Optimization Strategies for Cori

Jack Deslippe
NERSC User Services
Wednesday Feb 25, 2015
Introduction to Cori
What is different about Cori?

- **Cori will begin to transition the workload to more energy efficient architectures**
- **Cray XC system with over 9300 Intel Knights Landing (Xeon-Phi) compute nodes**
  - Self-hosted, (not an accelerator) manycore processor with over 60 cores per node
  - On-package high-bandwidth memory
- **Data Intensive Science Support**
  - NVRAM Burst Buffer to accelerate applications
  - 28PB of disk and >700 GB/sec I/O bandwidth

System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.
What is different about Cori?

<table>
<thead>
<tr>
<th>Edison (Ivy-Bridge):</th>
<th>Cori (Knights-Landing):</th>
</tr>
</thead>
<tbody>
<tr>
<td>● 12 Cores Per CPU</td>
<td>● 60+ Physical Cores Per CPU</td>
</tr>
<tr>
<td>● 24 Virtual Cores Per CPU</td>
<td>● 240+ Virtual Cores Per CPU</td>
</tr>
<tr>
<td>● 2.4-3.2 GHz</td>
<td>● Much slower GHz</td>
</tr>
<tr>
<td>● Can do 4 Double Precision Operations per Cycle (+ multiply/add)</td>
<td>● Can do 8 Double Precision Operations per Cycle (+ multiply/add)</td>
</tr>
<tr>
<td>● 2.5 GB of Memory Per Core</td>
<td>● &lt; 0.3 GB of Fast Memory Per Core</td>
</tr>
<tr>
<td>● ~100 GB/s Memory Bandwidth</td>
<td>● &lt; 2 GB of Slow Memory Per Core</td>
</tr>
<tr>
<td></td>
<td>● Fast memory has ~ 5x DDR4 bandwidth</td>
</tr>
</tbody>
</table>
What is different about Cori?

**Edison (Ivy-Bridge):**
- 12 Cores Per CPU
- 24 Virtual Cores Per CPU
- 2.4-3.2 GHz
- Can do 4 Double Precision Operations per Cycle (+ multiply/add)
- 2.5 GB of Memory Per Core
- ~100 GB/s Memory Bandwidth

**Cori (Knights-Landing):**
- 60+ Physical Cores Per CPU
- 240+ Virtual Cores Per CPU
- Much slower GHz
- Can do 8 Double Precision Operations per Cycle (+ multiply/add)
- < 0.3 GB of Fast Memory Per Core
- < 2 GB of Slow Memory Per Core
- Fast memory has ~ 5x DDR4 bandwidth
What is different about Cori?

Two Big Changes:

1. More on node parallelism. More cores, bigger vectors

2. Small amount of very fast memory. (similar-ish amounts of traditional DDR)
Key Concepts
MPI Vs. OpenMP For Multi-Core Programming

Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime.
PARATEC computes parallel FFTs across all processors.

Involves MPI all-to-all communication (small messages, latency bound).

Reducing the number of MPI tasks in favor OpenMP threads makes large improvement in overall runtime.
There is another important form of on-node parallelism: Vectorization. CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently.

\[
\begin{pmatrix}
ad_1 \\
\vdots \\
a_n \\
\end{pmatrix} =
\begin{pmatrix}
b_1 \\
\vdots \\
b_n \\
\end{pmatrix} +
\begin{pmatrix}
c_1 \\
\vdots \\
c_n \\
\end{pmatrix}
\]
There is another important form of on-node parallelism: **Vectorization**. CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently.

\[
\begin{align*}
\text{do } i &= 1, n \\
a(i) &= b(i) + c(i) \\
\text{enddo}
\end{align*}
\]

- **Intel Xeon Sandy-Bridge/Ivy-Bridge:** 4 Double Precision Ops Concurrently
- **Intel Xeon Phi:** 8 Double Precision Ops Concurrently
- **NVIDIA Kepler GPUs:** 32 SIMT threads
Things that prevent vectorization in your code

Compilers want to “vectorize” your loops whenever possible. But sometimes they get stumped. Here are a few things that prevent your code from vectorizing:

Loop dependency:

```plaintext
do i = 1, n
   a(i) = a(i-1) + b(i)
enddo
```

Task forking:

```plaintext
do i = 1, n
   if (a(i) < x) cycle
   if (a(i) > x) ...
enddo
```
Consider the following loop:

```verbatim
do i = 1, n
    do j = 1, m
        c = c + a(i) * b(j)
    enddo
enddo
```

Assume, \( n \) & \( m \) are very large such that \( a \) & \( b \) don’t fit into cache.

Then,

During execution, the number of loads From DRAM is

\[ n \times m + n \]
Consider the following loop: Assume, $n$ & $m$ are very large such that $a$ & $b$ don’t fit into cache.

\[
\begin{align*}
do i &= 1, n \\
do j &= 1, m \\
c &= c + a(i) \times b(j) \\
enddo \\
enddo
\end{align*}
\]

Assume, $n$ & $m$ are very large such that $a$ & $b$ don’t fit into cache.

Then,

During execution, the number of loads From DRAM is $n \times m + n$

Requires 8 bytes loaded from DRAM per FMA (if supported). Assuming 100 GB/s bandwidth on Edison, we can at most achieve 25 GFlops/second (2 Flops per FMA)

Much lower than 460 GFlops/second peak on Edison node. Loop is memory bandwidth bound.
Roofline Model For Edison

Edison Node Roofline Based on Stream of 85GB/s and Peak Flops of 460 GFlop/Sec

Operational Intensity (Flops/Byte) vs. Attainable GFlops/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling
Improving Memory Locality

Improving Memory Locality. Reducing bandwidth required.

Loads From DRAM:

\[ n \times m + n \]

Loads From DRAM:

\[ \frac{m}{\text{block}} \times (n + \text{block}) = n \times \frac{m}{\text{block}} + m \]
Improving Memory Locality Moves you to the Right on the Roofline

Edison Node Roofline Based on Stream of 85GB/s and Peak Flops of 460 GFlop/Sec

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

Operational Intensity (Flops/Byte) vs. Attainable GFlops/Sec
Optimization Strategy
Optimizing Code For Cori is like:

A. A Staircase ?
B. A Labyrinth ?
C. A Space Elevator?
An Ant Farm!

OpenMP scales only to 4 Threads

larges cache miss rate

Communication dominates beyond 100 nodes

Code shows no improvements when turning on vectorization

50% Walltime is IO

IO bottlenecks

Compute intensive doesn’t vectorize

MPI/OpenMP Scaling Issue

Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.

Can you use a library?

Increase Memory Locality

Create micro-kernels or examples to examine thread level performance, vectorization, cache use, locality.

Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.

The Dungeon: Simulate kernels on KNL. Plan use of on package memory, vector instructions.

Utilize performant / portable libraries
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Make Algorithm Changes

Run Example in “Half Packed” Mode

Is Performance affected by Half-Packing?

Yes

Your Code is at least Partially Memory Bandwidth Bound

No

Run Example at “Half Clock” Speed

Is Performance affected by Half-Clock Speed?

Yes

You are at least Partially CPU Bound

No

Use IPM and Darshan to Measure and Remove Communication and IO Bottlenecks from Code

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

- Yes
  - Can You Reduce Memory Requests Per Flop In Algorithm?
    - Yes
      - Explore Using HBM on Cori For Key Arrays
    - No
      - Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune
- No
  - You are at least Partially CPU Bound
    - Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune
  - Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
    - Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)
Use IPM and Darshan to Measure and Remove Communication and IO Bottlenecks from Code

Use IPM to Measure Communication Time

https://www.nersc.gov/users/software/debugging-and-profiling/ipm/

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>calls</th>
<th>&lt;%mpi&gt;</th>
<th>&lt;%wall&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallclock</td>
<td>953.272</td>
<td>29.7897</td>
<td>29.6092</td>
<td>29.9752</td>
</tr>
<tr>
<td>user</td>
<td>837.25</td>
<td>26.1641</td>
<td>25.71</td>
<td>26.92</td>
</tr>
<tr>
<td>system</td>
<td>60.6</td>
<td>1.89375</td>
<td>1.52</td>
<td>2.59</td>
</tr>
<tr>
<td>mpi</td>
<td>264.267</td>
<td>8.25834</td>
<td>7.73025</td>
<td>8.70985</td>
</tr>
<tr>
<td>%comm</td>
<td>27.7234</td>
<td>25.8873</td>
<td>29.3705</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[time]</th>
<th>[calls]</th>
<th>&lt;%mpi&gt;</th>
<th>&lt;%wall&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>188.386</td>
<td>639616</td>
<td>71.29</td>
<td>19.76</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>69.5032</td>
<td>639616</td>
<td>26.30</td>
<td>7.29</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>6.34936</td>
<td>639616</td>
<td>2.40</td>
<td>0.67</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>0.0177442</td>
<td>32</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>0.00540609</td>
<td>32</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Use Darshan to Measure IO Time/Performance

https://www.nersc.gov/users/software/debugging-and-profiling/darshan
Measuring Your Memory Bandwidth Usage (VTune)

Measure memory bandwidth usage in VTune. (Next Talk)

Compare to Stream GB/s.

If 90% of stream, you are memory bandwidth bound.

If less, more tests need to be done.
Are you memory or compute bound? Or both?

- Run Example in “Half Packed” Mode
  - Is Performance affected by Half-Packing?
    - Yes: Your Code is at least Partially Memory Bandwidth Bound
    - No: Run Example at “Half Clock” Speed
  - No: Run Example at “Half Clock” Speed
    - Is Performance affected by Half-Clock Speed?
      - Yes: You are at least Partially CPU Bound
      - No: Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
Are you memory or compute bound? Or both?

Run Example in “Half Packed” Mode

If you run on only half of the cores on a node, each core you do run has access to more bandwidth

```
aprun -n 24 -N 12 - S 6 ...
```

VS

```
aprun -n 24 -N 24 -S 12 ...
```

If your performance changes, you are at least partially memory bandwidth bound
Are you memory or compute bound? Or both?

Reducing the CPU speed slows down computation, but doesn’t reduce memory bandwidth available.

Run Example at “Half Clock” Speed

\[ \text{aprun --p-state=2400000 ...} \quad \text{VS} \quad \text{aprun --p-state=2200000 ...} \]

If your performance changes, you are at least partially compute bound
So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse

2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

Profit by getting ~ 5x more bandwidth GB/s.
So, you are Compute Bound?

What to do?

1. Make sure you have good OpenMP scalability. Look at VTune to see thread activity for major OpenMP regions.

![Diagram showing thread activity and concurrency]

2. Make sure your code is vectorizing. Look at Cycles per Instruction (CPI) and VPU utilization in vtune.

See whether intel compiler vectorized loop using compiler flag: -qopt-report=5
So, you are neither compute nor memory bandwidth bound?

You may be memory latency bound (or you may be spending all your time in IO and Communication).

If running with hyper-threading on Edison improves performance, you *might* be latency bound:

```
aprun -j 2 -n 48 .... VS aprun -n 24 ....
```

If you can, try to reduce the number of memory requests per flop by accessing contiguous and predictable segments of memory and reusing variables in cache as much as possible.

On Cori, each core will support up to 4 threads. Use them all.
BerkeleyGW Case Study
BerkeleyGW Use Case

★ Big systems require more memory. Cost scales as $N_{\text{atoms}}^2$ to store the data.
★ In an MPI GW implementation, in practice, to avoid communication, data is duplicated and each MPI task has a memory overhead.
★ Users sometimes forced to use 1 of 24 available cores, in order to provide MPI tasks with enough memory. 90% of the computing capability is lost.
Targeting Intel Xeon Phi Many Core Architecture

1. Target more on-node parallelism. (MPI model already failing users)
2. Ensure key loops/kernels can be vectorized.

**Example: Optimization steps for Xeon Phi Coprocessor**

Refactor to Have 3 Loop Structure:
Outer: MPI
Middle: OpenMP
Inner: Vectorization

Add OpenMP
Ensure Vectorization
Final Loop Structure

ngpown typically in 100's to 1000s. Good for many threads.

ncouls typically in 1000s - 10,000s. Good for vectorization.

Original inner loop. Too small to vectorize!

Attempt to save work breaks vectorization and makes code slower.
Hybrid MPI-OpenMP Scaling Improvements.

* Major Improvement between 1.0 and 1.1
* Trading MPI tasks for OpenMP threads, yields improved performance (mostly in MPI communication costs) and allows scaling to higher core counts.
The End