Agenda

• Process/thread/memory affinity for optimal performance
• Submitting jobs to run on Cori KNL nodes
• Compiling/building codes
• Library support
• Memkind/hbwmalloc etc.
Running jobs on Cori KNL nodes

Zhengji Zhao
A Cori KNL node has 68 cores/272 CPUs, 96GB DDR memory, 16GB high bandwidth on package memory (MCDRAM)

Three NUMA cluster modes are available as boot time options
MCDRAM memory can be configured in three different modes at boot time, cache, flat, and hybrid modes

**Cache Mode**
- 16GB MCDRAM
- 96GB DDR

**Flat Mode**
- 16GB MCDRAM
- 96GB DDR

**Hybrid Mode**
- 8 or 12GB MCDRAM
- 4 or 8 GB MCDRAM
- 96GB DDR

- No source code changes needed
- Misses are expensive
- Code changes required
- Exposed as a NUMA node
- Access via memkind library, job launchers, and/or numactl
- Combination of the cache and flat modes
Correct 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
• Thread affinity: pin each thread to run on a designated CPU or a range of CPUs
• Memory affinity allows processes to run on designated NUMA nodes only
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: the number of CPUs allocated 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 threads to CPUs**
  – OMP_PROC_BIND – Whether threads may be moved between CPUs; OMP_PLACES – Specifies which CPUs the threads should be placed in
  – Different compilers may have different default values for them, so the following are recommended:
    • OMP_PROC_BIND=true
    • OMP_PLACES=threads
    • OMP_DISPLAY_ENV=true
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 now, 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
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 working on various memory modes to set as defaults
Default Slurm behavior with respect to task/thread/memory binding

- Default CPU binding is set only when the MPI tasks per node x CPUs per task = the total number of CPUs allocated on the nodes
  - When Hyperthreading (--nasks-per-core=1) is not used, binds to cores
  - When hyperthreading (--ntasks-per-core=4) is used, binds to threads.
- Otherwise, srun’s --cpu_bind and –c options should be used to achieve optimal process/thread affinity
  - -c: the number of CPUs allocated per MPI task (process)
- No default memory binding is set (can allocate memory from all NUMA nodes), the –mem_bind (or numactl) should be used explicitly
- The default distribution, the –m option, is block:cyclic
  - Srun –m: default is block:cyclic; but –m block:block also works
  - Users are encouraged to experiment with block distribution
zz217@gert01:~> salloc -N 1 –p debug –t 30:00 -C knl,quad,cache
salloc: Granted job allocation 5545
salloc: Waiting for resource configuration
salloc: Nodes nid00044 are ready for job
zz217@nid00044:~> numactl -H
available: 1 nodes (0)
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
node 0 size: 96757 MB
node 0 free: 94207 MB
node distances:
node  0
  0: 10
Sample job script to request 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=verbose,cores ./a.out

export OMP_NUM_THREADS=8
export OMP_PROC_BIND=true
export OMP_PLACES=threads
srun --n16 --c16 --cpu_bind=verbose,cores ./a.out
```

---

**Process affinity outcome**

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

---

**Pure MPI**

```bash
export OMP_NUM_THREADS=1  # optional
srun -n64 -c4 --cpu_bind=verbose,cores ./a.out
```
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 CPUSoc=272 CPULoad=0.01
AvailableFeatures=knl,flat,splitequal,cache,a2a,snc2,snc4,hemi,quad
ActiveFeatures=knl,cache,quad
... State=IDLE ThreadsPerCore=4
... BootTime=2016-10-31T13:43:12
...
Sample job script to request 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 -n 64 -c 4 --cpu_bind=cores ./a.out

export OMP_NUM_THREADS=8
export OMP_PROC_BIND=true
export OMP_PLACES=threads
srun -n 16 -c 16 --cpu_bind=cores ./a.out
```

**Process affinity outcome**

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

- 14 -
Sample job script to request the **quad,flat** mode using MCDRAM

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

```bash
#!/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) memory, runs out of MCDRAM
export OMP_NUM_THREADS=4
srun -n64 -c4 --cpu_bind=cores --mem_bind=map_mem:1 ./a.out
```

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

```bash
#!/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 of your app does not fit on MCDRAM, spills to DDR
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
srun --n16 --c16 --cpu_bind=cores numactl --p 1 ./a.out
```
Available methods to check affinity

• Use `srun` flag: `--cpu_bind=verbose`
  – Need to read the cpu mask in hexadecimal format

• Use a Cray provided code `xthi.c`.

• 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)
A few recommended commands

- **Sinfo** –format="%C %b" for available features of nodes
  - A/I/O/T (allocated/idle/other/total)
- **Sinfo show node <nid>**
- **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 and job step creation time
  - Some options are only available at Job allocation time, such as –ntasks-per-core, some only work with certain plugins
  - the #SBATCH directives; job step creation time is for the command line (although srun can do allocation and job step creation at the same time)
Summary

- Use \(-C\) knl,<NUMA>,<MCDRAM> to request KNL nodes with the same partitions as Haswell nodes.
- Always explicitly use srun’s \(\text{-cpu\_bind}\) option to achieve an optimal process/task placement on the KNL node.
- Use the \(\text{-c}\) option to spread the MPI tasks evenly over the CPUs on the node.
- Use OpenMP envs, OMP\_PROC\_BIND and OMP\_PLACES to finely pin threads to CPUs.
- Use srun’s \(\text{-mem\_bind}\) and numactl \(\text{-p}\) to control memory affinity and access the MCDRAM memory.
  - Using memkind/autoHBW libraries to allocate only selected arrays to MCDRAM will be discussed shortly in the next talk.
Summary (2)

- Consider using 64 cores out of 68 in most cases
- We will post our recommendations and sample job scripts on the NERSC website
- 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
xthi.c code: compile with cc –qopenmp xthi.c (Intel); cc –fopenmp xthi.c (GNU); cc xthi.c (Cray)
Building for Cori KNL nodes

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 HBM
  – “High Bandwidth Memory” == “MCDRAM”
Building for Cori KNL nodes

Don’t Panic
(much)
KNL can run Haswell executables

KNL ISA

KNL implements all legacy instructions
- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions
- 512-bit FP/Integer Vectors
- 32 registers, & 8 mask registers
- Gather/Scatter

Conflict Detection: Improves Vectorization
Prefetch: Gather and Scatter Prefetch
Exponential and Reciprocal Instructions

Previous Code name Intel® Xeon® processors
Xeon Phi = Intel® Xeon Phi™ processor
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

First: 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 -mknl, -hmic-knl
- craype-haswell is default on login nodes
- craype-mic-knl is for KNL nodes
How to compile

First: 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
How to compile

Recommendations:
• For best performance, KNL-specific: Use both
  
  module swap craype-haswell craype-mic-knl
  CC -axMIC-AVX512,CORE-AVX2 -O3 -c myfile.c++

• The craype-mic-knl module ensures the compiler wrappers will link knl-optimized libraries
  — For cray-* modules, also by convention for NERSC-installed software (more soon)
How to compile

Recommendations:

• For KNL/Haswell portability, use craype-haswell but add KNL-optimized execution path

```
CC -axMIC-AVX512,CORE-AVX2 -O3 -c myfile.c++
```

• The (default) craype-haswell module ensures the compiler wrappers will link Haswell-compatible libraries

• The alternate execution path (-ax) enables KNL-optimized versions of functions
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
What to link

Performance-critical libraries

• MKL: has KNL-targeted optimizations
  – Note: need to link with 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
  – Could compile on KNL compute node, or..
  – 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 HBM
  – “High Bandwidth Memory” == “MCDRAM”
HBM 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 HBM (note: granularity is a page)
- “Flat” mode: appears to OS as separate NUMA node, with no local CPUs. Accessible via numactl, libnuma
HBM in a nutshell
How to use HBM

• **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 HBM

- **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 HBM

- **Option 3: Make your application NUMA-aware**
  - Flat mode
  - Use `libmemkind` to explicitly allocate selected arrays in HBM

```
#include <hbwmalloc.h>

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

NUMA-aware extensible heap manager

Malloc implementation emphasizing fragmentation avoidance and concurrency

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
  ```
real, allocatable :: a(:,:), b(:,:), c(:)
!DIR$ ATTRIBUTES FASTMEM :: a,b,c
  ```

• Caveat: only for dynamically-allocated arrays
  – Not local (stack) variables
  – Or Fortran pointers
Using libmemkind in code

• Which arrays to put in HBM?
  – Vtune memory-access measurements:
  – amplxe-cl -collect memory-access ...
Building with libmemkind

- module load memkind
- \texttt{(or module load cray-memkind)}

- Compiler wrappers will add
  - \texttt{-lmemkind -ljemalloc -lnuma}

- Fortran note: only Intel compiler supports FASTMEM directive (at the moment)
AutoHBW: Automatic memkind

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

Runtime environment variables:
export AUTO_HBW_SIZE=4K    # any allocation
    # >4KB will be placed in HBM
export AUTO_HBW_SIZE=4K:8K # allocations
    # between 4KB and 8KB will
    # be placed in HBM
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 HBM
  – 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