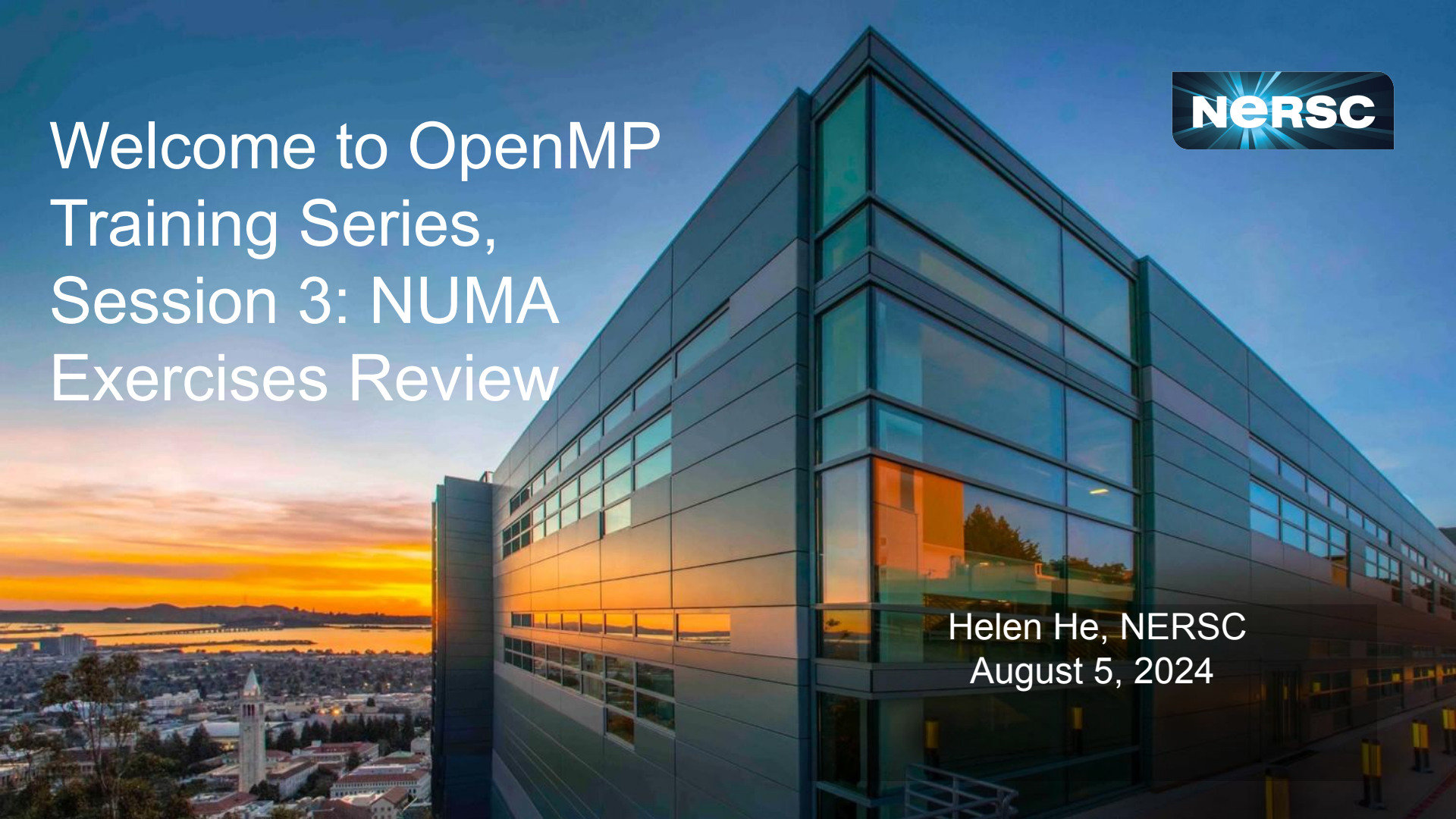


# Welcome to OpenMP Training Series, Session 3: NUMA Exercises Review

Helen He, NERSC
August 5, 2024

# Perlmutter CPU Node NUMA Info

**numactl -H:** provides NUMA info of CPUs

**128 cores and 2 processors total**
**64 cores and 4 NUMA domains per processor**

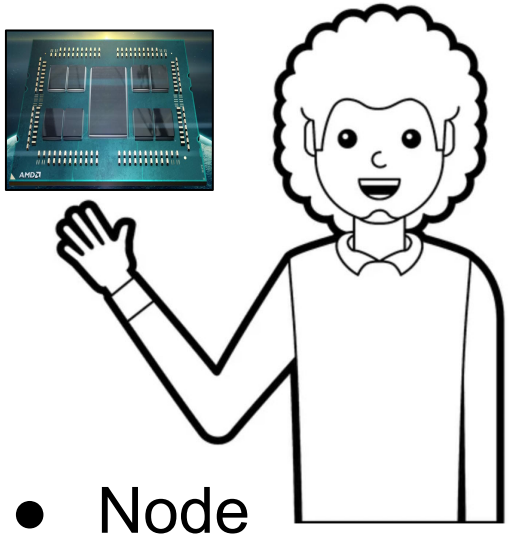
```
yunhe@nid005620:~> numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
...
node 7 cpus: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 240 241 242 243 244 245 246 247 248 249 250 251
252 253 254 255

node distances:
node 0 1 2 3 4 5 6 7
0: 10 12 12 12 32 32 32 32
1: 12 10 12 12 32 32 32 32
2: 12 12 10 12 32 32 32 32
3: 12 12 12 10 32 32 32 32
4: 32 32 32 32 10 12 12 12
5: 32 32 32 32 12 10 12 12
6: 32 32 32 32 12 12 10 12
7: 32 32 32 32 12 12 12 10
```

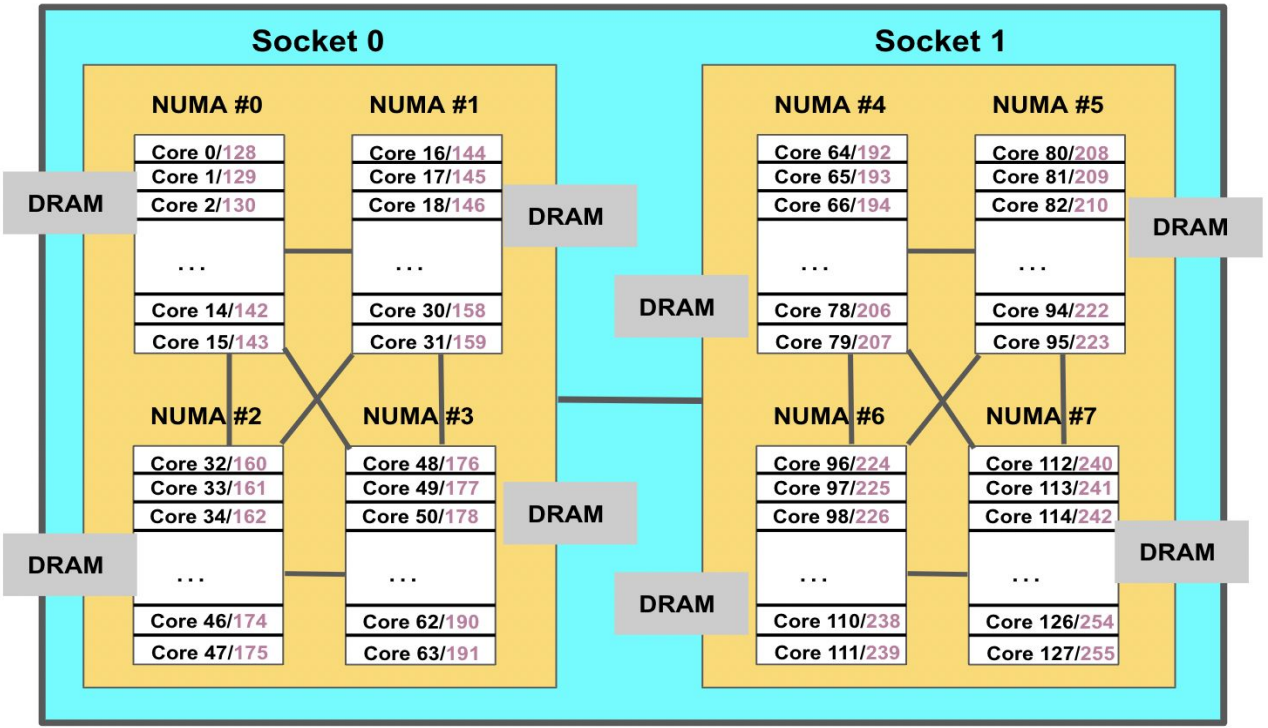
Shows relative cost of memory bandwidth
There is a factor of 3 accessing local vs remote NUMA domains in this example

These numbers are hardware thread ids (or logical core ids)

# Perlmutter CPU Compute Node



- Node
- Processor
- Physical Core
- *Logical* CPU



# xthi.c - verify thread affinity

```
int main(int argc, char *argv[])
{
        int rank, thread;
        …
        #pragma omp parallel private(thread, coremask, clbuf)
        {
                thread = omp_get_thread_num();
                (void)sched_getaffinity(0, sizeof(coremask), &coremask);
                cpuset_to_cstr(&coremask, clbuf);
                #pragma omp barrier
                printf("Hello from thread %d, on %s. (core affinity = %s)\n",
                        thread, hnbuf, clbuf);
        }
  return(0);
}
```

# Check Affinity Results

- Compile on a Perlmutter login node using the compiler wrapper.
  The default is GNU compiler
  % cc -fopenmp -o xthi_omp xthi_omp.c
  % cc -fopenmp -o xthi_nested_omp xthi_nested_omp.c
- Request a compute node via interactive batch
  % salloc -N 1 -C cpu -q interactive -t 30:00
  <You will then get on a compute node>
  Sample run command with the output ordered:
  nid004082% export OMP_NUM_THREADS=8
  nid004082% export OMP_PROC_BIND=spread
  nid004082% export OMP_PLACES=cores
  nid004082% ./xthi_omp |sort -k4,6

# Affinity Verification Methods (1)

- NERSC has provided pre-built binaries from an HPE code (xthi.c) to display process thread affinity: check-omp.gnu.pm, check-mpi.nvidia.pm, check-hybrid.cray.pm, etc. (use them instead of your own executable to verify)

```
% check-omp.gnu.pm | sort -nk 4
Hello from thread 0, on nid200039. (core affinity = 0,128)
Hello from thread 1, on nid200039. (core affinity = 16,144) …
…
% srun -n 8 -c 32 --cpu-bind=cores check-hybrid.gnu.pm | sort -nk 4
Hello from rank 0, thread 0, on nid200039. (core affinity = 0,128)
Hello from rank 0, thread 1, on nid200039. (core affinity = 2,130)
Hello from rank 0, thread 2, on nid200039. (core affinity = 4,132)
…
Hello from rank 1, thread 0, on nid200039. (core affinity = 64,192)
Hello from rank 1, thread 1, on nid200039. (core affinity = 66,194)
…
```

# Affinity Verification Methods (2)

- Use OMP_DISPLAY_AFFINITY and OMP_AFFINITY_FORMAT

```
% export OMP_DISPLAY_AFFINITY=true
% export OMP_AFFINITY_FORMAT="host=%H, thread_level=%L, thread_num=%n, thread affinity=%A”
% ./xthi_omp | sort -nk 4
host=nid200039, thread_level=1, thread_num=0, thread affinity=0,128
host=nid200039, thread_level=1, thread_num=1, thread affinity=16,144
host=nid200039, thread_level=1, thread_num=2, thread affinity=32,160
…
% export OMP_NUM_THREADS=4,3
% ./xthi_nested_omp | sort -nk 4
host=nid200039, thread_level=1, thread_num=0, thread affinity=0,128
host=nid200039, thread_level=1, thread_num=1, thread affinity=32,160
host=nid200039, thread_level=1, thread_num=2, thread affinity=64,192
host=nid200039, thread_level=1, thread_num=3, thread affinity=96,224
host=nid200039, thread_level=2, thread_num=0, thread affinity=0,128
host=nid200039, thread_level=2, thread_num=1, thread affinity=1,129
host=nid200039, thread_level=2, thread_num=2, thread affinity=2,130
```

# STREAM - Check Memory Bandwidth

```
/* Get initial value for system clock. */
for (j=0; j<STREAM_ARRAY_SIZE; j++) {
        a[j] = 1.0;
        b[j] = 2.0;
        c[j] = 0.0;
}
```

```
void tuned_STREAM_Triad(STREAM_TYPE scalar)
{
        ssize_t j;
#pragma omp parallel for
        for (j=0; j<STREAM_ARRAY_SIZE; j++)
                a[j] = b[j]+scalar*c[j];
}
```

# 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***

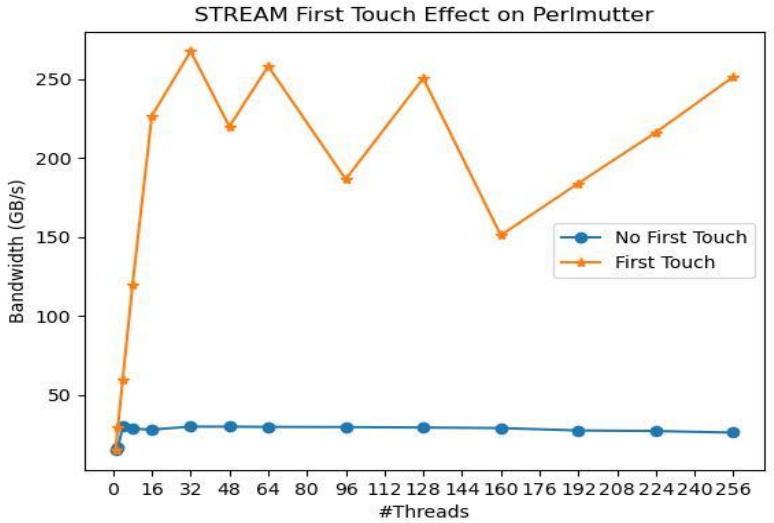
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

***Step 2 Compute***

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+scalar*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
Both with OMP_PROC_BIND=close




NERSC

# OMP_PROC_BIND Choices for STREAM Benchmark

**OMP_NUM_THREADS=32**
**OMP_PLACES=threads**

OMP_PROC_BIND=close
Threads 0 to 15 bind to CPUs 0,128,1,129,,…15,143. All threads are in the first socket (1st NUMA domain). The second socket is idle. Not optimal.

OMP_PROC_BIND=spread
Threads 0 to 31 bind to CPUs 0,4,8,… to 124. Both sockets (and all NUMA domains) and memory are used to maximize memory bandwidth.

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