What is different about Cori-II?

- Cori-II 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** (MIC) processor with up to 72 cores per node
  - On-package **high-bandwidth memory**
- **Data Intensive Science Support**
  - NVRAM **Burst Buffer** to accelerate applications

System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.
<table>
<thead>
<tr>
<th>Edison (Ivy-Bridge)</th>
<th>Cori (Knights-Landing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Cores Per CPU</td>
<td>Up to 72 Physical Cores Per CPU</td>
</tr>
<tr>
<td>24 Virtual Cores Per CPU</td>
<td>Up to 288 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 (slow) Memory Per Core (DDR4 DRAM)</td>
<td>&lt; 2 GB of Slow Memory Per Core (DDR4 DRAM)</td>
</tr>
<tr>
<td>~100 GB/s Memory Bandwidth</td>
<td>&lt; 0.3 GB of Fast Memory Per Core (MCDRAM)</td>
</tr>
<tr>
<td></td>
<td>Fast memory has ~ 5x DDR4 bandwidth</td>
</tr>
</tbody>
</table>
Basic Optimization Concepts
Need to move beyond plain MPI: MPI + X?

Need to explicitly consider both **inter-node** (MPI) and **on-node** parallelism (OpenMP, Pthreads, Vectorization etc.) in application.

**Existing applications may suffer from:**
- Memory overhead due to duplicated data in traditional MPI tasks
- Lack of SIMD/Vectorization expressiveness in app.
- Potential MPI latency in all-to-all communication patterns

**Possible Solutions:**
- MPI+OpenMP
- MPI+Pthreads
- PGAS (MPI+PGAS)
- Task Based Parallel Programming (HPX, Legion etc.)
Tools for on-node parallelism

• Threads

• Vectorization
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.

Figure Courtesy of Andrew Canning
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single {
        printf("Level %d: number of threads in the team: %d
", level, omp_get_num_threads());
    }
} 
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2) {
        report_num_threads(1);
        #pragma omp parallel num_threads(2) {
            report_num_threads(2);
            #pragma omp parallel num_threads(2) {
                report_num_threads(3);
            }
        }
    } 
    return(0);
}
Nested OpenMP

- Beneficial to use nested OpenMP to allow more fine-grained thread parallelism
- Achieving best **process and thread affinity is crucial** in getting good performance with nested OpenMP
- Combinations of OpenMP environment variables and run time flags are needed for different compilers and different batch schedulers on different NERSC systems.

**Example usage with Intel compiler with Torque/Moab on Edison:**

```
setenv OMP_NESTED true
setenv OMP_NUM_THREADS 4,3
setenv OMP_PROC_BIND spread,close
setenv KMP_HOT_TEAMS 1
setenv KMP_HOT_TEAMS_MAX_LEVELS 2
aprun -n 2 -S 1 -d 12 –cc numa_node ./nested.intel.edison
```

- Refer to NERSC “Nested OpenMP” web page for detailed instructions illustrated with sample hybrid MPI/OpenMP programs:
Tools for on-node parallelism

- Threads
- Vectorization
Vectorization

Another important form of on-node parallelism – SIMD (Single Instruction Multiple Data)

Vectorization: CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently.

\[
\begin{pmatrix}
a_1 \\
... \\
a_n
\end{pmatrix}
= 
\begin{pmatrix}
b_1 \\
... \\
b_n
\end{pmatrix}
+ 
\begin{pmatrix}
c_1 \\
... \\
c_n
\end{pmatrix}
\]
Another important form of on-node parallelism – SIMD (Single Instruction Multiple Data)

Vectorization: CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently.

\[
\begin{pmatrix}
a_1 \\
\vdots \\
a_n
\end{pmatrix} = \begin{pmatrix}
b_1 \\
\vdots \\
b_n
\end{pmatrix} + \begin{pmatrix}
c_1 \\
\vdots \\
c_n
\end{pmatrix}
\]

Intel Xeon Sandy-Bridge/Ivy-Bridge: 4 Double Precision Ops Concurrently
Intel Xeon Phi: 8 Double Precision Ops Concurrently
Things that prevent vectorization

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:

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

Task forking:

```fortran
do i = 1, n
   if (a(i) < x) cycle
   if (a(i) > x) ...
enddo
```
Things that prevent vectorization

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```c
for (many iterations) {
    ... many flops ...
    et = exp(outcome1)
    tt = pow(outcome2,3)
    IN = IN * et + tt
}
```
Things that prevent vectorization

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```c
for (many iterations) {
  ... many flops ...
  et = exp(outcome1)
  tt = pow(outcome2, 3)
  IN = IN * et + tt
}
```

```c
for (many iterations) {
  ... many flops ...
  et(i) = exp(outcome1)
  tt(i) = pow(outcome2, 3)
}

for (many iterations) {
  IN = IN * et(i) + tt(i)
}
```
Things that prevent vectorization

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```plaintext
for (many iterations) {
  ... many flops ...
  et = exp(outcome1)
  tt = pow(outcome2,3)
  IN = IN * et + tt
}
```

```plaintext
for (many iterations) {
  ... many flops ...
  et(i) = exp(outcome1)
  tt(i) = pow(outcome2,3)
}
```

```plaintext
for (many iterations) {
  IN = IN * et(i) + tt(i)
}
```

30% speed up for entire application!
Things that prevent vectorization

Example From Cray COE Work on XGC1

Original

```
real (8), dimension
(5, (col_f_nvr-1)*(col_f_nvz-1),
(col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
  do index jp = 1, mesh_Nrml
    index_2dp = index jp + mesh_Nrml*(index_ip-1)

    tmp_vol = cs2%local center_volume(index jp)
    tmp_f_half_v = f_half(index jp, index ip) * tmp_vol
    tmp_dfdr_v = dfdr(index jp, index ip) + tmp_vol
    tmp_dfdz_v = dfdz(index jp, index ip) * tmp_vol

    tmpr(1:3) = tmpr(1:3) + Ms(1:3, index_2dp, index_2D)* tmp_f_half_v
    tmpr(5) = tmpr(5) + Ms(4, index_2dp, index_2D)*tmp_dfdr_v +
```

Optimized

```
real (8), dimension
((col_f_nvr-1), 5, (col_f_nvz-1),
(col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzml
  do index jp = 1, mesh_Nrml
    index_2dp = index jp + mesh_Nrml*(index_ip-1)
    tmp_vol = cs2%local center_volume(index jp)
    tmp_f_half_v = f_half(index jp, index ip) * tmp_vol
    tmp_dfdr_v = dfdr(index jp, index ip) * tmp_vol
    tmp_dfdz_v = dfdz(index jp, index ip) * tmp_vol

    tmpr(index jp, 1) = tmpr(index jp, 1) + Ms(index jp, 1, index_ip, index_2D)*
      tmp_f_half_v
    tmpr(index jp, 2) = tmpr(index jp, 2) + Ms(index jp, 2, index_ip, index_2D)*
      tmp_f_half_v
    tmpr(index jp, 3) = tmpr(index jp, 3) + Ms(index jp, 3, index_ip, index_2D)*
      tmp_f_half_v
    tmpr(index jp, 4) = tmpr(index jp, 4) + Ms(index jp, 4, index_ip, index_2D)*
      tmp_dfdr_v
    tmpr(index jp, 5) = tmpr(index jp, 5) + Ms(index jp, 5, index_ip, index_2D)*
      tmp_dfdr_v
```

U.S. DEPARTMENT OF ENERGY
Office of Science

40 YEARS AT THE FRONT
crystal clear solutions

NERSC
BERKELEY LAB
Things that prevent vectorization

Example From Cray COE Work on XGC1

~40% speed up for kernel
Consider the following loop:

```
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 \textbf{number of loads From DRAM} is

\[ n \times m + n \]
Consider the following loop:

```
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

\[ n \times m + n \]

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

- Roofline
- Unbalanced Ceiling
- Unbalanced No SIMD Ceiling

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

Improving Memory Locality. Reducing bandwidth required.

\[
\text{Bandwidth requirement decreased by a factor } \text{block}.
\]

\[
\begin{align*}
\text{Loads From DRAM:} & \quad n \times m + n \\
\text{Improved:} & \quad m / \text{block} \times (n + \text{block}) \\
& = n \times m / \text{block} + m
\end{align*}
\]
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

Attainable GFlops/Sec vs Operational Intensity (Flops/Byte)
Optimization Strategy
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Make Algorithm Changes

- Explore Using HBM on Cori For Key Arrays

Is Performance affected by Half-Clock Speed?

- Run Example at “Half Clock” Speed
- Run Example in “Half Packed” Mode

Is Performance affected by Half-Packing?

- Yes
- No

Your Code is at least Partially Memory Bandwidth Bound

- Yes
- No

You are at least Partially CPU Bound

- Yes
- No

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Make Algorithm Changes
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Yes

Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

No

Explore Using HBM on Cori For Key Arrays

No

Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune

Yes

Can You Reduce Memory Requests Per Flop In Algorithm?

No

Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

No

You are at least Partially CPU Bound

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Yes

Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune

No

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)
Measure memory bandwidth usage in VTune. 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
      - 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

```
#SBATCH -N 8
#SBATCH --tasks-per-node=16
srun -n 128 ./
```

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

If performance changes, you are at least partially memory bandwidth bound.

---

**BOUT memory bandwidth test**

<table>
<thead>
<tr>
<th></th>
<th>Walltime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>300</td>
</tr>
<tr>
<td>Half-packed</td>
<td>250</td>
</tr>
</tbody>
</table>

---

---
Are you memory or compute bound? Or both?

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

If performance changes, you are at least partially compute bound.

Run Example at “Half Clock” Speed

srun -N 4 --cpu-freq 2300000 ./elm_pb

srun -N 4 --cpu-freq 1400000 ./elm_pb

BOUT core speed test

Walltime (s)

Regular

Slow-core

0 100 200 300 400 500 600 700

0 100 200 300 400

0 100 200 300 400
So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse

1. 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.

2. Make sure your code is vectorizing. Look at Cycles per Instruction (CPI) and VPU utilization in VTune. Check whether Intel compiler vectorized loop using compiler flag: `-qopt-report=5`
Not all flops are equal: Complex-Division
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/Cori-I improves performance, you *might* be latency bound:

```
#SBATCH -N 4
srun -n 256 ./elm_pb
```

```
#SBATCH -N 4
srun -n 128 ./elm_pb
```

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-II, each core will support up to 4 threads. Use them all (after making sure problem fits in memory).
Another example: BerkeleyGW

- Restructured code to have 3 loop levels
  - Outer loops for MPI (1000+ trip count)
  - Middle loop for OpenMP (100+ trip count, lots of work, low barrier overhead)
  - Inner loop (1000+ trip count, vectorizable)
Final Loop Structure

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

Original inner loop. Too small to vectorize!

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

Attempt to save work breaks vectorization and makes code slower.

```fortran
!$OMP DO reduction(+:achtemp)
do my_igp = 1, ngpown
    ...
    do iw=1,3
        scht=0D0
        wxt = wx_array(iw)
        do ig = 1, ncouls
            !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle
            wdiff = wxt - wtilde_array(ig,my_igp)
            delw = wtilde_array(ig,my_igp) / wdiff
            ...
            scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)
            scht = scht + scha(ig)
        enddo ! loop over g
        sch_array(iw) = sch_array(iw) + 0.5D0*scht
    enddo
    achtemp(:) = achtemp(:) + sch_array(:) * vcoul(my_igp)
enddo
```
How much performance can we get from 3 arrays in Fast Memory?

- Identify the candidate (key arrays) for HBM
  - VTune Memory Access tool can help to find key arrays
  - Using NUMA affinity to simulate HBM on a dual socket system
  - Use FASTMEM directives and link with jemalloc/memkind libraries

On Edison (NERSC Cray XC30):

```fortran
real, allocatable :: a(:,,:), b(:,,:), c(:)
!DIR$ ATTRIBUTE FASTMEM :: a, b, c
% module load memkind jemalloc
% ftn -dynamic -O3 -openmp mycode.f90
% export MEMKIND_HBW_NODES=0
% aprun -n 1 -cc numa_node numactl --membind=1 --cpunodebind=0 ./myexecutable
```

On Haswell:

```
Link with ‘-ljemalloc -lmemkind -lpthread -lnuma “
% numactl --membind=1 --cpunodebind=0 ./myexecutable
```

<table>
<thead>
<tr>
<th>Application</th>
<th>All memory on far memory</th>
<th>All memory on near memory</th>
<th>Key arrays on near memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>BerkeleyGW</td>
<td>baseline</td>
<td>52% faster</td>
<td>52.4% faster</td>
</tr>
<tr>
<td>EmGeo</td>
<td>baseline</td>
<td>40% faster</td>
<td>32% faster</td>
</tr>
<tr>
<td>XGC1</td>
<td>baseline</td>
<td></td>
<td>24% faster</td>
</tr>
</tbody>
</table>
Conclusions
1. Optimizing code for Cori-II is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.

2. Understanding **bandwidth and compute limitations** of hotspots are key to deciding how to improve code.

3. Use profiling tools like VTune and CrayPat on Edison/Cori-I to find and characterize hotspots.

4. Localized regions of code (preferably as stand-alone kernels) help sandboxing optimization efforts.