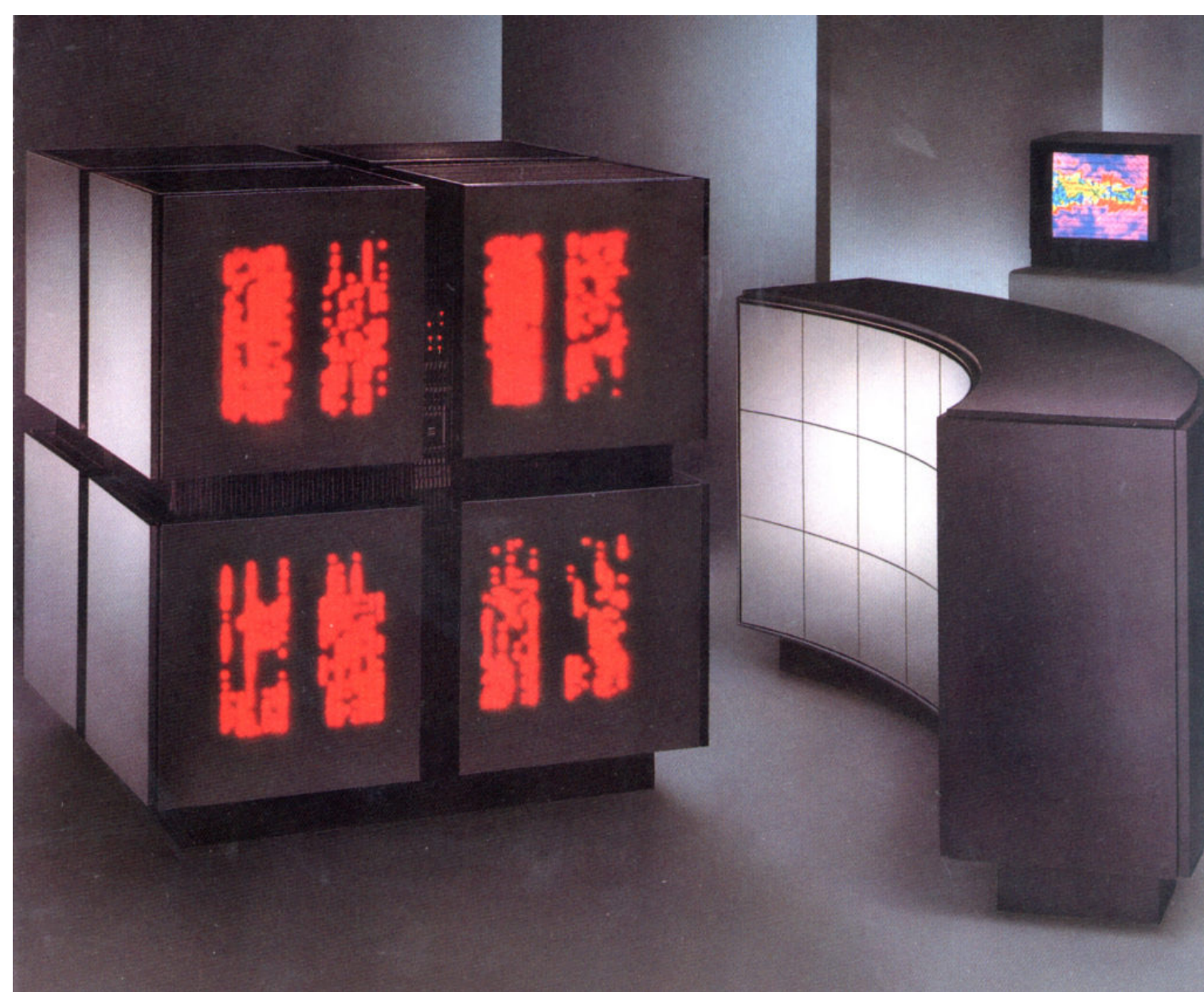


OpenMP Beyond the Common Core

Tim Mattson



The coolest looking computer ever built



- The coolest looking computer in history ..
 - The “Thinking Machines CM2”: A SIMD computer from the mid 1980’s with up to 64000 bit-serial processing elements.
- This was designed with old-style, symbolic AI in mind (back when we weren’t so obsessed with neural networks)
- It’s designer, Danny Hillis, came up with the greatest slogan ever produced by a computer company

“... we want to build a computer that will be proud of us”

We all love the common core...

The OpenMP Common Core: Most OpenMP programs only use these 21 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

**... but there is a bit beyond the
common core that's good to
know**

**Let's start with a bit more about
worksharing constructs**

The Loop Worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
  #pragma omp for
    for (I=0;I<N;I++){
      NEAT_STUFF(I);
    }
}
```

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause

Loop construct name:

- C/C++: for
- Fortran: do

Loop Worksharing Constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
 - **schedule(static [,chunk])**
 - Deal-out blocks of iterations of size “chunk” to each thread.
 - **schedule(dynamic[,chunk])**
 - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
 - **schedule(guided[,chunk])**
 - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
 - **schedule(runtime)**
 - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library) ... vary schedule without a recompile!
 - **Schedule(auto)**
 - Schedule is left up to the runtime to choose (does not have to be any of the above).

Loop Worksharing Constructs: The schedule clause

Schedule Clause	When To Use
static	Pre-determined and predictable by the programmer
dynamic	Unpredictable, highly variable work per iteration
guided	Special case of dynamic to reduce scheduling overhead
runtime	Schedule determined at runtime based on the assignment to the environment variable, OMP_SCHEDULE. Example: export OMP_SCHEDULE="dynamic,1"
auto	When the runtime can “learn” from previous executions of the same loop

Optimizing mandel.c

```
wtime = omp_get_wtime();  
#pragma omp parallel for collapse(2) schedule(runtime) firstprivate(eps) private(j,c)  
for (i=0; i<NPOINTS; i++) {  
    for (j=0; j<NPOINTS; j++) {  
        c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;  
        c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;  
        testpoint(c);  
    }  
}  
wttime = omp_get_wtime() - wtime;
```

```
$ export OMP_SCHEDULE="dynamic,100"
```

```
$ ./mandel_par
```

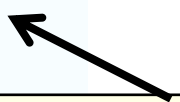
default schedule	0.48 secs
schedule(dynamic,100)	0.39 secs
collapse(2) schedule(dynamic,100)	0.34 secs

Four threads on a dual core Apple laptop (Macbook air ... 2.2 Ghz Intel Core i7 with 8 GB memory)
and the gcc version 9.1. Times are the minimum time from three runs

Nested Loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        . . . . .
    }
}
```



Number of loops
to be
parallelized,
counting from
the outside

- Will form a single loop of length $N \times M$ and then parallelize that.
- Useful if N is $O(\text{no. of threads})$ so parallelizing the outer loop makes balancing the load difficult.

Array Sections with Reduce

```
#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);
int main(){
    int i,j; float a[N], b[N][N]; init(N,b);
    for(i=0; i<N; i++) a[i]=0.0e0;
```

Works the same as any other reduce ... a private array is formed for each thread, element wise combination across threads and then with original array at the end

```
#pragma omp parallel for private(j) reduction(+:a[0:N])
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        a[j] += b[i][j];
    }
}
printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
return 0;
```

**We talked about the atomic construct
when we discussed the memory
model, but we never properly
introduced the construct**

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
    double B;
    B = DOIT();

    #pragma omp atomic
    X += big_ugly(B);
}
```


Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
    double B, tmp;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

Atomic only protects the read/update of X

The OpenMP 3.1 Atomics (1 of 2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

pragma omp atomic [read | write | update | capture]

- Atomic can protect loads

pragma omp atomic read

v = x;

- Atomic can protect stores

pragma omp atomic write

x = expr;

- Atomic can protect updates to a storage location (this is the default behavior ... i.e. when you don't provide a clause)

pragma omp atomic update

x++; or ++x; or x--; or --x; or

x binop= expr; or x = x binop expr;

This is the
original OpenMP
atomic

The OpenMP 3.1 Atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:

```
# pragma omp atomic capture  
statement or structured block
```

- Where the statement is one of the following forms:

```
v = x++;    v = ++x;    v = x--;    v = --x;    v = x binop expr;
```

- Where the structured block is one of the following forms:

{v = x; x binop = expr;}	{x binop = expr; v = x;}
{v=x; x=x binop expr;}	{X = x binop expr; v = x;}
{v = x; x++;}	{v=x; ++x;}
{++x; v=x;}	{x++; v = x;}
{v = x; x--;}	{v= x; --x;}
{--x; v = x;}	{x--; v = x;}

The capture semantics in atomic were added to map onto common hardware supported atomic operations and to support modern lock free algorithms

**You all know about mutual exclusion
synchronization using the critical construct.**

**There is another (and more flexible) way to do
mutual exclusion synchronization.**

Locks

Synchronization: Lock Routines

- Simple Lock routines:

- A simple lock is available if it is unset.
 - `omp_init_lock()`, `omp_set_lock()`,
`omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

A lock implies a memory fence (a “flush”) of all thread visible variables

- Nested Locks

- A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
 - **`omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`,
`omp_test_nest_lock()`, `omp_destroy_nest_lock()`**

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Locks with hints were added in OpenMP 4.5 to suggest a lock strategy based on intended use (e.g. contended, uncontended, speculative, unspeculative)

Synchronization: Simple Locks Example

- Count odds and evens in an input array(x) of N random values.

```
int i, ix, even_count = 0, odd_count = 0;
```

```
omp_lock_t odd_lck, even_lck;
```

```
omp_init_lock(&odd_lck);
```

```
omp_init_lock(&even_lck);
```

One lock per case ... even and odd

```
#pragma omp parallel for private(ix) shared(even_count, odd_count)
```

```
for(i=0; i<N; i++){
```

```
    ix = (int) x[i]; //truncate to int
```

```
    if(((int) x[i])%2 == 0) {
```

```
        omp_set_lock(&even_lck);
```

```
        even_count++;
```

```
        omp_unset_lock(&even_lck);
```

```
    }
```

```
    else{
```

```
        omp_set_lock(&odd_lck);
```

```
        odd_count++;
```

```
        omp_unset_lock(&odd_lck);
```

```
    }
```

```
}
```

```
omp_destroy_lock(&odd_lck);
```

```
omp_destroy_lock(&even_lck);
```

```
}
```

Enforce mutual exclusion updates,
but in parallel for each case.

Free-up storage when done.

Exercise

- In the file hist.c, we provide a program that generates a large array of random numbers and then generates a histogram of values.
- The program uses our own random number generator so build the code as:

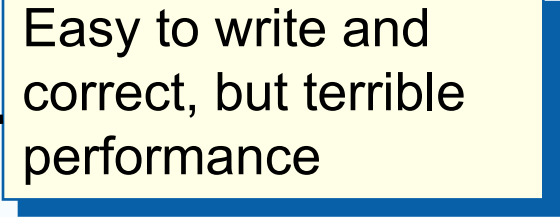
```
gcc -fopenmp hist.c random.c -lm
```

- This is a "quick and informal" way to test a random number generator ... if all goes well the bins of the histogram should be the same size.
- Parallelize the filling of the histogram You must assure that your program is race free and gets the same result as the sequential program.
- Using any of the OpenMP constructs we've discussed, try to minimize the time to generate the histogram.
- Time ONLY the assignment to the histogram. Can you beat the sequential time?

Histogram Program: Critical section

- A critical section means that only one thread at a time can update a histogram bin ... but this effectively serializes the loops and adds huge overhead as the runtime manages all the threads waiting for their turn for the update.

```
#pragma omp parallel for  
for(i=0;i<NVALS;i++){  
    ival = (int) x[i];  
    #pragma omp critical  
    hist[ival]++;  
}
```



Easy to write and
correct, but terrible
performance

Histogram program: one lock per histogram bin

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);    hist[i] = 0;
}
#pragma omp parallel for
for(i=0;i<NVALS;i++){
    ival = (int) x[i];
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}

#pragma omp parallel for
for(i=0;i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Histogram program: reduction with an array

- We can give each thread a copy of the histogram, they can fill them in parallel, and then combine them when done

```
#pragma omp parallel for reduction(+:hist[0:Nbins])
for(i=0;i<NVALS;i++){
    ival = (int) x[i];
    hist[ival]++;
}
```

Easy to write and correct, Uses a lot of memory on the stack, but its fast ... sometimes faster than the serial method.

sequential	0.0019 secs
critical	0.079 secs
Locks per bin	0.029 secs
Reduction, replicated histogram array	0.00097 secs

1000000 random values in X sorted into 50 bins. Four threads on a dual core Apple laptop (Macbook air ... 2.2 Ghz Intel Core i7 with 8 GB memory) and the gcc version 9.1. Times are for the above loop only (we do not time set-up for locks, destruction of locks or anything else)

Multithreading is all about shared address spaces.

We usually treat our hardware as symmetric multiprocessor systems (SMP)

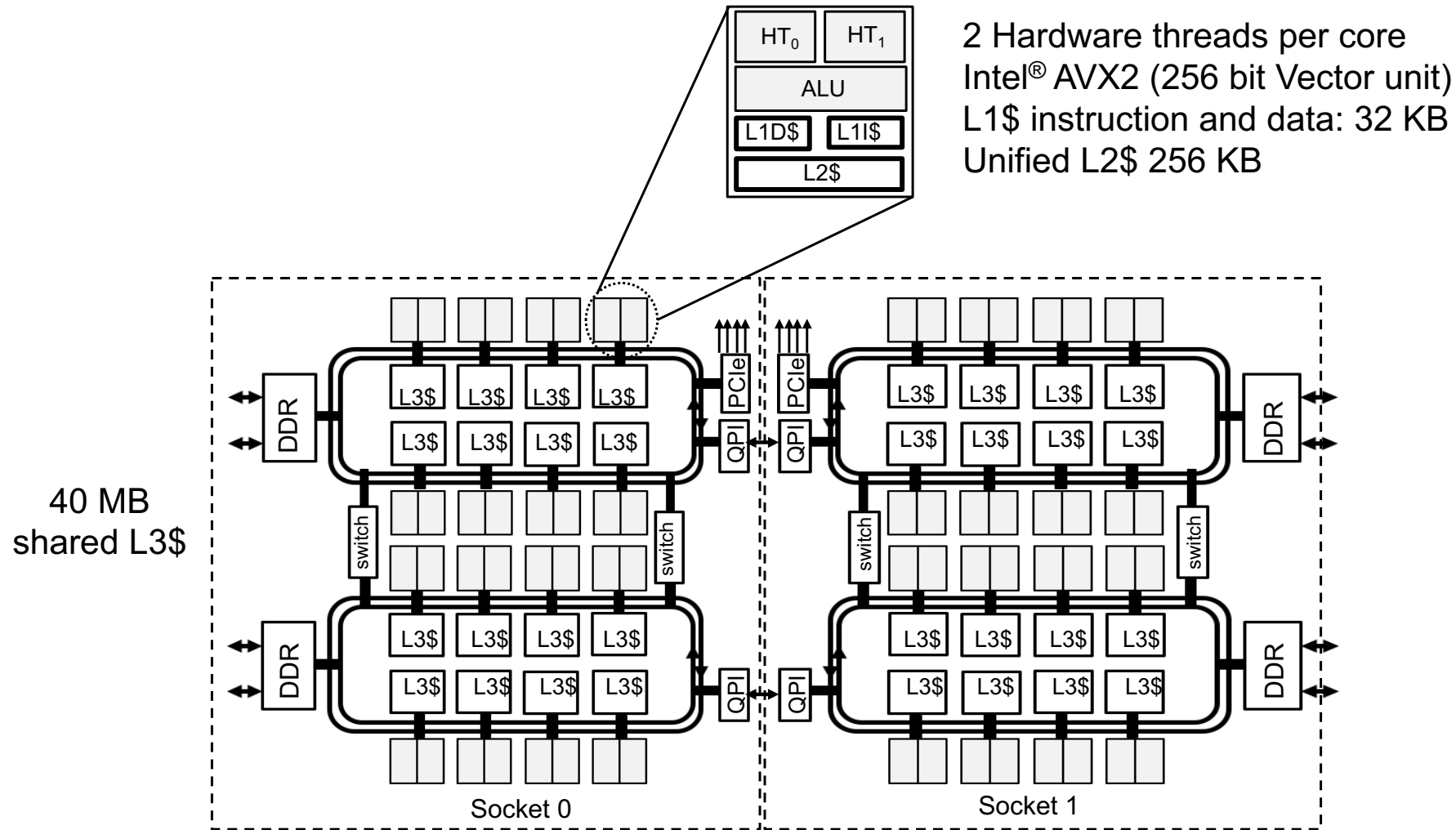
But they aren't ... they are non-uniform Memory systems

The following slides are based on ones from Dr. Helen He of the National Energy Research Scientific Computing Center

Note: None of this is likely to matter for your laptop. On an HPC server in a cluster, however, which a large manycore chip, NUMA effects can be huge.

A Typical CPU Node in an HPC System

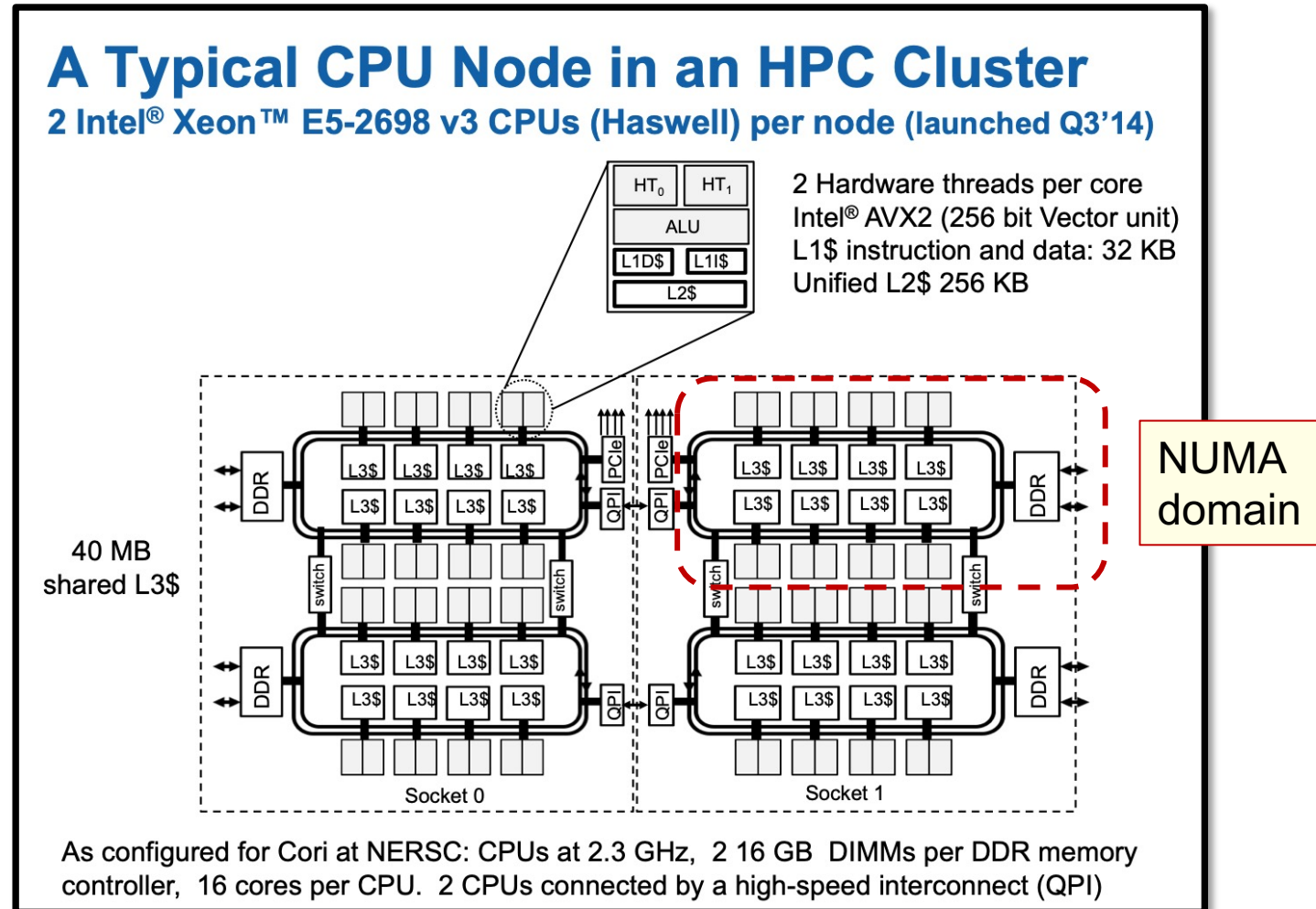
2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)



As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

Does this look like an SMP node to you?

There may be a single address space, but there are multiple levels of non-uniformity to the memory. This is a **Non-Uniform Memory Architecture (NUMA)**



Even a single CPU is properly considered a NUMA architecture

Process / Thread / Memory Affinity

- **Process Affinity**: also called "CPU pinning", binds processes (MPI tasks, etc.) to a CPU or a range 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 cores for threads (i.e., it is generally a bad idea to have more threads than cores).

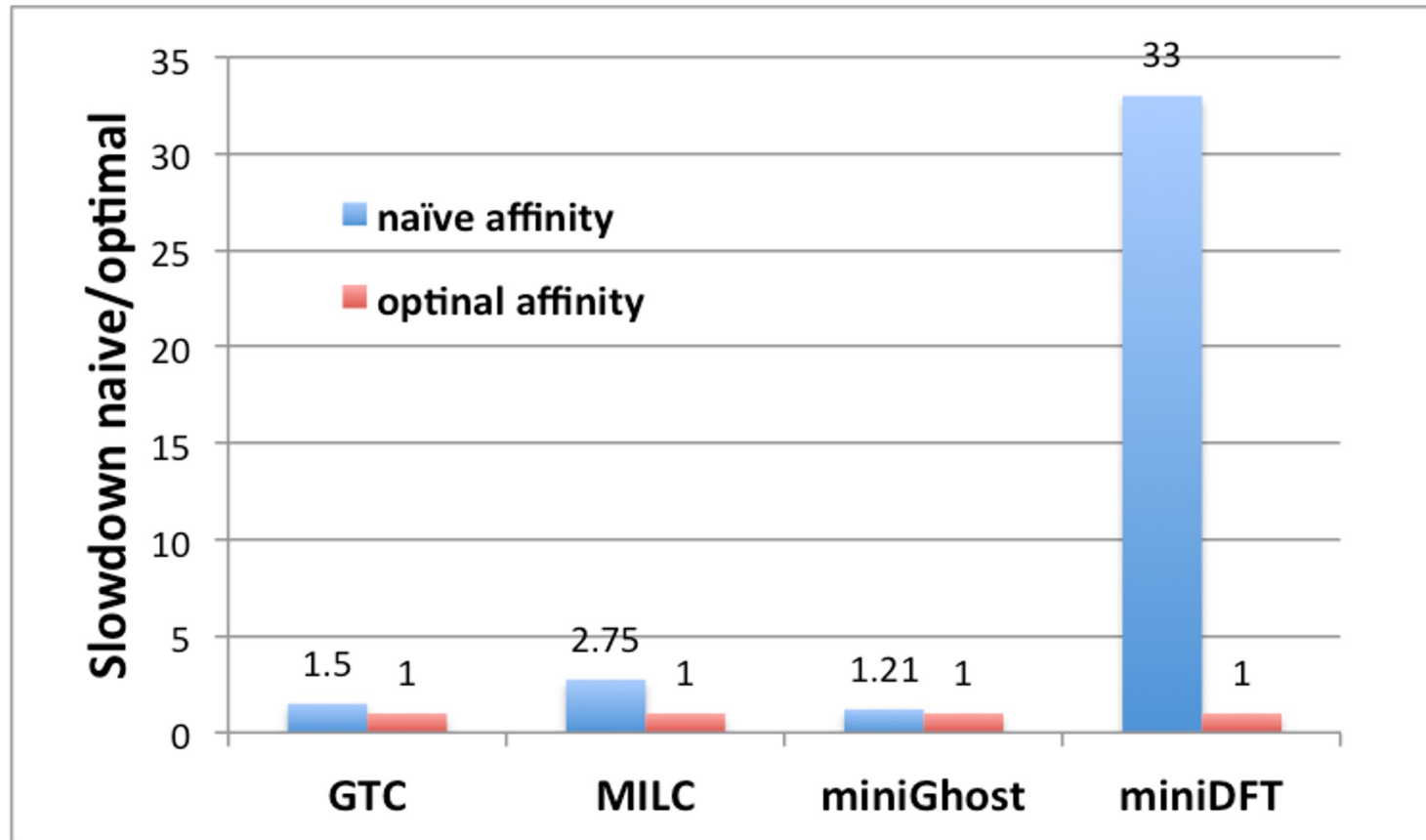
Process affinity is controlled by how you setup your MPI hostfile (something we'll discuss later)

Process / Thread / Memory Affinity

- **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**
- **Cache Locality**: reuse data in cache 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.

Naïve vs. Optimal Affinity

Application Benchmark Performance on Cori at NERSC (the node shown a few slides back)



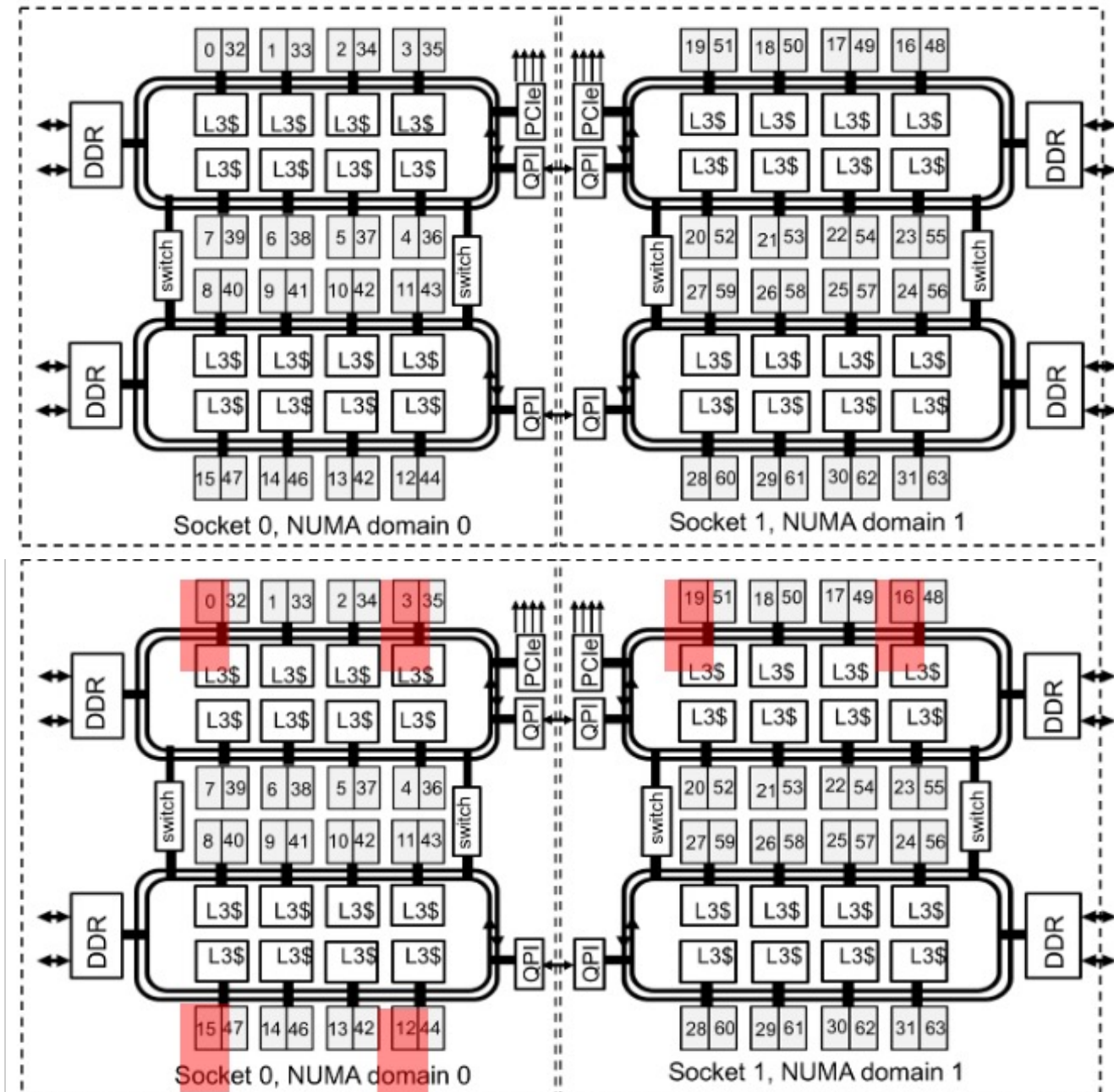
Lower is better

The Concept of Places

- The Operating System assigns logical CPU IDs to hardware threads.
- The linux command ***numactl -H*** returns those numbers.
- A place defines where threads can run
 - > export OMP_PLACES "{0, 3, 15, 12, 19, 16, 28, 31}"
 - > export NUM_THREADS= 6

```
#pragma omp parallel
{
    // do a bunch of cool stuff
}
```

Numactl is not installed by default on most Linux systems. A dedicated HPC system, will most likely have it installed



The Concept of Places

- The Operating System assigns logical CPU IDs to hardware threads.
- The linux command ***numactl -H*** returns those numbers.

Programmers can use OMP_PLACES for detailed control over the execution-units threads utilize. BUT ...

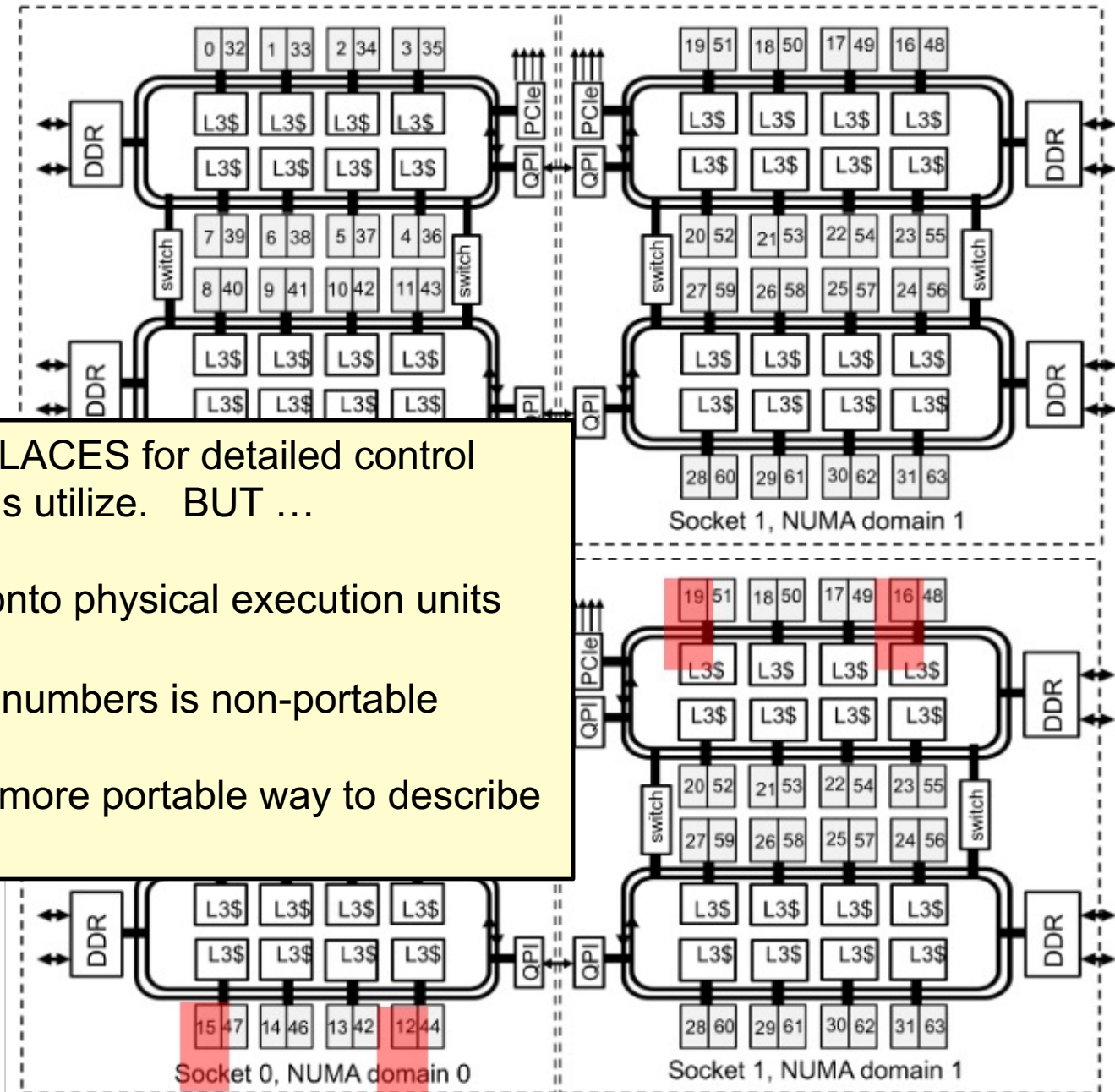
- A place defines where a task is executed
 - > export OMP_PLACES
 - > export NUM_THREADS
- The rules for mapping onto physical execution units are complicated.
- PLACES expressed as numbers is non-portable

```
#pragma omp parallel
{
```

```
// do a bunch of cool stuff
```

}

Numactl is not installed by default on most Linux systems. A dedicated HPC system, will most likely have it installed



Hardware Abstraction: 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`

Mapping Strategy: OMP_PROC_BIND

- 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 primary* thread
 - **spread**: bind threads as evenly distributed (spreaded) as possible
 - **primary**: bind threads to the same place as the primary thread
- The values **primary***, **close**, and **spread** imply the value **true**

Examples:

```
export OMP_PROC_BIND=spread  
export OMP_PROC_BIND=spread,close
```

This is for nested parallel regions ... the outermost is "spread" and the nested region is "close"

*Primary thread: this is the thread with ID=0 that encountered the parallel construct and created the team of threads

Mapping Strategy: OMP_PROC_BIND

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	

- **primary**: bind threads to the same place as the primary 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`

OMP_PROC_BIND Choices for STREAM Benchmark

OMP_NUM_THREADS=32

OMP_PLACES=threads

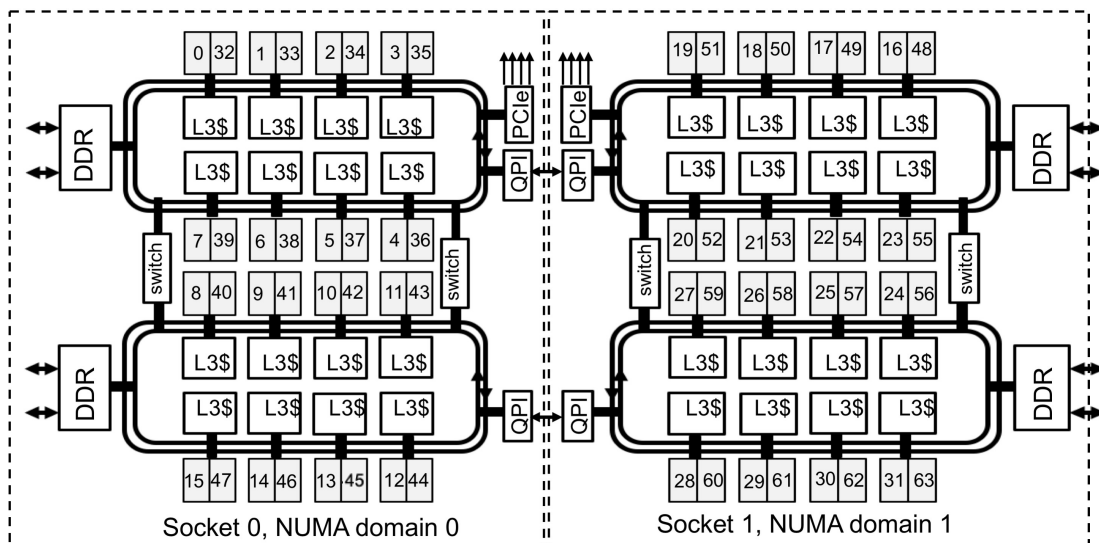
OMP_PROC_BIND=close

Threads 0 to 31 bind to cores

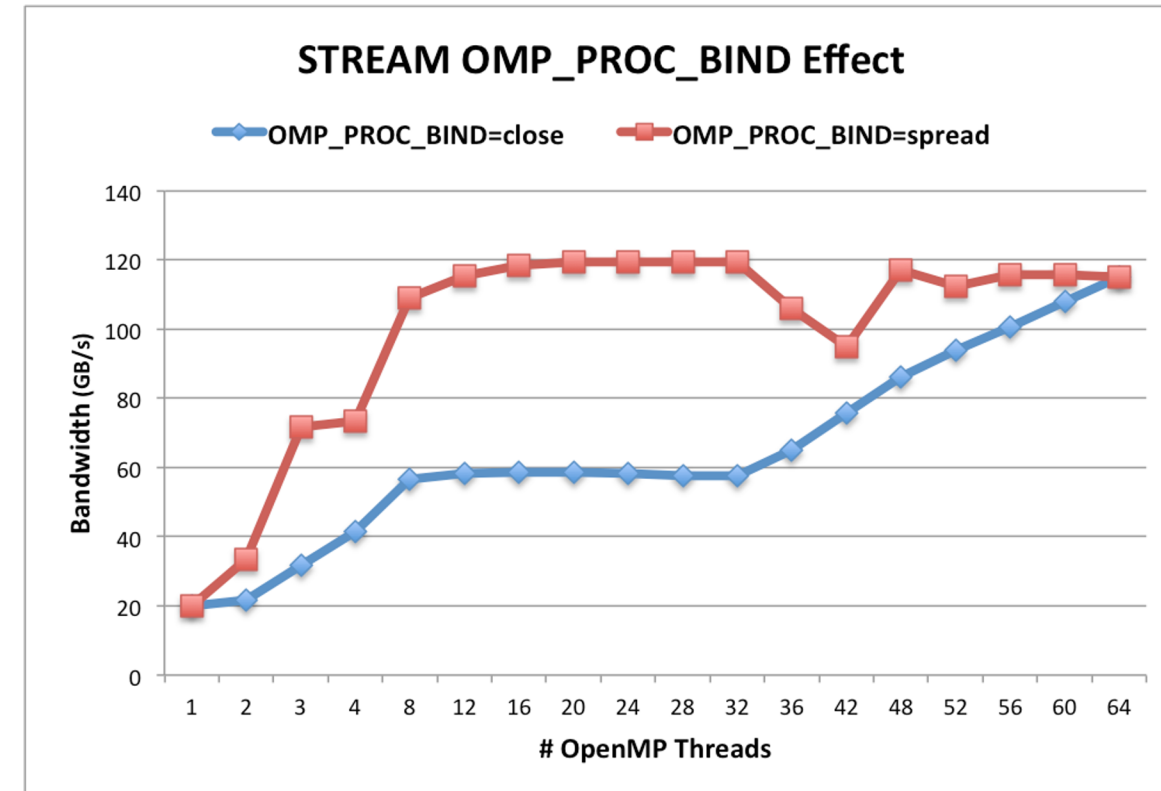
(0,32),(1,33),(2,34),... (15,47). All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread

Threads 0 to 31 bind to cores 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



Stream is a well known memory bandwidth benchmark based on simple vector operations on huge vectors

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;}
```

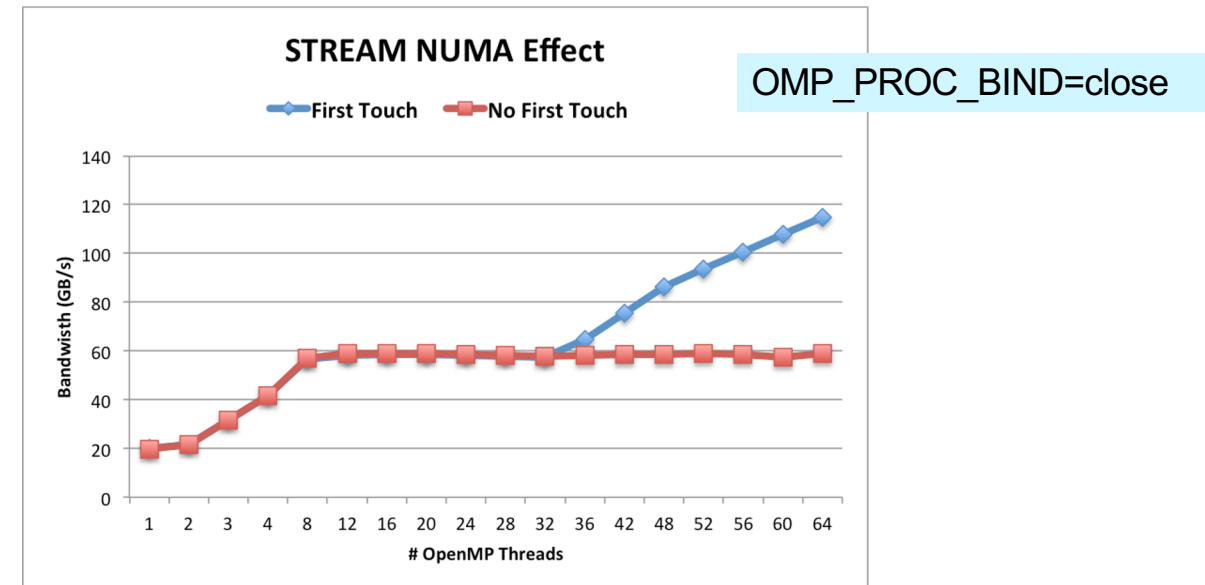
Step 2 Compute

```
#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.

Red: step 1.1 + step 2. No First Touch

Blue: step 1.2 + step 2. First Touch



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 not straightforward
 - 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 affinity first or check with OMP_DISPLAY_AFFINITY
 - For nested OpenMP, set OMP_PROC_BIND=spread,close is recommended
- Optimize code for memory affinity
 - Pay special attention to avoid false sharing
 - Exploit first touch data policy, or use at least 1 MPI task per NUMA domain
 - Optimize code for cache locality
 - Compare performance with put threads close or far apart (spread)
 - Use omp_allocator
 - Use numactl -m option to explicitly request memory allocation in specific NUMA domain (such as high bandwidth memory in KNL)