Process and Thread Affinity with MPI/OpenMP

5th Workshop on Coupling Technologies for Earth System Models (CW2020)
September 25, 2020

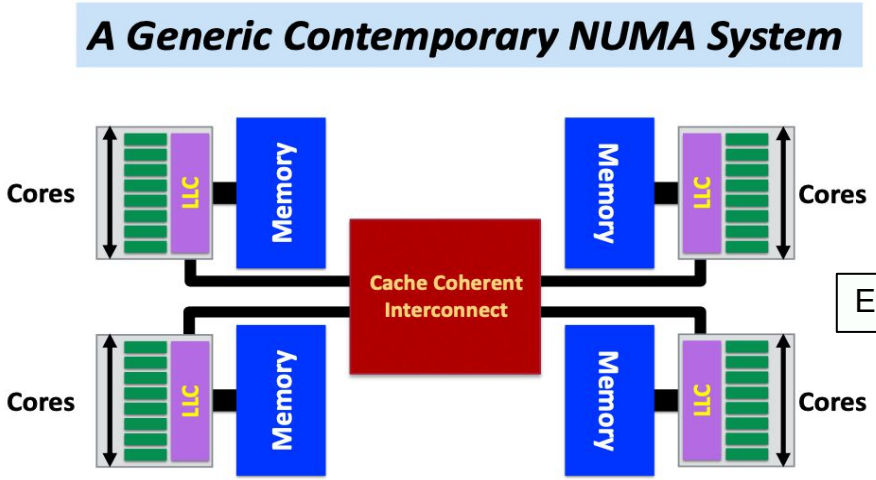
Yun (Helen) He
Lawrence Berkeley National Laboratory

CPU Architecture Trend

- Multi-socket nodes with rapidly increasing core counts
  - Memory per core decreases
  - Memory bandwidth per core decreases
  - Network bandwidth per core decreases
- Applications often use a hybrid programming model with three levels of parallelism
  - MPI between nodes or sockets
  - Shared memory (such as OpenMP) on the nodes/sockets
  - Increase vectorization for lower level loop structures

NUMA Systems

- Most systems today are Non-Uniform Memory Access (NUMA)
- Multiple NUMA domains per node or socket
- Accessing memory in remote NUMA is slower than accessing memory in local NUMA
- Accessing High Bandwidth Memory is faster than DDR



Each core may have multiple hyperthreads

All modern CPUs include caches therefore all modern systems are NUMA even though we often pretend they are UMA

Diagram courtesy Ruud van der Pas

Process / Thread / Memory Affinity (1)

- Process Affinity: also called "CPU pinning", binds processed (MPI tasks, etc.) to a CPU or a ranges of CPUs on a node
  - It is important to spread MPI ranks evenly onto cores in different NUMA domains
- Thread Affinity: further binding threads to CPUs that are allocated to their parent process
  - Thread affinity should be based on achieving process affinity first
  - Threads forked by a certain MPI task have thread affinity binding close to the process affinity binding of their parent MPI task
  - Do not over schedule CPUs for threads

Process / Thread / Memory Affinity (2)

- Memory Locality: allocate memory as close as possible to the core on which the task that requested the memory is running
  - Applications should use memory from local NUMA domain as much as possible
- Our goal is to promote OpenMP standard settings for portability
  - OMP_PLACES and OMP_PROC_BIND are preferred to vendor specific settings
- Correct process, thread and memory affinity is the basis for getting optimal performance. It is also essential for guiding further performance optimizations.

Tools to Check Compute Node Information (1)

- **numactl:** controls NUMA policy for processes or shared memory
  - **numactl -H:** provides NUMA info of the CPUs

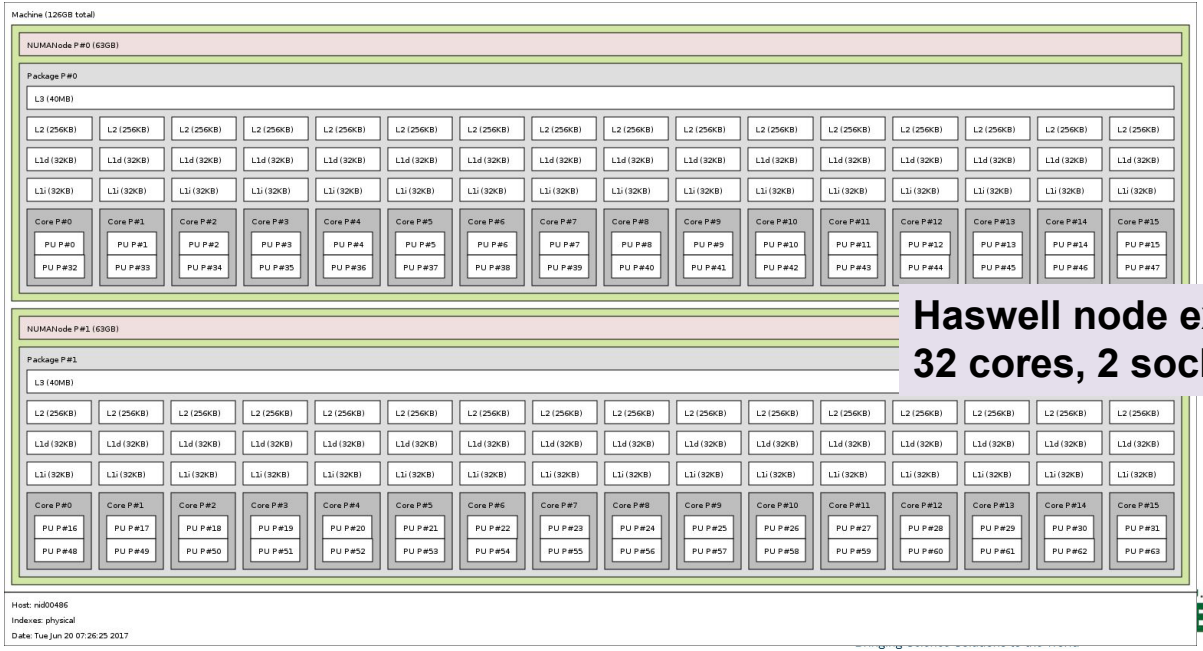
Haswell node example
32 cores, 2 sockets

```
% 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 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 64430 MB
node 0 free: 63002 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63
node 1 size: 64635 MB
node 1 free: 63395 MB
node distances:node   0   1
0:  10  21
1:  21  10
```

*Haswell: 16-core **Intel® Xeon™ Processor E5-2698 v3 at 2.3 GHz**

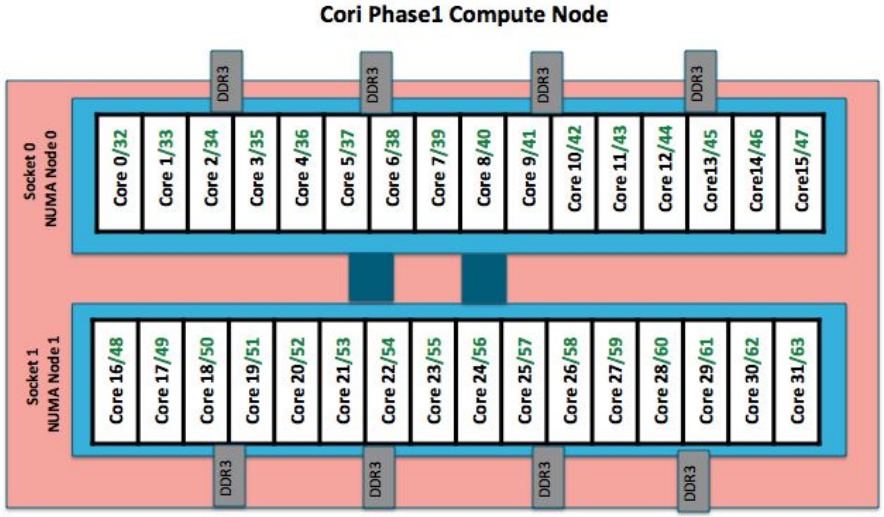
Tools to Check Compute Node Information (2)

- Portable Hardware Locality (hwloc)
  - hwloc-ls and lstopo: provides a text and graphical representation of the system topology, NUMA nodes, cache info, and the mapping of procs.



Haswell node example
32 cores, 2 sockets

Haswell Compute Nodes Example



To obtain processor info:

Get on a compute node:
% salloc -N 1 -C …
Then:
% numactl -H
or % cat /proc/cpuinfo
or % hwloc-ls

- Each Haswell node has 2 Intel Xeon 16-core Haswell processors
  - **2 NUMA domains (sockets) per node, 16 cores per NUMA domain. 2 hardware threads per physical core.**
  - **NUMA Domain 0: physical cores 0-15 (and logical cores 32-47)**
    NUMA Domain 1: physical cores 16-31 (and logical cores 48-63)
- Memory bandwidth is non-homogeneous among NUMA domains

MPI Process Affinity: Selected Slurm srun Options

- --cpu-bind=threads
  Automatically generate masks binding tasks to threads
- --cpu-bind=cores
  Automatically generate masks binding tasks to cores
- --cpu-bind=sockets
  Automatically generate masks binding tasks to sockets
- --cpu-bind=map_cpu:<cpulist>
  Bind by setting CPU masks on tasks (or ranks)
- --cpu-bind=map_ldom:<NUMA_domain_list>
  Bind by mapping NUMA locality domain IDs to tasks
  (ldom means logical domain)

Use numactl Command Line Tool

- numactl is a Linux tool to investigate and handle NUMA
- Can be used to request CPU or memory binding
  - Use "**numactl <options> ./myapp**" as the executable (instead of "./myapp")

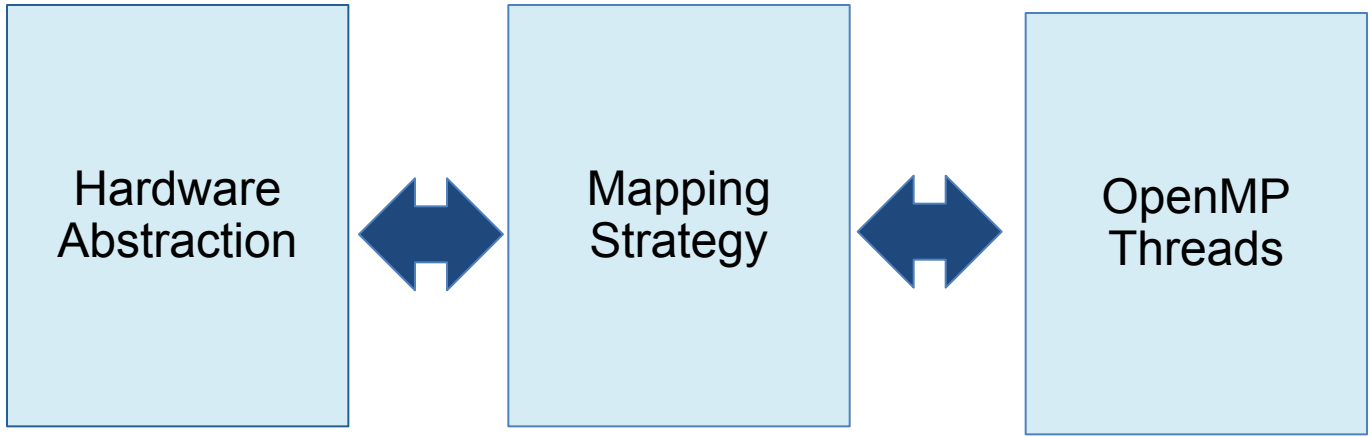
- CPU binding example:
  - % numactl --cpunodebind 0,1 ./code.exe

    only use cores of NUMA nodes 0 and 1
- Memory binding example:
  - % numactl --membind 1 ./code.exe

    only use memory in NUMA nodes 1, such as the MCDRAM (High Bandwidth Memory) in KNL quad,flat mode

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

OMP_NUM_THREADS
Environment Variable
or
num_threads() clause
of parallel region

Courtesy of Oscar Hernandez, ORNL

Runtime Environment Variable: OMP_PLACES

- OMP_PLACES environment variable
  - controls thread allocation
  - defines a series of places to which the threads are assigned
- It can be an abstract name or a specific list
  - threads: each place corresponds to a single hardware thread
  - cores: each place corresponds to a single core (having one or more hardware threads)
  - sockets: each place corresponds to a single socket (consisting of one or more cores)
  - a list with explicit place values of CPU ids, such as:
    - export OMP_PLACES=“ {0:4:2},{1:4:2}” (equivalent to “{0,2,4,6},{1,3,5,7}”)

- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Runtime Environment Variable: OMP_PROC_BIND (1)

- Controls thread affinity within and between OpenMP places
- Allowed values:
  - true: the runtime will not move threads around between processors
  - false: the runtime may move threads around between processors
  - close: bind threads close to the master thread
  - spread: bind threads as evenly distributed (spreaded) as possible
  - master: bind threads to the same place as the master thread
- The values **master**, **close** and **spread** imply the value **true**

- Examples:
  - export OMP_PROC_BIND=spread
  - export OMP_PROC_BIND=spread,close (for nested levels)

Runtime Environment Variable: OMP_PROC_BIND (2)

Prototype example: 4 cores total, 2 hyperthreads per core, 4 OpenMP threads

- none: no affinity setting
- close: Bind threads as close to each other as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0		1		2		3	

- master: bind threads to the same place as the master thread

Affinity Clauses for OpenMP Parallel Construct

- The num_threads and proc_bind clauses can be used
  - The values set with these clauses take precedence over values set by runtime environment variables
- Helps code portability

- Examples:
  - C/C++:

    #pragma omp parallel *num_threads(2) proc_bind(spread)*
  - Fortran:

    !$omp parallel *num_threads (2) proc_bind (spread)*

    …

    !$omp end parallel

Various Methods to Set Number of Threads

1) Use num_threads clause

```
#pragma omp parallel num_threads (4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

2) Call omp_set_num_threads API

```
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

3) Set runtime environment

```
export OMP_NUM_THREDS=4
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

4) Do none of the three above.

Code will use an implementation dependent default number of threads defined by the compiler.

- Precedence: 1) > 2) > 3) > 4)
- You may get fewer threads than you requested, check with omp_get_num_threads()

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

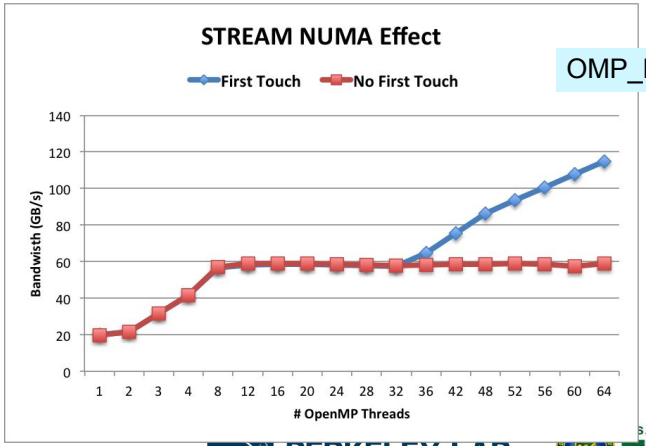
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

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.
- Hard to do “perfect touch” for real applications. General recommendation is to use number of threads fewer than number of CPUs (one or more MPI tasks) per NUMA domain.

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

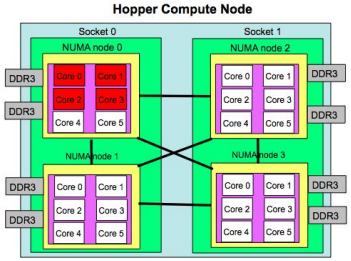


OMP_PROC_BIND=close

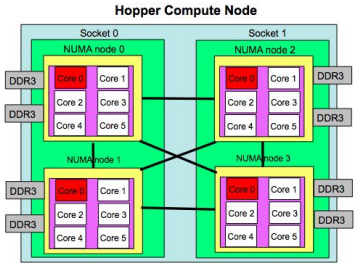
MPI Process Affinity Example: aprun "-S" Option

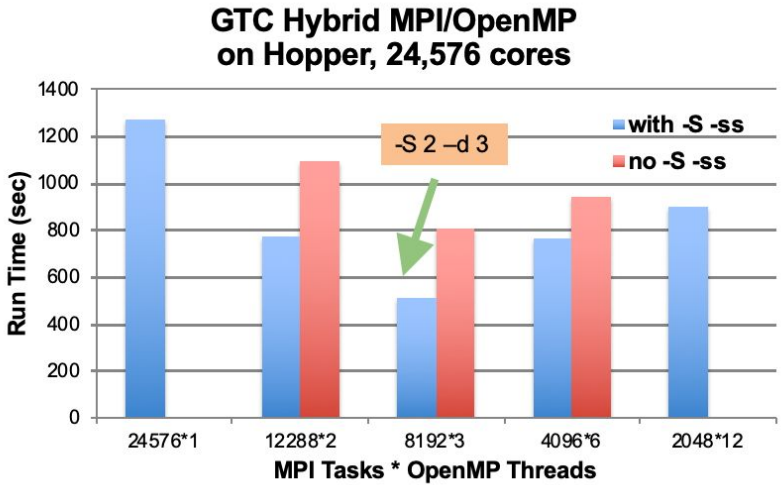
- Important to spread MPI ranks evenly onto different NUMA nodes
- Use the "-S" option: specify #MPI_tasks per NUMA domain
- The example below was from an XE6 system (NERSC Hopper)

aprun –n 4 –d 6



aprun –n 4 **–S 1**–d 6





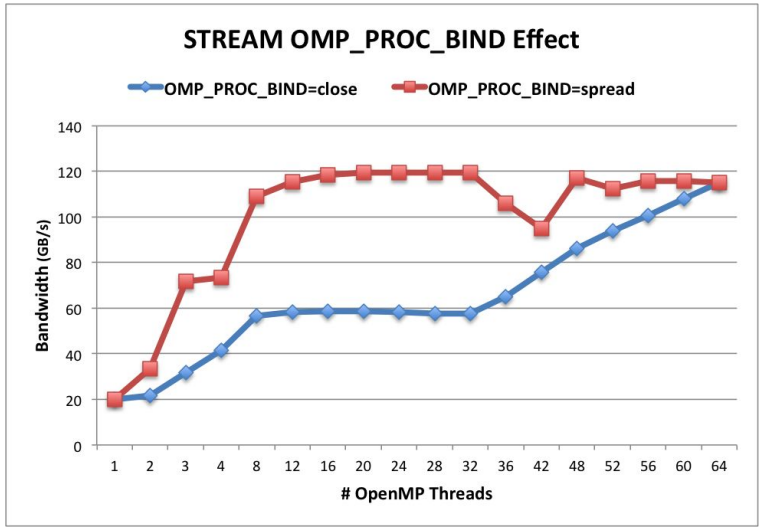
OMP_PROC_BIND Choices for STREAM Benchmark

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



Sample Nested OpenMP Program

```
#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\n", 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);
}
```

```
% ./a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
```

```
% export OMP_NESTED=true
% export OMP_MAX_ACTIVE_LEVELS=3
% ./a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
```

```
Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P7
```

Process and Thread Affinity in Nested OpenMP

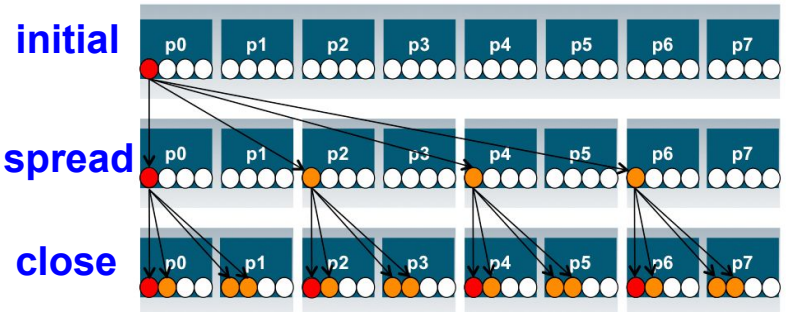
- A combination of OpenMP environment variables and runtime flags are needed for different compilers and different batch schedulers on different systems

```
#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 ./code.exe
```



- 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

KNL Compute Nodes Example

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

Arrangement of Hardware Threads for 68 Core KNL

Core # →	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
HW Thread #	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
	68	69	70	71	...	84	85	86	...	101	102	103	...	118	119	120	...	133	134	135
	136	137	138	139	...	152	153	154	...	169	170	171	...	186	187	188	...	201	202	203
	204	205	206	207	...	220	221	222	...	237	238	239	...	254	255	256	...	269	270	271

- A quad,cache node (default setting) has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory). MCDRAM is hidden from the "numactl -H" result since it is a cache.
- A quad,flat node has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory). NUMA node 1 has MCDRAM only
- A snc2,flat node has 4 NUMA domains with DDR memory and all CPUs on NUMA nodes 0 and 1

Can We Just Do a Naive srun?

Example: 16 MPI tasks x 8 OpenMP threads per task on a single 68-core KNL quad,cache node:

```
% export OMP_NUM_THREADS=8
% export OMP_PROC_BIND=spread    (other choice are “close”,”master”,”true”,”false”)
% export OMP_PLACES=threads       (other choices are: cores, sockets, and various ways to specify explicit lists, etc.)

% srun -n 16  ./xthi |sort -k4n,6n          or % mpirun –n 16 ./xthi
Hello from rank 0, thread 0, on nid02304. (core affinity = 0)
Hello from rank 0, thread 1, on nid02304. (core affinity = 144)      (on physical core 8)
Hello from rank 0, thread 2, on nid02304. (core affinity = 17)
Hello from rank 0, thread 3, on nid02304. (core affinity = 161)      (on physical core 25)
…
Hello from rank 1, thread 0, on nid02304. (core affinity = 0)
Hello from rank 1, thread 1, on nid02304. (core affinity = 144)
```

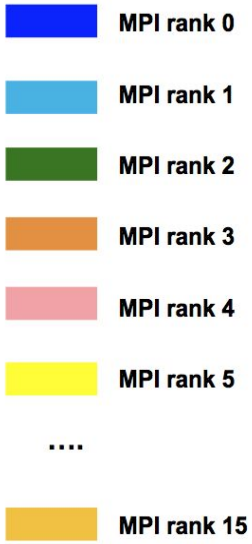
It is a mess! e.g., thread 0 for rank 0, and thread 1 for rank 1 are on same physical core 0

Example mpirun or srun Commands: Fix the Problem

- The reason is #MPI tasks is not divisible by 68!
  - Each MPI task is getting 68x4/#MPI tasks of logical cores as the domain size
  - MPI tasks are crossing tile boundaries
- Let's set number of logical cores per MPI task manually by wasting extra 4 cores on purpose, which is 256/#MPI tasks

- Cray MPICH with Aries network using native SLURM
  - % srun -n 16 -c 16 --cpu_bind=cores ./code.exe
    Notes: Here the value for -c is also set to number of logical cores per MPI task, i.e., 256/#MPI tasks.
- Intel MPI with Omni Path using mpirun:
  - % export I_MPI_PIN_DOMAIN=16
  - % mpirun -n 16 ./code.exe

Now It Looks Good!

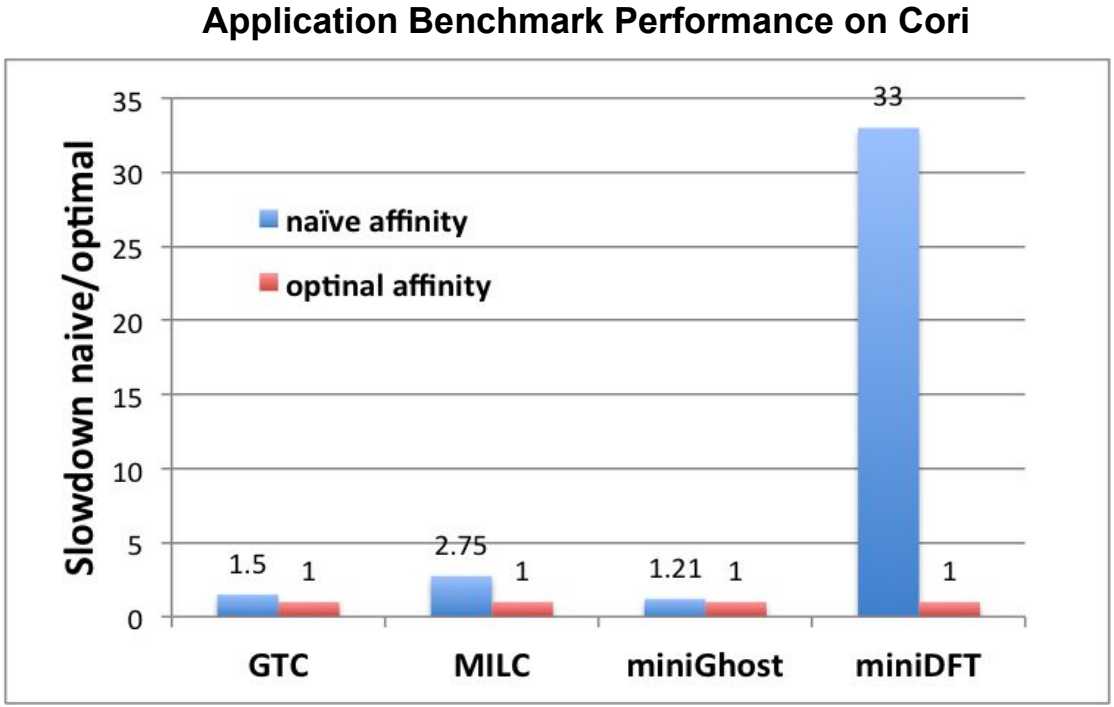


Process/thread affinity are good! (Marked first 6 and last MPI tasks only)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
204																220	221
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
86																102	103
154	155	156	157	158	159	160	161									170	171
222																238	239
36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51		
104																	
172																	
240																	
52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67		
120																	
188								196	197	198	199						
256																	

And so on for other MPI tasks and threads ….

Naïve *vs.* Optimal Affinity



OpenMP task-to-data Affinity (in OpenMP 5.0)

- Affinity hints can be provided for OpenMP tasks, resulting data to be closer to tasks
- Useful for multi-socket systems

```
void task_affinity() {
      double* B;
#pragma omp task shared(B) affinity(A[0:N])
      B = init_B_and_important_computation(A);

#pragma omp task firstprivate(B) affinity(B[0:N])
      important_computation_too(B);

#pragma omp taskwait
}
```

Affinity Verification Methods

- NERSC provides pre-built binaries from a Cray code (xthi.c) to display process thread affinity

```
% srun -n 32 -c 8 --cpu-bind=cores check-mpi.intel.cori | sort -nk 4
Hello from rank 0, on nid02305. (core affinity = 0,1,68,69,136,137,204,205)
Hello from rank 1, on nid02305. (core affinity = 2,3,70,71,138,139,206,207)
```

- Use portable OpenMP environment variables OMP_DISPLAY_AFFINITY and OMP_AFFINITY_FORMAT (in OpenMP 5.0)
  - Automatically displays affinity info when OMP_DISPLAY_AFFINITY=true
  - Can set custom OMP_DISPLAY_AFFINITY_FORMAT
  - Also has runtime APIs such as omp_display_affinity and omp_capture_affinity

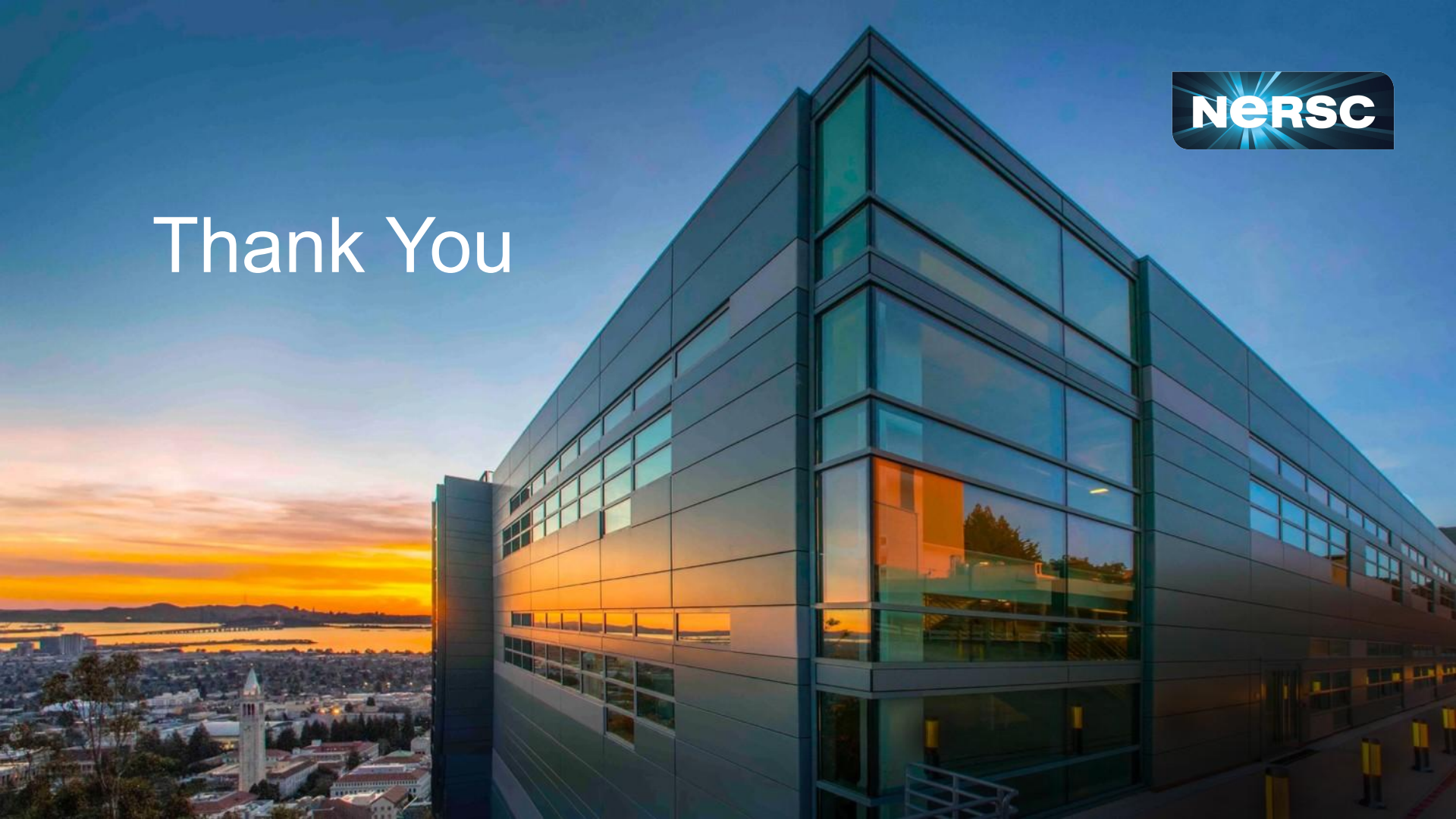
OMP_AFFINITY_FORMAT Fields

Short Name	Long name	Meaning
L	thread_level	from omp_get_level()
n	thread_num	from omp_get_thread_num()
a	thread_affinity	the numerical identifiers of the processors the current thread is binding to, in the format of a comma separated list of OpenMP thread places
h	host	host or node name
p	process_id	process id used by the implementation (such as the process id for the MPI process)
N	num_threads	from omp_get_num_threads()
A	ancestor_tnum	from omp_get_ancestor_thread_num(). One level up only.

```
% export OMP_DISPLAY_AFFINITY=true
% export OMP_AFFINITY_FORMAT="host=%h, pid=%p, thread_num=%n, thread affinity=%a”
host=nid02496, pid=150147, thread_num=0, thread affinity=0
host=nid02496, pid=150147, thread_num=1, thread affinity=4
% export  OMP_AFFINITY_FORMAT="Thread Affinity: %0.3L %.10n %.20{thread_affinity} %.15h”
Thread Affinity: 001        0        0-1,16-17        nid003
Thread Affinity: 001        1        2-3,18-19        nid003
```

Process and Thread Affinity Best Practices

- Achieving best data locality, and optimal process and thread affinity is crucial in getting good performance with MPI/OpenMP, yet it is not straightforward to do so
  - Understand the node architecture with tools such as “numactl -H” first
  - Set correct cpu-bind and OMP_PLACES options
  - Always use simple examples with the same settings for your real application to verify first or check with OMP_DISPLAY_AFFINITY
- Optimize code for memory affinity
  - Exploit first touch data policy, or use at least 1 MPI task per NUMA domain
  - Compare performance with put threads far apart (spread) or close
  - Use numactl -m option to explicitly request memory allocation in specific NUMA domain (for example: high bandwidth memory in KNL)


NERSC
Thank You