How to Compile & MCDRAM

Steve Leak
Building for Cori KNL nodes

- What’s different?
- How to compile
  - .. to use the new wide vector instructions
- What to link
- Making use of MCDRAM
  - High Bandwidth Memory
Don’t Panic
(much)
KNL can run Haswell executables
But ...

Haswell Executables can’t fully use KNL hardware

AVX2
(haswell)
Operation
on 4 DP
words

AVX-512 (knl)
Hardware
can compute
8 DP words
per
instruction
And ...

**KNL relies more on vectorization for performance**
And ...

**KNL memory hierarchy is more complicated**
How to compile

Best: Use compiler options to build for KNL
module swap craype-haswell craype-mic-knl

- The loaded craype-* module sets the target that the compiler wrappers (cc, CC, ftn) build for
  - Eg `--mkl` (GNU compiler), `--hmic-knl` (Cray compiler)
- craype-haswell is default on login nodes
- craype-mic-knl is for KNL nodes
How to compile

Best: Compiler settings to target KNL

Alternate:

CC -axMIC-AVX512,CORE-AVX2 <more-options> mycode.c++

- Only valid when using Intel compilers (cc, CC or ftn)
- -ax<arch> adds an “alternate execution paths” optimized for different architectures
  - Makes 2 (or more) versions of code in same object file
- NOT AS GOOD as the craype-mic-knl module
  - (module causes versions of libraries built for that architecture to be used - eg MKL)
How to compile

Recommendations:

• For best performance, use the craype-mic-knl module
  
  module swap craype-haswell craype-mic-knl
  CC -O3 -c myfile.c++

• If the same executable must run on KNL and Haswell nodes, use craype-haswell but add KNL-optimized execution path
  
  CC -axMIC-AVX512,CORE-AVX2 -O3 -c myfile.c++
What to link

Utility libraries
  • Not performance-critical (by definition)
    – KNL can run Xeon binaries .. can use Haswell-targeted versions
  • I/O libraries (HDF5, NetCDF, etc) should fit in this category too
    – (for Cray-provided libraries, compiler wrapper will use craype-* to select best build anyway)
What to link

Performance-critical libraries

• MKL: has KNL-targeted optimizations
  – Note: need to link with -lmemkind (more soon)
• PETsc, SLEPc, Caffe, Metis, etc:
  – (soon) has KNL-targeted builds
• Modulefiles will use craype-{haswell,mic-knl} to find appropriate library
• Key points:
  – Someone else has already prepared libraries for KNL
  – No need to do-it-yourself
  – Load the right craype-module
What to link

• **NERSC convention:**
  /usr/common/software/<name>/<version>/<arch>/[<PrgEnv>]

• **Eg:**
  /usr/common/software/petsc/3.7.2/hsw/intel
  /usr/common/software/petsc/3.7.2/knl/intel

• **KNL subfolder may be a symlink to hsw**
  – Libraries compiled with -axMIC-AVX512,CORE-AVX2

• **Modulefiles should do the right thing™**
  – Using CRAY_CPU_TARGET, set by craype-{haswell,mic-knl}
**Where to build**

- **Mostly: on the login nodes**
  - KNL is designed for scalable, vectorized workloads
  - Compiling is neither!
    - Will probably be much slower on KNL node than Xeon node

- **Cross-compiling**
  - You are compiling for a Xeon Phi (KNL) target, on a Xeon host
    - Tools like autoconf (./configure) may try to build-and-run small executables to test availability of libraries, etc .. which might not work
  - Compile on KNL compute node?
    - Slow (and currently not working)
  - craype-haswell + CFLAGS=-axMIC-AVX512,CORE-AVX2
Don’t Panic!

In Summary:

• Build on login nodes (like you do now)
• Use provided libraries (like you probably do now)

• Here’s the new bit:
• module swap craype-haswell craype-mic-knl
  – For KNL-specific executables, or
• CC -axMIC-AVX512,CORE-AVX2 …
  – For Haswell/KNL portability
What about MCDRAM?

• What’s different?
• How to compile
  – .. to use the new wide vector instructions
• What to link
  • Making use of MCDRAM
    – High Bandwidth Memory
MCDRAM in a nutshell

• **16GB on-chip memory**
  – cf 96GB off-chip DDR (Cori)

• **Not (exactly) a cache**
  – Latency similar to DDR

• **But very high bandwidth**
  – ~5x DDR

• **2 ways to use it:**
  – “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. Accessible via numactl, libnuma (page granularity)
MCDRAM in a nutshell - cache mode
MCDRAM in a nutshell - flat mode
How to use MCDRAM

• **Option 1: Let the system figure it out**
  – Cache mode, no changes to code, build procedure or run procedure
  – Most of the benefit, free, most of the time
How to use MCDRAM

• Option 2: Run-time settings only
  – Flat mode, no changes to code or build procedure
  – Does whole job fit within 16GB/node?
    • srun <options> numactl -m 1 ./myexec.exe
  – Too big?
    • srun <options> numactl -p 1 ./myexec.exe
How to use MCDRAM

• Option 3: Make your application NUMA-aware
  – Flat mode
  – Use libmemkind to explicitly allocate selected arrays in MCDRAM

```
#include <hbwmalloc.h>

malloc(size) -> hbw_malloc(size)
```

NUMA-aware extensible heap manager

Malloc implementation emphasizing fragmentation avoidance and concurrency

memkind

jemalloc

libnuma

API for NUMA allocation policy in Linux kernel
Using libmemkind in code

• C/C++  `hbw_malloc()` replaces `malloc()`
  ```
  #include <hbwmalloc.h>
  // malloc(size) -> hbw_malloc(size)
  ```

• 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
Using libmemkind in code

• 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.

Bandwidth Domain: DRAM, GB/sec
Building with libmemkind

• module load memkind
• (or module load cray-memkind)

• Compiler wrappers will add
  -lmemkind -ljemalloc -lnuma

• Fortran note: Not all compilers support FASTMEM directive
  – Currently Intel and maybe Cray
AutoHBW: Automatic memkind

- Uses array size to determine whether an array should be allocated to MCDRAM
- No code changes necessary!
- 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
```
Don’t Panic!

In Summary:
• Build on login nodes (like you do now)
• Use provided libraries (like you probably do now)
• Here’s the new bit:
  • module swap craype-haswell craype-mic-knl
    – For KNL-specific executables, or
  • CC -axMIC-AVX512,CORE-AVX2 ... 
    – For Haswell/KNL portability

And:
• Think about MCDRAM
  – numactl, memkind, autohbm
A few final notes

• Edison executables (probably) won’t work without recompile
  – ISA-compatible, but...
  – Cori has newer OS version, updated libraries
  – So: recompile for Cori

• KNL-optimized MKL uses libmemkind
  – Will need to link with -lmemkind -ljemalloc
  – Should be invisibly integrated in future version
Running jobs on Cori KNL nodes

Zhengji Zhao
Agenda

• What’s new on KNL nodes
• Process/thread/memory affinity
• Sample job scripts
• Summary
1 Socket/Node
68 Cores (272 CPUs) /Node
36 Tiles/Node (34 active)
2 Cores/Tile; 4 CPUs/Core
1.4 GB/Core DDR memory
235 MB/Core MCDRAM

• A Cori KNL node has 68 cores/272 CPUs running at 1.4 GHz, 96 GB DDR memory, 16 GB high (~5xDDR) 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.
**KNL overview – MCDRAM modes**

**Cache Mode**
- No source code changes needed
- Misses are expensive

**Flat Mode**
- Code changes required
- Exposed as a NUMA node
- Access via memkind library, job launchers, and/or numactl

**Hybrid Mode**
- Combination of the cache and flat modes

**MCDRAM can be configured in three different modes at boot time – cache, flat, and hybrid modes**
What’s new on KNL nodes (in comparison with Cori Haswell nodes from the perspective of running jobs)

1. A lot more (slower) cores on the node
2. Much reduced per core memory
3. Dynamically configurable NUMA and MCDRAM modes

...
A proper process/thread/memory affinity is the basis for optimal performance

• Process affinity (or CPU pinning): bind a (MPI) process to a CPU or a range of CPUs on the node, so that the process executes within the designated CPUs instead of drifting around to other CPUs on the node.

• Thread affinity: fine pin each thread of a process to a CPU or CPUs within the CPUs that are designated to the process.
  – Threads live in the process that owns them, so the process and thread affinity are not separable.

• Memory affinity: restrict processes to allocate memories from the designated NUMA nodes only rather than any NUMA nodes.
The minimum goal of process/thread/memory affinity is to achieve best resource utilization and to avoid NUMA performance penalty

- Spread MPI tasks and threads onto the cores and CPUs on the nodes as evenly as possible so that no cores and CPUs are oversubscribed while others stay idle. This can ensure the resources available on the node, such as cores, CPUs, NUMA nodes, memory and network bandwidths, etc., can be best utilized.

- Avoid accessing remote NUMA nodes as much as possible so to avoid performance penalty.

- In context of KNL, enable and control the MCDRAM access.
Using srun’s --cpu_bind option and OpenMP environment variables to achieve desired process/thread affinity

• **Use srun --cpu_bind to bind tasks to CPUs**
  – Often needs to work with the --c option of srun to evenly spread MPI tasks on the CPUs on the nodes
  – The srun --c <n> (or --cpus-per-task=n) allocates (reserves) n number of CPUs per task (process)
  – --cpu_bind=[{verbose,quiet},]type, type: cores, threads, map_cpu:<list of CPUs>, mask_cpu:<list of masks>, none, ...

• **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 are recommended, which yield a more 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
  – Use **OMP_DISPLAY_ENV=true** to display the OpenMP environment variables set (useful when checking the default compiler behavior)
Using `srun`'s `--mem_bind` option and/or `numactl` to achieve desired memory affinity

- **Use `srun` `--mem_bind` for memory affinity**
  
  - `--mem_bind=[{verbose,quiet},]type: local, map_mem:<NUMA id list>, mask_mem:<NUMA mask list>, none,...`
  
  - E.g., `--mem_bind=<MCDRAM NUMA id>` when allocations fit into MDCRAM in flat mode

- **Use `numactl` `--p <NUMA id>`**
  
  - Srun does not have this functionality currently (16.05.6), will be supported in Slurm 17.02.
  
  - E.g., `numactl --p <MCDRAM NUMA id> ./a.out` so that allocations that don’t fit into MCDRAM spill over to DDR memory
Default Slurm behavior with respect to process/thread/memory binding

• By Slurm default, a decent CPU binding is set only when the MPI tasks per node x CPUs per task = the total number of CPUs allocated per node, e.g., 68x4=272
• Otherwise, Slurm does not do anything with CPU binding. The srun’s --cpu_bind and –c options must be used explicitly to achieve optimal process/thread affinity.
• No default memory binding is set by Slurm. Processes can allocate memory from all NUMA nodes. The --mem_bind (or numactl) should be used explicitly to set memory bindings.
The default distribution, the –m option of srun, is block:cyclic on Cori.
   - The cyclic distribution method distributes allocated CPUs for binding to a given task consecutively from the same socket, and from the next consecutive socket for the next task, in a round-robin fashion across sockets.

The –m block:block also works. You are encouraged to experiment with –m block:block as some applications perform better with the block distribution.
   - The block distribution method distributes allocated CPUs consecutively from the same socket for binding to tasks, before using the next consecutive socket.

The –m option is relevant to the KNL nodes when they are configured in the sub-NUMA cluster modes, e.g., SNC2, SNC4, etc. Slurm treats “NUMA nodes with CPUs” as “sockets”, although KNL is a single socket node.
Available partitions and NUMA/MCDRAM modes on Cori KNL nodes (not finalized view yet)

• **Same partitions as Haswell**
  – `#SBATCH --p regular`
  – `#SBATCH --p debug`
  – Type `sinfo -s` for more info about partitions and nodes

• **Using the --C knl,<NUMA>,<MCDRAM> options of sbatch to request KNL nodes with desired features**
  – `#SBATCH --C knl,quad,flat`

• **Supports combination of the following NUMA/MCDRAM modes:**
  – AllowNUMA=a2a,snc2,snc4,hemi,quad
  – AllowMCDRAM=cache,split,equal,flat
  – Quad,flat is the default for now (not finalized)

• **Nodes can be rebooted automatically**
  – Frequent reboots are not encouraged, as they currently take a long time
  – We are testing various memory modes so to set a proper default mode
Example of running interactive batch job with KNL nodes in the \textit{quad,cache} mode

\texttt{zz217@gert01:~> \$salloc -N 1 -p debug -t 30:00 -C knl,quad,cache}
\texttt{salloc: Granted job allocation 5545}
\texttt{salloc: Waiting for resource configuration}
\texttt{salloc: Nodes nid00044 are ready for job}

\texttt{zz217@nid00044:~> \$numactl -H}
\texttt{available: 1 nodes (0)}
\texttt{node 0 cpus: 0 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 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 ...... 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271}
\texttt{node 0 size: 96757 MB}
\texttt{node 0 free: 94207 MB}
\texttt{node distances:}
\texttt{node 0}
\texttt{0: 10}

- Run the \texttt{numactl -H} command to check if the actual NUMA configuration matches the requested NUMA,MCDRAM mode
- The \texttt{quad,cache} mode has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory)
- The MCDRAM is hidden from the \texttt{numactl -H} command (it is a cache).
Sample job script to run under the quad,cache mode

Sample Job script (Pure MPI)

```bash
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH -C knl,quad,cache
export OMP_NUM_THREADS=1 #optional
srun -n 64 -c 4 --cpu_bind=cores ./a.out
```

*) The use of “export OMP_NUM_THREADS=1” is optional but recommended even for pure MPI codes. This is to avoid unexpected thread forking (compiler wrappers may link your code to the multi-threaded system provided libraries by default).

This job script requests 1 KNL node in the quad,cache mode. The srun command launches 64 MPI tasks on the node, allocating 4 CPUs per task, and binds processes to cores. The resulting task placement is shown in the right figure. The Rank 0 will be pinned to Core0, Rank1 to Core1, ..., Rank63 will be pinned to Core63. Each MPI task may move within the 4 CPUs in the cores.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>68</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>136</td>
<td>204</td>
<td>137</td>
<td>205</td>
</tr>
<tr>
<td>Rank 0</td>
<td>Rank 1</td>
<td>Rank 2</td>
<td>Rank 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core 60</th>
<th>Core 61</th>
<th>Core 62</th>
<th>Core 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>128</td>
<td>61</td>
<td>129</td>
</tr>
<tr>
<td>196</td>
<td>264</td>
<td>197</td>
<td>265</td>
</tr>
<tr>
<td>Rank 60</td>
<td>Rank 61</td>
<td>Rank 62</td>
<td>Rank 63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core 64</th>
<th>Core 65</th>
<th>Core 66</th>
<th>Core 67</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>132</td>
<td>65</td>
<td>133</td>
</tr>
<tr>
<td>200</td>
<td>268</td>
<td>201</td>
<td>269</td>
</tr>
</tbody>
</table>

Each 2x2 box above is a core with 4 CPUs (hardware threads). The numbers shown in each CPU box is the CPU ids. The last 4 cores are not used in this example. The cores 4-59 were not be shown.
Sample job script to run under the **quad,cache** mode

**Sample Job script (Pure MPI)**

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

export OMP_NUM_THREADS=1  #optional*
srun -n 16 --cpu_bind=cores ./a.out
```

This job script requests 1 KNL node in the quad,cache mode. The `srun` command launches 16 MPI tasks on the node, allocating 16 CPUs per task, and binds each process to 4 cores/16 CPUs. The resulting task placement is shown in the right figure. The Rank 0 is pinned to Core 0-3, and Rank 1 to Core 4-7, ..., Rank 15 to Core 60-63. The MPI task may move within the 16 CPUs in the 4 cores.

Process affinity outcome

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 68 1 69 3 70 4 71</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>136 204 137 205 138 206 139 207</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... Rank 0 ...

<table>
<thead>
<tr>
<th>Core 60</th>
<th>Core 61</th>
<th>Core 62</th>
<th>Core 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 128 61 129 62 130 63 131</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>196 264 197 265 198 266 199 267</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... Rank 15 ...

<table>
<thead>
<tr>
<th>Core 64</th>
<th>Core 65</th>
<th>Core 66</th>
<th>Core 67</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 132 65 133 66 134 67 135</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 268 201 269 202 270 203 271</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sample job script to run under the quad,cache mode

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

```bash
#!/bin/bash
#SBATCH –N 1
#SBATCH –p regular
#SBATCH –t 1:00:00
#SBATCH -C knl,quad,cache

export OMP_NUM_THREADS=4
srun -n64 -c4 --cpu_bind=cores ./a.out
```

This job script requests 1 KNL node in the quad,cache mode to run 64 MPI tasks on the node, allocating 4 CPUs per task, and binds each task to the 4 CPUs allocated within the cores. Each MPI task runs 4 OpenMP threads. The resulting task placement is shown in the right figure. The Rank 0 will be pinned to Core 0, Rank 1 to Core 1, ..., Rank 63 to Core 63. The 4 threads of each task are pinned within the core. Depending on the compilers used to compile the code, the 4 threads in each core may or may not move between the 4 CPUs.
Sample job script to run under the **quad,cache** mode

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

```bash
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH --C knl,quad,cache
export OMP_PROC_BIND=true
export OMP_PLACES=threads
export OMP_NUM_THREADS=4
srun -n64 -c4 --cpu_bind=cores ./a.out
```

**Process/thread affinity outcome**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>68</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>136</td>
<td>204</td>
<td>137</td>
<td>205</td>
</tr>
<tr>
<td>Rank 0</td>
<td>Rank 1</td>
<td>Rank 2</td>
<td>Rank 3</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>3</td>
<td>71</td>
</tr>
<tr>
<td>138</td>
<td>206</td>
<td>139</td>
<td>207</td>
</tr>
</tbody>
</table>

... Core 60, Core 61, Core 62, Core 63, ... Core 64, Core 65, Core 66, Core 67

<table>
<thead>
<tr>
<th>Core 64</th>
<th>Core 65</th>
<th>Core 66</th>
<th>Core 67</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>132</td>
<td>65</td>
<td>133</td>
</tr>
<tr>
<td>200</td>
<td>268</td>
<td>201</td>
<td>269</td>
</tr>
<tr>
<td>202</td>
<td>270</td>
<td>203</td>
<td>271</td>
</tr>
</tbody>
</table>

With the above two OpenMP envs, each thread is pinned to a single CPU within each core. The resulting thread affinity (and task affinity) is shown in the right figure. E.g., for Rank 0, Thread 0 is pinned to CPU 0, Thread 1 to CPU 68, Thread 2 to CPU 136, and Thread 3 is pinned to CPU 204.
Sample job script to run under the **quad,cache** mode

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

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

export OMP_NUM_THREADS=8
srun -n 16 --cpu_bind=cores ./a.out
```

Depending on the compiler implementations, the 8 threads in each task may or may not move between 4 cores/16 CPUs allocated to the host task.
Sample job script to run under the **quad,cache** mode

Sample Job script (**MPI+OpenMP**)  
```bash  
#!/bin/bash  
#SBATCH --N 1  
#SBATCH --p regular  
#SBATCH --t 1:00:00  
#SBATCH --C knl,quad,cache  
export OMP_PROC_BIND=true  
export OMP_PLACES=threads  
export OMP_NUM_THREADS=8  
srun -n16 --c16 --cpu_bind=cores ./a.out  
```

Process/thread affinity outcome

With the above two OpenMP envs, each thread is pinned to a single CPU on the cores allocated to the task. The resulting process/thread is shown in the right figure. E.g., for Rank 0, Thread 0 is pinned to the CPU 0 (on Core 0), Thread 1 to the CPU 1 (on Core1), Threads 2 to CPU 2 (on Core 2), and so on.
Example of running under the **quad,flat** mode interactively

```
zz217@gert01:~> salloc -p debug -t 30:00 -C knl,quad,flat
zz217@nid00037:~> numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 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 26 27
  28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
  56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74  ....  262 263 264 265 266
  267 268 269 270 271
node 0 size: 96759 MB
node 0 free: 94208 MB
node 1 cpus:
node 1 size: 16157 MB
node 1 free: 16091 MB
node distances:
node  0  1
  0: 10 31
  1: 31 10
```

```
zz217@cori10:~> scontrol show node nid10388
NodeName=nid10388 Arch=x86_64 CoresPerSocket=68
  CPUAlloc=0 CPUErr=0 CPUTot=272 CPULoad=0.01
AvailableFeatures=knl,flat,split,equal,cache,a2a,snc2,snc4,hemi,quad
  ActiveFeatures=knl,cache,quad
  ...
  State=IDLE ThreadsPerCore=4
  ...
  BootTime=2016-10-31T13:43:12
  ...
```
Sample job script to run under the **quad,flat** mode

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

```bash
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH -C knl,quad,flat
export OMP_NUM_THREADS=4
srun -n64 -c4 --cpu_bind=cores ./a.out
export OMP_NUM_THREADS=8
srun -n16 -c16 --cpu_bind=cores ./a.out
```

**Process affinity outcome**

```
Core 0  | Core 1  | Core 2  | Core 3  |
-------|--------|--------|--------|
0      | 68     | 1      | 69     | 2      | 70     | 3      | 71     |
136    | 204    | 137    | 205    | 138    | 206    | 139    | 207    |
```

```
Core 60 | Core 61 | Core 62 | Core 63 |
--------|--------|--------|--------|
60      | 178    | 61     | 129    | 62     | 130    | 63     | 131    |
196     | 264    | 197    | 265    | 198    | 266    | 199    | 267    |
```

```
Core 0  | Core 1  | Core 2  | Core 3  |
-------|--------|--------|--------|
0      | 68     | 1      | 69     | 2      | 70     | 3      | 71     |
136    | 204    | 137    | 205    | 136    | 204    | 137    | 205    |
```

```
Core 60 | Core 61 | Core 62 | Core 63 |
--------|--------|--------|--------|
60      | 128    | 61     | 129    | 62     | 130    | 63     | 131    |
196     | 264    | 196    | 264    | 196    | 264    | 196    | 264    |
```

---

**- 50 -**
Sample job script to run under the **quad,flat** mode

### Sample Job script (MPI+OpenMP)
```bash
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH --C knl,quad,flat

export OMP_PROC_BIND=true
export OMP_PLACES=threads

export OMP_NUM_THREADS=4
srun -n64 -c4 --cpu_bind=cores ./a.out

export OMP_NUM_THREADS=8
srun -n16 -c16 --cpu_bind=cores ./a.out
```

### Process/thread affinity outcome

```
<table>
<thead>
<tr>
<th>Rank</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
</tr>
<tr>
<td>1</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>205</td>
<td>206</td>
<td>207</td>
<td>208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>129</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td></td>
<td>196</td>
<td>197</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>264</td>
<td>265</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>130</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td></td>
<td>197</td>
<td>198</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td></td>
<td>265</td>
<td>266</td>
<td>267</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>129</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td></td>
<td>196</td>
<td>197</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>264</td>
<td>265</td>
<td>266</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Rank</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>129</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td></td>
<td>196</td>
<td>197</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>264</td>
<td>265</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>130</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td></td>
<td>197</td>
<td>198</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td></td>
<td>265</td>
<td>266</td>
<td>267</td>
<td></td>
</tr>
</tbody>
</table>
```
Sample job script to run under the quad, flat mode using MCDRAM

Sample Job script (MPI+OpenMP)
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH -C knl,quad,flat

#When the memory footprint fits in 16GB of MCDRAM (NUMA node 1), runs out of MCDRAM
export OMP_NUM_THREADS=4
export OMP_PROC_BIND=true
export OMP_PLACES=threads

srun -n64 -c4 --cpu_bind=cores --mem_bind=map_mem:1 ./a.out

#or using numactl -m
srun -n64 -c4 --cpu_bind=cores numactl -m 1 ./a.out

Sample Job script (MPI+OpenMP)
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH -C knl,quad,flat

#Prefers running on MCDRAM (NUMA node 1) if memory footprint does not fit on MCDRAM, spills to DDR
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=true
export OMP_PLACES=threads

srun --n16 --c16 --cpu_bind=cores numactl -p 1 ./a.out
How to check the process/thread affinity

- **Use** `srun` **flag**: `--cpu_bind=verbose`
  - Need to read the cpu masks in hexadecimal format
- **Use a Cray provided code** `xthi.c` *(see backup slides).*
- **Use** `--mem_bind=verbose,<type>` **to check memory affinity**
- **Use** the `numastat –p <PID>` command **to confirm while a job is running**
- **Use** environmental variables *(Slurm, compiler specific)*
  - `SLURM_CPU_BIND_VERBOSE --cpu_bind verbosity` *(quiet,verbose).*
A few useful commands

- `sinfo --format="%F %b"` for available features of nodes, or `sinfo --format="%C %b"`
  - A/I/O/T (allocated/idle/other/total)
- `scontrol show node <nid>`
- `scontrol show job <jobid> -ddd`
- `sinfo --s` to see available partitions and nodes
- `sbatch`, `srun`, `squeue`, `sinfo` and other Slurm command man pages
  - need to distinguish the job allocation time (`#SBATCH`) and job step creation time (`srun` within a job script)
  - Some options are only available at Job allocation time, such as `--ntasks-per-core`, some only work when certain plugins are enabled
Summary

• Use –C knl,<NUMA>,<MCDRAM> to request KNL nodes with the same partitions as Haswell nodes (debug, or regular)
• Always explicitly use srun’s --cpu_bind and -c option 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 tasks
• Use srun’s --mem_bind and numactl –p to control memory affinity and access MCDRAM
  – Using memkind/autoHBW libraries can be used to allocate only selected arrays/memory allocations to MCDRAM (Steve’s talk)
Summary (2)

• Consider using 64 cores out of 68 in most cases
• More sample job scripts can be found in our website
  — http://www.nersc.gov/users/computational-systems/cori/
    running-jobs/running-jobs-on-cori-knl-nodes/
• We have provided a job script generator to help you to generate batch job scripts for KNL (and Haswell, Edison)
• Slurm KNL features are in continuous development and some instructions are subject to change

------

URL for the job script generator:
https://my.nersc.gov/script_generator.php
Backup slides
Cray provided a code to check process/thread affinity

**xthi.c** ([http://portal.nersc.gov/project/training/KNLUserTraining20161103/UsingCori/xthi.c/](http://portal.nersc.gov/project/training/KNLUserTraining20161103/UsingCori/xthi.c/))

- **To compile,**

  - `cc -qopenmp -o xthi.intel xthi.c`  # Intel compilers
  - `cc -fopenmp -o xthi.gnu xthi.c`  # GNU compilers
  - `cc -o xthi.gnu  xthi.c`  # Cray compilers

- **To run,**

  - `salloc -N 1 -p debug -C knl,quad,flat`  # start an interactive 1 node job
  
  - `export OMP_DISPLAY_ENV=true`  # to display envs used/set by openmp runtime
  - `export OMP_NUM_THREADS=4`
  - `srun -n 64 -c4 --cpu_bind=verbose,cores xthi.intel`  # run 64 tasks with 4 threads each
  
  - `srun -n 16 -c16 --cpu_bind=verbose,cores xthi.intel`  # run 16 tasks 4 threads each
sinfo --format="%F %b" to show the number of nodes with active features

zz217@cori01:~> date
Mon Nov 28 15:10:36 PST 2016

zz217@cori01:~> sinfo --format="%F %b"
NODES(A/I/O/T) ACTIVE_FEATURES
1805/157/42/2004 haswell
0/107/1/108 knl,flat,a2a
0/0/1/1 knl
0/225/6/231 knl,flat,quad
2500/1942/60/4502 knl,cache,quad
2676/995/6/3677 quad,cache,knl
768/0/0/768 snc4,flat,knl
0/8/0/8 quad,flat,knl
0/8/1/9 snc2,flat,knl

This command shows that there are 8179 KNL nodes in quad,cache mode; 239 nodes in quad,flat mode; 768 nodes in snc4,flat mode; 9 in snc2,flat mode.

The order of features does not make difference.
The --cpu_bind option of `srun` enables CPU bindings

```
salloc -N 1 -p debug -C knl,quad,flat
...
```

```
srun -n 4 ./a.out  # no CPU bidings. Tasks can move around within 68 cores/272 CPUs
```

```
srun -n 4 --cpu_bind=cores ./a.out
```

```
srun -n 4 --cpu_bind=threads ./a.out
```

```
srun -n 4 --cpu_bind=map_cpu:0,204,67,271 ./a.out
```
The **--cpu_bind** option: the **-c option** spreads tasks (evenly) on the CPUs on the node

```
salloc -N 1 -p debug -C knl,quad,flat
...

srun -n 4 -c8 --cpu_bind=cores ./a.out  
srun -n 16 -c16 --cpu_bind=cores ./a.out
```

- **Rank 0, Core 0**: First 8 cores/32 CPUs are used; rest 60 cores/240 CPUs stay idle
- **Rank 1, Core 60**: Last 4 cores/16 CPUs are idle
The --cpu_bind option (continued): the –c option spread tasks (evenly) on the CPUs on the node

salloc -N 1 -p debug -C knl,quad,flat
...

srun -n 4 -c8 -cpu_bind=threads ./a.out    srun -n 16 -c16 -cpu_bind=threads ./a.out

---

First 8 cores/32 CPUs are used; rest 60 cores/240 CPUs stay idle

Last 4 cores/16 CPUs are idle
The -c option: --cpu_bind=cores vs --cpu_bind=threads

salloc -N 1 -p debug -C knl,quad,flat
...

srun -n 4 -c 6 --cpu_bind=cores ./a.out

srun -n 4 -c 6 --cpu_bind=threads ./a.out

First 8 cores/32 CPUs are used; rest 60 cores/240 CPUs stay idle

First 6 cores/24 CPUs are used; rest 62 cores/248 CPUs stay idle
zz217@nid11512:~> numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 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 26 27 28 29 30 31 32 33 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237
node 0 size: 48293 MB
node 0 free: 46047 MB
node 1 size: 48466 MB
node 1 free: 44949 MB
node 2 cpus:
node 2 size: 8079 MB
node 2 free: 7983 MB
node 3 cpus:
node 3 size: 8077 MB
node 3 free: 7980 MB
node distances:
node 0 1 2 3
0: 10 21 31 41
1: 21 10 41 31
2: 31 41 10 41
3: 41 31 41 10

Snc2,flat (salloc -N 1 -p regular -C knl,snc2,flat)
The default distribution on Cori is \texttt{-m block:cyclic}

\begin{verbatim}
salloc -N 2 -p debug -C knl,snc2,flat
...
srun -n 11 -c4 -cpu_bind=cores ./a.out
#or
srun -n 11 -c4 -cpu_bind=cores -m block:cyclic ./a.out
\end{verbatim}
The default distribution on Cori is `-m block:cyclic` (continued)

```plaintext
salloc --N 2 --p debug --C knl,snc2,flat
...
srun --n 11 --c8 --cpu_bind=cores ./a.out
# or
srun --n 11 --c8 --cpu_bind=cores -m block:cyclic ./a.out
```

![Diagram showing the default distribution on Cori](image-url)
Block distribution: `-m block:block`

Salloc `–N 2 –p debug –C knl,snc2,flat`
...
Srun `–n 11 –c4 –cpu_bind=cores –m block:block ./a.out`
Block distribution: \texttt{-m block:block} (continued)

Salloc \texttt{--N 2 --p debug --C knl,snc2,flat}

\ldots

Srun \texttt{--n 11 --c8 --cpu_bind=cores --m block:block ./a.out}
Sample job script to run MPI+OpenMP under the snc2,flat mode

#!/bin/bash
#SBATCH –N 1
#SBATCH –p regular
#SBATCH –t 1:00:00
#SBATCH –L SCRATCH
#SBATCH –C knl,snc2,flat
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=true
export OMP_PLACES=threads

# using 2 CPUs (hardware threads) per core using DDR memory (or using MCDRAM via libmemkind)
srun --cpu_bind=cores --mem_bind=local ./a.out

# using 2 CPUs (hardware threads) per core using MCDRAM
srun --cpu_bind=cores --mem_bind=map_mem:2,3 ./a.out
# or
numactl --m 2,3 .a.out

# using 2 CPUs (hardware threads) per core with MCDRAM preferred
srun --cpu_bind=cores numactl --p 2,3 ./a.out

# using 4 CPUs (hardware threads) per core with MCDRAM preferred
srun --cpu_bind=cores numactl --p 2,3 ./a.out
Sample job script to run large jobs (> 1500 MPI tasks)
(quad,flat)

Sample job script (MPI+OpenMP)
#!/bin/bash -l
#SBATCH --N 100
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH --L SCRATCH
#SBATCH -C knl,quad,flat

export OMP_NUM_THREADS=4
export OMP_PROC_BIND=true
export OMP_PLACES=threads

#using 4 CPUs (hardware threads) per core using DDR memory (or using MCDRAM via libmemkind)
srun --bcast=/tmp/a.out --n6400 --c4 --cpu_bind=cores --mem_bind=local ./a.out

#or
#using 4 CPUs (hardware threads) per core with MCDRAM preferred
sbcast ./a.out /tmp/a.out  #copy a.out to the /tmp of each compute node allocated first
srun--n6400 --c4 --cpu_bind=cores numactl --p 1 /tmp/a.out
Sample job script to use core specialization (quad, flat)

Sample job script (MPI+OpenMP)
#!/bin/bash -l
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH --L SCRATCH
#SBATCH --C knl,quad,flat
#SBATCH --S 1

export OMP_NUM_THREADS=4
export OMP_PROC_BIND=true
export OMP_PLACES=threads

#using 4 CPUs (hardware threads) per core using DDR memory (or using MCDRAM via libmemkind)
Srun -n64 -c 4 --cpu_bind=cores --mem_bind=local ./a.out

#or
#using 4 CPUs (hardware threads) per core with MCDRAM preferred
Srun -n64 -c 4 --cpu_bind=cores numactl -p 1 ./a.out
Sample job script to run with Intel MPI (quad,flat)

Sample job script (MPI+OpenMP)
#!/bin/bash
#SBATCH --N 1
#SBATCH --p regular
#SBATCH --t 1:00:00
#SBATCH -C knl,quad,flat

export OMP_NUM_THREADS=4
export OMP_PROC_BIND=true
export OMP_PLACES=threads

module load impi
export I_MPI_PMI_LIBRARY=/usr/lib64/slurmpmi/libpmi.so
#export I_MPI_FABRICS=shm:tcp

#using 4 CPUs (hardware threads) per core using DDR memory (or using MCDRAM via libmemkind)
srun --n64 --c4 --cpu_bind=cores --mem_bind=local ./a.out

#or
#using 4 CPUs (hardware threads) per core with MCDRAM preferred
srun --n64 --c4 --cpu_bind=cores numactl --p 1 ./a.out
Thank you!