Best Practices for Best Performance on Edison

Zhenjgi Zhao
NERSC User Services Group

NUG 2014
Feb 6, 2014
Agenda

• System Overview
• Compile time optimization
• Run time tuning options
• Node placements on Edison
• A couple of tips for Lustre I/O
• Python applications at scale
• CCM usage

• Will not cover libraries, profiling tools.
• Interconnect will be covered by next talk
System Overview
Edison at Glance

- First Cray XC30
- Intel Ivy Bridge 12-core, 2.4GHz processors
- 64GB memory node
- Aries interconnect with Dragonfly topology for great scalability
- Software environment similar to Hopper
- Performs 2.2x sustained performance relative to Hopper

- 3 Lustre scratch file systems configured as 2:2:3 for capacity and bandwidth
- Access to NERSC’s GPFS global file system via DVS
- 12 x 512GB login nodes to support visualization and analytics
- Ambient cooled for extreme energy efficiency
## Vital Statistics

<table>
<thead>
<tr>
<th></th>
<th>Hopper</th>
<th>Edison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cabinets</td>
<td>68</td>
<td>28</td>
</tr>
<tr>
<td>Compute Nodes</td>
<td>6,384</td>
<td>5,192</td>
</tr>
<tr>
<td>CPU Cores (<em>Total / Per-node</em>)</td>
<td>152,408 / 24</td>
<td>124,608 / 24</td>
</tr>
<tr>
<td>CPU Frequency (GHz)</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td>Peak Flops (PF)</td>
<td>1.29</td>
<td>2.4</td>
</tr>
<tr>
<td>Memory (TB) (<em>Total / Per-node</em>)</td>
<td>217 / 32</td>
<td>333 / 64</td>
</tr>
<tr>
<td>Memory (Stream) BW* (TB/s)</td>
<td>331</td>
<td>462.8</td>
</tr>
<tr>
<td>Memory BW/node* (GB/s)</td>
<td>52</td>
<td>89</td>
</tr>
<tr>
<td>File system(s)</td>
<td>2 PB @ 70 GB/s</td>
<td>7.56 PB @ 180 GB/s</td>
</tr>
<tr>
<td>Peak Bisection BW (TB/s)</td>
<td>5.1</td>
<td>11</td>
</tr>
<tr>
<td>Power (MW Linpack)</td>
<td>2.9</td>
<td>1.9</td>
</tr>
</tbody>
</table>
## Baseline performance

### NERSC-6 Application Benchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>CAM</th>
<th>GAMESS</th>
<th>GTC</th>
<th>IMPACT-T</th>
<th>MAESTRO</th>
<th>MILC</th>
<th>PARATEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>240</td>
<td>1024</td>
<td>2048</td>
<td>1024</td>
<td>2048</td>
<td>8192</td>
<td>1024</td>
</tr>
<tr>
<td>Streams/Core</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Edison Time (s)</td>
<td>273.08</td>
<td>1,125.80</td>
<td>863.88</td>
<td>579.78</td>
<td>935.45</td>
<td>446.36</td>
<td>173.51</td>
</tr>
<tr>
<td>Hopper Time(s)</td>
<td>348</td>
<td>1389</td>
<td>1338</td>
<td>618</td>
<td>1901</td>
<td>921</td>
<td>353</td>
</tr>
<tr>
<td>Speedup¹)</td>
<td><strong>1.3</strong></td>
<td><strong>1.2</strong></td>
<td><strong>1.5</strong></td>
<td><strong>1.1</strong></td>
<td><strong>2.0</strong></td>
<td><strong>2.1</strong></td>
<td><strong>2.0</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>SSP²)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Edison</td>
<td><strong>258</strong></td>
</tr>
<tr>
<td>Hopper</td>
<td><strong>144</strong></td>
</tr>
</tbody>
</table>

¹) Speedup = Time(Hopper)/Time(Edison)

²) SSP stands for sustained system performance
Compile time options
# Compilers and NERSC recommended compiler optimization flags

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Recommended Optimization flags</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel (default)</td>
<td>-fast –no-ipo</td>
<td>Comparable to the -O2 optimization level; Compiler wrappers add -xAVX</td>
</tr>
<tr>
<td>Cray</td>
<td>Default</td>
<td>High optimization; Compiler wrappers add: -hcpu=ivybridge</td>
</tr>
<tr>
<td>GNU</td>
<td>-Ofast</td>
<td>No optimization; Compiler wrappers add: -march=core-avx-i</td>
</tr>
</tbody>
</table>

Use the verbose option of the compiler wrappers to see the exact compile/link options
- `ftn -v hello.f90`
- `cc -v hello.c`
- `CC -v hello.C`

Module show craype-ivybridge
Compilers and NERSC recommended compiler optimization flags

Relative Performance of Compilers on Edison

Compiler flags used:
- Intel: `-fast -no-ipo`
- Cray: `default`
- GNU: `-Ofast`

Courtesy of Mike Stewart at NERSC
Users are responsible to validate the codes to generate correct results

- Intel compiler fails to catch the type mismatch in the example below, and generates wrong results.
- Cray and GNU compilers do a better job by aborting the compilation

```fortran
zz217@edison02:~> cat test1.f90
program test1
real y
y=1.0
print *, "y, a(y) = ", y, a(y)
end

real*8 function a(z)
real*8 z
a=z
end

zz217@edison02:~> ftn test1.f90
zz217@edison02:~> ./a.out
y, a(y) = 1.000000 0.0000000E+00
```

• Strictly follow language standards in your coding is highly recommended as compilers follow the language standard more strictly now.
Compiler flags that are helpful to generate useful warnings

```
zz217@edison02:~> ftn -warn all test1.f90
test1.f90(6): warning #6717: This name has not been given an explicit type.  [A]
y=a(x)
   ^
test1.f90(6): warning #6717: This name has not been given an explicit type.  [A]
y=a(x)
^
test1.f90(6): error #7977: The type of the function reference does not match the type of the function definition.  [A]
y=a(x)
   ^
test1.f90(6): error #6633: The type of the actual argument differs from the type of the dummy argument.  [X]
y=a(x)
   ^
   ^
compilation aborted for test1.f90 (code 1)
```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>-warn all</td>
</tr>
<tr>
<td>Cray</td>
<td>-m msg_lvl</td>
</tr>
<tr>
<td>GNU</td>
<td>-Wall</td>
</tr>
</tbody>
</table>
Rum time options
Default process/thread affinity and default OMP_NUM_THREADS

Machine default: one-on-one process/thread to core binding

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Process/thread affinity</th>
<th>Default OMP_NUM_THREADS</th>
</tr>
</thead>
</table>
| Intel    | • Pure MPI codes, the process affinity works fine if running on fully packed nodes  
• There are issues with thread affinity - all threads from an MPI task are pined to a single core where the MPI task is placed.  
• An extra thread created by the Intel OpenMP runtime interacts with the CLE thread binding mechanism and causes poor performance | The number of cpu slots available |
| Cray     | Works fine              | OMP_NUM_THREADS=1        |
| GNU      | Works fine              | The number of cpu slots available |
Process/thread binding with a binary compiled with an Intel compiler

Export OMP_NUM_THREADS=6
aprun --n4 -N4 -S2 -d6 xthi.intel

Hello from rank 0, thread 0, on nid02877. (core affinity = 0)
Hello from rank 0, thread 1, on nid02877. (core affinity = 0)
Hello from rank 0, thread 2, on nid02877. (core affinity = 0)
Hello from rank 0, thread 3, on nid02877. (core affinity = 0)
Hello from rank 0, thread 4, on nid02877. (core affinity = 0)
Hello from rank 0, thread 5, on nid02877. (core affinity = 0)

Hello from rank 1, thread 0, on nid02877. (core affinity = 6)
Hello from rank 1, thread 1, on nid02877. (core affinity = 6)
Hello from rank 1, thread 2, on nid02877. (core affinity = 6)
Hello from rank 1, thread 3, on nid02877. (core affinity = 6)
Hello from rank 1, thread 4, on nid02877. (core affinity = 6)
Hello from rank 1, thread 5, on nid02877. (core affinity = 6)

Hello from rank 2, thread 0, on nid02877. (core affinity = 12)
Hello from rank 2, thread 1, on nid02877. (core affinity = 12)
Hello from rank 2, thread 2, on nid02877. (core affinity = 12)
Hello from rank 2, thread 3, on nid02877. (core affinity = 12)
Hello from rank 2, thread 4, on nid02877. (core affinity = 12)
Hello from rank 2, thread 5, on nid02877. (core affinity = 12)

Hello from rank 3, thread 0, on nid02877. (core affinity = 18)
Hello from rank 3, thread 1, on nid02877. (core affinity = 18)
Hello from rank 3, thread 2, on nid02877. (core affinity = 18)
Hello from rank 3, thread 3, on nid02877. (core affinity = 18)
Hello from rank 3, thread 4, on nid02877. (core affinity = 18)
Hello from rank 3, thread 5, on nid02877. (core affinity = 18)

QE performance slowdown from a bad process/thread affinity

<table>
<thead>
<tr>
<th>Aprun command line/OMP_NUM_THREADS</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aprun -n 48</td>
<td>154.465.5</td>
</tr>
<tr>
<td>aprun -n 24 -N 2 -S 6 -d 2 numa_node</td>
<td>167.5</td>
</tr>
<tr>
<td>aprun -n 24 -N 12 -S 6 -d 2 numa_node</td>
<td>2775.1</td>
</tr>
<tr>
<td></td>
<td>83.75</td>
</tr>
</tbody>
</table>
Aprun’s –S option need to be used to evenly distribute MPI tasks to the two NUMA nodes

aprun -n 12 -N12 xthi.intel
Hello from rank 0, thread 0, on nid06119. (core affinity = 0)
Hello from rank 1, thread 0, on nid06119. (core affinity = 1)
Hello from rank 2, thread 0, on nid06119. (core affinity = 2)
Hello from rank 3, thread 0, on nid06119. (core affinity = 3)
Hello from rank 4, thread 0, on nid06119. (core affinity = 4)
Hello from rank 5, thread 0, on nid06119. (core affinity = 5)
Hello from rank 6, thread 0, on nid06119. (core affinity = 6)
Hello from rank 7, thread 0, on nid06119. (core affinity = 7)
Hello from rank 8, thread 0, on nid06119. (core affinity = 8)
Hello from rank 9, thread 0, on nid06119. (core affinity = 9)
Hello from rank 10, thread 0, on nid06119. (core affinity = 10)
Hello from rank 11, thread 0, on nid06119. (core affinity = 11)

aprun -n 12 -N12 -S6 xthi.intel
Hello from rank 0, thread 0, on nid06119. (core affinity = 0)
Hello from rank 1, thread 0, on nid06119. (core affinity = 1)
Hello from rank 2, thread 0, on nid06119. (core affinity = 2)
Hello from rank 3, thread 0, on nid06119. (core affinity = 3)
Hello from rank 4, thread 0, on nid06119. (core affinity = 4)
Hello from rank 5, thread 0, on nid06119. (core affinity = 5)
Hello from rank 6, thread 0, on nid06119. (core affinity = 6)
Hello from rank 7, thread 0, on nid06119. (core affinity = 7)
Hello from rank 8, thread 0, on nid06119. (core affinity = 8)
Hello from rank 9, thread 0, on nid06119. (core affinity = 9)
Hello from rank 10, thread 0, on nid06119. (core affinity = 10)
Hello from rank 11, thread 0, on nid06119. (core affinity = 11)
Manipulate process/thread affinity

- S, -sn, -sl, -cc, and -ss options control how your application uses the NUMA nodes.
  - -n Number of MPI tasks.
  - -N (Optional) Number of MPI tasks per Edison Node. Default is 24.
  - -S (Optional) Number of tasks per NUMA node. Values can be 1-12; default 12
  - -sn (Optional) Number of NUMA nodes to use per Edison node. Values can be 1-2; default 2
  - -ss (Optional) Demands strict memory containment per NUMA node. The default is the opposite - to allow remote NUMA node memory access.
  - -cc (Optional) Controls how tasks are bound to cores and NUMA nodes. The default setting on Edison is -cc cpu which restricts each task to run on a specific core.

- These options are important on Edison if you use OpenMP or if you don't fully populate the Edison nodes.

http://portal.nersc.gov/project/training/EdisonPerformance2013/affinity
Recommended aprun options to assure appropriate process/thread affinity

- Running on unpacked nodes

  #PBS –l mppwidth=48  #2 nodes
  aprun –n 24 –N 12 –S 6 ./a.out

- Running with OpenMP threads

  #for threads per task <= 12
  setenv OMP_NUM_THREADS 12
  #for binaries compiled with Intel compilers
  aprun –n 4 –N 2 -S 1 –d 12 –cc numa_node ./a.out
  # for binaries compiled with GNU or Cray compilers.
  aprun -n 4 -N 2 -S1 -d 12 ./a.out

  #for threads per task>12 and  <= 24
  export OMP_NUM_THREADS=24
  #for binaries compiled with Intel compilers
  aprun –n 2 –N 1 –d 24 –cc none ./a.out
  # for binaries compiled with GNU or Cray compilers.
  aprun –n 2 –N 1 -d 24 ./a.out
Hyper-Threading (HT) on Edison

- Cray compute nodes booted with Hyper-Threads always ON
- Users can choose to run with one or two tasks/threads per core
- Use aprun –j2 option to use Hyper-threading
  - aprun –j1 –n ... Single Stream mode, one rank/thread per core
  - aprun –j2 –n ... Dual Stream mode, two ranks/threads per core
  - Default is Single Stream mode
- Dual Stream is often better if
  - throughput is more important
  - your code scales extremely well
  - When running at relatively low core counts
- Single Stream is often better if ...
  - single job performance matters more
  - code does not scale well
- NERSC-6 SSP applications 4 out of 7 ran with HT
- However, HT may hurt code performance, use with caution.

https://www.nersc.gov/users/computational-systems/edison/performance-and-optimization/hyper-threading/
Core specialization

- System ‘noise’ on compute nodes may significantly degrade scalability for some applications
- Core Specialization can mitigate this problem
  - M core(s)/cpu(s) per node will be dedicated for system work (service core)
  - As many system interrupts as possible will be forced to execute on the service core
  - The application will not run on the service cpus
- Use aprun -r to get core specialization
  - aprun –r[1-8] –n 100 a.out
  - Highest numbered cpus will be used
  - Starts with cpu 48 on Ivy Bridge e nodes
  - Independent of aprun –j setting
- Apcount provided to compute total number of cores required
- Tests with NERSC-6 benchmark codes shows that the impact of core specialization is at best negligible and often negative.
  
Hugepages may improve your code performance

- Hugepages may improve memory performance for common access patterns on large data sets.
- The Aries may perform better with HUGE pages than with 4K pages.
  - HUGE pages use less Aries resources than 4k pages
  - More important when remotely access large percentage of nodes memory in an irregular manner
- May get “cannot run errors” if there are not enough Hugepages memory available (memory page fragmentation)
- Use modules to change default page sizes (man intro_hugepages)
- Users are recommended to experiment with hugepages
- This feature is implemented at link and run time, to use
  - Module load craype-hugepages2M
  - cc -o my_app my_app.c
  - Then run with the same hugepages module loaded
Hugepages may improve your code performance

Maestro run time comparison with/without using hugepages

Mastro run time improves by 11% by average when using hugepage memory compared to not using the hugepages.
Node placements on Edison
NERSC-6 application benchmark production and dedicated time comparison

<table>
<thead>
<tr>
<th>Application</th>
<th>CAM</th>
<th>GAMESS</th>
<th>GTC</th>
<th>IMPACT-T</th>
<th>MAESTRO</th>
<th>MILC</th>
<th>PARATEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>240</td>
<td>1024</td>
<td>2048</td>
<td>1024</td>
<td>2048</td>
<td>8192</td>
<td>1024</td>
</tr>
<tr>
<td>Streams/Core</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dedicated Time (s)</td>
<td>273.08</td>
<td>1,125.80</td>
<td>863.88</td>
<td>579.78</td>
<td>935.45</td>
<td>446.36</td>
<td>173.51</td>
</tr>
<tr>
<td>Production Time (s)</td>
<td>277.07</td>
<td>1,218.17</td>
<td>871.06</td>
<td>597.25</td>
<td>996.70</td>
<td>482.87</td>
<td>198.45</td>
</tr>
<tr>
<td>Slowdown 1)</td>
<td>1.5%</td>
<td>8.2%</td>
<td>0.8%</td>
<td>3.0%</td>
<td>6.5%</td>
<td>8.2%</td>
<td>14.4%</td>
</tr>
</tbody>
</table>

1) Slowdown = Time(Production) / Time(Dedicated)
Edison cabinet groups:

Group 0: C0-0 C1-0
Group 1: C2-0 C3-0
Group 2: C4-0 C5-0
Group 3: C6-0 C7-0
Group 4: C0-1 C1-1
Group 5: C2-1 C3-1
Group 6: C4-1 C5-1
Group 7: C6-1 C7-1
Group 8: C0-2 C1-2
Group 9: C2-2 C3-2
Group 10: C4-2 C5-2
Group 11: C6-2 C7-2

Group 14: C4-3 C5-3
Group 15: C6-3 C7-3

Note:

- The groups 12, and 13 are missing in our layout
- Use cnselect x_coord.eq.3 to choose the node list in the cabinet group 3
Node placements and run time

MAESTRO Run time

- Dedicated run with same 86 nodes in one cabinet group
- Dedicated, one job in each cabinet group, 14 jobs simultaneously
- Production runn but one job in each cabine, 7 jobs simultaneously

Time (s)

Runs

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Node placements and run time

Maestro Run Time

- One job in one cabinet --p-state 2.4GHz
- One job in each cabinet group with --p-state=2.4GHz
- One job in one cabinet
- One job in each cabinet group

Time (s)

Runs

- 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83

- 940 - 1120
I/O performance
Edison has three Lustre file systems

<table>
<thead>
<tr>
<th></th>
<th>Size (PB)</th>
<th>Aggregate Peak Performance (GB/s)</th>
<th># of Disks</th>
<th># of OSSs</th>
<th># of OSTs</th>
<th>Default stripe count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SCRATCH/scratch1</td>
<td>2.1</td>
<td>48</td>
<td>12</td>
<td>24</td>
<td>96</td>
<td>2</td>
</tr>
<tr>
<td>$SCRATCH/scratch2</td>
<td>2.1</td>
<td>48</td>
<td>12</td>
<td>24</td>
<td>96</td>
<td>2</td>
</tr>
<tr>
<td>/scratch3</td>
<td>3.2</td>
<td>72</td>
<td>18</td>
<td>24</td>
<td>144</td>
<td>8</td>
</tr>
</tbody>
</table>

https://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/edison-scratch3-directory-request-form/

Users are encouraged to experiment with Lustre stripe count, size to obtain a good I/O performance for their workloads, with a general guidance that a larger stripe count may increase bandwidth but subject more contention, and vise versa.

lfs setstripe
lfs getstripe
man lfs
Many factors may affect the I/O performance of your jobs

• Contentions for the resources with other users
• Hardware failure or downgraded performance
• File system fragmentations
• Bad user practices
  – A user used fixed offset, and stripecount 1 and filled up one of the OSTs a couple of times.
  – Using too large stripe counts for small file I/O inviting contention with other users unnecessarily and get widely varying I/O time
Python applications at scale
DLFM method effectively reduces python application startup time

WARP startup time using DLFM on Edison

Warp startup time is ~1 minutes at 38.4K cores!
Using DLFM module for large scale python applications

• DLFM, developed by Mike Davis at Cray, Inc, is a library tool to reduce the python application startup time at large scale.
• To access, do module load dlfm
• Compile your code using the python available via the dlfm module
• Run with two steps
  – Pilot run with small node count, eg., using 2 nodes collect the needed shared libraries and python modules imported
  – Real run with large number of cores, only one core read in the shared libraries and python imported modules
• More info is in the DLFM website
Cluster Compatibility Mode
Cluster compatibility mode (CCM)

- CCM is available on Edison to run TCP/IP applications or ISV (Independent Software Vendor) applications.
- G09 and Wien2k run via CCM because they need ssh to compute nodes.
- Running g09 over multiple nodes are not recommended due to a performance issue with CCM and also g09’s relatively low parallel scalability.

- [https://www.nersc.gov/users/computational-systems/edison/cluster-compatibility-mode/](https://www.nersc.gov/users/computational-systems/edison/cluster-compatibility-mode/)
Thank you.