**Big Red 2**

**A few words about Structure**

1. Actually, may be thought of as three connected machines.

2. Login nodes are for typical interactive work like compilations

3. Aprun nodes manage the jobs running on the compute nodes.

4. The compute nodes are the bulk of the machine with high speed network fabric to run parallel jobs.

5. Two modes for jobs. Extreme Scalability Mode (ESM) and Cluster Compatibility Mode (CCM).

6. ESM is launched with aprun. It has limited outside connections.

7. CCM is basically recreating a Quarry-like environment on Big Red 2’s nodes. Nothing is prelinked (like MPI) but all LINUX functionality exists. May login with ccmlogin or just submit a job with ccmrun.

**Compiling Jobs.**

**Four types of Compilers:  CCE (Cray), PGI, Intel and GCC**

**Compiler wrapper:**

1. All Big Red 2 compilers use the same wrapper. Which compiler that will be used is controlled by the loaded Prgenv module.

2. Commands are: CC for C++, cc for C, and ftn for Fortran.

3. Limited wrapper switches. The default is –static (for speed) but may be overridden with –shared or –dynamic.

4. The default page size is 4K. may use –lhugetlbfs to increase this to 2MB. Other modules may be used to increase this value to 16M but not through the compiler wrapper.

5. Must have the PrgEnv module of choice preloaded before it is possible to see that compiler’s specific man pages.

6. Wrapper will link in MPI automatically.
Cray CCE Compilers:

1. Generally compile faster executables than gcc for the same level of optimization.
2. A little stricter to the Standards than gcc.
3. Extensive man pages, but not nearly as burdensome as GCC (average about 2,000 lines vs. more than 15,000 lines).

C & C++:

Machine switches:
These switches are always the same for the platform.

1. –h cpu=abhudabi assumes one thread per cpu while –h cpu=abhudabi-cu assumes one thread per floating point unit (There are two cpus per one fpu).

User (compatibility) switches:

1. The default is –h c99  (C only)
2. –h gnu recognizes GCC version 4.4.4 language extensions (off by default).
3. –h tolerant is handy with old C codes Many modern errors (like assigning ints as pointers) become warnings. (C only)

Optimization switches:

. Often, more is not better. Testing and experimenting is about the only way to properly tune an executable. Cray sets optimization defaults fairly aggressively already.

Fortran:

Machine switches:
These switches are always the same for the platform.

1. –h cpu=abhudabi assumes one thread per cpu while –h cpu=abhudabi-cu assumes one thread per floating point unit (There are two cpus per one fpu).

User switches:
1. By default, double precision arithmetic is disabled. Need –ep in conjunction with –s real64 or –s default64. This has the effect of promoting declared DOUBLE PRECISION types to 128 bits and DOUBLE COMPLEX to 256 bits.

2. –ev will allocate variables to static storage (emulates SAVE declaration). Does not apply to automatic or explicitly allocated variables.

3. –h acc enables OpenACC directives. Should now be enabled by default.

**Optimization switches:**

1. -O modinline  Inlines fortran modules.

2. –h aggress  Will evaluate entire program for optimization.

3. –O ipa[0 to 5] will inline from nothing (0) to actually cloning library routines (5).

3. –O2 is the default optimization level without overriding it.

Again, testing and experimenting is about the only way to properly tune an executable.

**Threads and System Math Libraries:**

Threads are automatically recognized at compile time. Math libraries will be linked in if the xt-libsci module is loaded at compile time.

**GCC:**

1. Default is version 4.9.1. The oldest version available is 4.4.4.

2. Works the same as gcc everywhere else.

3. 64-bit compilations are the default but may be explicitly forced with –m64.

4. -march=opteron or –march=opteron-sse3  Identifies our processor type and implies –mtune of the same type..

5. Without –mmodel=medium set, the maximum code size is 2GB
6. man page is extensive (more than 15,000 lines).

**GCC is on Quarry/Karst too:**

1. Quarry default is version 4.1.2. Version 4.7.2 is available as a module.

**Machine switch:**

The switch “-march=nocona” turns on 64-bt extensions as well as SSE vector instructions.

**Optimization Switches:**

1. The default is no optimizations at all.

2. Many options are known to slow the executable. Often mentioned in the man page.

Example:

- **-funroll-all-loops**
  
  Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. **-funroll-all-loops** implies the same options as **-funroll-loops**.

3. Macro, “-O2” is structured to not increase executable’s size.

4. Macro, “-O3” is usually a good idea. It turns on many options:

   - **-fforce-mem -foptimize-sibling-calls -fstrength-reduce -fcse-follow-jumps**

   - **-fcse-skip-blocks -frerun-cse-after-loop -frerun-loop-opt -fgcse -fgcse-lm -fgcse-sm -fgcse-las**


   - **-fpeephole2 -freorder-blocks -freorder-functions -fstrict-aliasing -funit-at-a-time**

   - **-falign-functions -falign-jumps -falign-loops -falign-labels -fcrossjumping -finline-functions**

   - **-fweb, -frename-registers and -funswitch-loops**
Native Intel compiler:

1. Started with GCC as a base set beginning with Intel 8.
2. Emphasis is with ease of use.
3. Generally about 1/3 faster code is generated with the same amount of effort.

C compiler is icc, C++ is icpc, Fortran is ifort

1. Addressing size is determined by either “-m64” or “-m32” switches. Actually, -m32 builds binaries for an IA-32 architecture. Quarry is X86_64 and will always build 64-bit programs for itself without any size switch.

2. A good optimization switch to try is “-fast.” This implies -O3, -ipo, -static, -no-prec-div, and -xhost. Of course, “-O3” is a macro itself. It implies loop transformation as well as many others. Sometimes this can be trouble. If so, try “-O2” and the rest of the above switches.

3. “-xSSSE3” is usually a good architecture choice (replaced –xT).

A note on C++ (icpc)/GCC interoperability from the man page:

C++ compilers are interoperable if they can link object files and libraries generated by one compiler with object files and libraries generated by the second compiler, and the resulting executable runs successfully. Some GNU gcc* versions are not interoperable, some versions are interoperable. By default, the Intel compiler will generate code that is interoperable with the version of gcc it finds on your system.

Portland Group compiler:

The third party compilers from the Portland Group are installed on Quarry. This compiler suite is useful when software packages (often built on other platforms) support it but not Intel. Some users like the idea of a compiler that is supported across platforms (like GCC) but still provides professional support and bug fixing.

Compilation precision considerations

Addressing size often has no direct impact on precision. Some compilers do default to higher precisions because pointers are the address size and some C codes declare integers as pointers, but this is arbitrary. Most common defaults are 4-byte, 32-bits. How big a deal is that? Let’s investigate. For many codes, like popular molecular dynamics modelers or finite difference
codes, 12 bits of mathematical precision are necessary. Often the wind speed in meteorological

codes was originally only measured with 4 digits of precision.

A computer is an imprecise machine. Just adding and subtracting will cause errors. Below is a
simple code. It creates and writes out 1,000,000 ASCII numbers to a file. It then adds them
forward \{1, 2, 3…1,000,000\} and adds them backwards \{1,000,000, 999,999 999,998…1\} and
prints the result.

```
program time
  real*4 temp, paste
  real*4 A(1000000)
  real*4 sumf, sumb
  integer j
  integer counter

  counter = 1000000
  temp = 0.0
  sumf = 0.0
  sumb = 0.0
  open(unit = 10, file = 'data.dat' ,status = 'unknown')
  paste = 1.0 / 77.
  call random_seed()
  do 30 j = 1, counter
    call random_number(temp)
    temp = temp * paste
    A(j) = temp
  30 continue
  do 50 j = 1, counter, 7
    A(j) = 785454654 * A(j)
  50 continue
  do 70 j = 1,counter , 3
    A(j) = A(j) * 0.0000000001
  70 continue
  do 90 j = 1, counter
    write(10, 22) A(j)
    sumf = sumf + A(j)
  22  format( f48.25)
  90 continue
  close(10)
  do 80 j = counter, 1, -1
    sumb = sumb + A(j)
  80 continue
  print *, "forward sum is = ", sumf
```
print *, "reverse sum is = ", sumb
end program

So, let's look at the result.

32-bit precision:

/ray/round_error> ./round_32.out

forward sum is = 0.4852211057E+12
reverse sum is = 0.4852207124E+12

First discrepancy is in the 6th digit.

64-bit precision:

/ray/round_error> ./round_64.out

forward sum is = 485220744826.172424
reverse sum is = 485220744826.173584

First discrepancy is in the 15th digit.

128-bit precision (IBM extension to the language):

/ray/round_error> ./round_128.out

forward sum is = 485220744826.18382927712407234600134
reverse sum is = 485220744826.18382927712407234630628

First discrepancy is in the 31st digit.

**Simple OpenMP (so simple, let's just do it...)**

OpenMP is a high level, threading directive set. It takes the place of the original pthreads by hiding the mechanics from the user. Primarily, its use is to divide the work of loops among multiple processors. These loops must not be dependent on other loops. Historically, only the outer loop of a set of nested loops may be parallelized this way, but work has been done to


include multiple levels of loops if it useful. The threads must all be able to see the same memory. This is often referred as a “shared everything” environment.

In our summing program, we had a lot of independent loops. So, why not lets parallelize them with OpenMP?
We only need to add three new lines and get 4 processors to our serial code!

```fortran
program time
real*4 temp, paste
real*4 A(1000000)
real*4 sumf, sumb
integer j
integer counter

call OMP_SET_NUM_THREADS(4)

counter = 1000000
temp = 0.0
sumf = 0.0
sumb = 0.0
open(unit = 10, file = 'data.dat' ,status = 'unknown')
paste = 1.0 / 77.
call random_seed()

!$OMP PARALLEL PRIVATE(j)
    do 30 j = 1, counter
        call random_number(temp)
        temp = temp * paste
        A(j) = temp
    30  continue
    
    do 50 j = 1, counter, 7
        A(j) = 785454654 * A(j)
    50  continue
    
    do 70 j = 1,counter , 3
        A(j) = A(j) * 0.0000000001
    70  continue

!$OMP END PARALLEL
    do 90 j = 1, counter
        write(10, 22) A(j)
        sumf = sumf + A(j)
    22  format( f48.25)
```
90 continue
close(10)
do 80 j = counter, 1, -1
  sumb = sumb + A(j)
80 continue
  print *, "forward sum is = ", sumf
  print *, "reverse sum is = ", sumb
end program

O.K. So, how did we do?

Serial timing:
time ./input__serial_test
  forward sum is = 0.4852211057E+12
  reverse sum is = 0.4852207124E+12

real  0m3.136s
user  0m1.713s
sys   0m0.115s

OpenMP timing:
time ./input__smp_test
  forward sum is = 0.3957986771E+29
  reverse sum is = 0.3957989605E+29

real  0m9.086s
user  0m5.963s
sys   0m1.959s

Oops!

Let's add "temp" as a private variable in the line:

!$OMP  DO PRIVATE(j, temp)

We get:
> time ./new_smp
  forward sum is =  0.2485100971E+30
  reverse sum is =  0.2485099460E+30

real    0m8.551s
user    0m5.796s
sys     0m2.007s
---------------------

Better but, still pretty poor.  Lets try identifying each loop to OpenMP.  Here is our parallel section now:

!$OMP PARALLEL
  !$OMP DO PRIVATE(j, temp)
       do 30 j = 1, counter
           call random_number(temp)
           temp = temp * paste
           A(j) = temp
  30 continue

  !$OMP END DO
  !$OMP DO PRIVATE(j)
       do 50 j = 1, counter, 7
           A(j) = 785454654 * A(j)
  50 continue

  !$OMP END DO
  !$OMP DO PRIVATE(j)
       do 70 j = 1, counter, 3
           A(j) = A(j) * 0.0000000001
  70 continue

  !$OMP END DO
  !$OMP END PARALLEL

Looks good, so how about now?

> time ./new_smp
  forward sum is =  0.4869933957E+12
  reverse sum is =  0.4869911347E+12
Now we are only twice as slow as serial. Any ideas why?

Threads tutorial overview:  [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)

---

**Submitting A Job**

Big Red 2, Mason and Quarry use PBS

Both LoadLeveler (used in the original Big Red) and PBS need to do the same job (and share the same scheduler package) so many keywords map to each other:

### Common commands

<table>
<thead>
<tr>
<th>TORQUE command</th>
<th>LL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job submission</td>
<td>qsub [scriptfile]</td>
</tr>
<tr>
<td>Job deletion</td>
<td>qdel [job_id]</td>
</tr>
<tr>
<td>Job status (for user)</td>
<td>qstat -u [username]</td>
</tr>
<tr>
<td>Extended job status</td>
<td>qstat -f [job_id]</td>
</tr>
<tr>
<td>Hold a job temporarily</td>
<td>qhold [job_id]</td>
</tr>
<tr>
<td>Resume job on hold</td>
<td>qrls [job_id]</td>
</tr>
<tr>
<td>List of usable queues</td>
<td>qstat -Q</td>
</tr>
<tr>
<td>GUI for batch system</td>
<td>xpbs</td>
</tr>
</tbody>
</table>

### Environment variables

<table>
<thead>
<tr>
<th>TORQUE command</th>
<th>LL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job ID</td>
<td>$PBS_JOBID</td>
</tr>
<tr>
<td>Submission directory</td>
<td>$PBS_O_WORKDIR</td>
</tr>
<tr>
<td>Processor list</td>
<td>$PBS_NODEFILE</td>
</tr>
</tbody>
</table>
# Resource specifications

<table>
<thead>
<tr>
<th></th>
<th><strong>TORQUE command</strong></th>
<th><strong>LL command</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td><code>#PBS -q [queue]</code></td>
<td><code>@ class=[queue]</code></td>
</tr>
<tr>
<td>Nodes</td>
<td><code>#PBS -l nodes=[#]</code></td>
<td><code>@ node=[#]</code></td>
</tr>
<tr>
<td>Processors</td>
<td><code>#PBS -l ppn=[#]</code></td>
<td><code>@ tasks_per_node=[#]</code></td>
</tr>
<tr>
<td>Wall clock limit</td>
<td><code>#PBS -l walltime=[hh:mm:ss]</code></td>
<td><code>@ wall_clock_limit=[hh:mm:ss]</code></td>
</tr>
<tr>
<td>Standard output file</td>
<td><code>#PBS -o [file]</code></td>
<td><code>@ output=[file]</code></td>
</tr>
<tr>
<td>Standard error</td>
<td><code>#PBS -e [file]</code></td>
<td><code>@ error=[file]</code></td>
</tr>
<tr>
<td>Copy environment</td>
<td><code>#PBS -V</code></td>
<td><code>@ environment=COPY_ALL</code></td>
</tr>
<tr>
<td>Notification event</td>
<td><code>#PBS -m abe</code></td>
<td>`@ notification=start</td>
</tr>
<tr>
<td>Email address</td>
<td><code>#PBS -M [email]</code></td>
<td><code>@ notify_user=[email]</code></td>
</tr>
<tr>
<td>Job name</td>
<td><code>#PBS -N [name]</code></td>
<td><code>@ job_name=[name]</code></td>
</tr>
<tr>
<td>Job restart</td>
<td>`#PBS -r [y</td>
<td>n]`</td>
</tr>
<tr>
<td>Job type</td>
<td>N/A</td>
<td><code>@ job_type=[type]</code></td>
</tr>
<tr>
<td>Initial directory</td>
<td>N/A</td>
<td><code>@ initialdir=[directory]</code></td>
</tr>
<tr>
<td>Node usage</td>
<td>N/A?</td>
<td><code>@ node_usage=not_shared</code></td>
</tr>
<tr>
<td>Memory requirement</td>
<td><code>#PBS -l vmem=xxxx</code></td>
<td><code>@ requirements=(Memory &gt;= NumMegaBytes)</code></td>
</tr>
</tbody>
</table>

# Common Moab scheduler commands
Big Red and Quarry use the Moab job scheduler. Frequently used Moab scheduler commands include:

<table>
<thead>
<tr>
<th>Command</th>
<th>LL and TORQUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show currently running/queued jobs</td>
<td>showq</td>
</tr>
<tr>
<td>Check a job's status</td>
<td>checkjob [jobid] or checkjob -v [jobid] (for details)</td>
</tr>
<tr>
<td>Show when your job might start</td>
<td>showstart [job_id]</td>
</tr>
<tr>
<td>Show fairshare information (priority, etc.)</td>
<td>diagnose -f</td>
</tr>
<tr>
<td>Check a nodes status</td>
<td>checknode [nodename]</td>
</tr>
<tr>
<td>Show current reservations</td>
<td>showres</td>
</tr>
</tbody>
</table>

Sample PBS Job on Quarry

```bash
#!/bin/bash

#PBS -l nodes=4:ppn=2,vmem=1024mb,walltime=3:00:00
#PBS -m ae
#PBS -N raytest
#PBS -o test_pbs2.out
#PBS -e test_pbs2.err

mpirun -np 8 -machinefile $PBS_NODEFILE hellomi

echo `cat $PBS_NODEFILE`

echo "this is just a test"

NOTES: Script is executed. Resource arguments are all on one line.

Check The Job:

There are lots of ways to check on your job. The simplest is:

llq | grep {your_username}
If your job isn't running, use the job number with "showstart" to find out when it is scheduled. An example:  "showstart s10c2b5.10488.0"

A more in-depth command would be "checkjob"  Here is an example:

    checkjob s10c2b5.10692.0
    job s10c2b5.10692.0

    AName: 0
    State: Running
    Creds: user:rsheppar group:hpc account:NONE class:MED
    WallTime:  00:00:00 of 00:20:00
    SubmitTime: Wed May 30 19:00:03
                (Time Queued  Total: 00:00:41  Eligible: 00:00:41)

    StartTime: Wed May 30 19:00:44
    Total Requested Tasks: 8

    Req[0]  TaskCount: 8  Partition: base
    Memory >= 0  Disk >= 0  Swap >= 0
    Opsys: Linux2  Arch: PPC64  Features: ---

    Allocated Nodes:
    [s6c2b10.dim:2][s6c2b11.dim:2][s6c2b12.dim:2][s6c2b13.dim:2]

    IWD:          /N/gpfsbr/namd_example
    Executable:   /N/gpfsbr/namd_example/mpich_namd.bash
    StartCount:
    Flags:        BACKFILL,RESTARTABLE
    Attr:         BACKFILL
    StartPriority: 3804
    Reservation 's10c2b5.10692.0' (-00:00:09 -> 00:19:51  Duration: 00:20:00)

**Time for Lab?**

**Message Passing Libraries**

MPI is an actual instruction set that vendors have agreed to standardize. In MPI, the processors “share nothing.” Each must be given its task and report its result through the use of messages
passed between them. This allows a much greater number to work on a solution than is possible with threads (which must all be able to see the same memory).

**Big Red 2 is a parallel machine! It is not structured for serial codes!**

Serial codes waste 97% of the processor cores as well as the interconnect switch (which is about 1/3 the cost of the machine by itself). Serial codes should normally run on Quarry.

1. Message Passing Packages are selected with modules on Quarry and are automatic on Big Red 2.

2. No MPI libraries exist in the default environment on Quarry.

4. Once an MPI library is added, compiles are made through a wrapper to the compiler that built the library. For instance, on Quarry, mpif90 is actually just a wrapper to either ifort or gfortran (depending on the MPI module). The same switches used by the underlying compiler are available to the mpif90 wrapper.

A super link to go deeper into this subject: [http://kb.iu.edu/data/autn.html](http://kb.iu.edu/data/autn.html)

**MPICH:**

Argonne Original. MPICH 1 is the original. MPICH 3 is the current version. Uses mpirun.

1. It has a limited runtime environment. Best Practice: Pass paths as environment variables.

**OpenMPI:**

Replaced LAM. No more lamboot. Has look and feel of MPICH to the user. Indiana University product. Improving with each new release.

**MPI Tutorial overview:** [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)

**Time for Lab?**

Notes by: Ray Sheppard, HPCST, RT, UITS, Indiana University