Using MPI+OpenMP for current and future architectures

- September 24th, 2018

OpenMPCon 2018

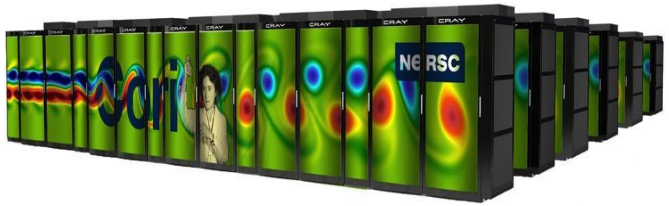
Oscar Hernandez
Yun (Helen) He
Barbara Chapman

EXASCALE COMPUTING PROJECT

National Nuclear Security Administration
U.S. DEPARTMENT OF ENERGY | Office of Science

DOE's Office of Science Computation User Facilities

- DOE is leader in open High-Performance Computing
- Provide the world's most powerful computational tools for open science
- Access is free to researchers who publish
- Boost US competitiveness
- Attract the best and brightest researchers



NERSC
Cori is 30 PF

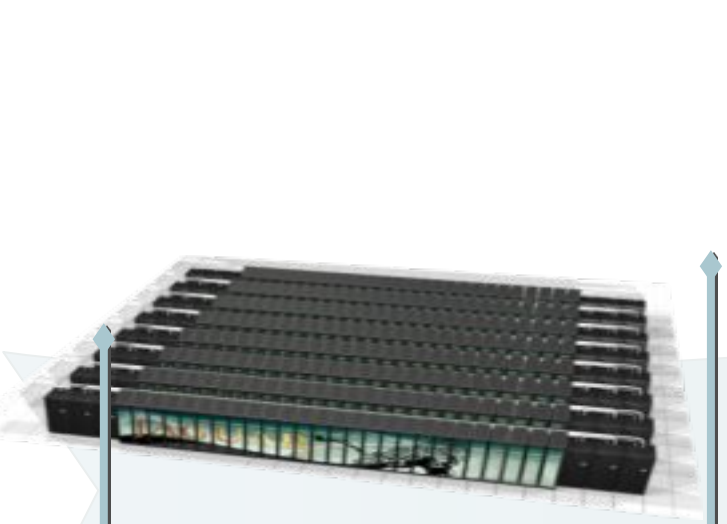
ALCF
Theta is 11.7 PF

OLCF
Titan is 27 PF

Roadmap to Exascale (ORNL)

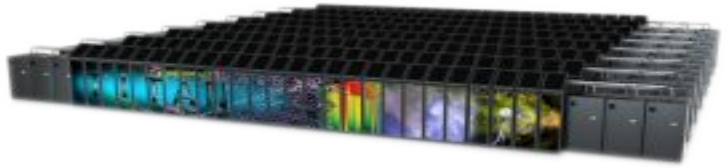
Since clock-rate scaling ended in 2003, HPC performance has been achieved through increased parallelism. Jaguar scaled to 300,000 CPU cores.

Titan and beyond deliver hierarchical parallelism with very powerful nodes. MPI plus thread level parallelism through OpenMP or OpenACC plus vectors



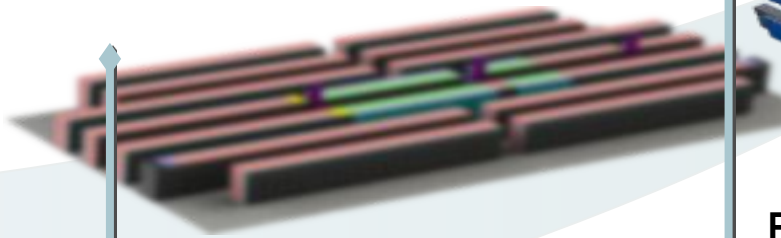
Jaguar: 2.3 PF
Multi-core CPU
7 MW

2010



Titan: 27 PF
Hybrid GPU/CPU
9 MW

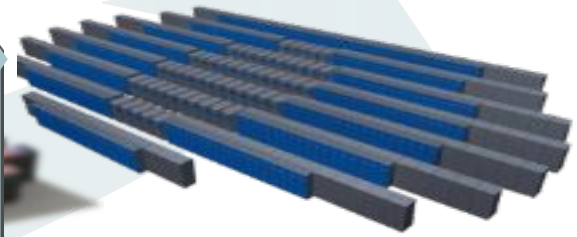
2012



Summit: 5-10x Titan
Hybrid GPU/CPU
10 MW

CORAL System

2017

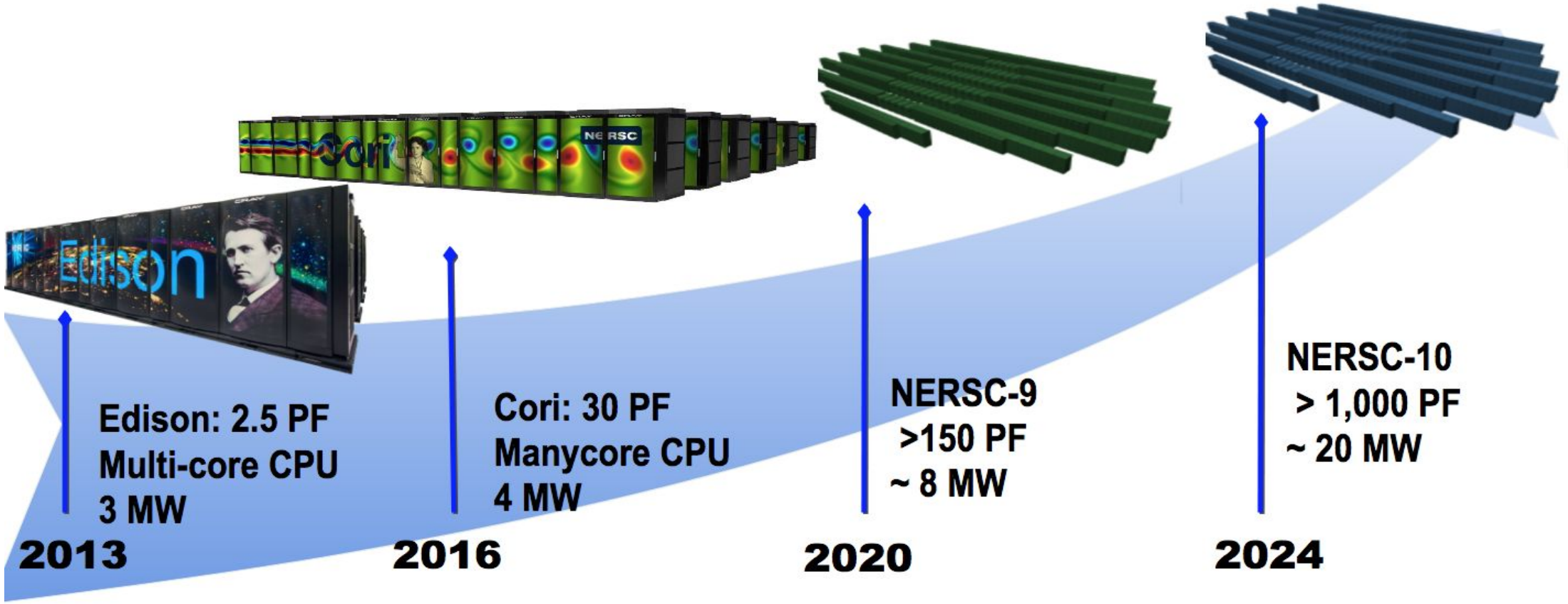


Frontier: 5-10x Summit
~40 MW

2021-3

Exascale Systems Roadmap (NERSC)

NERSC recommends MPI+OpenMP for N8 and N9 systems for programming model continuity and performance portability



Two Architecture Paths for Today and Future Leadership Systems

Power concerns for large supercomputers are driving the largest systems to either Heterogeneous or Homogeneous (manycore) architectures

Heterogeneous Systems (e.g. Titan, Summit)

- CPU(s) / GPU(s)
- Multiple CPUs and GPUs per node.
- Small number of very powerful nodes
- Expect data movement issues to be much easier than previous systems – coherent shared memory within a node
- Multiple levels of memory – on package, DDR, and non-volatile

Homogeneous (e.g. Sequoia/Mira/Theta/Cori)

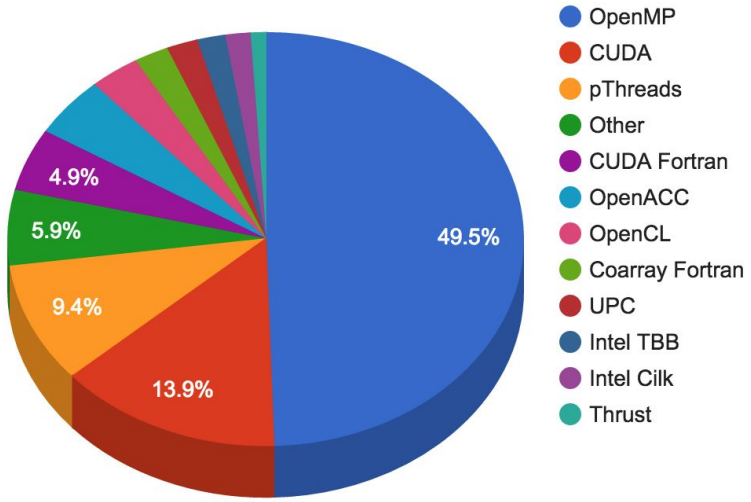
- 10's of thousands of nodes with millions of cores
- Homogeneous cores
- Multiple levels of memory – DDR, MCDRAM (e.g. cached or user-managed mode), SSD (Theta)

System Attributes	NERSC	OLCF	ALCF	NERSC	ALCF	OLCF
Name	Edison	Titan	Mira	Cori	Theta	**Summit**
System peak (PF)	2.57	27	10	Haswell: 2.81 KNL: 29.5	11.69	**200**
Peak Power (MW)	1.9	9	4.8	4.2	1.7	**13.3**
Total system memory	357 TB	710TB	768 TB	Haswell: 298.5 TB DDR4 KNL: 1.06 PB DDR4 + High Bandwidth Memory	1475 TB: 843 DDR4 + 70 MCDRAM + 562 SSD	**>2.4 PB:** **DDR4,** **HBM2,** **PB persistent, memory**
Node performance (TF)	0.461	1.452	0.204	Haswell: 1.178 KNL: 3.046	2.66	**>40**
Node Processors	Intel Ivy Bridge	AMD Opteron NVIDIA K20x	64-bit PowerPC A2	Intel Haswell Intel KNL	Intel KNL	**2 POWER9** **6 NVIDIA Volta GPUs**
System Size (nodes)	5,586 nodes	18,688 nodes	49,152	Haswell; 2,388 nodes KNL: 9,688 nodes	4,392 nodes	**~4600 nodes**
System Interconnect	Aries	Gemini	5D Torus	Aries	Aries	**Dual Rail EDR-IB**
File System	7.6 PB 168 GB/s Lustre	32 OB 1 TB/s Lustre	26 PB 300 GB/s GPFS	28 PB >700 GB/s Lustre	10 PB 744 GB/s Lustre	**120 PB** **1 TB/s** **GPFS**

Choice of Programming Models

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for shared memory or intra-node, and now supports accelerators as well (intra means within)
- Hybrid Programming (MPI+X) is when we use a solution with different programming models for inter *vs.* intra-node parallelism
- Several solutions including
  - Pure MPI
  - MPI + Shared Memory (OpenMP)
  - MPI + Accelerator programming
    - OpenMP 4.5 shared memory + offload, OpenACC, CUDA, etc
  - MPI message passing + MPI shared memory
  - PGAS: UPC/UPC++, Fortran 2008 coarrays, GA, OpenSHMEM, etc
  - Runtime tasks (Legion, HPX, HiHat (draft), etc)
  - Other hybrid based on Kokkos, Raja, SYCL, C++17 (C++20 draft)

NERSC data from 2015:
When asked: If you use MPI + X, what is X ?



Why Hybrid MPI + OpenMP?

- Homogeneous and Heterogeneous systems have large core counts per node
  - Cori: Xeon Phi (KNL) 68 cores, 4 hardware threads per core. Total of 272 threads per node
  - Summit: Total 176 (SMT4) Power9 threads + 6 Volta GPUs per node
- Application may run with MPI everywhere, but possibly not good performance
  - Needs hybrid programming to manage threading, improve SIMD, accelerator programming
- Many applications will not fit into the node memory using Pure MPI (e.g. per core) because of the memory overhead for each MPI task
- Hybrid MPI/OpenMP is a recommended programming model to achieve scaling capability and code portability, new trend
- Incremental parallelism with OpenMP for cores and accelerators
- Some applications have two levels of parallelism naturally
- Avoids extra communication overhead within the node
- Adds fine granularity (larger message sizes) and allows increased dynamic load balancing across MPI tasks

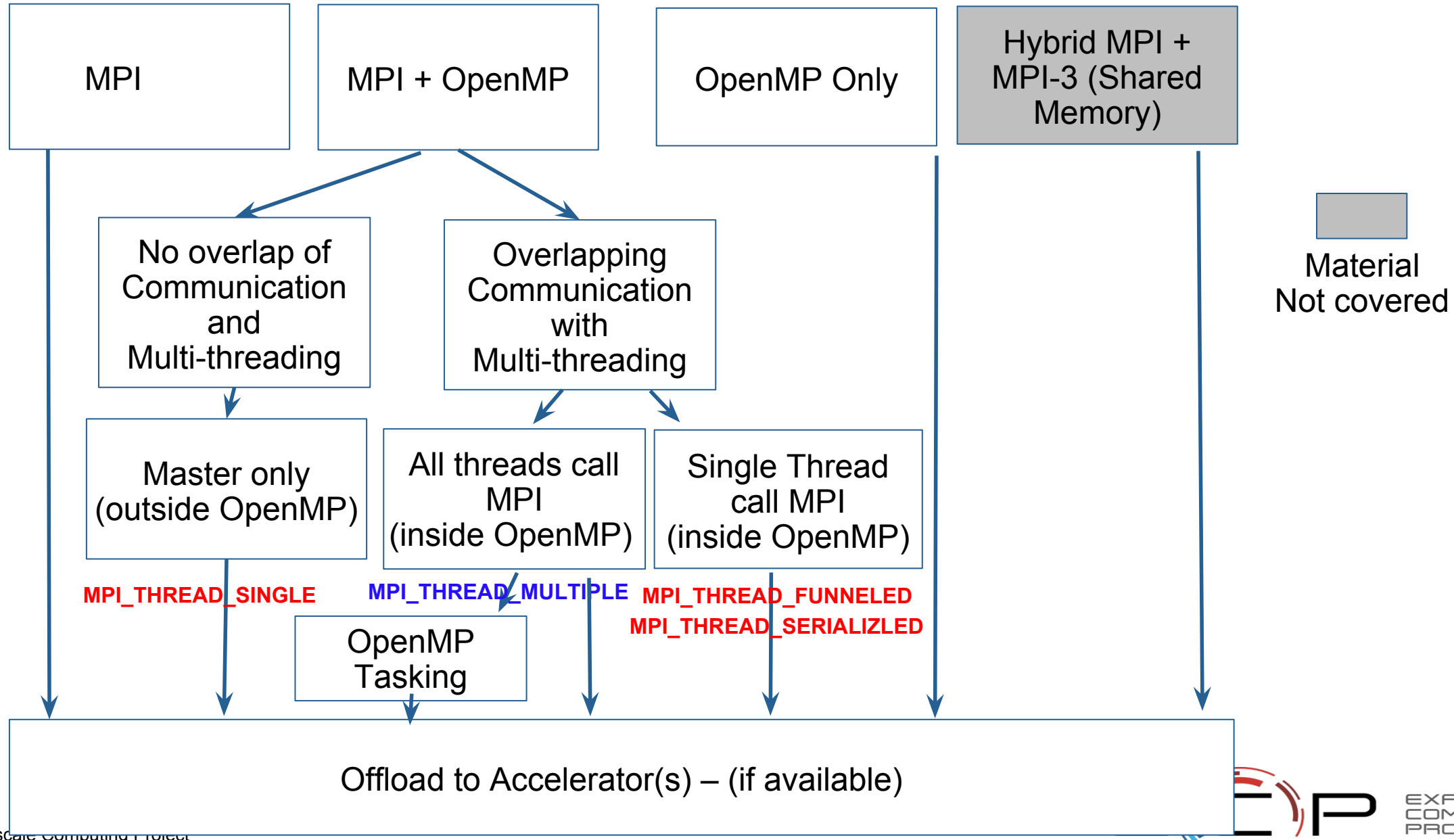
Example of Hybrid MPI/OpenMP

```
Program hybrid
 call MPI_INIT_THREAD (required, provided, ierr)
 call MPI_COMM_RANK (...)
 call MPI_COMM_SIZE (...)
  ...  some computation and MPI communication
  call OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL DO PRIVATE(i)
 !$OMP&                    SHARED(n)
    do i=1,n
        ... computation
    enddo
  !$OMP END PARALLEL DO
  ...  some computation and MPI communication
 call MPI_FINALIZE (ierr)
 end
```

Supported Levels of Thread Safety

- Defined by MPI standard in the form of **commitments** a multithreaded **application** makes to the MPI implementation. Not specific to hybrid MPI/OpenMP.
- Use MPI_INIT_THREAD (required, provided, ierr), as an alternative to MPI_INIT (ierr)
  - IN: “required”, desired level of thread support (integer)
  - OUT: “provided”, provided level of thread support (integer)
  - Returned “provided” maybe lower than “required”
- Thread support levels:
  - MPI_THREAD_SINGLE: Only one thread will execute
  - MPI_THREAD_FUNNELED: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are ’’funneled" to master thread)
  - MPI_THREAD_SERIALIZED: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are ’’serialized")
  - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions

Hybrid MPI+OpenMP Programming Styles



Overlap MPI Communication and OpenMP Multi-threading

- Is a good strategy for improving performance
  - Use MPI inside parallel region with thread-safe MPI
- Need at least MPI_THREAD_FUNNELED
- Many “easy” hybrid programs only need MPI_THREAD_FUNNELED
  - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks
  - While this single master is making MPI calls, other threads are computing
- Must be able to separate codes that can run before or after ghost zone or halo info is received. Can be very hard conceptually
- May lose compiler optimizations such as vectorization

```
!$OMP PARALLEL
    if (my_thread_rank < 1) then
        call MPI_xxx(...)
    else
        do some computation
    endif
!$OMP END PARALLEL
```

Overlap MPI Communication and OpenMP Taskiing

- Is a good strategy for improving performance with hosts or accelerators
  - Use MPI inside task region with thread-safe MPI
- Need at least MPI_THREAD_MULTIPLE
- Use task dependencies to orchestrate communication, computation on the host or device
- Approach can be used to hide latencies of MPI, target regions and data movement to/from devices
- May be harder to understand the performance or where time is spent
- Can be used to pipeline
- Needs to spawn multiple threads

On Host:

```
#pragma omp parallel single num_threads(N)
{
#pragma omp task depend(..)
      do some computation on host
#pragma omp task depend(...)
      call MPI_xxx(...)
#pragma omp taskwait
 }
```

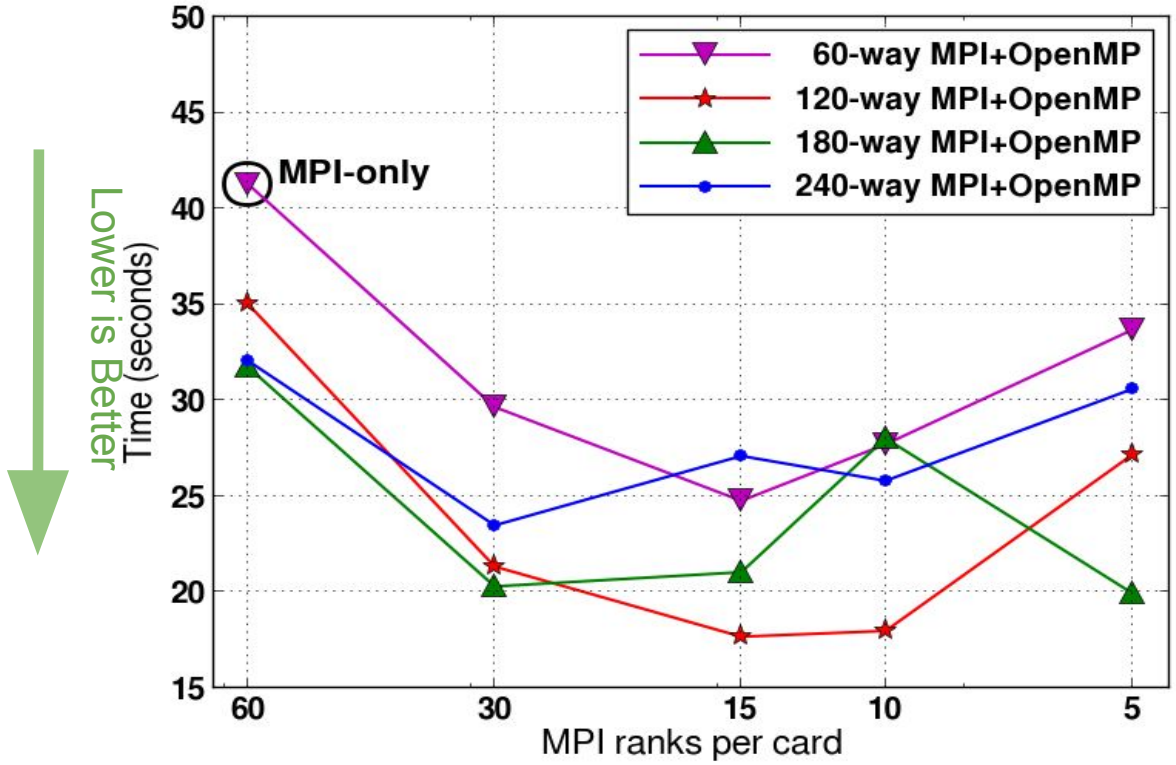
On Device:

```
#pragma omp target nowait depend(..)
      do some computation on device
#pragma omp target update from(..) nowait depend(...)
      update results on host
#pragma omp task depend(...)
      call MPI_xxx(...)
#pragma omp target update to(..) depend(..)
      update results on device
```

Best Practices for Hybrid MPI/OpenMP

- Use profiling tools to find hotspots. Add OpenMP and check correctness incrementally.
- Choose between OpenMP fine grain or coarse grain parallelism implementation.
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme
- Decide whether to overlap MPI communication with thread computation.
- Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth.
- Be aware of NUMA domains. Test different process and thread affinity options.
- Leave some cores idle on purpose, for memory capacity or bandwidth capacity.

MPI *vs.* OpenMP Scaling Analysis for Optimal Configuration



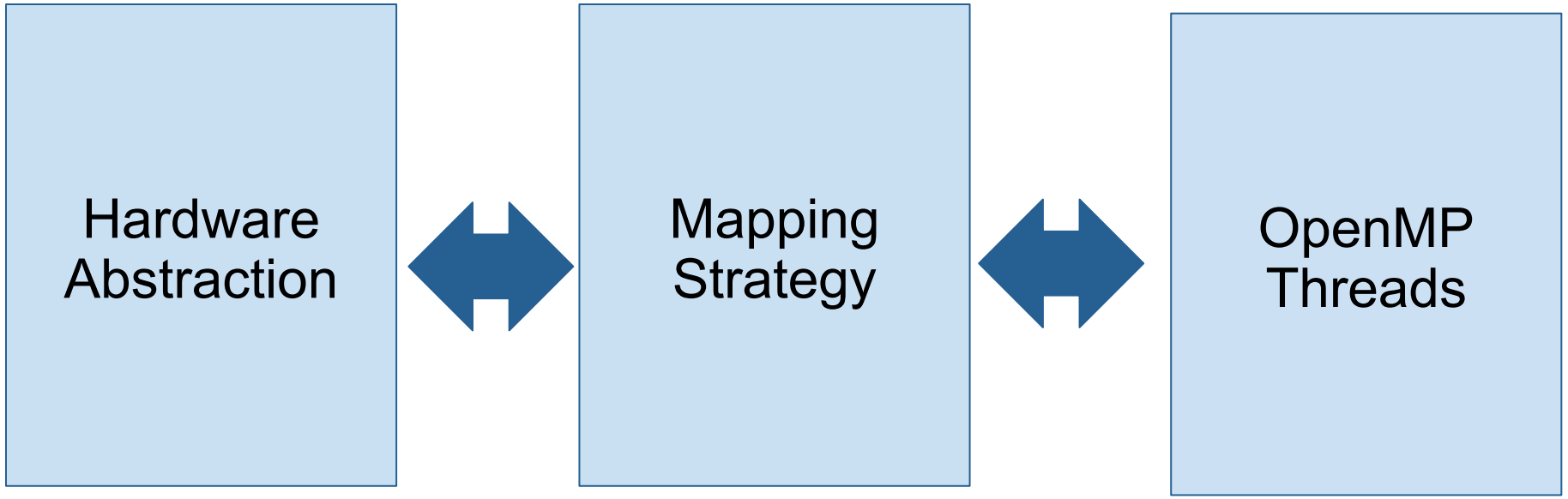
Courtesy of Chris Daley, NERSC

- Each line represents multiple runs using fixed total number of cores = #MPI tasks x #OpenMP threads/task
- Scaling may depend on the kernel algorithms and problem sizes
- In this test case, 15 MPI tasks with 8 OpenMP threads per task is optimal

Find the sweet spot for hybrid MPI/OpenMP

OpenMP Thread Affinity

- Three main concepts:



OMP_PLACES
Environment Variable
(e.g. threads, cores, sockets)

OMP_PROC_BIND
Environment Variable
Or
proc_bind() clause
of parallel region
(e.g. close, spread, master)

OMP_NUM_THREADS
Environment Variable
Or
num_threads() clause
of parallel region

Considerations for OMP_PROC_BIND Choices

- Selecting the “right” binding is dependent on the architecture topology but also on the application characteristics
- Putting threads apart (e.g. different sockets): spread
  - Can help to improve aggregated memory bandwidth
  - Combine the cache sizes across cores
  - May increase the overhead of synchronization across far apart threads
  - Aggregates memory bandwidth to/from accelerator(s)
- Putting threads near (e,g. hardware threads or cores sharing caches): master, close
  - Good for synchronization and data reuse
  - May decrease total memory bandwidth

OMP_PROC_BIND Choices for STREAM

OMP_NUM_THREADS=32
OMP_PLACES=threads

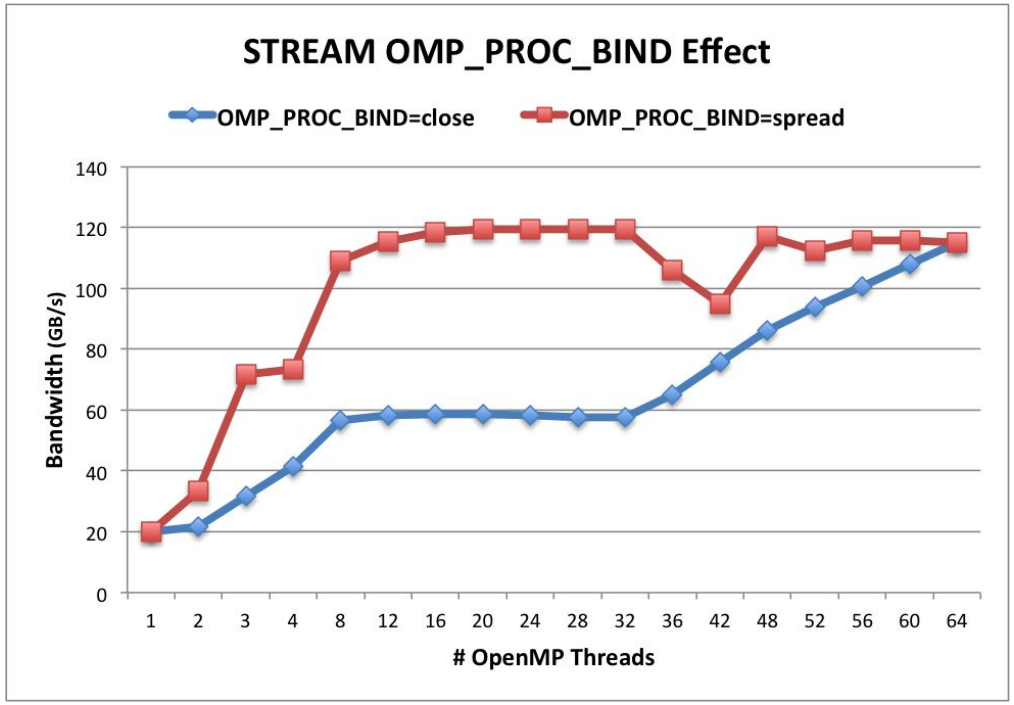
OMP_PROC_BIND=close

Threads 0 to 31 bind to CPUs 0,32,1,33,2,34,…15,47. All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread

Threads 0 to 31 bind to CPUs 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.

Blue: OMP_PROC_BIND=close
Red: OMP_PROC_BIND=spread
Both with First Touch



Memory Affinity: “First Touch” Memory

Step 1.1 Initialization
by master thread only

```
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 1.2 Initialization
by all threads

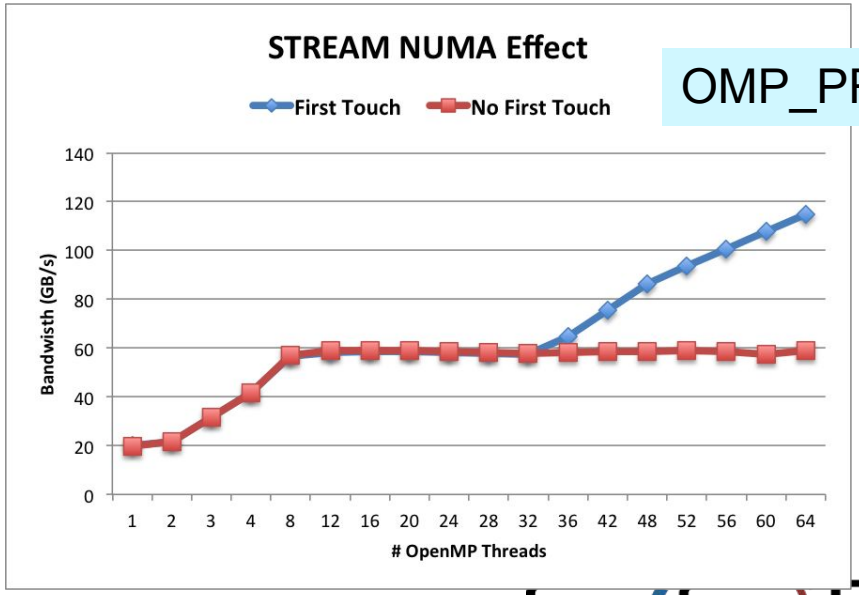
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 2 Compute

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

Memory affinity is not defined when memory was allocated, instead it will be defined at initialization.
Memory will be local to the thread which initializes it. This is called **first touch** policy.

Red: step 1.1 + step 2. No First Touch
Blue: step 1.2 + step 2. First Touch



OMP_PROC_BIND=close

“Perfect Touch” is Hard

- Hard to do “perfect touch” for real applications
- General recommendation is to use number of threads equal to or fewer than number of CPUs per NUMA domain
- Previous example: 16 cores (32 CPUs) per NUMA domain
  Sample run options:
  - 2 MPI tasks, 1 MPI task per NUMA domain: 32 OpenMP threads (use hyperthreads) or 16 OpenMP threads (no hyperthreads) per MPI task
  - 4 MPI tasks, 2 MPI tasks per NUMA domain: 16 OpenMP threads (use hyperthreads) or 8 OpenMP threads (no hyperthreads) per MPI task
  - 8 MPI tasks, and so on ...

Process and Thread Affinity in Nested OpenMP

- A combination of OpenMP environment variables and runtime flags are needed for different compilers and different batch schedulers on different systems

```
#pragma omp parallel proc_bind(spread)
    #pragma omp parallel proc_bind(close)
```

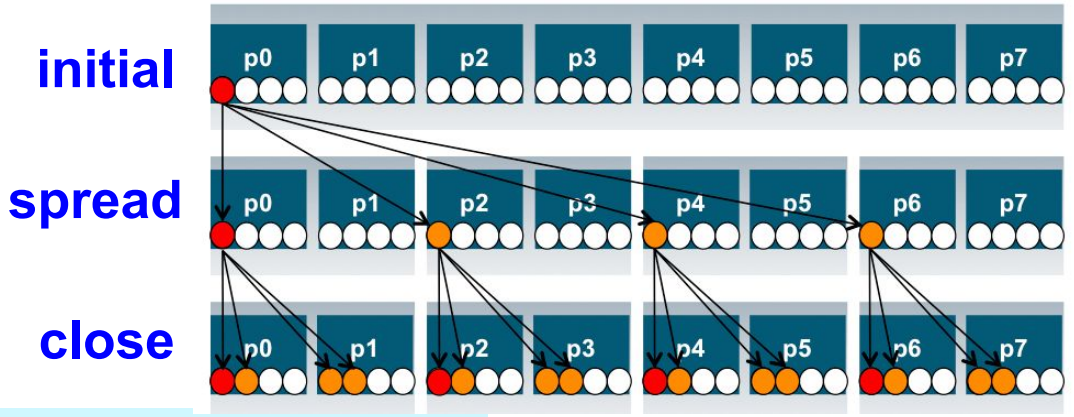


Illustration of a system with:
2 sockets, 4 cores per socket,
4 hyper-threads per core

Example: Use Intel compiler with SLURM on Cori Haswell:

```
export OMP_NESTED=true
export OMP_MAX_ACTIVE_LEVELS=2
export OMP_NUM_THREADS=4,4
export OMP_PROC_BIND=spread,close
export OMP_PLACES=threads
srun -n 4 -c 16 --cpu_bind=cores ./nested.intel.cori
```

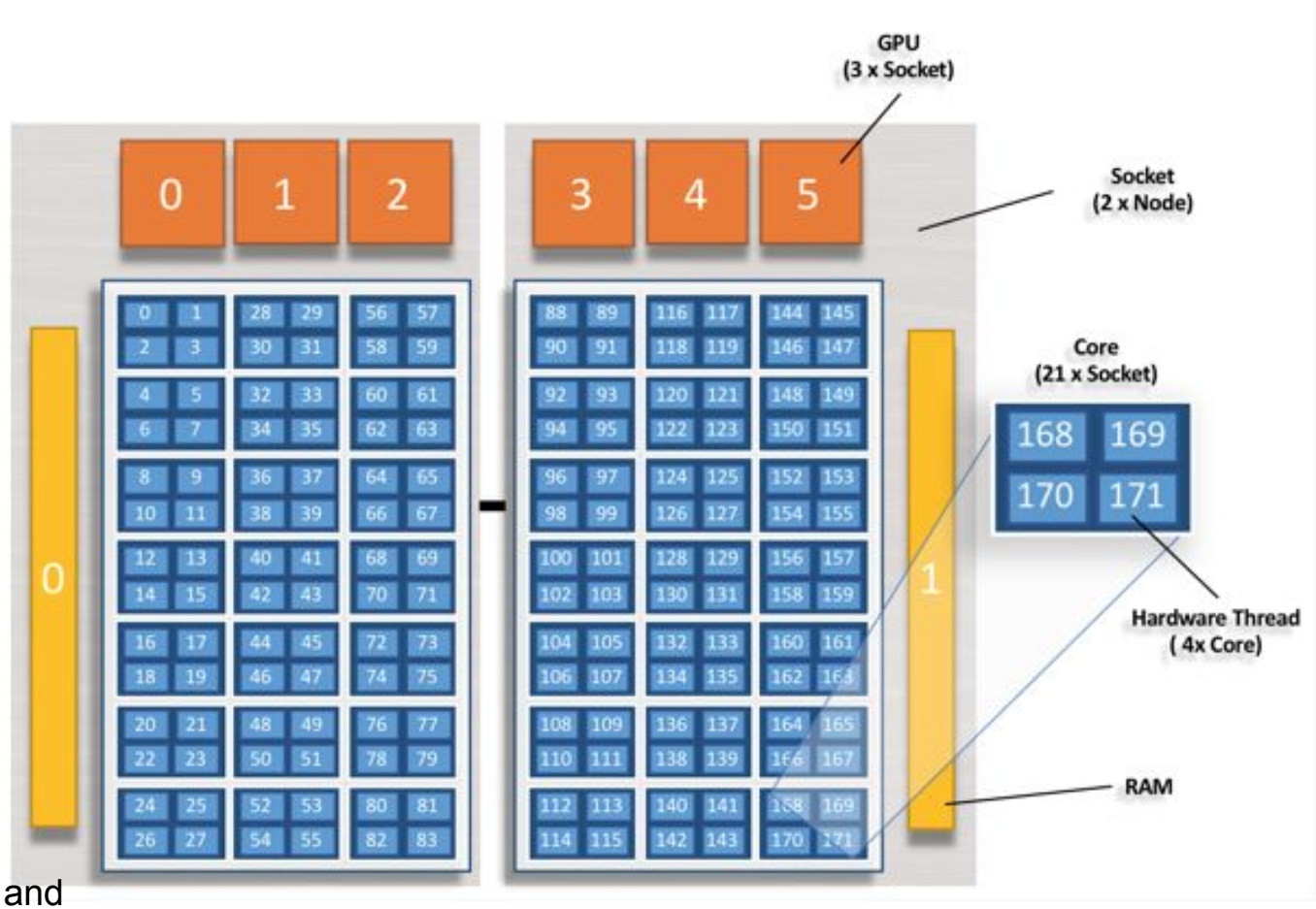
- Use num_threads clause in source codes to set threads for nested regions
- For most other non-nested regions, use OMP_NUM_THREADS environment variable for simplicity and flexibility

MPI+OpenMP: From Titan To Summit

Feature	Summit	Titan
Application Performance	5-10x Titan	Baseline
Number of Nodes	~3,400	18,688
Node performance	> 40 TF	1.4 TF
Memory per Node	>512 GB (HBM + DDR4)	38GB (GDDR5+DDR3)
NVRAM per Node	800 GB	0
Node Interconnect	NVLink (5-12x PCIe 3)	PCIe 2
System Interconnect (node injection bandwidth)	Dual Rail EDR-IB (23 GB/s)	Gemini (6.4 GB/s)
Interconnect Topology	Non-blocking Fat Tree	3D Torus
Processors	2 IBM POWER9 6 NVIDIA Volta™	AMD Opteron™ NVIDIA Kepler™
File System	120 PB, 1 TB/s, GPFS™	32 PB, 1 TB/s, Lustre®
Peak power consumption	10 MW	9 MW

OpenMP MPI+OpenMP affinity on Summit

6 NV100 GPUs
2 Power9 CPU sockets
42 CPU Cores
172 CPU Hardware threads



A Summit node

Note: 1 core on each socket has been set aside for overhead and is not available for allocation through jsrun. The core has been omitted and is not shown in the above image.

* Diagram courtesy of OLCF

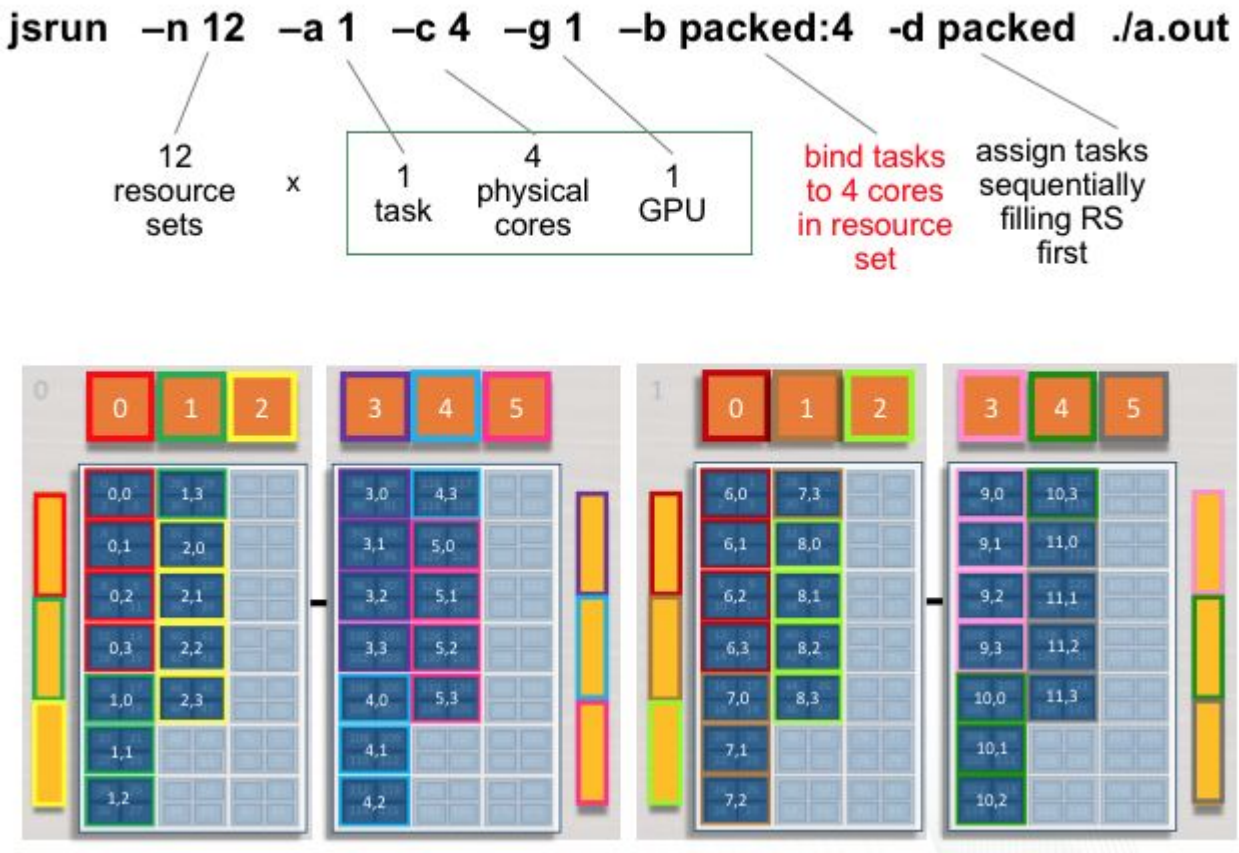
Summit – MPI+OpenMP Affinity

- jsrun is used to launch MPI, MPI+OpenMP, OpenMP (only) applications
- jsrun utility controls MPI task placement and OpenMP places
- MPI+OpenMP affinity set by jsrun
  - MPI task placement
  - **Sets OpenMP places ICV and OMP_PLACES environment variable**
  - Maps one OpenMP place per hardware thread.
  - OpenMP thread binding is set **by user** using OMP_PROC_BIND or proc_bind() clause
- mpirun is also available (not preferred)
- For more information on jsrun:
  - https://beta.olcf.ornl.gov/for-users/system-user-guides/summit/running-jobs/

Summit - JSRUN – Resource Sets

- A **resource set**
  - Controls how resources are managed within the node
  - Can create one or more **resource sets** within a node
  - Each **resource sets contain 1 or more cores and 0 or more GPUs**
- Understand how the application interacts with the system
  - How many tasks/threads per GPU
  - Does each task expect to see a single GPU? (or multiple GPUs)
- Create resource sets to specify:
  - GPUs per **resource set**
  - Cores per **resource set**
  - MPI tasks per **resource set**
    - Distribution
- Decide the # of resource set needed in applications

Example of resource sets on Summit

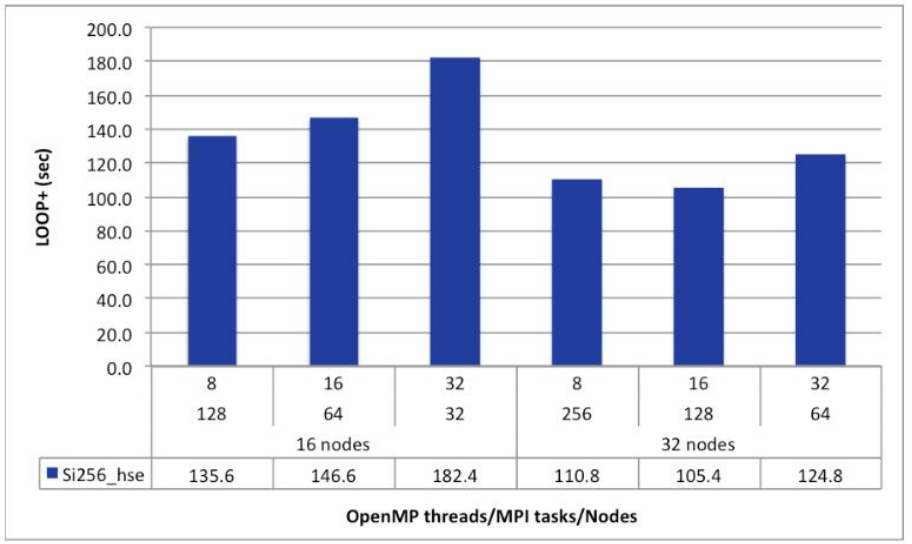


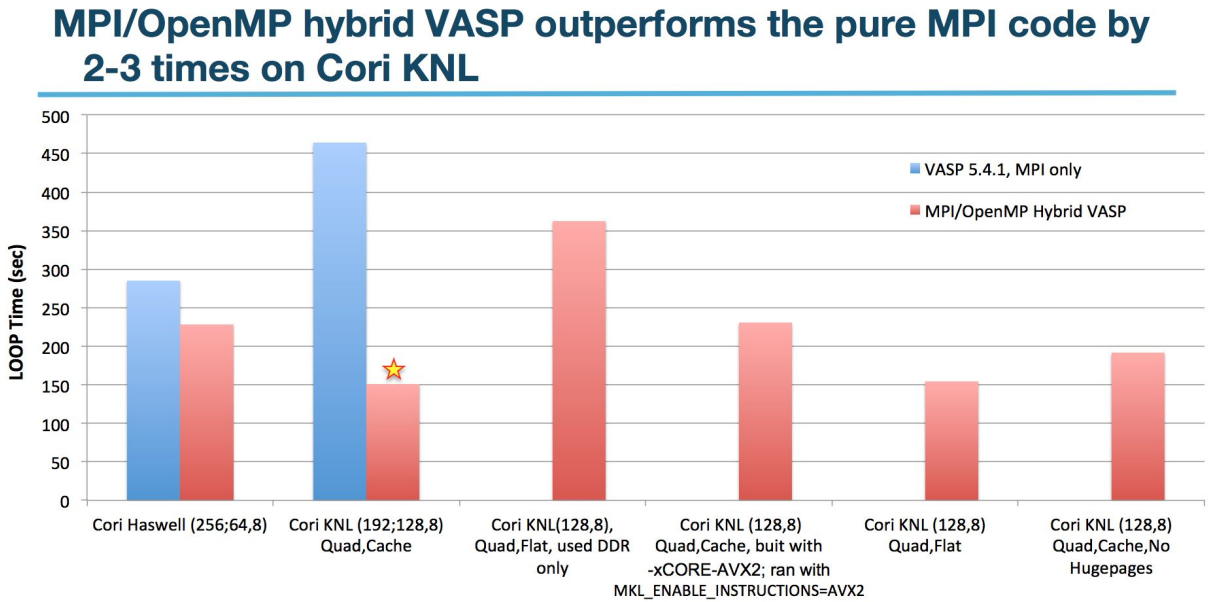

```
%setenv OMP_NUM_THREADS 4
%jsrun -n12 -a1 -c4 -g1 -b packed:4 -d packed ./a.out
Rank: 0; RankCore: 0; Thread: 0; ThreadCore: 0; Hostname: a33n06; OMP_NUM_PLACES: {0},{4},{8},{12}
Rank: 0; RankCore: 0; Thread: 1; ThreadCore: 4; Hostname: a33n06; OMP_NUM_PLACES: {0},{4},{8},{12}
Rank: 0; RankCore: 0; Thread: 2; ThreadCore: 8; Hostname: a33n06; OMP_NUM_PLACES: {0},{4},{8},{12}
Rank: 0; RankCore: 0; Thread: 3; ThreadCore: 12; Hostname: a33n06; OMP_NUM_PLACES: {0},{4},{8},{12}

Rank: 1; RankCore: 16; Thread: 0; ThreadCore: 16; Hostname: a33n06; OMP_NUM_PLACES: {16},{20},{24},{28}
Rank: 1; RankCore: 16; Thread: 1; ThreadCore: 20; Hostname: a33n06; OMP_NUM_PLACES: {16},{20},{24},{28}
Rank: 1; RankCore: 16; Thread: 2; ThreadCore: 24; Hostname: a33n06; OMP_NUM_PLACES: {16},{20},{24},{28}
Rank: 1; RankCore: 16; Thread: 3; ThreadCore: 28; Hostname: a33n06; OMP_NUM_PLACES: {16},{20},{24},{28}

...

Rank: 10; RankCore: 104; Thread: 0; ThreadCore: 104; Hostname: a33n05; OMP_NUM_PLACES: {104},{108},{112},{116}
Rank: 10; RankCore: 104; Thread: 1; ThreadCore: 108; Hostname: a33n05; OMP_NUM_PLACES: {104},{108},{112},{116}
Rank: 10; RankCore: 104; Thread: 2; ThreadCore: 112; Hostname: a33n05; OMP_NUM_PLACES: {104},{108},{112},{116}
Rank: 10; RankCore: 104; Thread: 3; ThreadCore: 116; Hostname: a33n05; OMP_NUM_PLACES: {104},{108},{112},{116}

Rank: 11; RankCore: 120; Thread: 0; ThreadCore: 120; Hostname: a33n05; OMP_NUM_PLACES: {120},{124},{128},{132}
Rank: 11; RankCore: 120; Thread: 1; ThreadCore: 124; Hostname: a33n05; OMP_NUM_PLACES: {120},{124},{128},{132}
Rank: 11; RankCore: 120; Thread: 2; ThreadCore: 128; Hostname: a33n05; OMP_NUM_PLACES: {120},{124},{128},{132}
Rank: 11; RankCore: 120; Thread: 3; ThreadCore: 132; Hostname: a33n05; OMP_NUM_PLACES: {120},{124},{128},{132}
```

* Example courtesy of OLCF

VASP: MPI/OpenMP Scaling Study

- **Original MPI parallelization**
  - Over the bands (high level)
  - Over Fourier coefficient of the bands (low level)
- **MPI + OpenMP parallelization**
  - MPI over bands (high level)
  - OpenMP threading over the coefficients of bands, either by explicitly adding OpenMP directives or via using threaded FFTW and LAPACK/BLAS3 libraries
  - No nested OpenMP
  - SIMD vectorization is deployed extensively
  - MPI/OpenMP scaling study to find the sweet spot
  - Other tuning options





Courtesy of Zhengji Zhao,NERSC

PARSEC: Overlap Comp and Comm (1)

Original Force Pseudocode

```
do type
   do atom
      calc A & B
      reduceAll A & B to master
      calc Δforce = f(A&B) on master
      store force on master
   end atom
end type
```

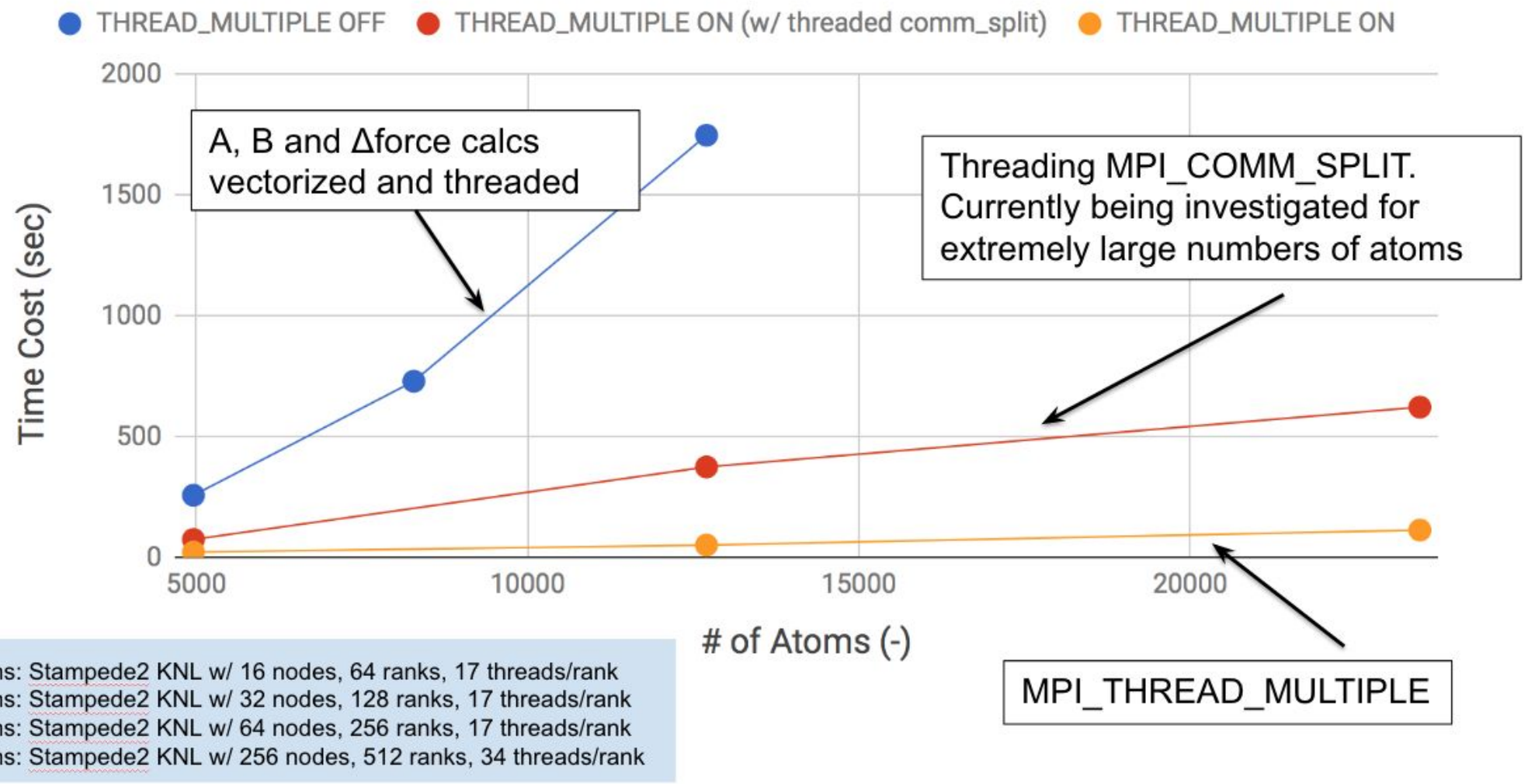
- Preemptively create an **array of comms**, one for each atom, to allow mpi ranks without data to move to the next atom
- **Atom loop is threaded**, allowing multiple atoms to be solved simultaneously
- Use **MPI_THREAD_MULTIPLE,** multiple threads call MPI

Improved Version with MPI_THREAD_MULTIPLE

```
do type
   MPI_COMM_SPLIT(atom, rank_has_data)
   !$OMP DO
   do atom
      if comm(atom) = MPI_COMM_NULL, cycle
      calc A & B
      reduceAll(comm(atom), A)
      calc Δforce = f(A&B)
      reduceAll(comm(atom), Δforce)
      store locally with master of comm(atom)
   end atom
   !$OMP END DO
end type
```

Courtesy of Kevin Gott, NERSC

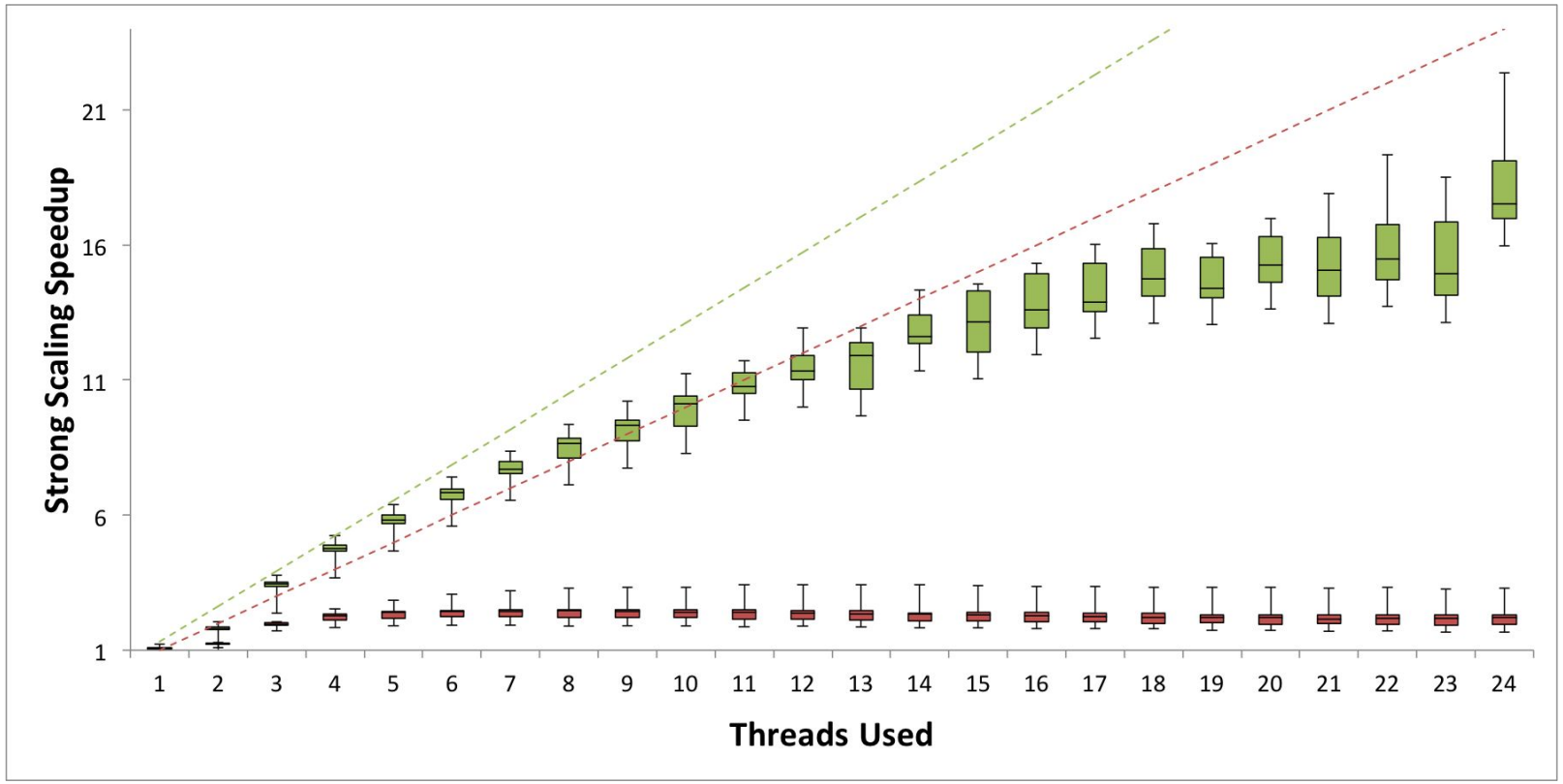
PARSEC: Overlap Comp and Comm (2)



Courtesy of Kevin Gott, NERSC

HMMER3: Use OpenMP Task Directives

- Replace pthread implementation limited by performance of master thread
  - OpenMP tasks facilitate overlap of I/O and Compute
  - Forking of child tasks and task groups allow simple work stealing implementation
- Thread scaling result on 1 Edison node (24 cores of Intel Xeon Ivy Bridge)



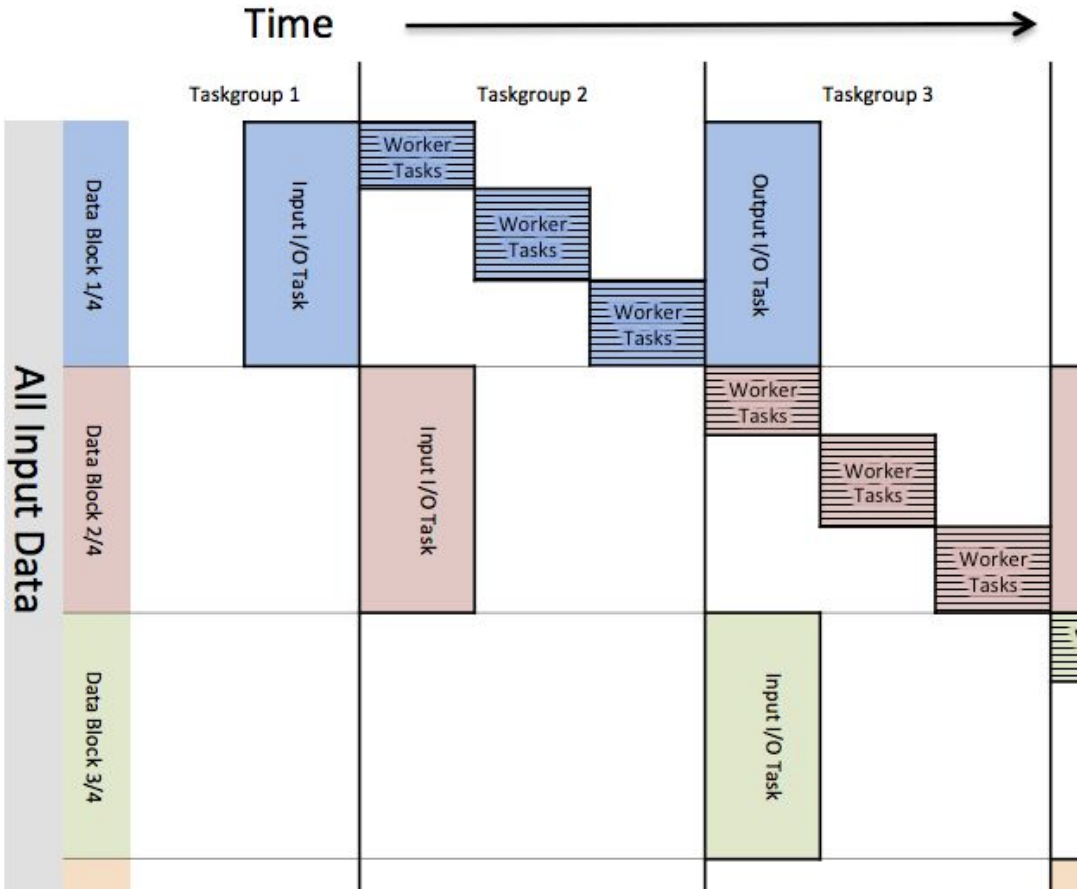
- pthread HMMER3 Red
- OpenMP HMMER3 Green

- Dashed lines show theoretical peak (two lines because serial performance is also improved)

Courtesy of Willaim Arndt, NERSC

HMMER3: Use task and taskgroup to Overlap I/O and Compute

```
#pragma omp parallel {
    #pragma omp single {
        #pragma omp task { load_seq_buffer(); }
        #pragma omp task { load_hmm_buffer(); }
        #pragma omp taskwait
        while( more HMMs ) {
            #pragma omp task { write_output();
                               load_hmm_buffer(); }
            while( more sequences ) {
                #pragma omp taskgroup {
                    #pragma omp task {
                    load_seq_buffer(); }
                    for ( each hmm in hmm_buffer )
                        #pragma omp task {
                        task_kernel(); }
                    swap_I/O_and_working_seq_buffers()
                    ;
                }
            }
            #pragma omp taskwait
            swap_I/O_and_working_hmm_buffers();
        }
    }
}
```

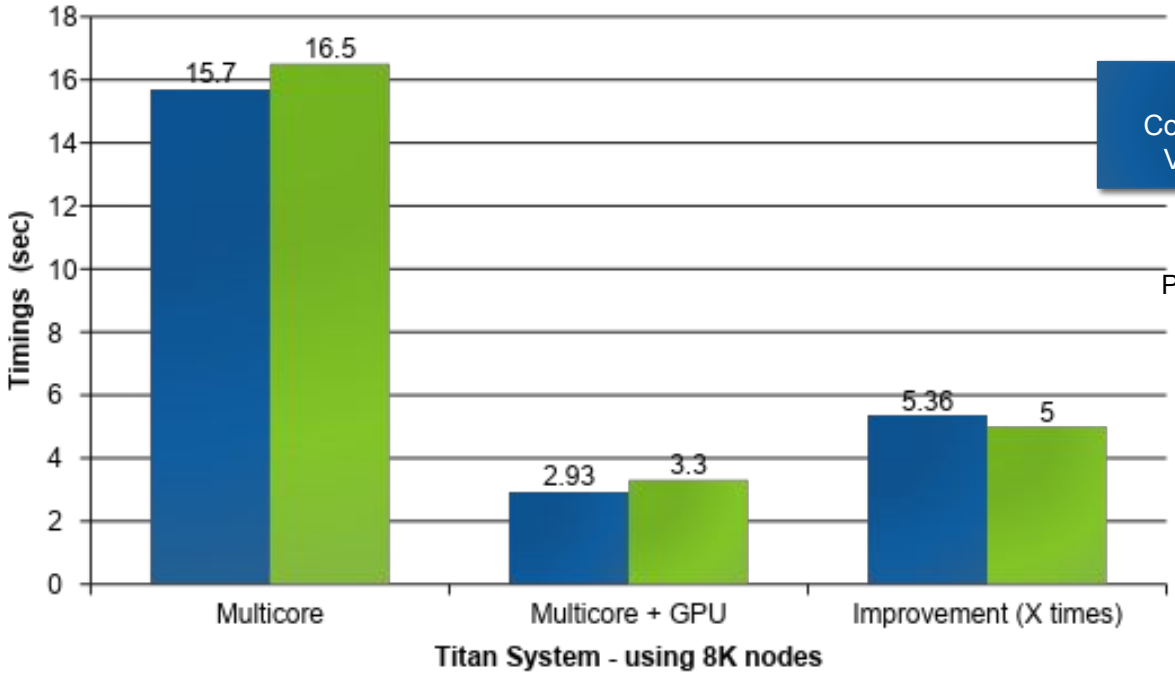


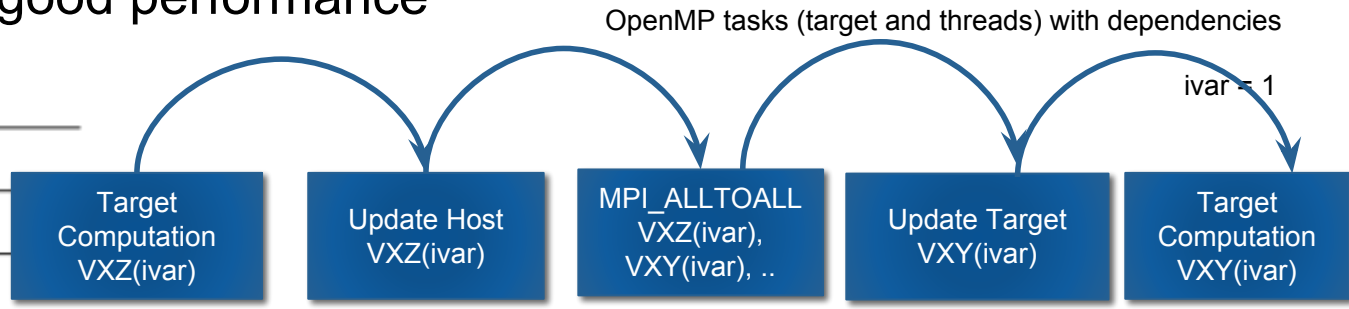
Courtesy of William Arndt, NERSC

MPI+OpenMP : Turbulence Mixing Application

P.K Yeung, et al.
INCITE

- Direct numerical simulations of turbulence and turbulent mixing
  - Pseudo-spectral for velocity, compact finite differences for scalar.
- Uses OpenMP to orchestrate the work among CPU cores, accelerator and network
  - Accelerate scalar field computation - 5x improvement
  - Use OpenMP tasks to hide network, GPU data transfer and computation latencies (14% improv.)
  - OpenMP Target tuning is needed to achieve good performance





Using 8K nodes/GPUs

EXASCALE COMPUTING PROJECT

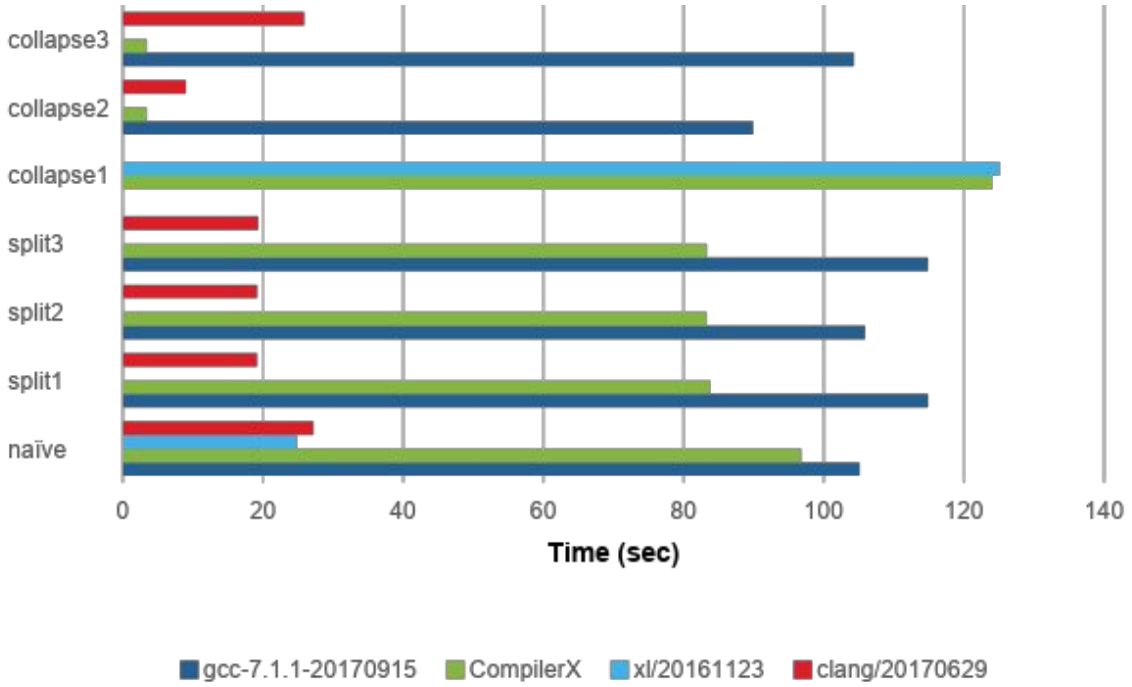
Overlap Communication and Computation

- MPI_THREAD_MULTIPLE: any thread can call MPI
- Can be used with OpenMP tasks to hide latencies, overheads, wait times, and to keep the interconnect busy etc.
- Profitable when using two or more OpenMP threads

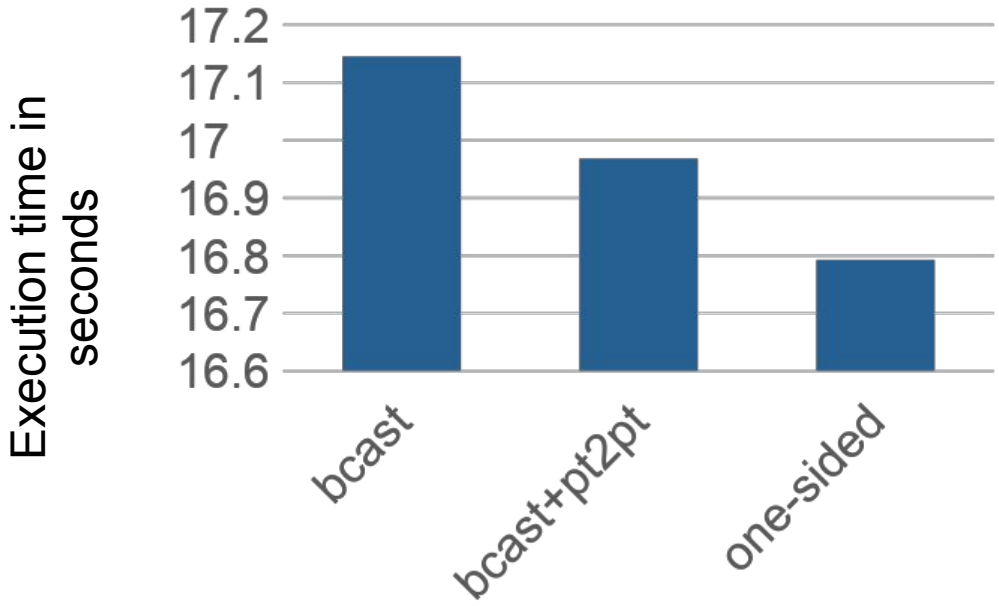
```
!$OMP PARALLEL DO NUM_THREADS(2 or more)
   do i=0, N
    !$OMP TARGET DEPEND(OUT:i) NOWAIT
         ! do some computation on device (e.g. dataXY(i))
    !$OMP TARGET UPDATE FROM(dataXY(i)) DEPEND(INOUT:i) NOWAIT
          ! update data on the host (e.g. dataXY(i))
   !$OMP TASK DEPEND(INOUT:i)
         call MPI_xxx(...) to exchange data (e.g. dataXY(i), dataYZ(i))
    $OMP TARGET UPDATE TO(dataYZ(i)) DEPEND(INOUT:i) NOWAIT
          ! update data on the device (e.g. dataYZ(i))
     !$OMP TARGET DEPEND(IN:i) NOWAIT
         ! do some computation on device on dataYZ(i)
      enddo
!$OMP END PARALLEL DO
```

MPI + OpenMP with GPU offload– Lessons Learned

- Matrix Multiplication: MPI for intranode communications and OpenMP to offload computation to GPUs
  - Application developers must pay **extreme** attention on tuning or performance won't be there.
  - Compilers/runtimes need to help for performance portability across platforms.



Summitdev:: Power8, NVIDIA GP100, NVLINK, IB EDR



EXASCALE COMPUTING PROJECT

Major features in OpenMP 5.0

- Task reductions
- Memory allocators
- Detachable tasks
- C++14 and C++17 support
- Fortran 2008 support
- Unified shared memory
- Loop construct
- Collapse non-rec loops
- Multi-level parallelism (*)
- Scan
- Task to data affinity
- Meta-directives
- Data serialization for offload
- Display affinity
- Reverse offload
- Dependence objects
- Improved task dependencies
- User-defined function variants
- OMPT/OMPD tools API

MPI Current Proposals/Investigated Concepts for Threads

- Endpoints: let threads have their own MPI rank
- Finepoints: let threads contribute to MPI operations without having a rank per thread (thread-based buffer partitioning)
- MPI & task-based environment: provide a MPI_TASK_MULTIPLE mode that guarantee progress of tasks and communications
- MPI_COMM_TYPE_ADDRESS_SPACE: groups ranks that share an address space (can be used in conjunction of endpoints)
- MPI implementations (not a standard proposal): runtime coordination to optimally partition on-resources between MPI and OpenMP