Using Cori KNL Nodes

Helen He

Cori KNL Hackathon, March 7, 2018
Outline

• How to Compile for KNL?
  – How to compile and link
  – How to use MCDRAM

• How to Run on KNL?
  – Process/thread/memory affinity
  – Script examples
  – Recommendations

• How to use Performance Tools
  – Vtune (and APS)
  – Advisor
  – Inspector
How to Compile for Cori KNL
Cori Login and Compute Nodes

- Haswell compute nodes and KNL compute nodes
- All login nodes are Haswell nodes
- Binaries built for Haswell can run on KNL nodes, but not vice versa
- **We need to cross-compile**
  - Directly compile on KNL compute nodes is very slow
  - Meaning to compile on Haswell login nodes to generate binaries for KNL compute nodes
How to Compile and Link (1)

- The default loaded architecture target module is “craype-haswell” on the Haswell login nodes.

- This module sets CRAY_CPU_TARGET to haswell

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>GNU</th>
<th>Cray</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cori Haswell</strong></td>
<td>-xCORE-AVX2 or -xHASWELL</td>
<td>-march=core-avx2</td>
<td>-h cpu=haswell</td>
<td>craype-haswell</td>
</tr>
<tr>
<td><strong>Cori KNL</strong></td>
<td>-xMIC-AVX512 or -xKNL</td>
<td>-march=knl</td>
<td>-h cpu=mic-knl</td>
<td>craype-mic-knl</td>
</tr>
</tbody>
</table>

- **Best recommendation to build for KNL target**
  
  % module swap craype-haswell craype-mic-knl
  
  which will set CRAY_CPU_TARGET to mic-knl
How to Compile and Link (2)

• Then the compiler wrappers will take care of adding specific KNL target (such as “-h cpu=mic-knl” for CCE, or “-X MIC-AVX512” or “-xKNL” for Intel compilers), and it will also link the optimized KNL libraries, if exist, from the modules loaded (such as cray-libsci, cray-petsc,crsay-tpsl, etc.)

• Some libraries that are not performance-critical on KNL will use the Haswell version (such as cray-netcdf, cray-hdf5, etc.), will also be linked in transparently with the compiler wrappers from the modules loaded.
How to Compile and Link (3)

• Alternate: build a fat binary by adding MIC-AVX512 in target option

  % cc -axMIC-AVX512,CORE-AVX2 <options> mycode.c

Or % cc -axKNL,HASWELL <options> mycode.c

• Only valid when using Intel compilers (cc, CC or ftn)
• -ax<arch> adds an “alternate execution path” optimized for different architectures
  – Makes 2 (or more) versions of code in same object file
  – Runtime will choose which version to use

• NOT AS GOOD as the craype-mic-knl module
  – craype-haswell module will still cause the Haswell versions of libraries being used, *e.g.* MKL
autoconf and cmake Solutions

• In some build systems, certain steps need to run a small test program in order to proceed, *e.g.*, to generate a Makefile.
  – It will fail with cross-compilation

• **Workaround for autoconf**
  % module load craype-haswell
  % ./configure CC=cc FTN=ftn CXX=CC ...
  % module swap craype-haswell craype-mic-knl
  % make

• **Solution for cmake**
  % export CRAYOS_VERSION=6
  % cmake -DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment ...
Using MCDRAM

- Cori KNL has 16 GB on-chip memory per node
  - and 96 GB off-chip DDR (Cori)
- Not (exactly) a cache
  - Latency similar to DDR
- But very high bandwidth
  - ~5x DDR
- MCDRAM modes
  - “Cache” mode: invisible to OS, memory pages are cached in MCDRAM (cache-line granularity)
  - “Flat” mode: appears to OS as separate NUMA node, with no local CPUs.
  - “Hybrid” mode: some memory as cache, some not
  - If not Cache: Accessible via numactl, libnuma, srun --mem_bind, autohbw, libmemkind
Which Arrays to Put in MCDRAM?

- VTune memory-access measurements:
  \% amplxe-cl -collect memory-access ...

---

**System Bandwidth**

This section provides various system bandwidth-related properties detected by the product. These values are used to define default High, Medium and Low bandwidth utilization thresholds for the Bandwidth Utilization Histogram and to scale overtime bandwidth graphs in the Bottom-up view.

- Max DRAM System Bandwidth: 128 GB
- Max DRAM Single-Package Bandwidth: 64 GB

**Bandwidth Utilization Histogram**

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.
Essential Runtime Settings for MCDRAM Memory Affinity

• In cache mode, no special setting is needed to use MCDRAM
• In flat mode, using quad,flat as an example: NUMA node 1 is MCDRAM (need to request a reservation to use this mode)

  • #SBATCH -C knl,quad,flat
    – Enforced memory mapping to MCDRAM
    – If using >16 GB, malloc will fail

      \[ \text{srun } -n \ 16 \ -c \ 16 \ -\text{cpu\_bind=cores} \ --\text{mem\_bind=map\_mem:1} \ ./a.out \]
      Or:
      \[ \text{srun } -n \ 16 \ -c \ 16 \ -\text{cpu\_bind=cores} \ ... \ \text{numactl } -m \ 1 \ ./a.out \]
    – Preferred memory mapping to MCDRAM:
    – If using >16 GB, malloc will spill to DDR

      \[ \text{srun } -n \ 16 \ -c \ 16 \ -\text{cpu\_bind=cores} \ --\text{mem\_bind=preferred,map\_mem:1} \ ./a.out \]
      Or:
      \[ \text{srun } -n \ 16 \ -c \ 16 \ -\text{cou\_bind=cores} \ \text{numactl } -p \ 1 \ ./a.out \]
Building with libmemkind

- `% module load cray-memkind`
  Compiler wrappers will add
    - `--dynamic -lmemkind -lnuma`
- The **binary is built dynamically** by default

- Or use NERSC built module:
  - `% module load memkind`
  Compiler wrappers will add
    - `--lmemkind -ljemalloc -lnuma`
- The **binary is built statically** by default
Use libmemkind

- Use libmemkind to explicitly allocate selected arrays in MCDRAM
- C/C++  hbw_malloc() replaces malloc()
  ```c
  #include <hbwmalloc.h>
  // malloc(size) -> hbw_malloc(size)
  ```
- Fortran
  ```fortran
  !DIR$ MEMORY(bandwidth) a,b,c  ! cray
  real, allocatable :: a(:,,:), b(:,,:), c(:)
  !DIR$ ATTRIBUTES FASTMEM :: a,b,c  ! intel
  ```
- Caveat: only for dynamically-allocated arrays
  - Not local (stack) variables
  - Or Fortran pointers
AutoHBW: Automatic memkind

- Uses array size to determine whether an array should be allocated to MCDRAM
- No code changes necessary!
  ```bash
  module load autohbw
  ```
- Link with -lautohbw

Runtime environment variables:
```bash
export AUTO_HBW_SIZE=4K    # any allocation
  # >4KB will be placed in MCDRAM
export AUTO_HBW_SIZE=4K:8K # allocations
  # between 4KB and 8KB will
  # be placed in MCDRAM
```
Summary: How to Compile and Link

• Build on login nodes
  • module swap craype-haswell craype-mic-knl
    — For KNL-specific executables
  or
  • CC -axKNL,HSWELL ...
    — For Haswell/KNL portability
• Libraries will be automatically linked by compiler wrappers when specific modules are loaded
• Think about MCDRAM
  — --mem_bind, numactl, memkind, autohbm
• More details at
  — https://www.nersc.gov/users/computational-systems/cori/programming/compiling-codes-on-cori/#toc-anchor-6
How to Run Jobs on Cori KNL
Cori KNL Example Compute Nodes

- A Cori KNL node has 68 cores/272 CPUs, 96 GB DDR memory, 16 GB high bandwidth on package memory (MCDRAM)
- Three cluster modes, all-to-all, quadrant, sub-NUMA clustering, are available at boot time to configure the KNL mesh interconnect

A quad,cache node has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory). MCDRAM is hidden from the “numactl -H” result since it is a cache.

A quad,flat node has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory). NUMA node 1 has MCDRAM only

A snc2,flat node has 4 NUMA domains with DDR memory and all CPUs on NUMA nodes 0 and 1. (NUMA node 0 has physical cores 0 to 33 and all corresponding hyperthreads, and NUMA node 1 has physical cores 34 to 67 and all corresponding hyperthreads). NUMA nodes 2 and 3 have MCDRAM only
Can we just do a naïve `srun`?

Example: 16 MPI tasks x 8 OpenMP threads per task on a single 68-core KNL quad, cache node:

```bash
% export OMP_NUM_THREADS=8
% export OMP_PROC_BIND=spread
% export OMP_PLACES=threads
```

(other choice are “close”, “master”, “true”, “false”)

(other choices are: cores, sockets, and various ways to specify explicit lists, etc.)

```bash
% srun -n 16 ./xthi | sort -k4n,6n
```

Hello from rank 0, thread 0, on nid02304. (core affinity = 0)
Hello from rank 0, thread 1, on nid02304. (core affinity = 144) (on physical core 8)
Hello from rank 0, thread 2, on nid02304. (core affinity = 17)
Hello from rank 0, thread 3, on nid02304. (core affinity = 161) (on physical core 25)
Hello from rank 0, thread 4, on nid02304. (core affinity = 34)
Hello from rank 0, thread 5, on nid02304. (core affinity = 178) (on physical core 42)
Hello from rank 0, thread 6, on nid02304. (core affinity = 51)
Hello from rank 0, thread 7, on nid02304. (core affinity = 195) (on physical core 59)
Hello from rank 1, thread 0, on nid02304. (core affinity = 0)
Hello from rank 1, thread 1, on nid02304. (core affinity = 144)

It is a mess!
Importance of -c and --cpu_bind Options

• **The reason: 68 is not divisible by #MPI tasks!**
  – Each MPI task is getting 68x4/#MPI tasks of logical cores as the domain size
  – MPI tasks are crossing tile boundaries

• **Let’s set number of logical cores per MPI task (-c) manually by wasting extra 4 cores on purpose, which is 256/#MPI tasks.**
  – Meaning to use 64 cores only on the 68-core KNL node., and spread the logical cores allocated to each MPI task evenly among these 64 cores.
  – Now it looks good!
  – % srun -n 16 -c 16 --cpu_bind=cores ./xthi
    Hello from rank 0, thread 0, on nid09244. (core affinity = 0)
    Hello from rank 0, thread 1, on nid09244. (core affinity = 136)
    Hello from rank 0, thread 2, on nid09244. (core affinity = 1)
    Hello from rank 0, thread 3, on nid09244. (core affinity = 137)
Now It Looks Good!

### Process/thread affinity are good! (Marked first 6 and last MPI tasks only)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI rank 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
</tr>
<tr>
<td>MPI rank 1</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
</tr>
<tr>
<td>MPI rank 2</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>220</td>
<td>221</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 2</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>MPI rank 3</td>
<td>86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>MPI rank 3</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
<td>160</td>
<td>161</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>170</td>
</tr>
<tr>
<td>MPI rank 4</td>
<td>222</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 5</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 6</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 6</td>
<td>172</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 7</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 7</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 8</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 8</td>
<td>188</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI rank 9</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And so on for other MPI tasks and threads ....
Essential runtime settings for process/thread affinity

- Use `srun -c` and `--cpu_bind` flags to bind tasks to CPUs
  - `-c <n>` (or `--cpus-per-task=n`) allocates (reserves) `n` CPUs per task (process). It helps to evenly spread MPI tasks
  - Use `--cpu_bind=cores` (no hyperthreads) or `--cpu_bind=threads` (if hyperthreads are used)

- Use OpenMP envs, `OMP_PROC_BIND` and `OMP_PLACES` to fine pin each thread to a subset of CPUs allocated to the host task
  - Different compilers may have different default values for them.
  - The following provide compatible thread affinity among Intel, GNU and Cray compilers:
    - `OMP_PROC_BIND=true` # Specifying threads may not be moved between CPUs
    - `OMP_PLACES=threads` # Specifying a thread should be placed in a single CPU
Sample Job Script to Run on KNL quad, cache Nodes

Sample Job script (**MPI+OpenMP**)

```
#!/bin/bash 
#SBATCH -N 2
#SBATCH -q regular
#SBATCH -t 1:00:00
#SBATCH -L SCRATCH
#SBATCH -C knl,quad,cache

export OMP_NUM_THREADS=4
srun -n 128 -c 4 --cpu_bind=cores ./a.out
```

This job script requests 2 KNL quad, cache nodes to run 128 MPI tasks, 64 MPI tasks per node, allocating 4 CPUs per task, and binds each task to the 4 CPUs allocated within each core. The 4 OpenMP threads per MPI task are bound to the 4 CPUs in the core.

*Courtesy of Zhengji Zhao, NERSC*
Sample Job Script to Run on KNL quad,cache Nodes

Sample Job script (MPI+OpenMP)

```bash
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -q regular
#SBATCH -t 1:00:00
#SBATCH -L SCRATCH
#SBATCH -C knl,quad,cache

export OMP_PROC_BIND=true
export OMP_PLACES=threads
export OMP_NUM_THREADS=4
srtn -n 128 -c 4 --cpu_bind=cores ./a.out
```

With the above two OpenMP envs, each thread is now pinned to a single CPU within each core

Courtesy of Zhengji Zhao, NERSC
Essential runtime settings for MCDRAM memory affinity

- In cache mode, no special setting is needed to use MCDRAM
- In flat mode, using quad,flat as an example: NUMA node 1 is MCDRAM (need to request a reservation to use this mode)
- `#SBATCH -C knl,quad,flat`
  - Enforced memory mapping to MCDRAM
  - If using >16 GB, malloc will fail
  
  `srun -n 16 -c 16 -cpu_bind=cores --mem_bind=map_mem:1 ./a.out`
  
  Or:
  
  `srun -n 16 -c 16 -cpu_bind=cores ... numactl -m 1 ./a.out`
  
  - Preferred memory mapping to MCDRAM:
  
  - If using >16 GB, malloc will spill to DDR
  
  `srun -n 16 -c 16 -cpu_bind=cores --mem_bind=preferred,map_mem:1 ./a.out`
  
  Or:
  
  `srun -n 16 -c 16 -cou_bind=cores numactl -p 1 ./a.out`
How to Run Debug and Interactive Jobs

• Debug
  – Max 512 nodes, 30 min, run limit 1, queue limit 5
  % salloc -N 20 -q debug -C knl,quad,cache -t 30:00

• interactive
  – Instant allocation (or reject), run limit 1, max walltime 4 hrs, up to 64 nodes on Cori (Haswell and KNL) per repo
  % salloc -N 2 -q interactive -C knl,quad,cache -t 2:00:00

Reservation is made for today (500 KNL nodes total):
% salloc -N 2 -q regular -C knl,quad,cache -t 1:00:00
--reservation=KNL_hack-a-thon -A ntrain
Recommend: Use Hugepages

- The default page size is 4K.
- General recommend to use hugepages on KNL
- % module load craype-hugepages2M
  - There are many other modules with different hugepage sizes. 4M, 8M, ... 512M
  - 2M is generally helpful already
  - More details in “man intro_hugepages”
- See benefits with many applications: VASP, MILC, MFDn, etc.
- Hugepage availability decreases the longer a node is up. Recommend system admins monitor and reboot from time to time.

Effect of Hugepages on VASP Performance on Cori KNL Nodes
(Quad,Cache, 8 OpenMP threads per MPI task, Test case:Si256_HSE)
Recommend: Use `sbcast` (for Larger Jobs)

- By default, SLURM does not copy the executable onto each node. Large delay can happen between first and last node starting the job.
- Use "sbcast" to broadcast the executable to each compute node prior to job start, especially for jobs >1500 MPI tasks.

```
% sbcast ./mycode.exe /tmp/mycode.exe
% srun <srun options> numactl <numactl options> /tmp/mycode.exe
```

# or in the case of when `numactl` is not needed:
% srun --bcast=/tmp/mycode.exe <srun options> ./mycode.exe
Recommend: Use -S for Core Specialization

- Core specialization is a feature designed to isolate system overhead (system interrupts, etc.) to designated cores on a compute node.

- `#SBATCH -S 4` or `#SBATCH -S 2`

- Only works with `sbatch`, can not be used with `salloc`, which is already a wrapper script for `srun`. 
Recommend: Use Switches to Minimize Performance Variations

- Request max count of switches and max wait time in the queue
- `#SBATCH --switches=<count>[@<max-time>]`
- For example, requesting 1024 nodes, it needs roughly 3 switches (1 switch for ~384 nodes)
  - `#SBATCH --switches=3@24:00:00`
- Helps to minimize inter-group network communication to improve latency and bandwidth and limit variations for codes with many MPI calls. (especially helpful if only 1 switch is needed).
Affinity Verification Methods (1)

- NERSC has provided pre-built binaries from a Cray code (xthi.c) to display process thread affinity: check-mpi.intel.cori, check-mpi.cray.cori, check-hybrid.intel.cori, etc.

  \%
  -run -n 32 -c 8 --cpu_bind=cores check-mpi.intel.cori|sort -nk 4

  Hello from rank 0, on nid02305. (core affinity = 0,1,68,69,136,137,204,205)
  Hello from rank 1, on nid02305. (core affinity = 2,3,70,71,138,139,206,207)
  Hello from rank 2, on nid02305. (core affinity = 4,5,72,73,140,141,208,209)
  Hello from rank 3, on nid02305. (core affinity = 6,7,74,75,142,143,210,211)

- Intel compiler has a run time environment variable KMP_AFFINITY, when set to "verbose":

  OMP: Info #242: KMP_AFFINITY: pid 255705 thread 0 bound to OS proc set {55}
  OMP: Info #242: KMP_AFFINITY: pid 255660 thread 1 bound to OS proc set {10,78}
  OMP: Info #242: OMP_PROC_BIND: pid 255660 thread 1 bound to OS proc set {78} ...

- Cray compiler has a similar env CRAY_OMP_CHECK_AFFINITY, when set to "TRUE":

  [CCE OMP: host=nid00033 pid=14506 tid=17606 id=1] thread 1 affinity: 90
  [CCE OMP: host=nid00033 pid=14510 tid=17597 id=1] thread 1 affinity: 94 ...
Affinity Verification Methods (2)

• Use srun flag --cpu_bind=verbose to check thread affinity
  – Need to read the cpu masks in hexadecimal format

• Use srun flag --mem_bind=verbose,<type> to check memory affinity

• Use the numastat -p <PID> command to confirm while a job is running
Use Burst Buffer for faster IO

• Cori has 1.8PB of SSD-based “Burst Buffer” to support I/O intensive workloads
• Jobs can request a job-temporary BB filesystem, or a persistent (up to a few weeks) reservation
• More info at http://www.nersc.gov/users/computational-systems/cori/burst-buffer/
Running Jobs on KNL Summary

- Use -C knl,<NUMA>,<MCDRAM> to request KNL nodes
- Consider using 64 cores out of 68 in most cases
- Always explicitly use `srun`'s -c and --cpu_bind flags to spread the MPI tasks evenly over the cores/CPUs on the nodes
- Use OpenMP envs, OMP_PROC_BIND and OMP_PLACES to fine pin threads to the CPUs allocated to the MPI tasks
- Use `srun`'s --mem_bind or `numactl` to control memory affinity and access MCDRAM
  - memkind/autoHBW libraries can be used to allocate only selected arrays/memory allocations to MCDRAM
- Use the NERSC Job Script Generator tool
- More details at
  - [http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/](http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/)
How to Use Intel Tools on Cori KNL
VTune

- VTune is a performance analysis tool
- Intel presentation
- NERSC documentation
- Use Lustre file system (scratch) instead of GPFS (home or project)
- Compile: Use Cray compiler wrappers such as “cc’ with “-g -dynamic”
- Run: add --perf=vtune in sbatch or salloc; use “srun” to launch a job
- View: use GUI to view results
Using VTune Example

Compile and Run

% module swap craype-haswell craype-mic-knl
% cc -g -dynamic -qopenmp -O3 -o mycode.exe mycode.c  # can still use “-O3” along with “-g”

% module load vtune
% salloc -N 2 -t 30:00 -C knl,quad,cache --reservation=KNL_hack-a-thon -q regular -A
ntrain --perf=vtune    # today only
Or % salloc -N 2 -t 30:00 -C knl,quad,cache -q interactive --perf=vtune  # or ”-q debug”
% export OMP_NUM_THREADS=8
% export OMP_PROC_BIND=spread
% export OMP_PLACES=threads
% srun -n 8 -c 64 --cpu_bind=cores amplxe-cl -collect hotspots -r res_dir -trace-mpi
-finalization-mode=none -- ./mycode.exe
# can add -data-limit=0 to get >500 MB profiling data
% srun -n 8 -c 64 --cpu_bind=cores amplxe-cl -collect memory-access -knob analyze-
openmp=true -knob analyze-mem-objects=true -r res_dir -trace-mpi -- ./mycode.exe
Finalize Result on a login node:
% module load vtune
% amplxe-cl -finalize --finalization-mode=full -r <your_result_dir> -search-dir <path_to_your_binary>

View with GUI (NX is recommended)
% ssh -Y <username>@cori.nersc.gov
% module load vtune
% amplxe-gui
Current VTune Limitations

• Due to the impact of Spectre/Meltdown security issues, the patches installed are not compatible with the Vtune kernel driver, so currently the some of the collections using hardware event-based counters are not available, such as:
  – Advanced hotspots
  – General Exploration
  – Memory Access
APS (Application Performance Snapshot)

• Available within VTune
• Intel presentation
• Use Lustre file system (scratch) instead of GPFS (home or project)
• Compile: Use Cray compiler wrappers such as “cc’ with “-g - dynamic”
• Run: add --perf=vtune in sbatch or salloc; use “srun” to launch a job
• View: view text or html results
Using APS Example

Compile and Run

% module swap craype-haswell craype-mic-knl
% cc -g -dynamic -qopenmp -O3 -o mycode.exe mycode.c

% module load vtune
% salloc -N 2 -t 1:00:00 -C knl,quad,cache --reservation=KNL_hack-a-thon -q regular -A ntrain
--perf=vtune. # today only
Or % salloc -N 2 -t 1:00:00 -C knl,quad,cache -q interactive # or ”-t 30:00 -q debug”
% . $APS_DIR/apsvars.sh
% export OMP_NUM_THREADS=8
srun -n 8 -c 64 --cpu_bind=cores aps mycode.exe
aps --report <my_report_name>   # generate text and html reports
aps-report -t <my_report_name>   # generate MPI statistics
Advisor

• Advisor includes threading advisor and vectorization advisor
• Intel presentation
• NERSC documentation
• Use Lustre file system (scratch) instead of GPFS (home or project)
• Compile: Use cray compiler wrappers such as “cc” with “-g -dynamic” or use native Intel compilers such as “mpiicc” with “-g”
• Run: use “srun” or “mpirun” to launch a job
• View: use GUI to view results
Using Advisor Roofline Example

Compile and Run

% module swap craype-haswell craype-mic-knl
% cc -g -dynamic -qopenmp -O3 -o mycode.exe mycode.c  # can still use “-O3” along with “-g”

% salloc -N 2 -t 1:00:00 -C knl,quad,cache --reservation=KNL_hack-a-thon -q regular -A ntrain
# today only
Or % salloc -N 2 -t 1:00:00 -C knl,quad,cache -q interactive  # or ”-t 30:00 -q debug”
% module load advisor
% export OMP_NUM_THREADS=8
# From advisor/2018.up1 (which is the default) one step collection for non-MPI codes
# still needs to do 2-step collection for MPI codes
% srun -n 8 -c 64 --cpu_bind=cores advixe-cl -collect roofline -project-dir=./myproj -data-limit=0
-trace-mpi -- ./mycode.exe
# add “-no-auto-finalize” if the collection is too slow
# “-collect roofline” does the following two steps automatically:
% srun -n 8 -c 64 --cpu_bind=cores advixe-cl -collect survey -project-dir=./myproj -trace-mpi -- ./mycode.exe
% srun -n 8 -c 64 --cpu_bind=cores advixe-cl -collect tripcounts -flop -project-dir=./myproj -trace-mpi -- ./mycode.exe
Using Advisor Roofline Example

View with GUI (NX is recommended)
% ssh -Y <username> cori.nersc.gov
% module load advisor
% advixe-gui

Write Report into CSV
% advixe-cl --report survey --project-dir <my-project-dir> --show-all-columns --format=csv --report-output <my-output-file.csv>
Inspector

- Inspector is a memory and threading error checking tool
- Intel presentation
- NERSC documentation
- Use Lustre file system (scratch) instead of GPFS (home or project)
- Compile: Use native Intel compilers such as “mpiicc” (instead of Cray compiler wrappers such as “cc’) with “-g -dynamic”
- Run: use “mpirun” to launch a job
- View: use GUI to view results
Using Inspector Example

**Compile and Run**

% module swap craype-haswell craype-mic-knl
% module load impi
% mpiicc -g -dynamic -qopenmp -O3 -o mycode.exe mycode.c  # can still use “-O3” along with “-g”
% salloc -N 2 -t 30:00 -C knl,quad,cache --reservation=KNL_hack-a-thon -q regular -A ntrain  # today only
Or % salloc -N 2 -t 30:00 -C knl,quad,cache -q interactive  # or -q debug
% module load inspector
% module load impi
% export I_MPI_PMI_LIBRARY=/usr/lib64/slurmpmi/libpmi.so
# use small number of MPI ranks and OpenMP threads, the goal is to detect threading and memory issues.
% export OMP_NUM_THREADS=2
% mpirun -n 2 inspxe-cl -collect ti2 -result-dir=./myresult -- ./mycode.exe

**View with GUI** (NX is recommended)

% ssh -Y <username>@cori.nersc.gov
% module load inspector
% inspxe-gui
Thank you.