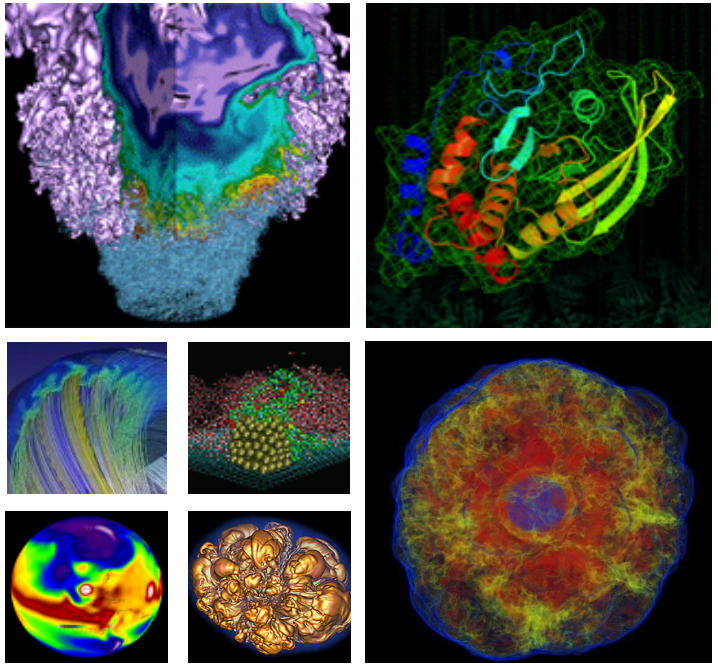
Optimization Strategies for Cori



Jack Deslippe

NERSC User Services

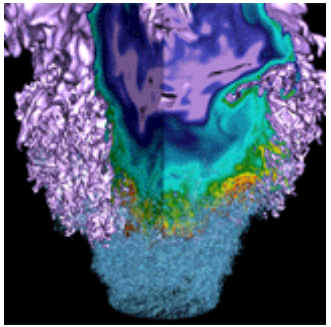
Wednesday Feb 25, 2015

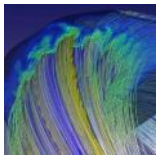
NERSC 40 YEARS at the FOREFRONT

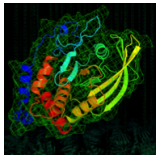
U.S. DEPARTMENT OF ENERGY | Office of Science

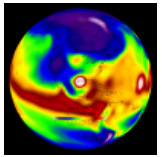
BERKELEY LAB

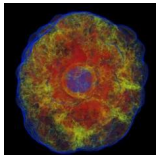
Introduction to Cori

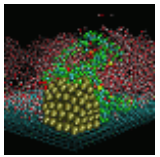












NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

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?

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?

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):

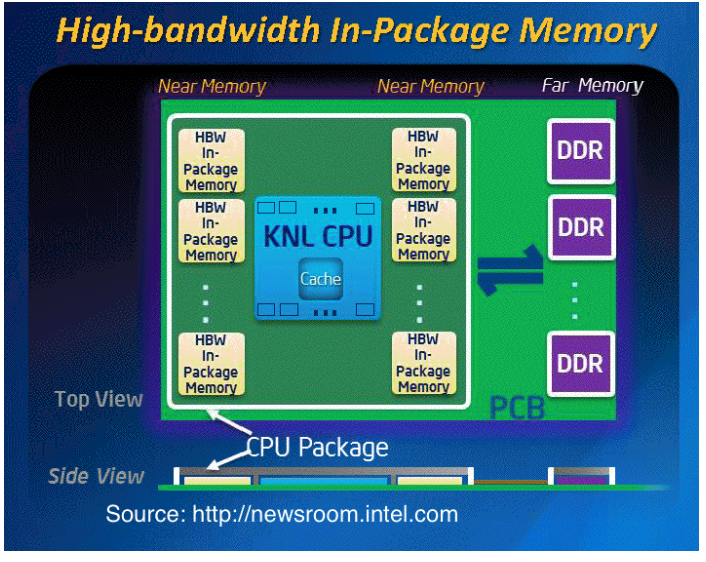
- 60+ Physical Cores Per CPU
- 240+ Virtual Cores Per CPU
- Much slower GHz
- Can do 8 Double Precision Operations per Cycle (+ multiply/add)
- < 0.3 GB of Fast Memory Per Core
  < 2 GB of Slow Memory Per Core
- Fast memory has ~ 5x DDR4 bandwidth

What is different about Cori?

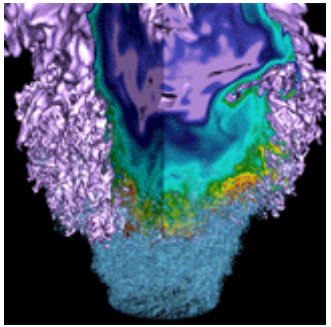
Two Big Changes:

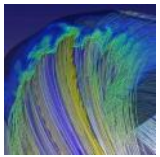
1. More on node parallelism. More cores, bigger vectors

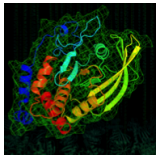
2. Small amount of very fast memory. (similar-ish amounts of traditional DDR)

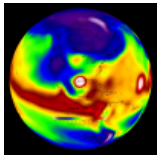


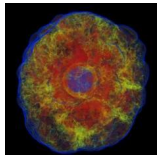
Key Concepts

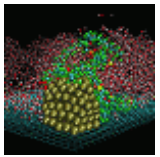












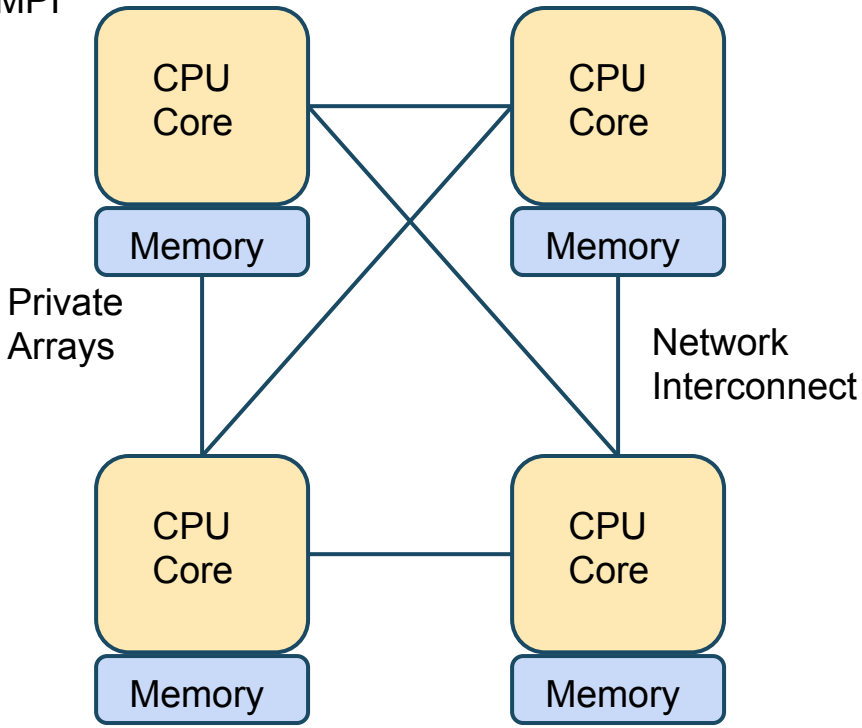
NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

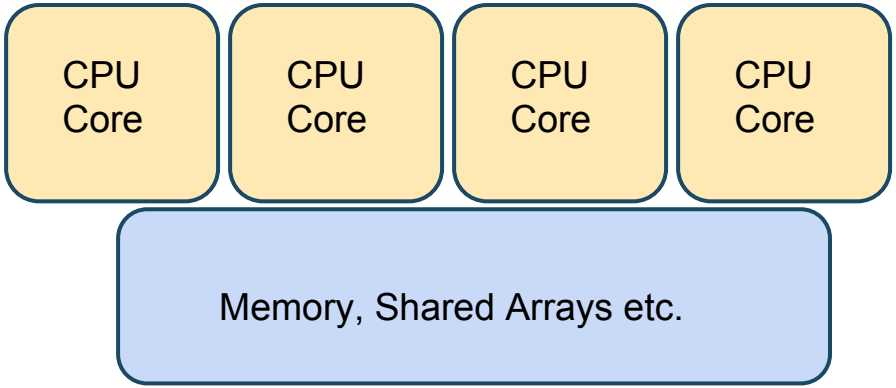
BERKELEY LAB

MPI Vs. OpenMP For Multi-Core Programming

MPI



OpenMP



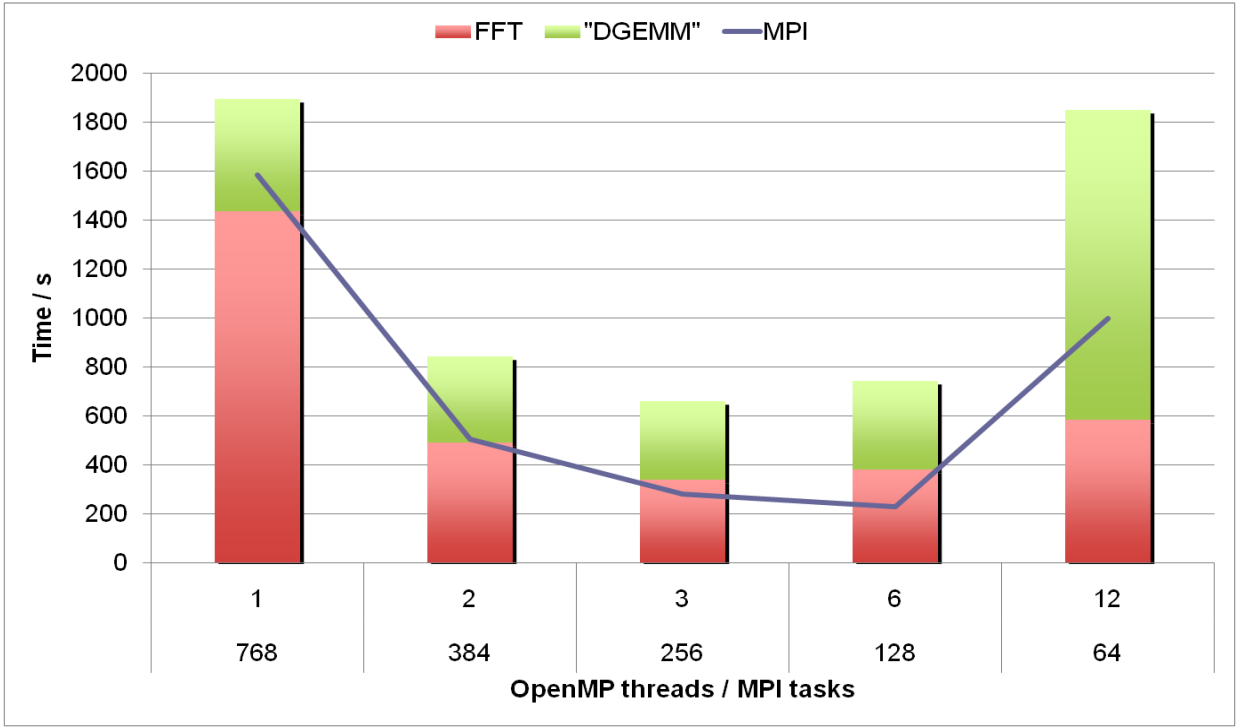
Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime

PARATEC Use Case For OpenMP

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.



Vectorization

There is a another important form of on-node parallelism

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

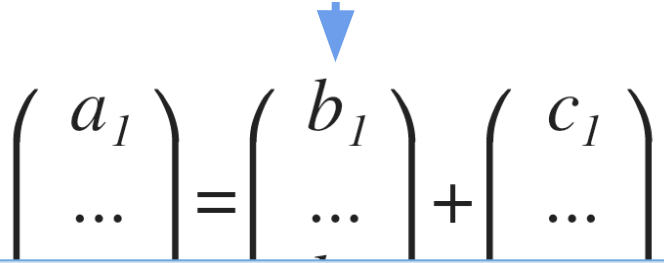
$$\begin{pmatrix} a_1 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \dots \\ b_n \end{pmatrix} + \begin{pmatrix} c_1 \\ \dots \\ c_n \end{pmatrix}$$

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

Vectorization

There is a another important form of on-node parallelism

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



Vectori s of the above l

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:

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

Task forking:

```
do i = 1, n
    if (a(i) < x) cycle
    if (a(i) > x) …
enddo
```

Memory Bandwidth

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*m + n

Memory Bandwidth

Consider the following loop: Assume, n & m are very large such that a & b don't fit into cache.

```
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,

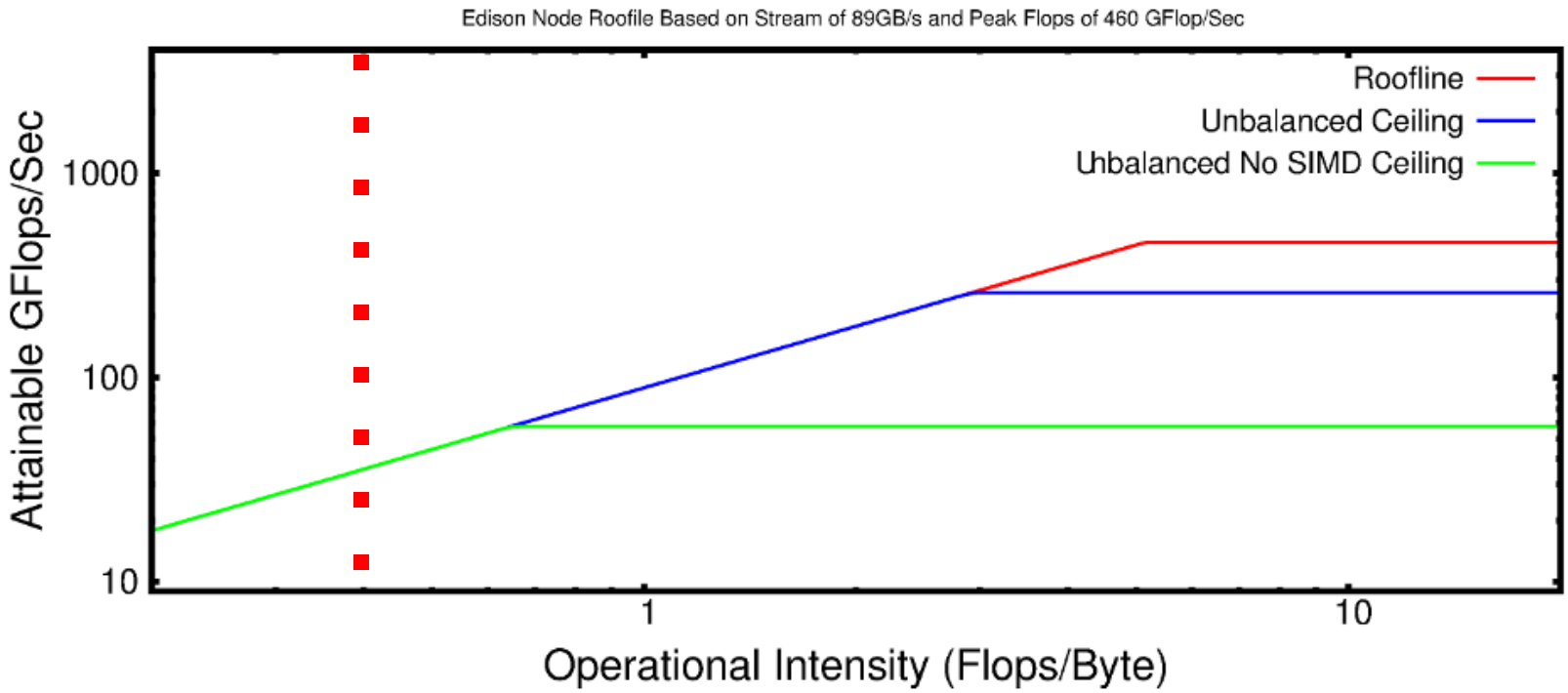
During execution, the **number of loads From DRAM** is

n*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



Improving Memory Locality

Improving Memory Locality. Reducing bandwidth required.

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



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

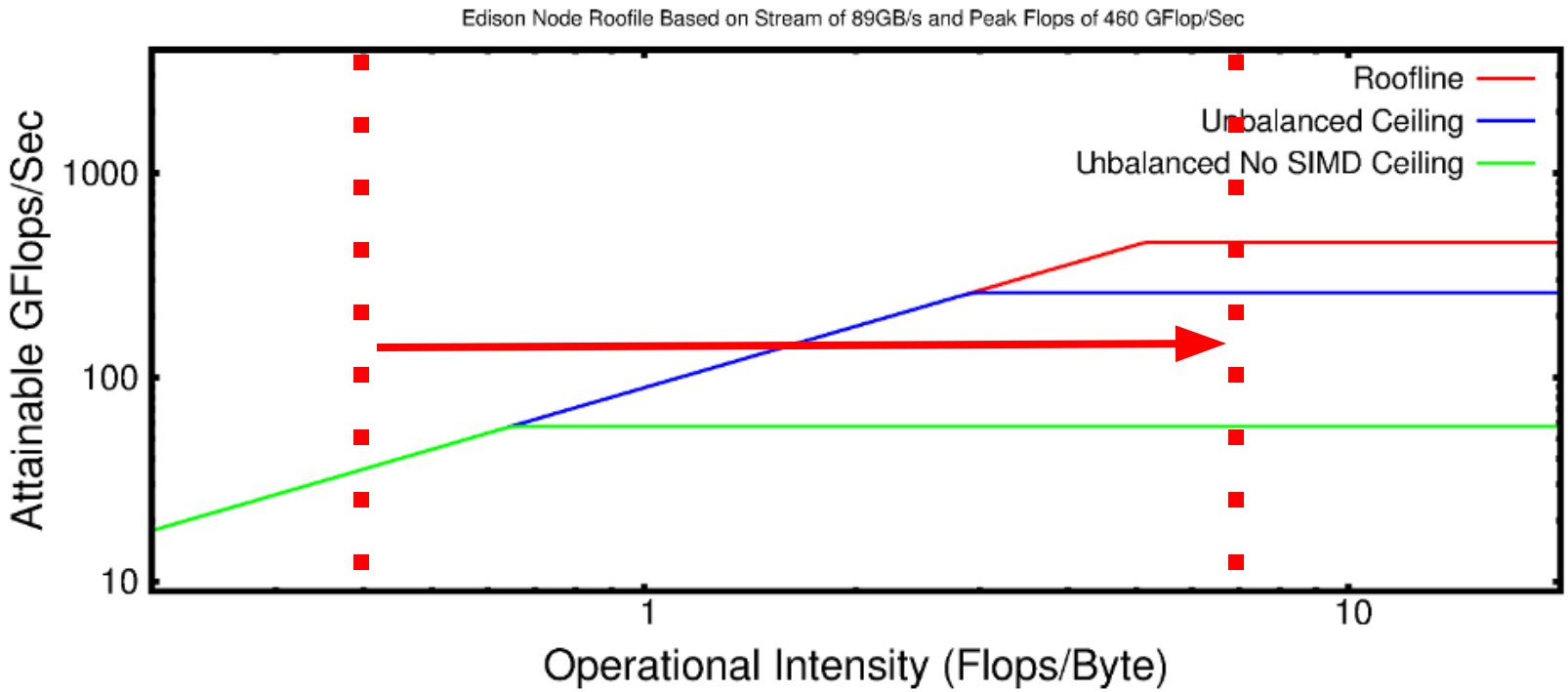
Loads From DRAM:

n*m + n

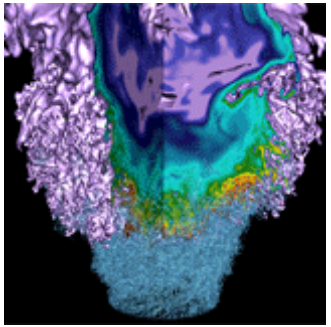
Loads From DRAM:

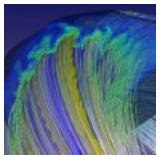
m/block * (n+block)
= n*m/block + m

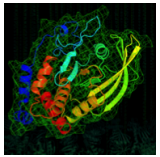
Improving Memory Locality Moves you to the Right on the Roofline

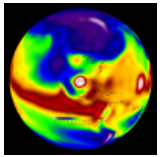


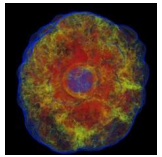
Optimization Strategy

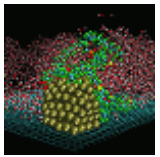












NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

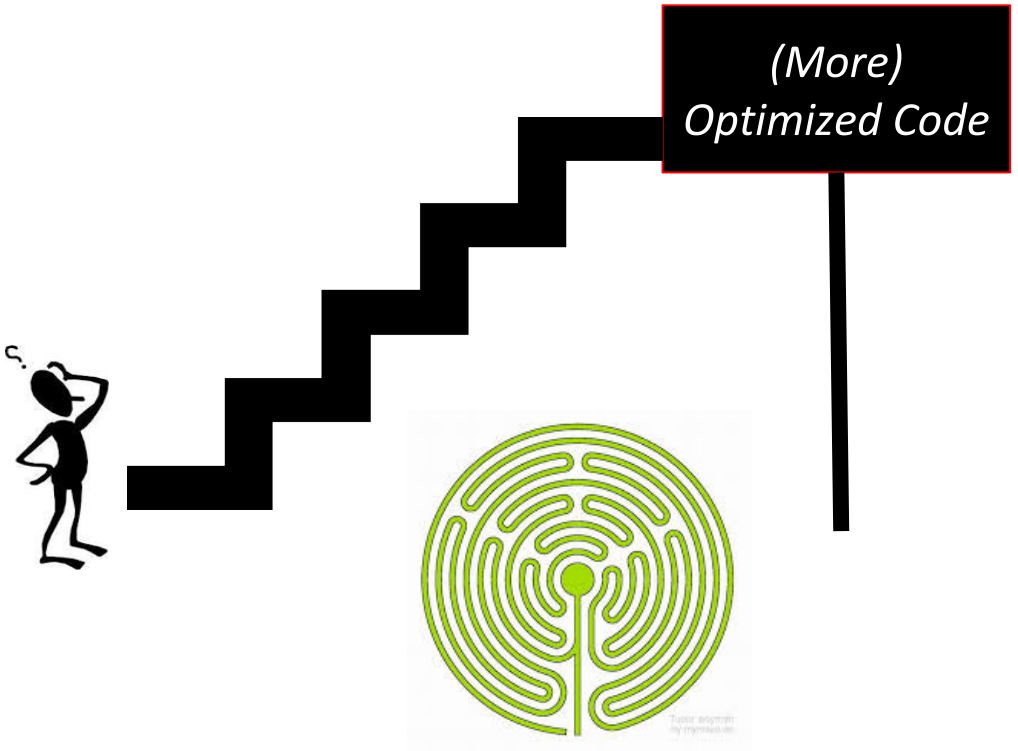
BERKELEY LAB

Optimizing Code For Cori is like:

A. **A Staircase ?**

B. **A Labyrinth ?**

C. **A Space Elevator?**



An Ant Farm!

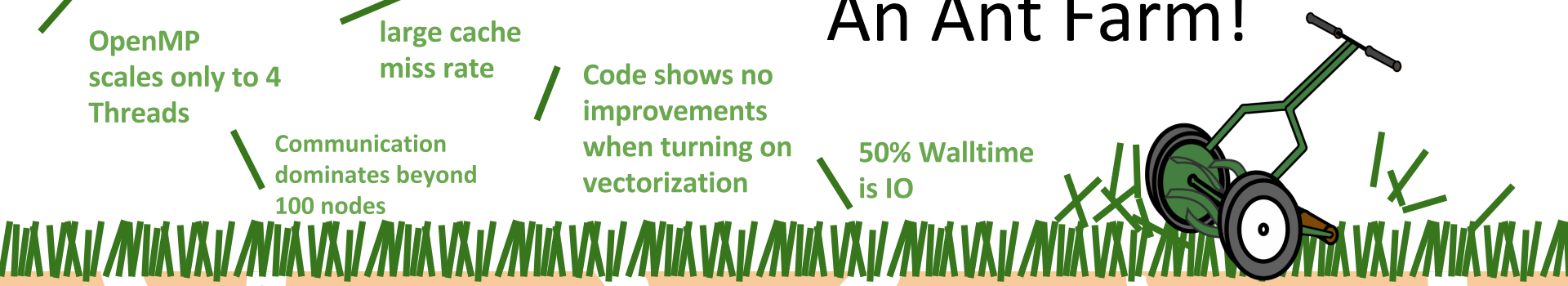
OpenMP scales only to 4 Threads

large cache miss rate

Code shows no improvements when turning on vectorization

Communication dominates beyond 100 nodes

50% Walltime is IO



Compute intensive doesn't vectorize

Memory bandwidth bound kernel

IO bottlenecks

MPI/OpenMP Scaling Issue

Can you use a library?

Increase Memory Locality

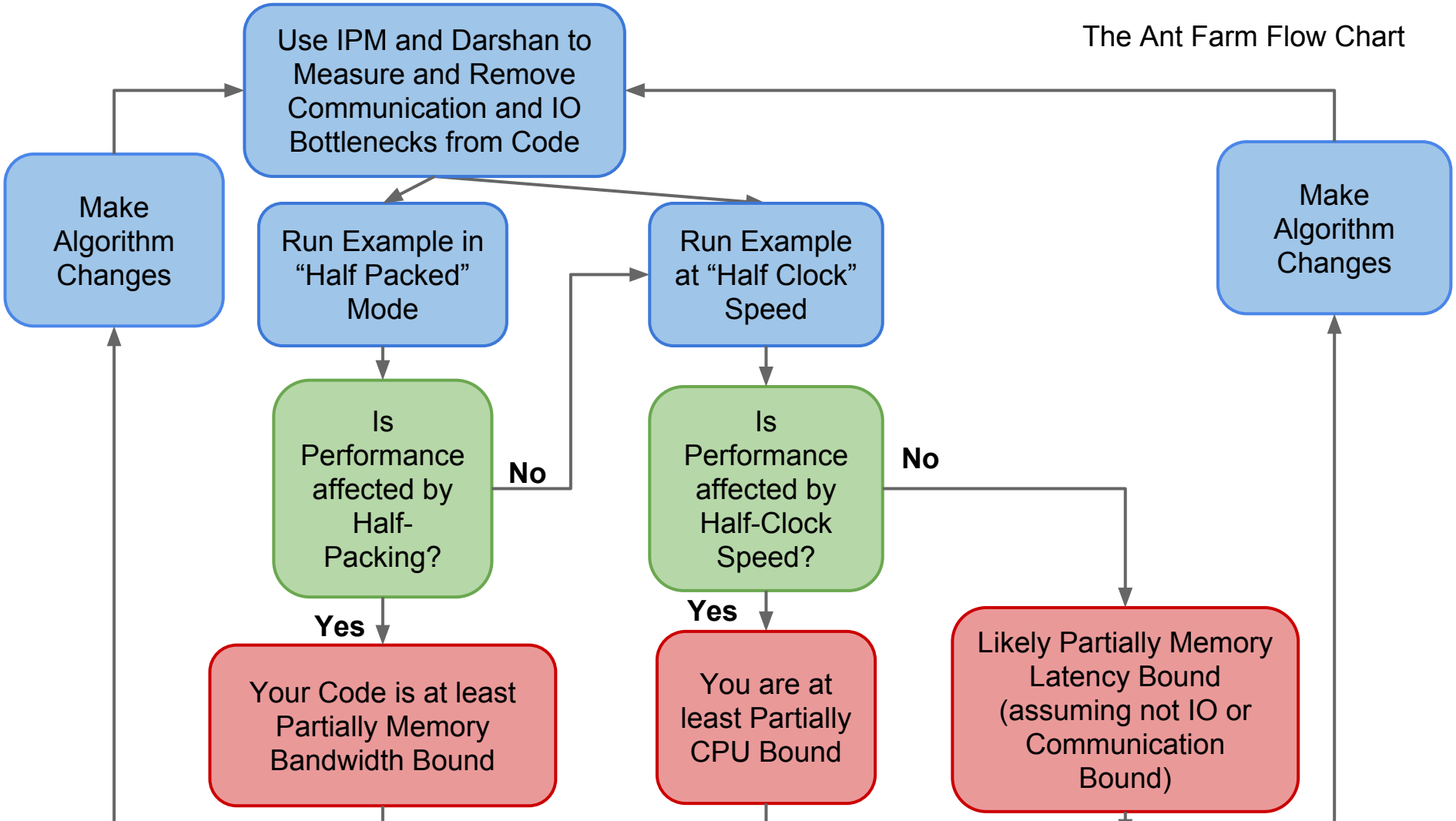
Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.

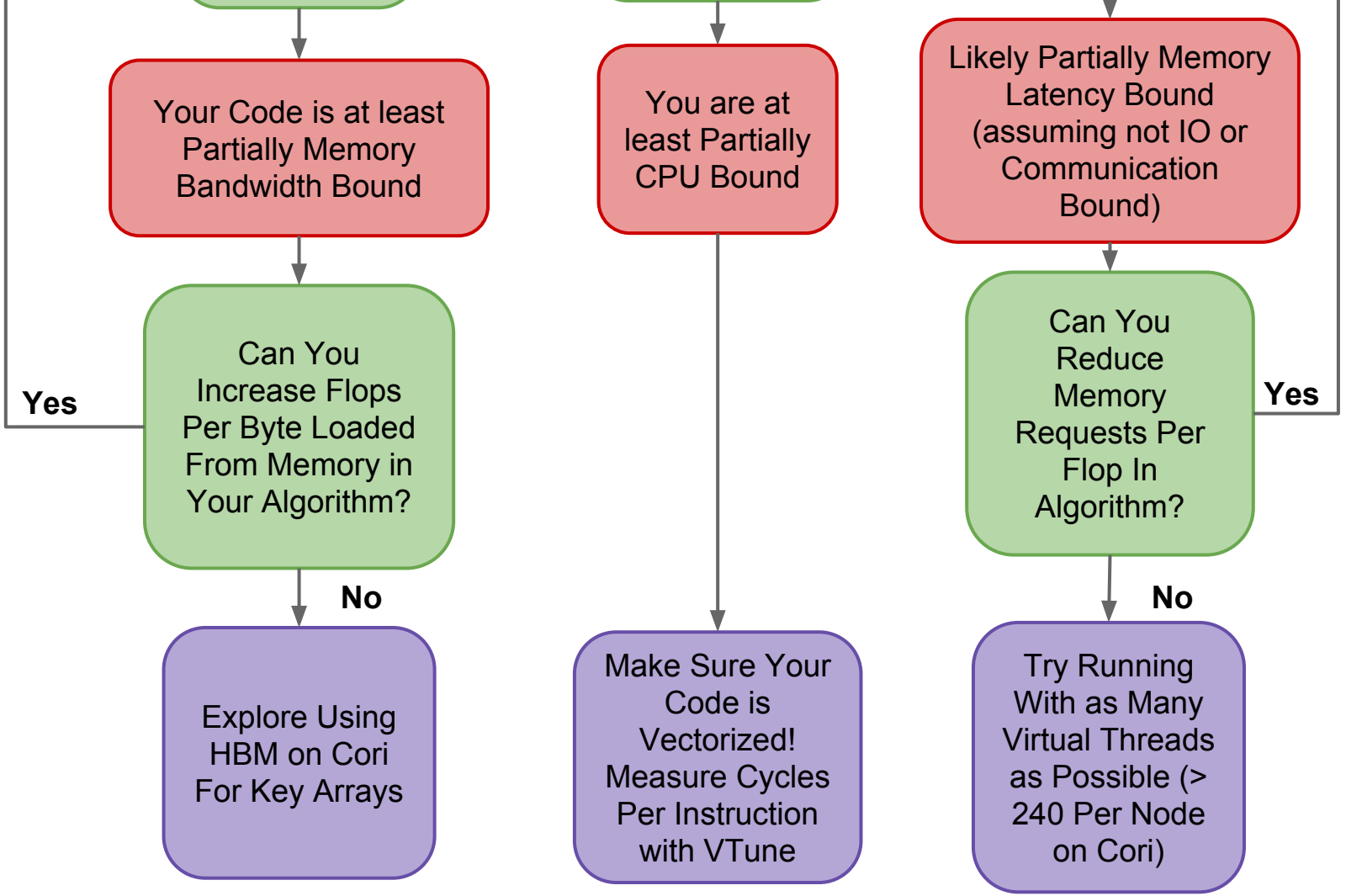
Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.

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

Utilize performant / portable libraries

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


The Ant Farm Flow Chart
Use IPM and Darshan to Measure and Remove Communication and IO Bottlenecks from Code
Make Algorithm Changes
Run Example in "Half Packed" Mode
Run Example at "Half Clock" Speed
Make Algorithm Changes
Is Performance affected by Half-Packing?
No
Yes
Is Performance affected by Half-Clock Speed?
No
Yes
Your Code is at least Partially Memory Bandwidth Bound
You are at least Partially CPU Bound
Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)


Your Code is at least Partially Memory Bandwidth Bound
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?
Yes
No
Explore Using HBM on Cori For Key Arrays
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)
Can You Reduce Memory Requests Per Flop In Algorithm?
Yes
No
Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

Use IPM to Measure Communication Time

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

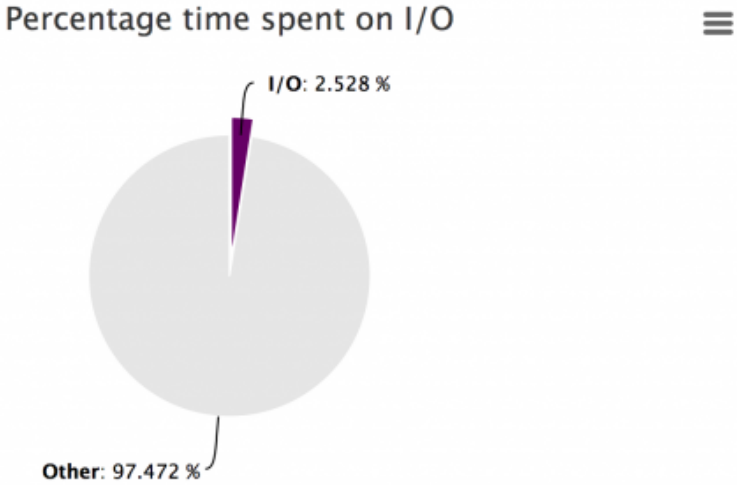
https://www.nersc.gov/users/software/debugging-and-profiling/ipm/

```
# wallclock           953.272      29.7897      29.6092      29.9752
# user                837.25       26.1641        25.71        26.92
# system               60.6        1.89375         1.52         2.59
# mpi                 264.267      8.25834      7.73025      8.70985
# %comm                            27.7234      25.8873      29.3705

#                      [time]      [calls]        <%mpi>      <%wall>
# MPI_Send            188.386       639616        71.29        19.76
# MPI_Wait            69.5032       639616        26.30         7.29
# MPI_Irecv           6.34936       639616         2.40         0.67
# MPI_Barrier       0.0177442           32         0.01         0.00
# MPI_Reduce        0.00540609           32         0.00         0.00
```

Use Darshan to Measure IO Time/Performance

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



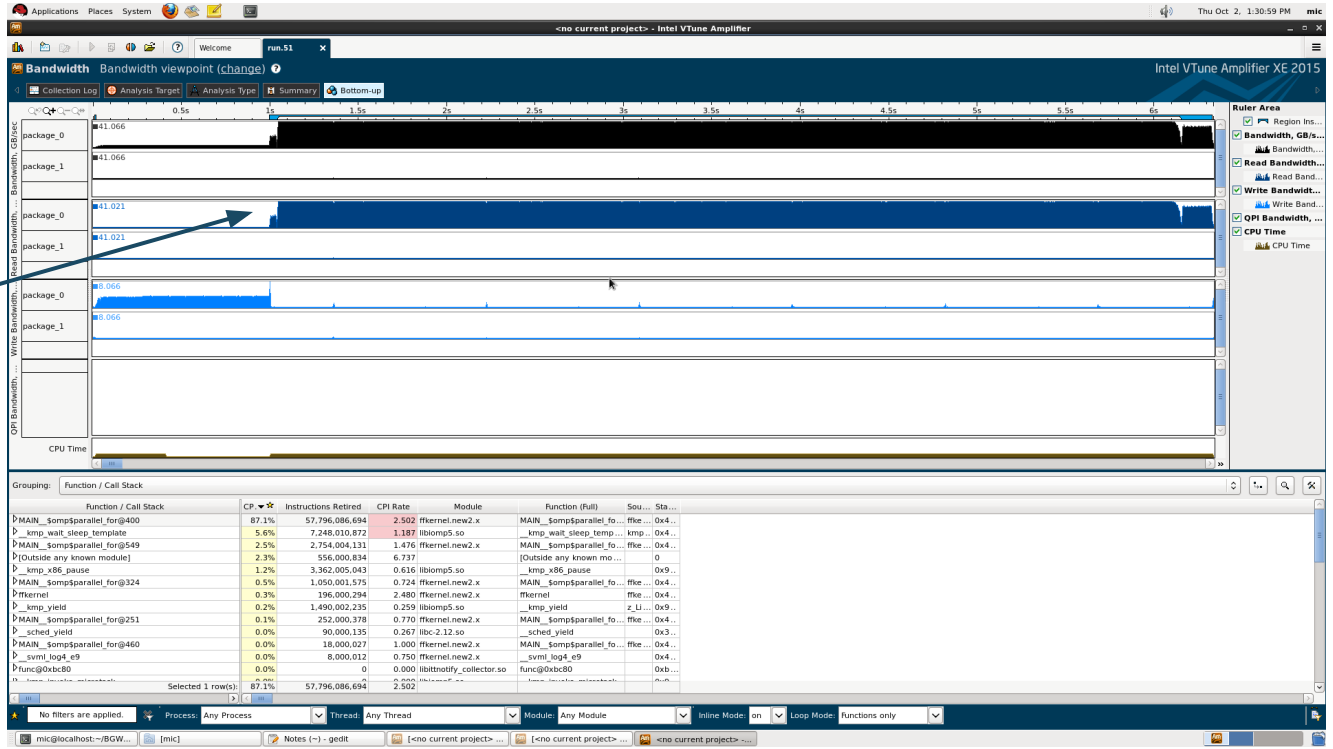
Measuring Your Memory Bandwidth Usage (VTune)

Measure memory bandwidth usage in VTune. (Next Talk)

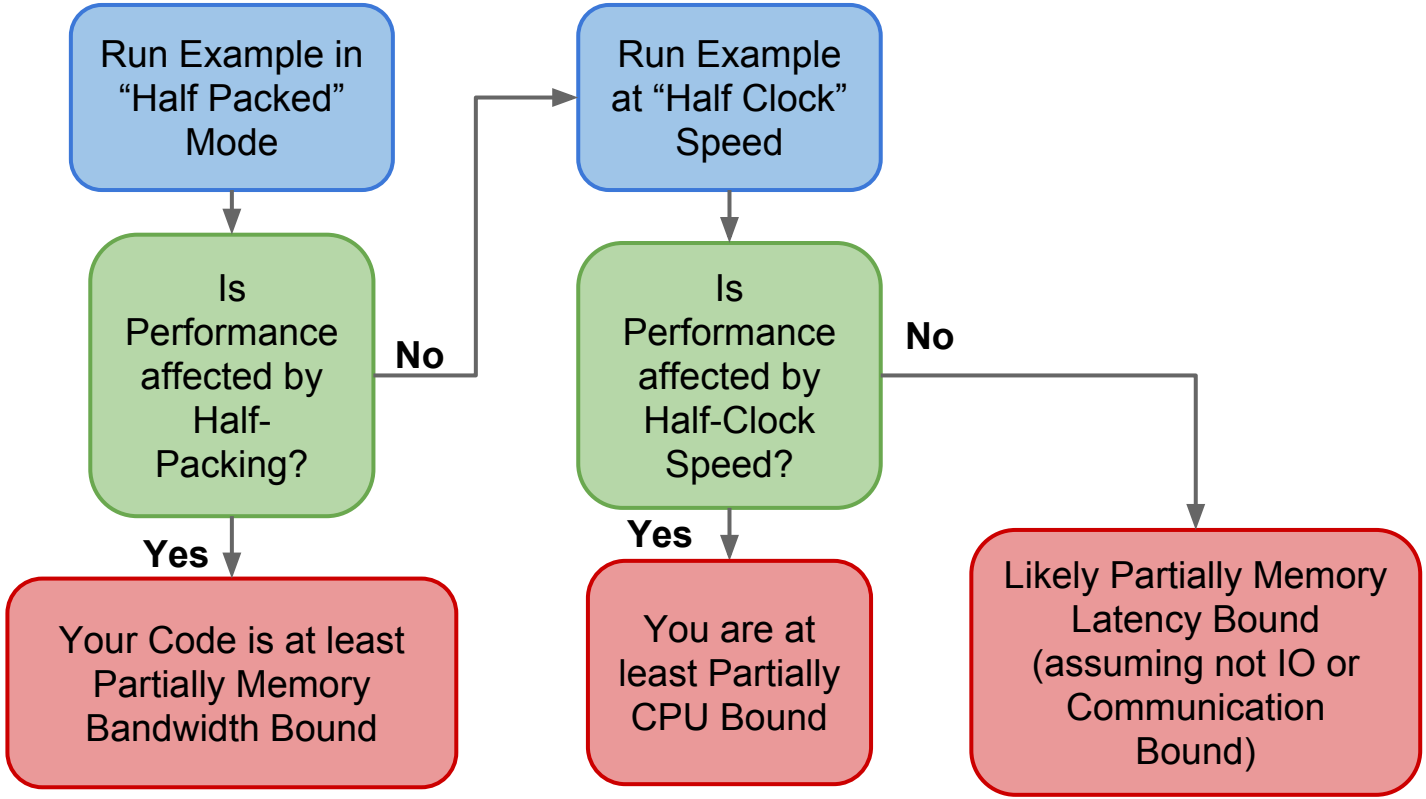
Compare to Stream GB/s.

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?



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?

Run Example at “Half Clock” Speed

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

```
aprun --p-state=2400000 ...
```

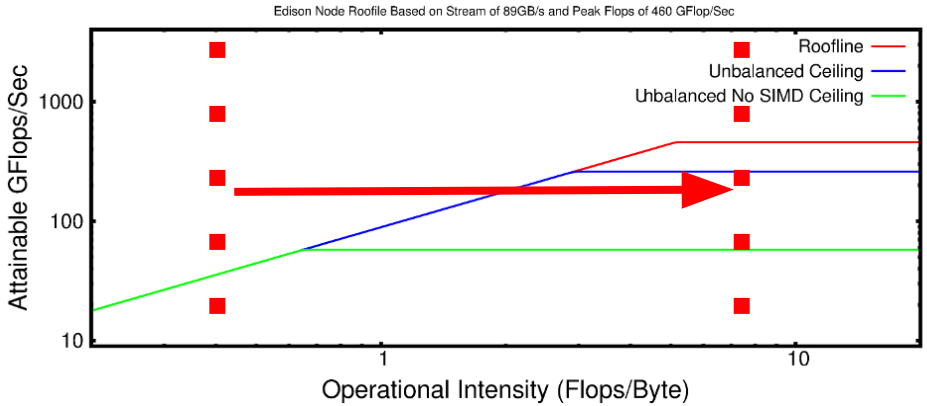
VS

```
aprun --p-state=2200000 ...
```

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

What to do?

1. Try to improve memory locality, cache reuse

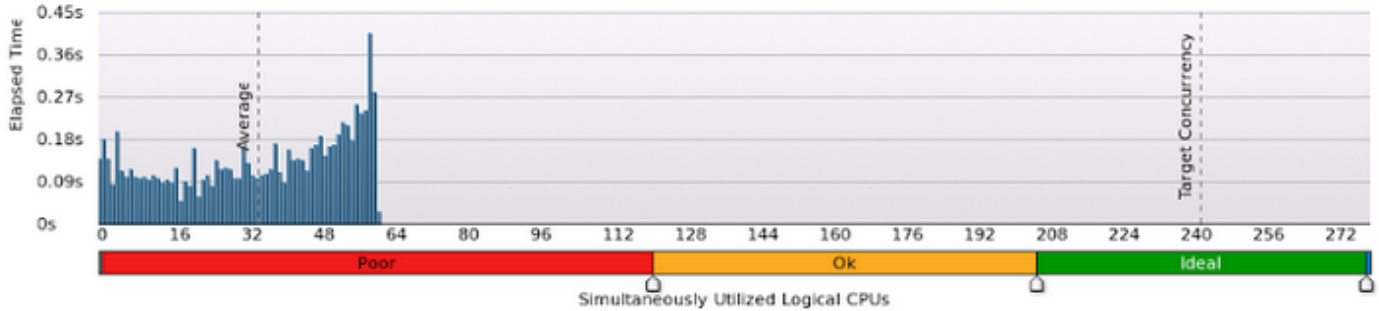


2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

   Profit by getting ~ 5x more bandwidth GB/s.

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.

   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).

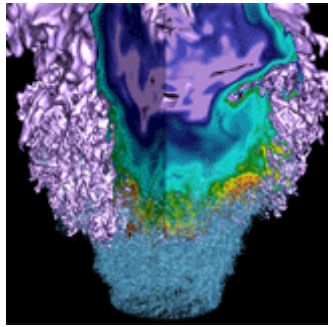
If running with hyper-threading on Edison improves performance, you *might* be latency bound:

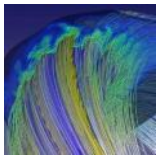
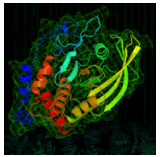
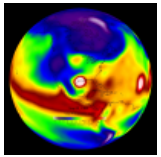
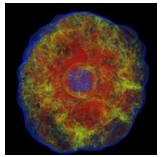
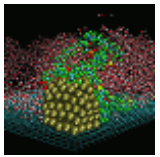
`aprun -j 2 -n 48 ….` VS `aprun -n 24 ….`

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



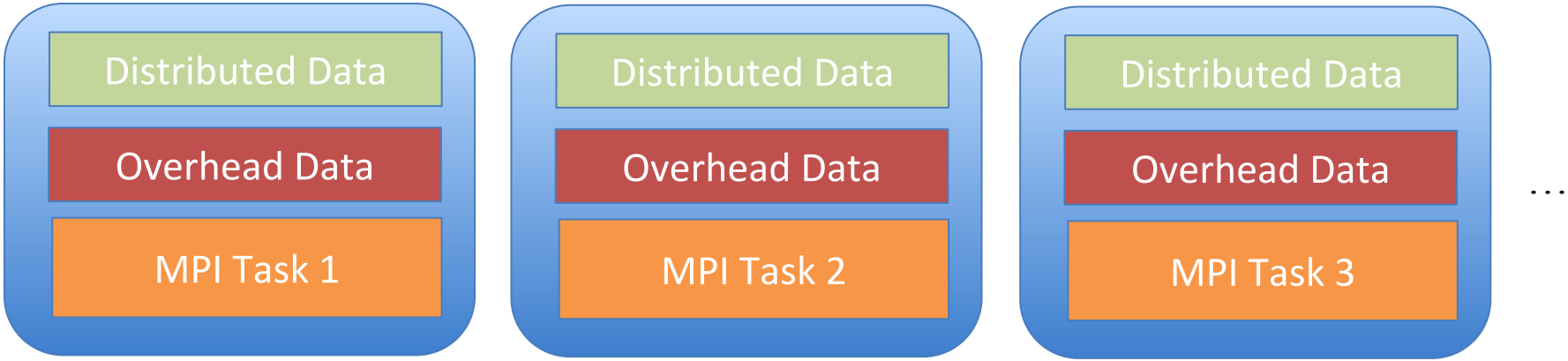
NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

BerkeleyGW Use Case

- ★ Big systems require more memory. Cost scales as N_{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**.



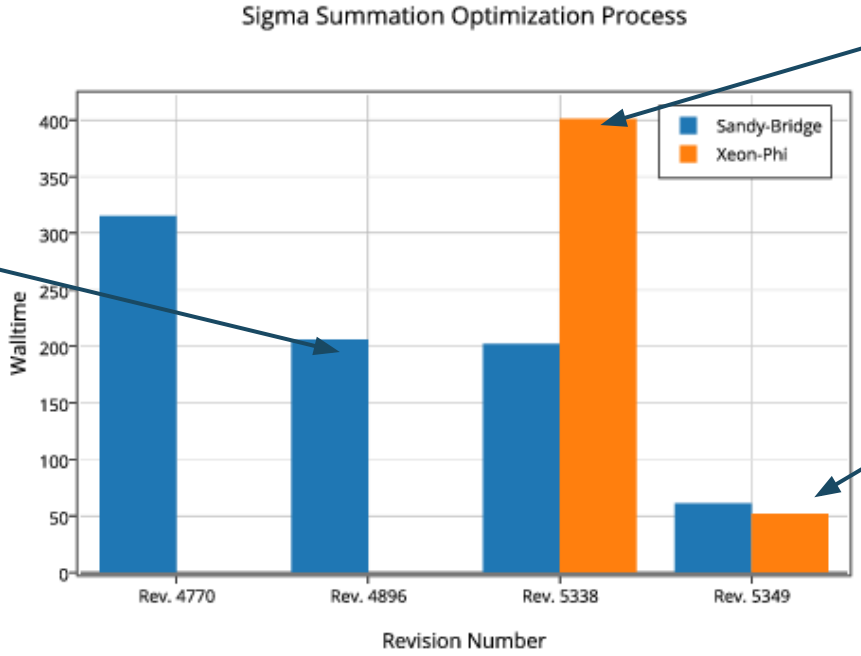
Targeting Intel Xeon Phi Many Core Architecture

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



Final Loop Structure

```
!$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
```

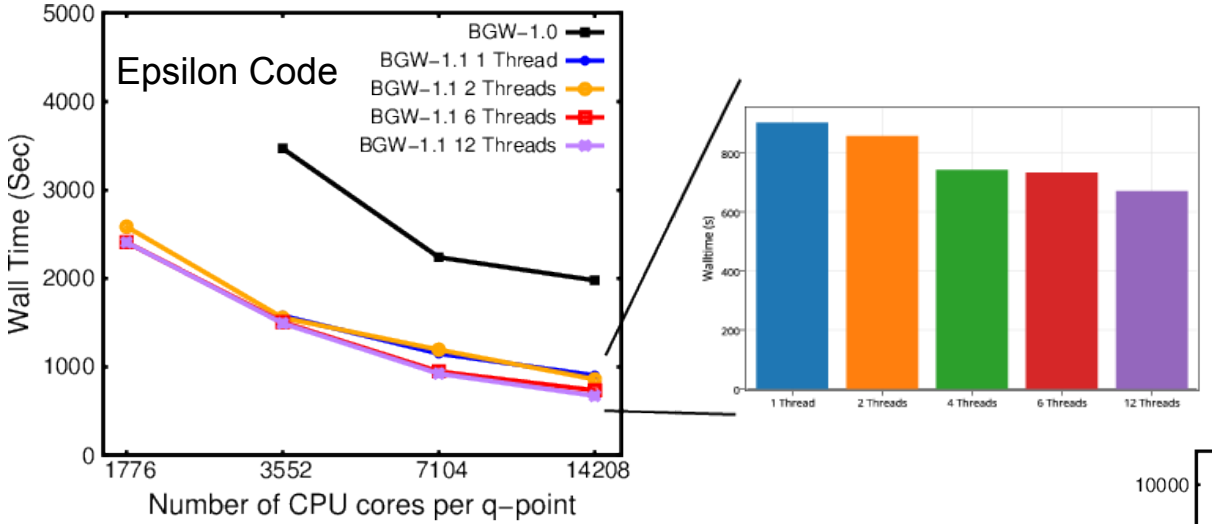
ngpown typically in 100's to 1000s. Good for many threads.

Original inner loop. Too small to vectorize!

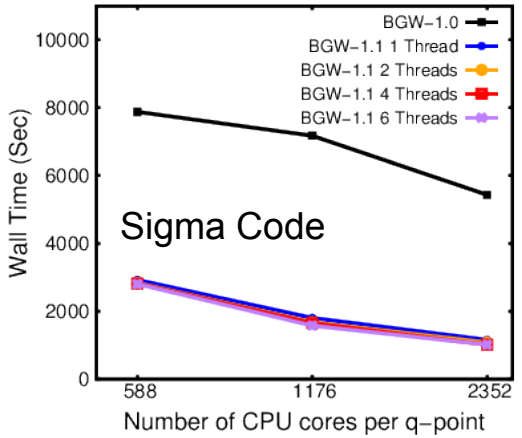
ncouls typically in 1000s - 10,000s. Good for vectorization.

Attempt to save work breaks vectorization and makes code slower.

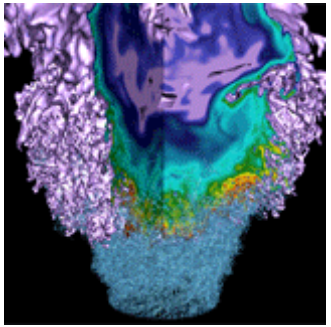
Hybrid MPI-OpenMP Scaling Improvements.

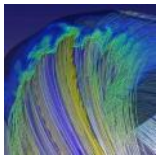


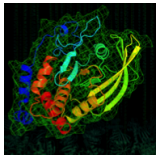
* Major Improvement between 1.0 and 1.1
* Trading MPI tasks for OpenMP threads, yields improved performance (mostly in MPI communication costs) and allows scaling to higher core counts.

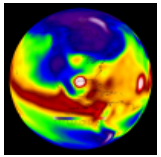


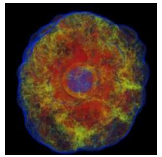
The End

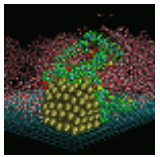












NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB