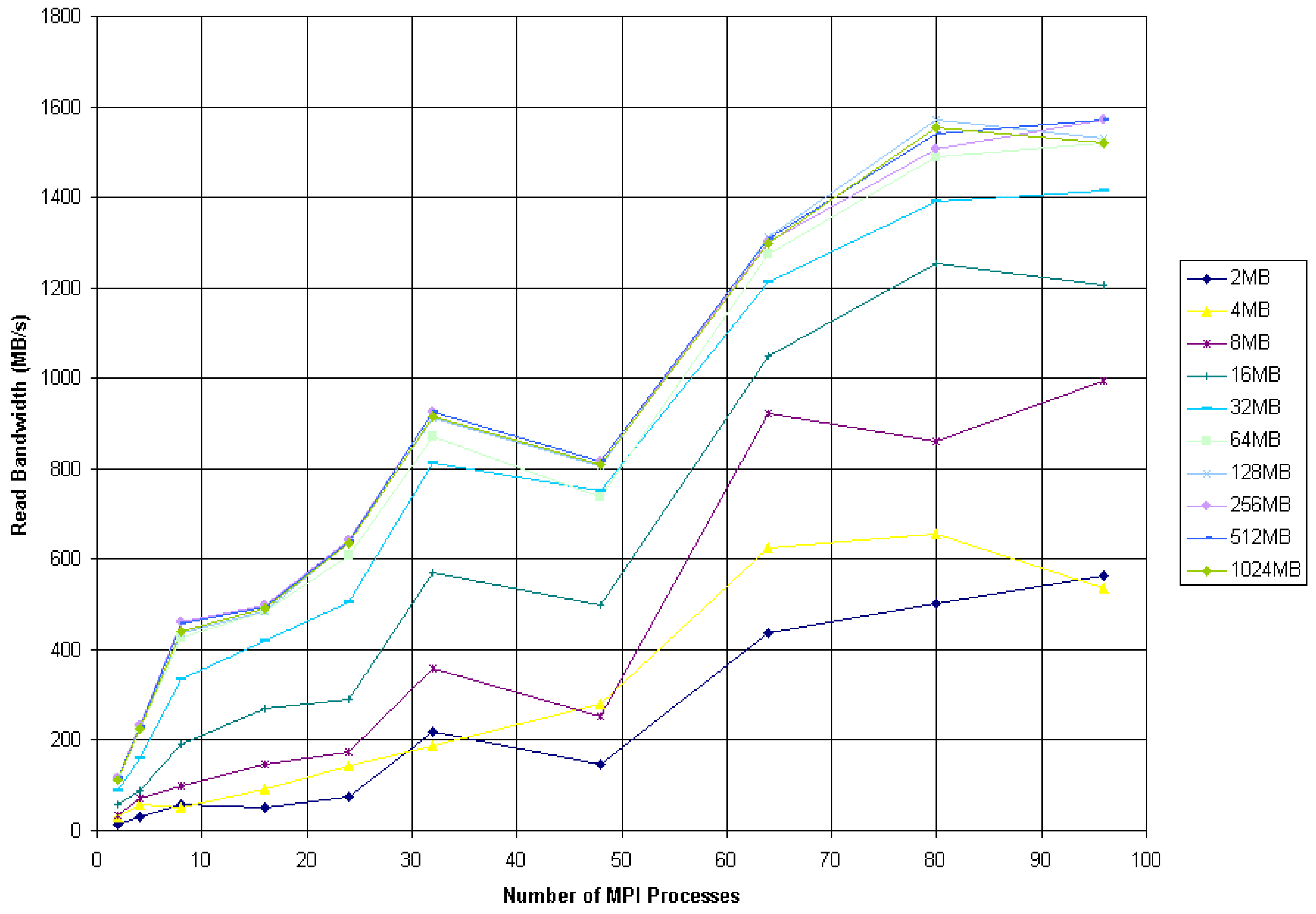
```
[oscbw]$ cat mpi-io.pbs
#PBS -N mpi-io
#PBS -j oe
#PBS -l nodes=8:ppn=2:pvfs
#PBS -l walltime=24:00:00
cd $HOME/myscience
pbsdcp parallel-io-app $TMPDIR
cp input.dat $PFSDIR
cd $PFSDIR
mpiexec $TMPDIR/parallel-io-app
cp output.dat $HOME/myscience
```

When to Use PVFS

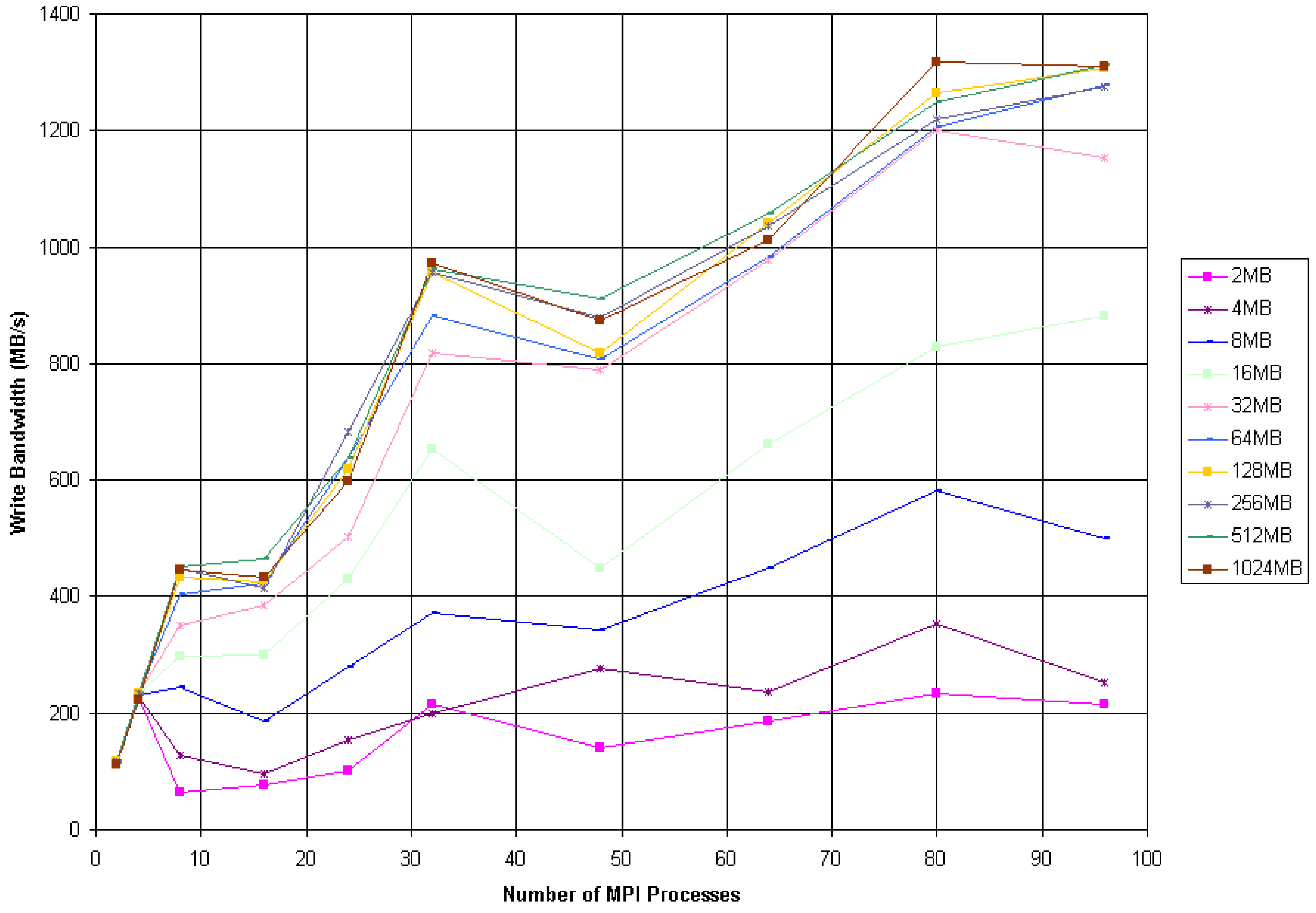
- Jobs which need very large scratch files: The local `/tmp` on each compute node has roughly 70 GB of available space. By comparison, `/pvfs` has just short of 8 TB (i.e. over two orders of magnitude more) available, is globally accessible, and is about as fast as a locally attached single disk in most cases.

- Jobs which use MPI-IO: Parallel programs which use the MPI-2 I/O routines (`MPI_File_*()`) to PVFS will see significant performance improvements over doing I/O to `/tmp` or `/home`. I/O rates of over 1.5 GB/s have been observed for jobs with large node counts, and rates of 100-400 MB/s are commonplace.

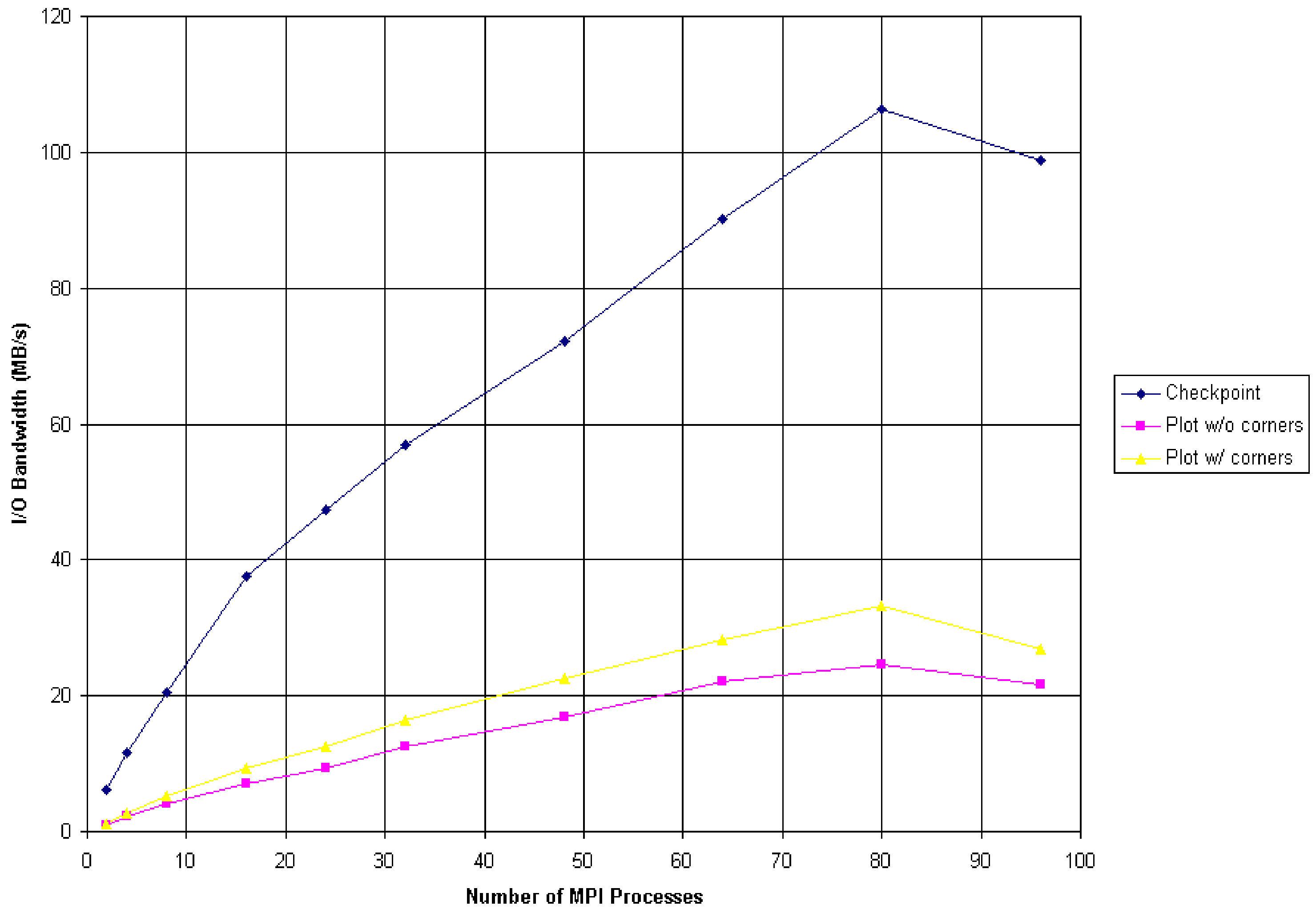
MPI Parallel Read Performance to PVFS



MPI Parallel Write Performance to PVFS



ASCI Flash Parallel I/O Benchmark



PVFS Caveats

- `/pvfs` is <u>**NOT**</u> backed up!!! It is intended strictly for temporary use; do not keep files on it over the long term without also storing them in your home directory.
- Do not store executables on `/pvfs`; while this will often work, it occasionally will cause odd problems.

Other Sources of Information

- Online manuals
- Software documentation on the web
- Related workshop courses

Online Manuals

- Like most UNIX-like systems, Linux includes a set of reference manuals as part of the operating system. This can be accessed by typing `man cmdname`, where `cmdname` is the name of the command or library routine for which you need information.
- You can also do a keyword search of all of the currently accessible manual pages by running `man -k keyword`.

Software Documentation on the Web

Many current software packages have documentation in web-accessible HTML format in addition to (or as a replacement for) standard UNIX man pages. Some examples of this include:

- The Myricom GM low-level communications library for Myrinet (http://www.myri.com/GM/doc/gm_toc.html)
- The PBS queuing system (http://pbs.mrj.com/docs.html)
- The Portland Group compiler suite (http://www.pgroup.com/ppro_docs/)

Other software documentation can be found at http://oscinfo.osc.edu/software/.

Related Workshop Courses

OSC offers several other courses which many be of interest to users of the OSC cluster:

- C Programming
- Features of the C++ Programming Language
- An Introduction to Fortran 90
- Parallel Programming with MPI
- Parallel Programming with OpenMP
- Parallel Code Optimization
- Parallel Numerical Libraries (ScaLAPACK)

More information on these courses and more can be found at http://oscinfo.osc.edu/training/.