The OpenMP* Common Core:
A hands-on Introduction

Tim Mattson
Intel Corp.

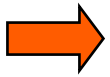
Yun (Helen) He
Berkeley Lab

Alice Koniges
Univ. of Hawai'i

David Eder
Univ. of Hawai'i

Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/OmpCommonCore.git

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

OpenMP* Overview

```
C$OMP FLUSH
#pragma omp critical
#pragma omp single
C$OMP THREADPRIVATE(/ABC/)
C$OMP ATOMIC
CALL OMP_SET_NUM_THREADS(10)
```

OpenMP: An API for Writing Parallel Applications

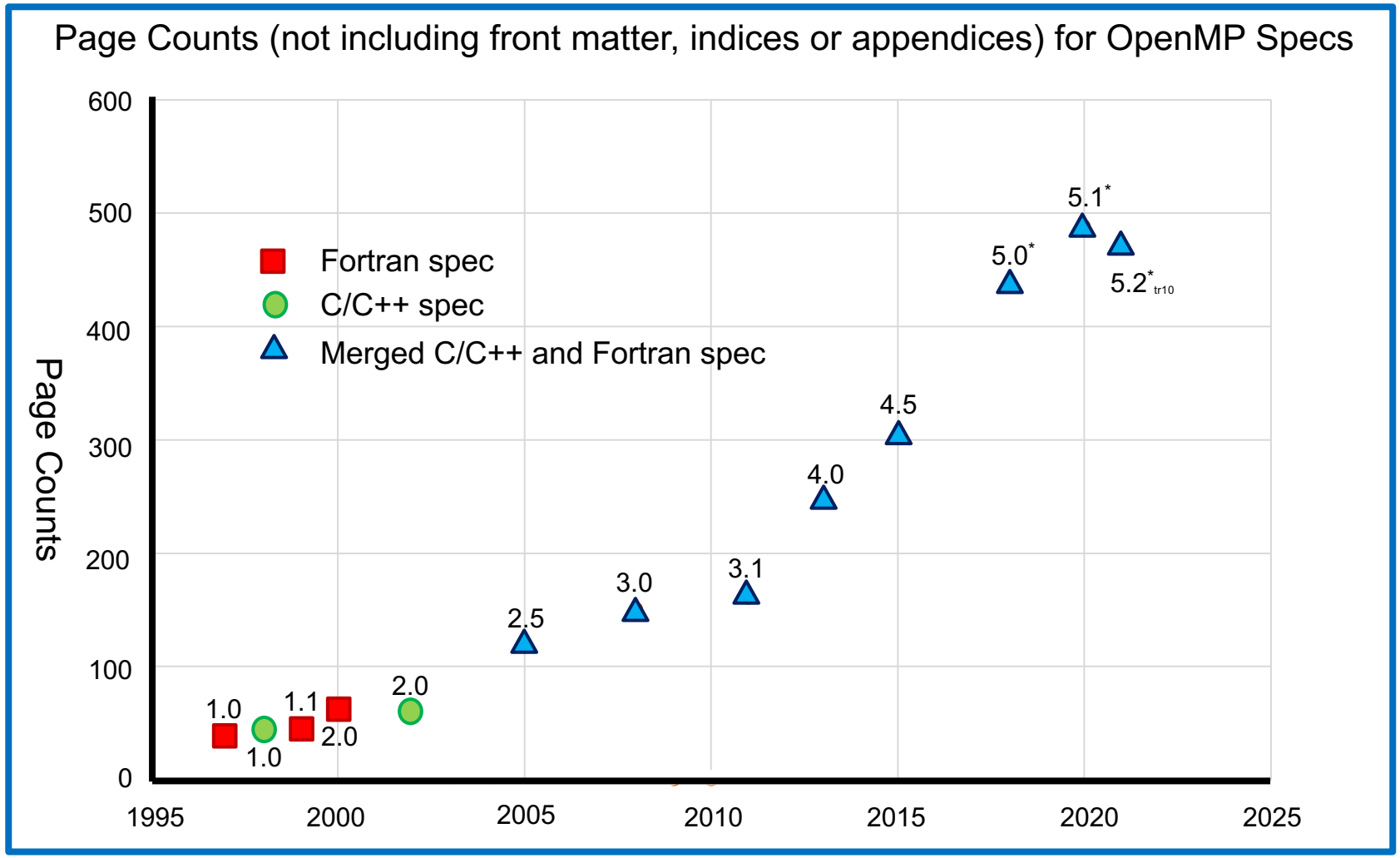
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Also supports non-uniform memories, vectorization and GPU programming

```
#pragma omp parallel for private(A, B)
C$OMP PARALLEL REDUCTION (+: A, B)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
#pragma omp atomic seq_cst
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers

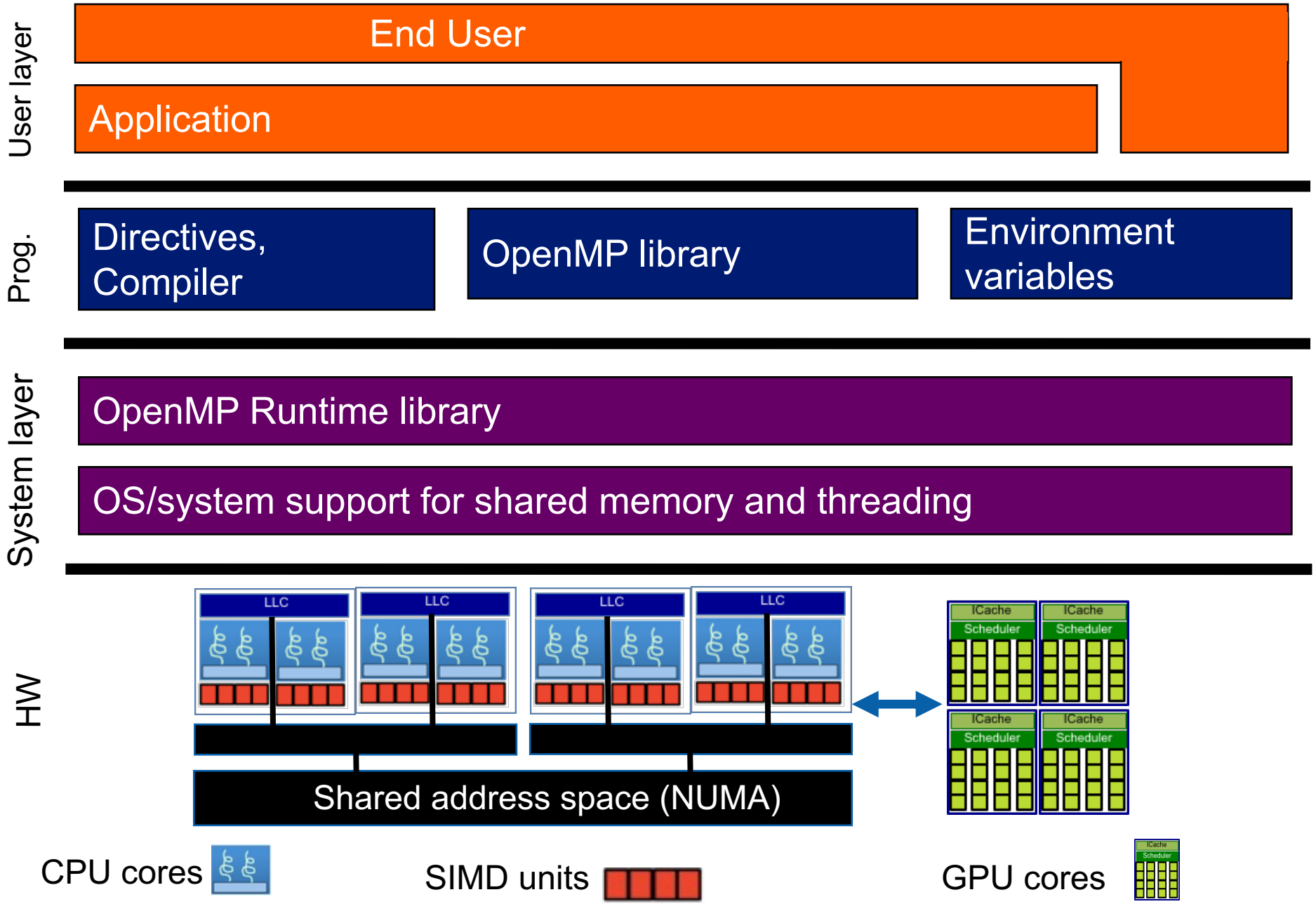


The full spec is overwhelming. We focus on the Common Core: the 21 items most people restrict themselves to

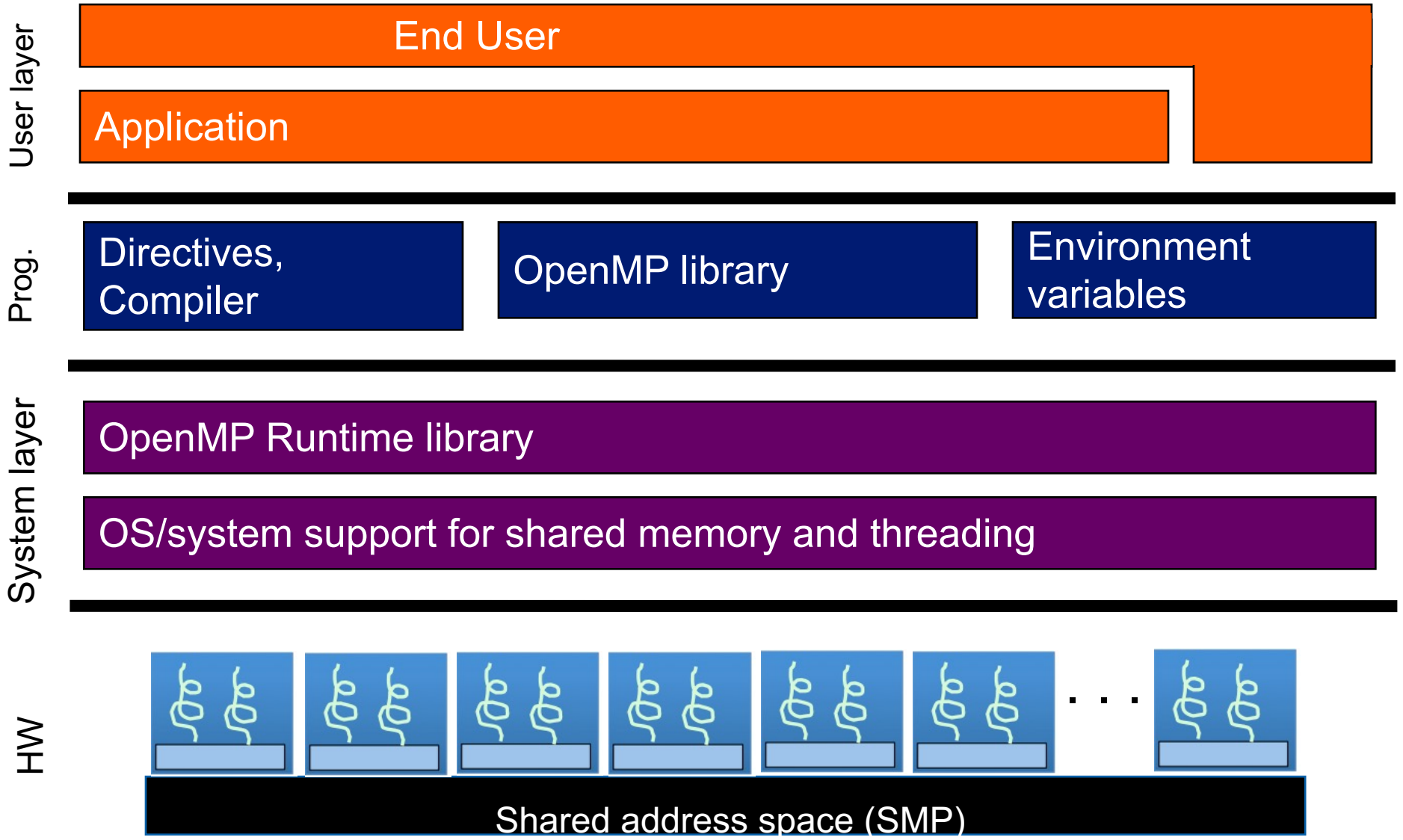
The OpenMP Common Core: Most OpenMP programs only use these 21 items

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.

OpenMP Basic Definitions: Basic Solution Stack



OpenMP Basic Definitions: Basic Solution Stack



For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case ….
i.e., lots of threads with “equal cost access” to memory

OpenMP Basic Syntax

- Most of the constructs in OpenMP are compiler directives.

C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause]…]	***!$OMP construct [clause [clause] …]***
Example	
#pragma omp parallel private(x) ***{*** ***}***	***!$OMP PARALLEL PRIVATE(X)*** ***!$OMP END PARALLEL***
Function prototypes and types:	
#include <omp.h>	***use OMP_LIB***

- Most OpenMP constructs apply to a “structured block”.
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

Exercise, Part A: Hello World

Verify that your environment works

- Write a program that prints “hello world”.

git clone https://github.com/tgmattso/OpenMPCommonCore.git

```
#include<stdio.h>
int main()
{




    printf(“ hello ”);
    printf(“ world \n”);

}
```

- To download the slides:

https://github.com/tgmattso/OmpCommonCore.git

Exercise, Part B: Hello World

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

git clone https://github.com/tgmattso/OpenMP_Common_Core.git

```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {


    printf(“ hello ”);
    printf(“ world \n”);
 }
}
```

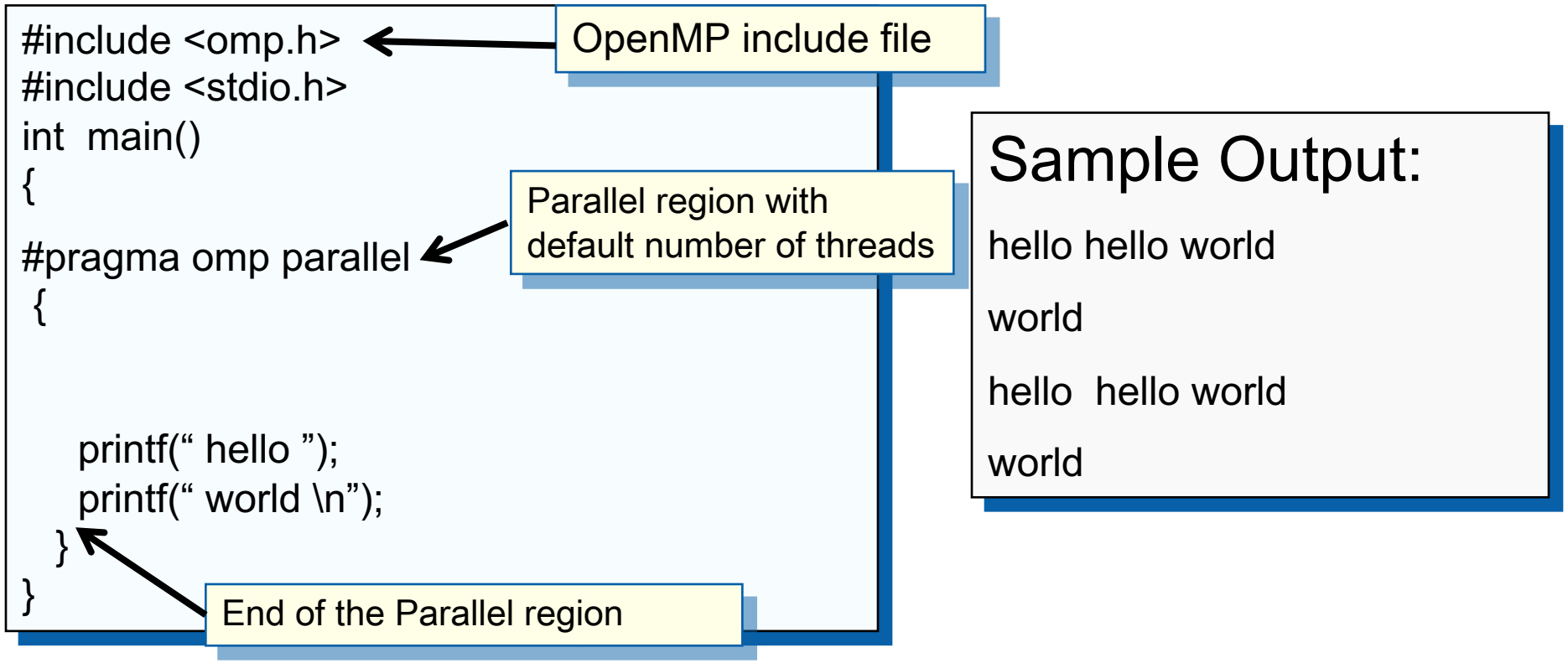
Switches for compiling and linking

gcc -fopenmp	**Gnu (Linux, OSX)**
cc -qopenmp	**Intel (Linux@NERSC)**
icl /Qopenmp	**Intel (windows)**
icc -fopenmp	**Intel (Linux, OSX)**

Solution

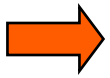
A Multi-Threaded “Hello World” Program

- Write a multithreaded program where each thread prints “hello world”.



The statements are interleaved based on how the operating schedules the threads

Outline

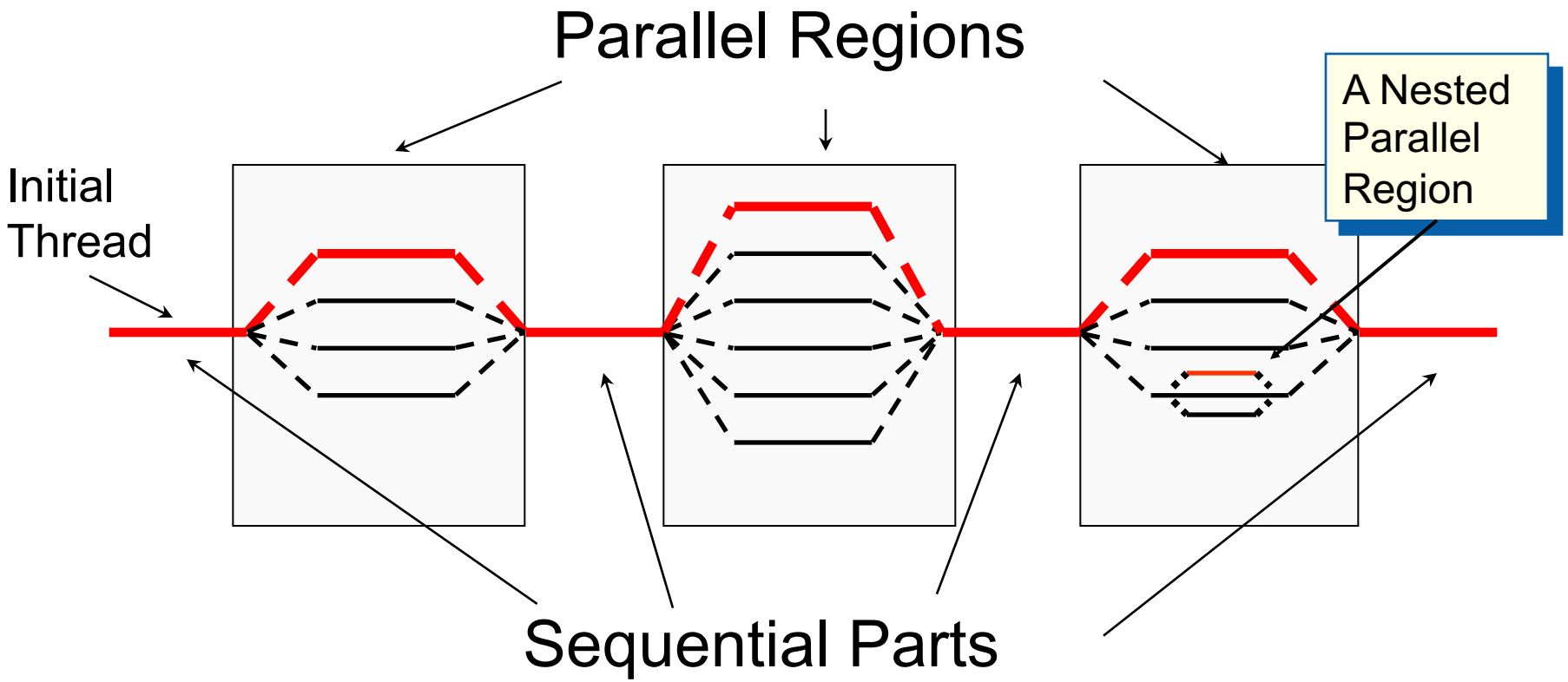


- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

OpenMP Execution model:

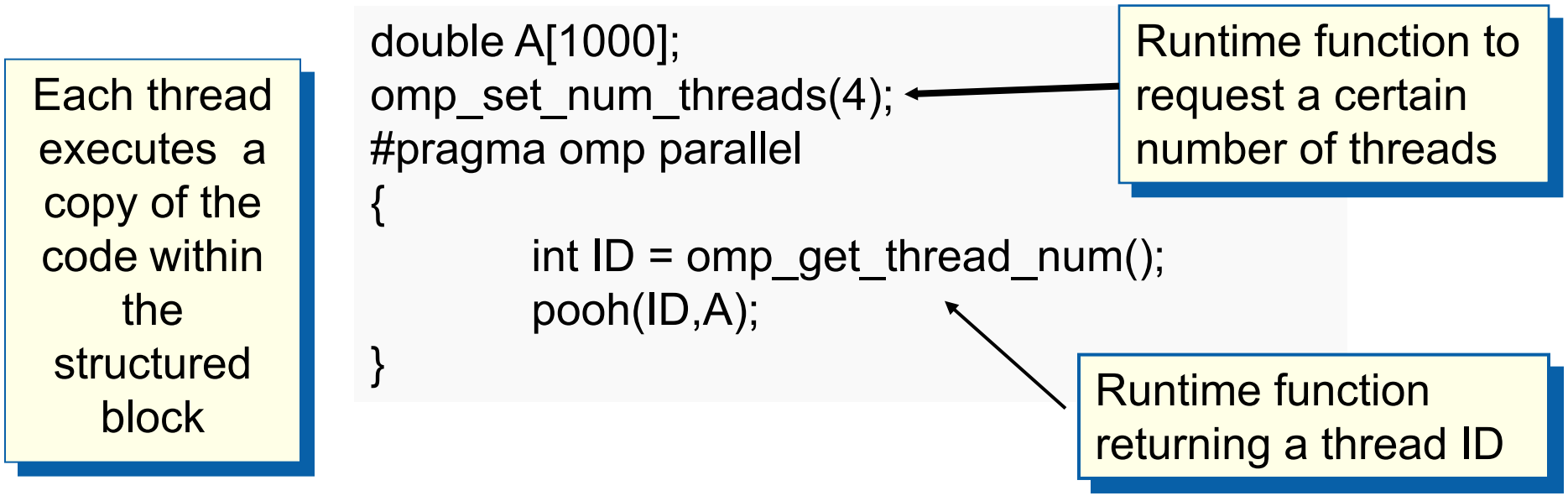
Fork-Join Parallelism:

- Initial thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, to create a 4 thread Parallel region:

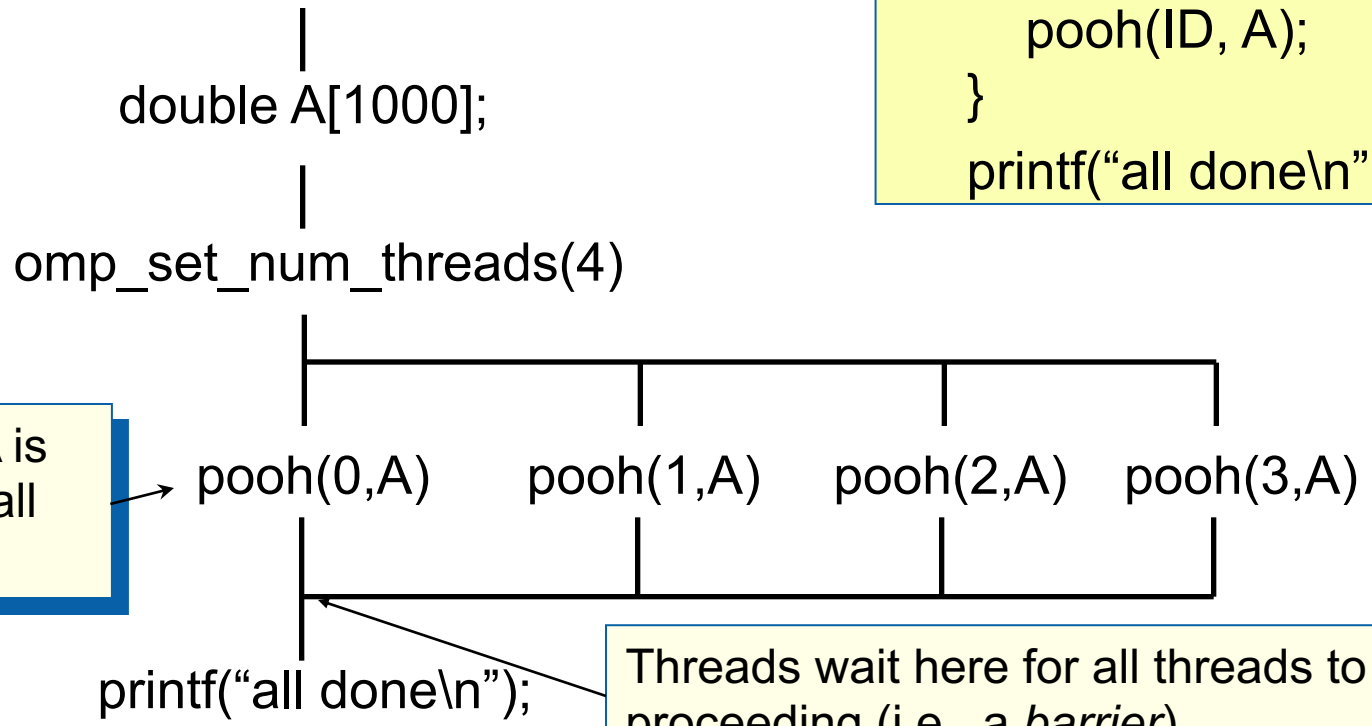


- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e., a *barrier*)

Thread creation: How many threads did you actually get?

- Request a number of threads with omp_set_num_threads()
- The number requested may not be the number you actually get.
  - An implementation may silently give you fewer threads than you requested.
  - Once a team of threads has launched, it will not be reduced.

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID      = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

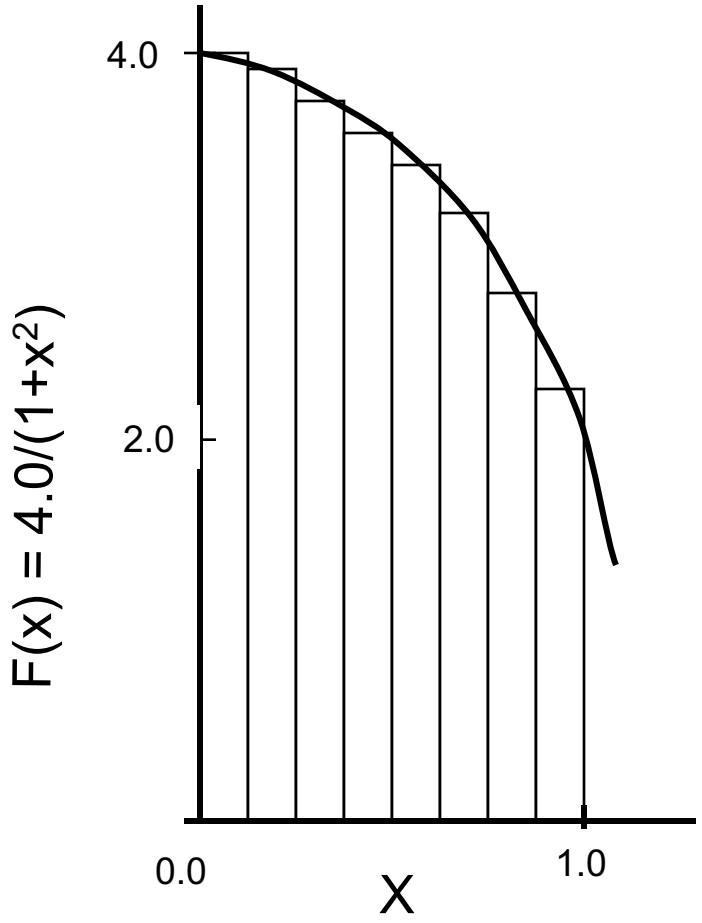
Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

- Each thread calls pooh(ID,A) for ID = 0 to nthrds-1

An Interesting Problem to Play With

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x = \Delta x \sum_{i=0}^{N} F(x_i) \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

See ParProgCourse/Exercises/OpenMP/pi.c

Serial PI Program

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{       int i;      double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;
        double tdata = omp_get_wtime();
        for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
        tdata = omp_get_wtime() - tdata;
        printf(" pi = %f in %f secs\n",pi, tdata);
}
```

The library routine get_omp_wtime() is used to find the elapsed "wall time" for blocks of code

See ParProgCourse/Exercises/OpenMP/pi.c

Exercise: the Parallel Pi Program

- Create a parallel version of the pi program using a parallel construct:

  #pragma omp parallel

- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads();

Number of threads in the team

Thread ID or rank

Time in seconds since a fixed point in the past

Request a number of threads in the team

Hints: the Parallel Pi Program

- Use a parallel construct:

  #pragma omp parallel

- The challenge is to:
  - divide loop iterations between threads (use the thread ID and the number of threads).
  - Create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

- In addition to a parallel construct, you will need the runtime library routines
  - int omp_set_num_threads();
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Example: A simple SPMD pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
     {
            int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
              }
      }
     for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
            }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

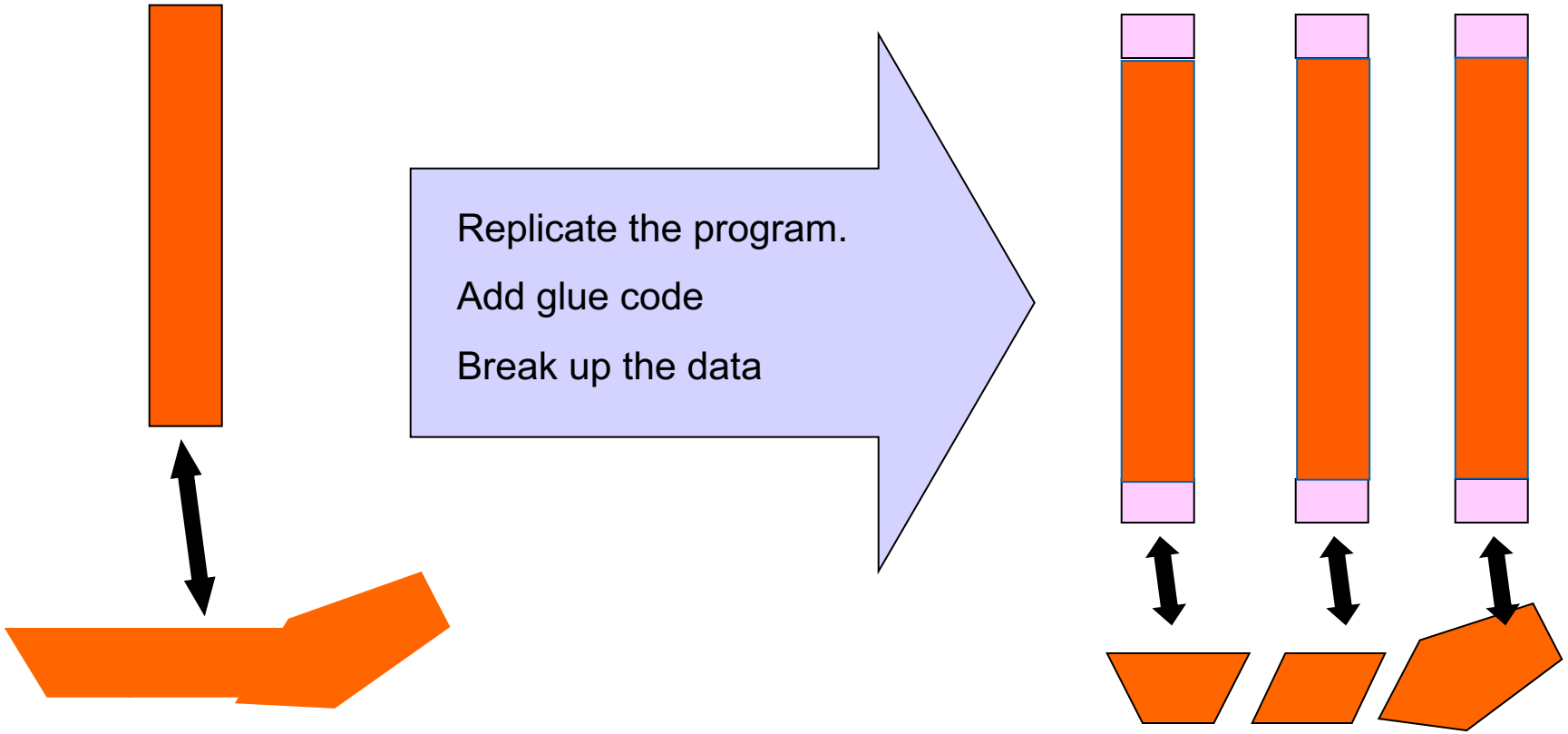
threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

*SPMD: Single Program Multiple Data

SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.

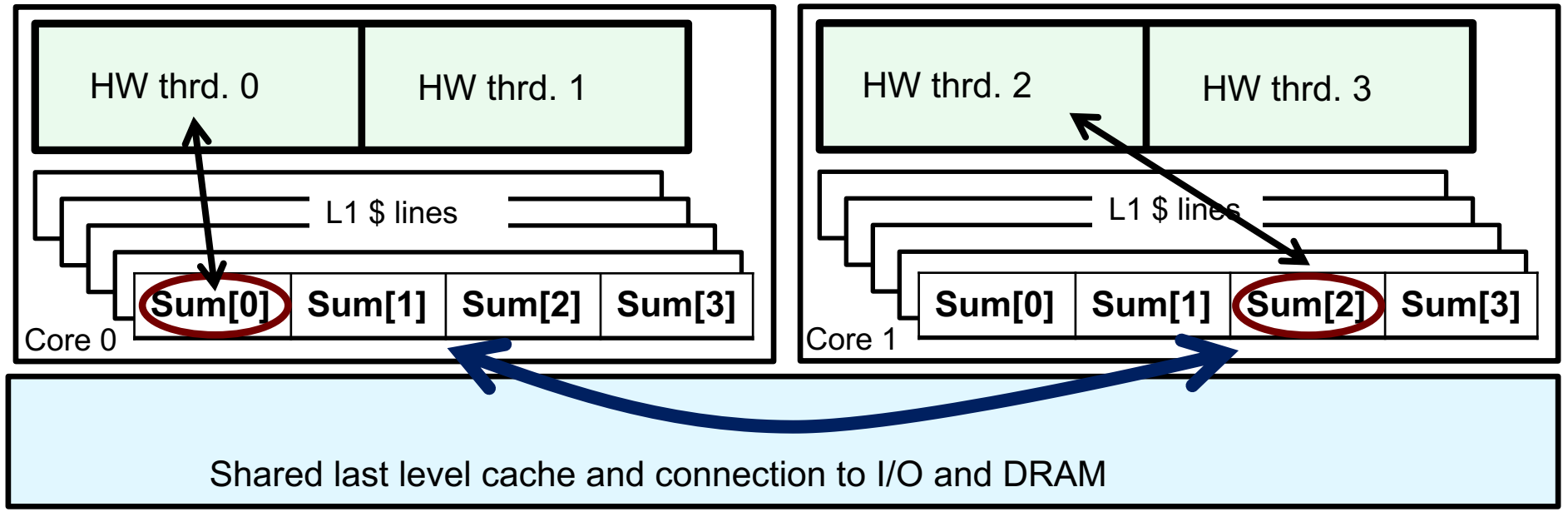


- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Why Such Poor Scaling? False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads … This is called **“false sharing”**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
#define PAD 8        // assume 64 byte L1 cache line size
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS][PAD] ;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel
     {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)    nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               sum[id][0] += 4.0/(1.0+x*x);
          }
     }
     for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

Results*: PI Program, Padded Accumulator

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
#define PAD 8        // assume 64 byte L1 cache line size
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS][PAD] ;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel
     {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)    nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               sum[id][0] += 4.0/(1.0+x*x);
          }
     }
     for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Four Ways to Request a Number of Threads for a Parallel Region

1. The system has an Internal Control Variable (ICV) that defines the default number of threads to request for a parallel region.

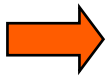
2. When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate internal control variable to the value of OMP_NUM_THREADS
   - For example, to set the default number of threads on my apple laptop
     - export OMP_NUM_THREADS=12

3. The omp_set_num_threads() runtime function overrides the value from the environment and resets the ICV to a new value.

4. A clause on the parallel construct requests a number of threads for that parallel region, but it does not change the ICV
   - #pragma omp parallel num_threads(4)

Order of Precedence for setting the requested number of threads: (4) > (3) > (2) > (1)

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization included in the common core:
  - critical
  - barrier

- Other, more advanced, synchronization operations:
  - atomic
  - ordered
  - flush
  - locks (both simple and nested)

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

Threads wait their turn – only one thread at a time calls consume()

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     B =  big_SPMD_job(id, nthrds);
     #pragma omp critical
          res += consume (B);

}
```

Synchronization: barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code … it is an executable statement.

```
double Arr[8], Brr[8];          int numthrds;

omp_set_num_threads(8)

#pragma omp parallel

{    int id, nthrds;

     id = omp_get_thread_num();

     nthrds = omp_get_num_threads();

     if (id==0) numthrds = nthrds;

     Arr[id] = big_ugly_calc(id, nthrds);

#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

Threads wait until all threads hit the barrier. Then they can go on.

PI Program with False Sharing

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               sum[id] += 4.0/(1.0+x*x);
          }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Recall that promoting sum to an array made the coding easy, but led to false sharing and poor performance.

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
       int i, id, nthrds;    double x, sum;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
        if (id == 0)   nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
              x = (i+0.5)*step;
              sum += 4.0/(1.0+x*x);
         }
        #pragma omp critical
                pi += sum * step;
   }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Results*: pi program critical section

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
       int i, id, nthrds;    double x, sum;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
        if (id == 0)   nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
              x = (i+0.5)*step;
              sum += 4.0/(1.0+x*x);
         }
        #pragma omp critical
                pi += sum * step;
   }
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
       int i, id, nthrds;    double x, sum;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
        if (id == 0)   nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
              x = (i+0.5)*step;
              #pragma omp critical
                   sum += 4.0/(1.0+x*x);
         }
    }
}
```

What would happen if you put the critical section inside the loop?

Synchronization

Synchronization is used to impose order constraints between threads and to protect access to shared data

- High level synchronization included in the common core:
  - critical
  - barrier

Covered earlier

- Other, more advanced, synchronization operations:
  - atomic
  - ordered
  - flush
  - locks (both simple and nested)

Covered in this section

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

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

Synchronization: Lock Routines

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

- 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()

- 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);

#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);
}
```

One lock per case … even and odd

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.

- 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 everything we’ve covered today, manage updates to shared data in two different ways. 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.

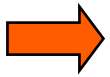
```
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
       omp_init_lock(&hist_locks[i]);    hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
     ival = (int)  x[i];
     omp_set_lock(&hist_locks[ival]);
          hist[ival]++;
     omp_unset_lock(&hist_locks[ival]);
   }

 #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.

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

The Loop Worksharing Construct

- 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);
        }
}
```

Loop construct name:

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

The loop control index I is made "private" to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop

Loop Worksharing Construct

A motivating example

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

(SPMD Pattern)

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * (N / Nthrds)-1;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

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.
- Example:
  - #pragma omp for schedule(dynamic, 10)

Schedule Clause	When To Use
STATIC	**Pre-determined and predictable by the programmer**
DYNAMIC	**Unpredictable, highly variable work per iteration**

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

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).

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

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
AUTO	When the runtime can “learn” from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

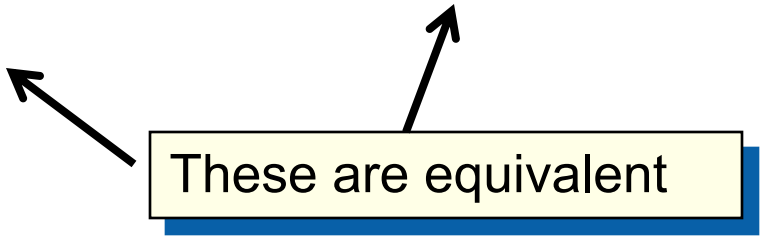
Most work at runtime : complex scheduling logic used at run-time

Combined Parallel/Worksharing Construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

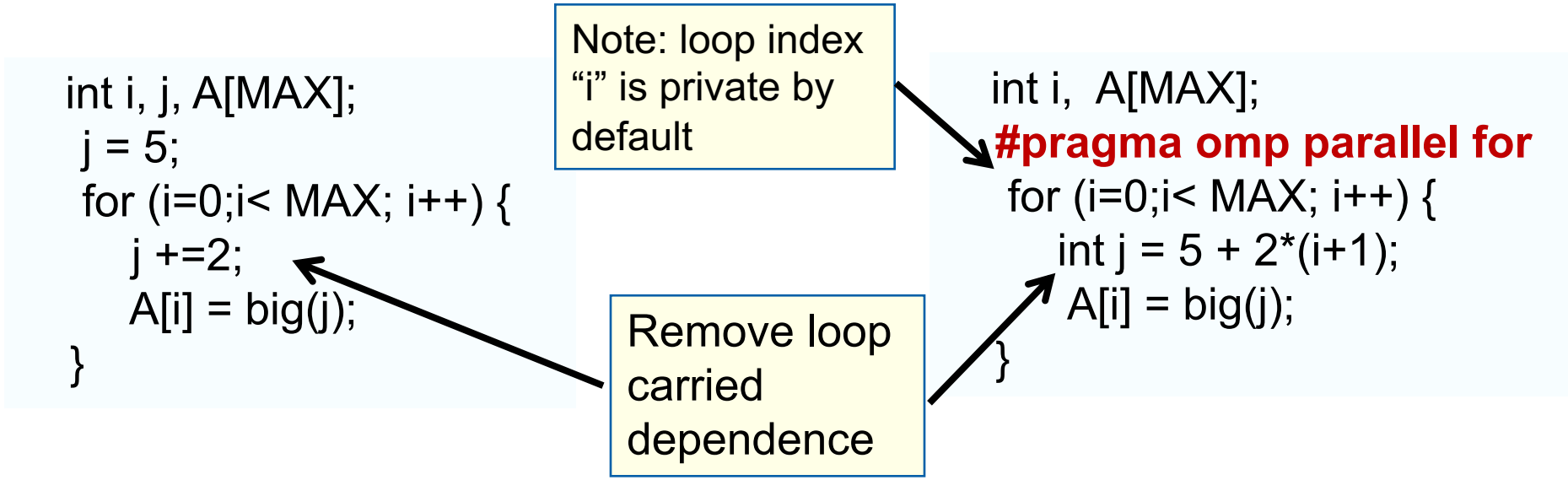
```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```



Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test



Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];
int i;
for (i=0;i< MAX; i++) {
      ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed.
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

- The variables in “list” must be shared in the enclosing parallel region.

```
 double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 }
 ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

OpenMP includes user defined reductions and array-sections as reduction variables (we just don't cover those topics here)

Example: PI with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
          double x;
         #pragma omp for reduction(+:sum)
             for (i=0;i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
             }
       }
            pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Example: PI with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{
    double pi, sum = 0.0;
     step = 1.0/(double) num_steps;

     #pragma omp parallel for reduction(+:sum)
     for (int i=0;i< num_steps; i++){
          double x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     pi = step * sum;
}
```

Using modern C style, we put declarations close to where they are used … which lets me use the parallel for construct.

Results*: PI with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a

```
#include <omp.h>
static long num_steps = 1000
void main ()
{   int i;          double x, pi, su
    step = 1.0/(double) num_s
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
            for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
            }
    }
        pi = step * sum;
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

The nowait clause

- Barriers are really expensive. You need to understand when they are implied and how to skip them when it's safe to do so.

```
double A[big], B[big], C[big];

#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

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

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 NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Sections Worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{
   #pragma omp sections
   {
   #pragma omp section
            x_calculation();
   #pragma omp section
             y_calculation();
   #pragma omp section
             z_calculation();
   }
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

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;

#pragma omp parallel for reduction(+:a[0:N]) private(j)
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;
```

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

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)

Exercise

- Go back to your parallel mandel.c program.
- Using what we've learned in this block of slides can you improve the runtime?

Optimizing mandel.c

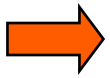
```
  wtime = omp_get_wtime();
#pragma omp parallel for collapse(2) schedule(runtime) firstprivate(eps) private(j,c)
  for (i=0; i<NPOINTS; i++) {
    for (j=0; j<NPOINTS; j++) {
      c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
      c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
      testpoint(c);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

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

Data Environment: Default storage attributes

- Shared memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

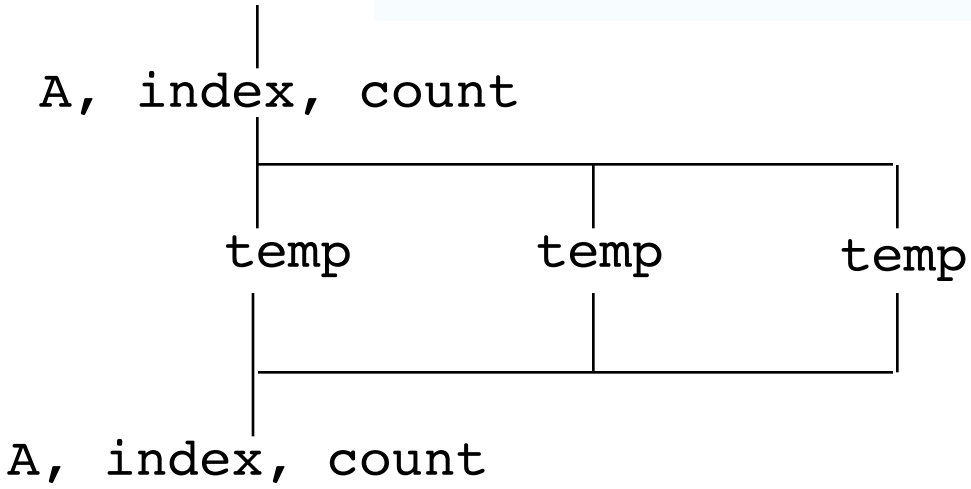
Data Sharing: Examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
    work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Data Sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses* (note: ***list*** is a comma-separated list of variables)
  - shared(list)
  - private(list)
  - firstprivate(list)
- These can be used on parallel and for constructs … other than shared which can only be used on a parallel construct

- Force the programmer to explicitly define storage attributes
  - default (none)

default() can only be used on parallel constructs

Data Sharing: Private clause

- private(var) creates a new local copy of var for each thread.

```
int N = 1000;
extern void init_arrays(int N, double *A, double *B, double *C);


void example () {
    int i, j;
    double A[N][N], B[N][N], C[N][N];
    init_arrays(N, *A, *B, *C);


    #pragma omp parallel for private(j)
    for (i = 0; i < 1000; i++)
        for( j = 0; j<1000; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

OpenMP makes the loop control index on the parallel loop (i) private by default … but not for the second loop (j)

Data Sharing: Private clause

- private(var) creates a new local copy of var for each thread.
  - The value of the private copies is uninitialized
  - The value of the original variable is unchanged after the region

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
            tmp += j;
      printf("%d\n", tmp);
}
```

tmp was not initialized

When you need to refer to the variable tmp that exists prior to the construct, we call it the **original variable**.

tmp is 0 here

Data Sharing: Private and the original variable

- The original variable's value is unspecified if it is referenced outside of the construct
  - Implementations may reference the original variable or a copy ….. a dangerous programming practice!
  - For example, consider what would happen if the compiler inlined work()?

```
int tmp;
void danger() {
      tmp = 0;
#pragma omp parallel private(tmp)
      work();
     printf("%d\n", tmp);
}
```

tmp has unspecified value

```
extern int tmp;
void work() {
      tmp = 5;
}
```

unspecified which copy of tmp

Firstprivate clause

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

Lastprivate clause

At the end of the worksharing region, the original variable for each of the variables in the last private list will be assigned the value from the last iteration of the loop a defined by a sequential execution.

```
#pragma omp for lastprivate(ierr)  {
 for (int i=0; i<N; i++)
      ierr = work(i);
}
```

Each thread gets its own copy of ierr

It is illegal for a single variable to appear in more than one data environment clause with one exception: OpenMP allows you to put the same variables in the firstprivate and lastprivate clauses.

Private clause example with Pi

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
   int i;
   double x, pi, sum = 0.0;
   double start_time, run_time;
   step = 1.0 / (double) num_steps;
   for (i = 1; i <= 4; i++) {
      sum = 0.0;
      omp_set_num_threads(i);
      start_time = omp_get_wtime();
      #pragma omp parallel
      {
         #pragma omp single
            printf(" num_threads = %d", omp_get_num_threads());
         #pragma omp for reduction(+:sum) private(x)
            for (i = 0; i < num_steps; i++){
               x = (i + 0.5) * step;
               sum = sum + 4.0 / (1.0 + x*x);
         }
      }
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("\n pi is %f in %f seconds and %d threads\n", pi, run_time, i);
   }
}
```

Each thread accumulates its local sum that is later combined into the global sum with the reduction operation. Variable x is declared as **private** with a data environment clause.

Data sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are private to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Data Sharing: Default clause

- **default(none)**: `Forces you to define the` storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain. Good programming practice!
- You can put the default clause on parallel and parallel + workshare constructs.

The static extent is the code in the compilation unit that contains the construct.

```
#include <omp.h>
int main()
{
    int i, j=5;      double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

The compiler would complain about j and y, which is important since you don't want j to be shared

The full OpenMP specification has other versions of the default clause, but they are not used very often so we skip them in the common core

Data Sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: **COMMON** blocks
  - C: File scope and static variables, static class members
- Different from making them **PRIVATE**
  - with **PRIVATE** global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using **COPYIN** or at time of definition (using language-defined initialization capabilities)

A Threadprivate Example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

A Threadprivate Example

Counting tasks executions with a threadprivate counter

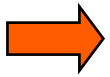
```
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <omp.h>
4  int counter = 0;
5  #pragma omp threadprivate(counter)
6
7  void inc_count(){
8     counter++;
9  }
10
11 int main() {
12    init_list(p);
13    head = p;
14    #pragma omp parallel
15    {
16       #pragma omp single
17       {
18          p = head;
19          while (p) {
20             #pragma omp task firstprivate(p)
21             {
22                inc_count();
23                processwork(p);
24             }
25          p = p->next;
26          }
27       }
28    printf("thread \%d ran \%d tasks\n",omp_get_thread_num(),counter);
29    }
30    freeList(p);
31    return 0;
32 }
```

we will soon learn about omp tasks

This program traverses a linked list in parallel with tasks doing a random amount of work for each node in the list. A **threadprivate** variable is used to keep track of how many tasks were executed by each thread.

Note: we do not provide the functions used for the list nor the list processing.

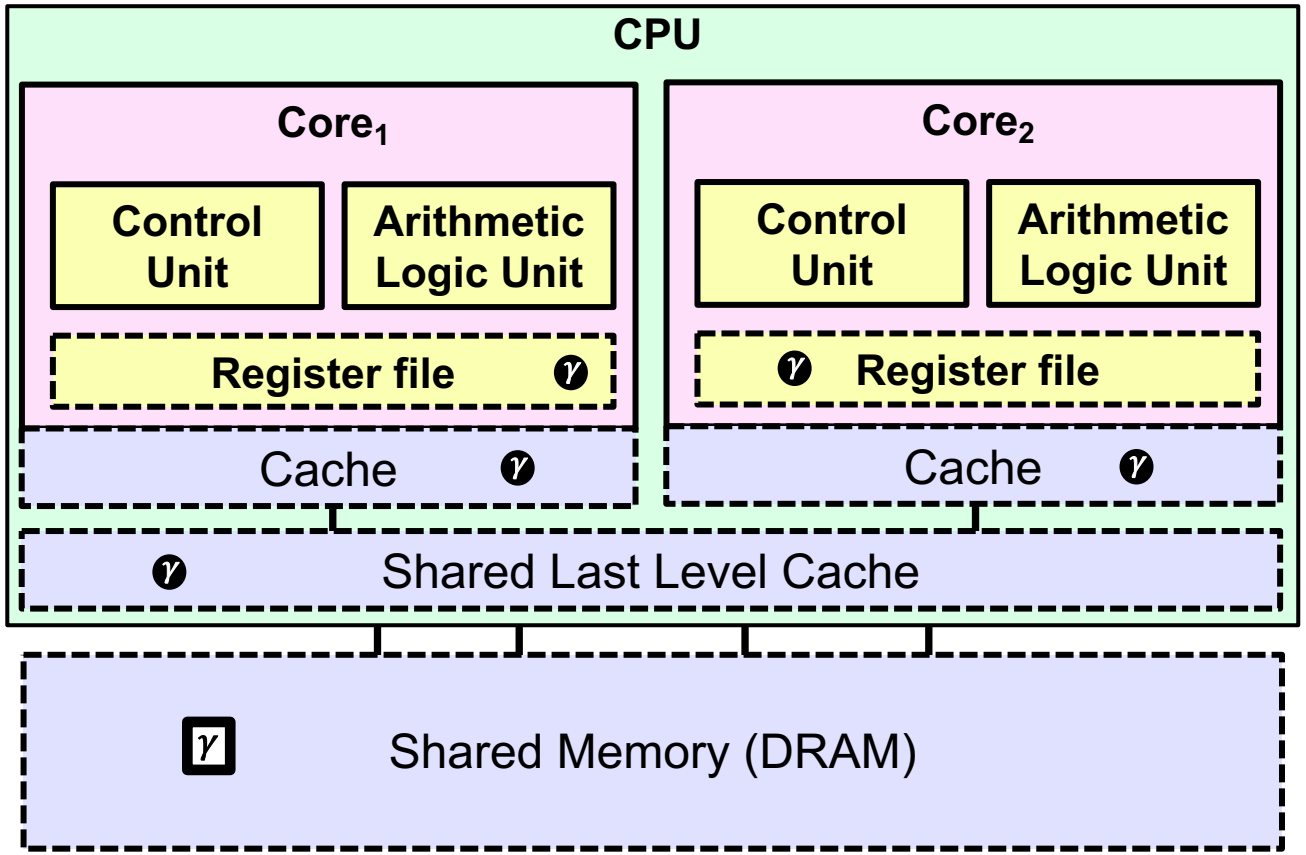
Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory Models and Point-to-Point Synchronization
  - Programming your GPU with OpenMP

Memory Models …

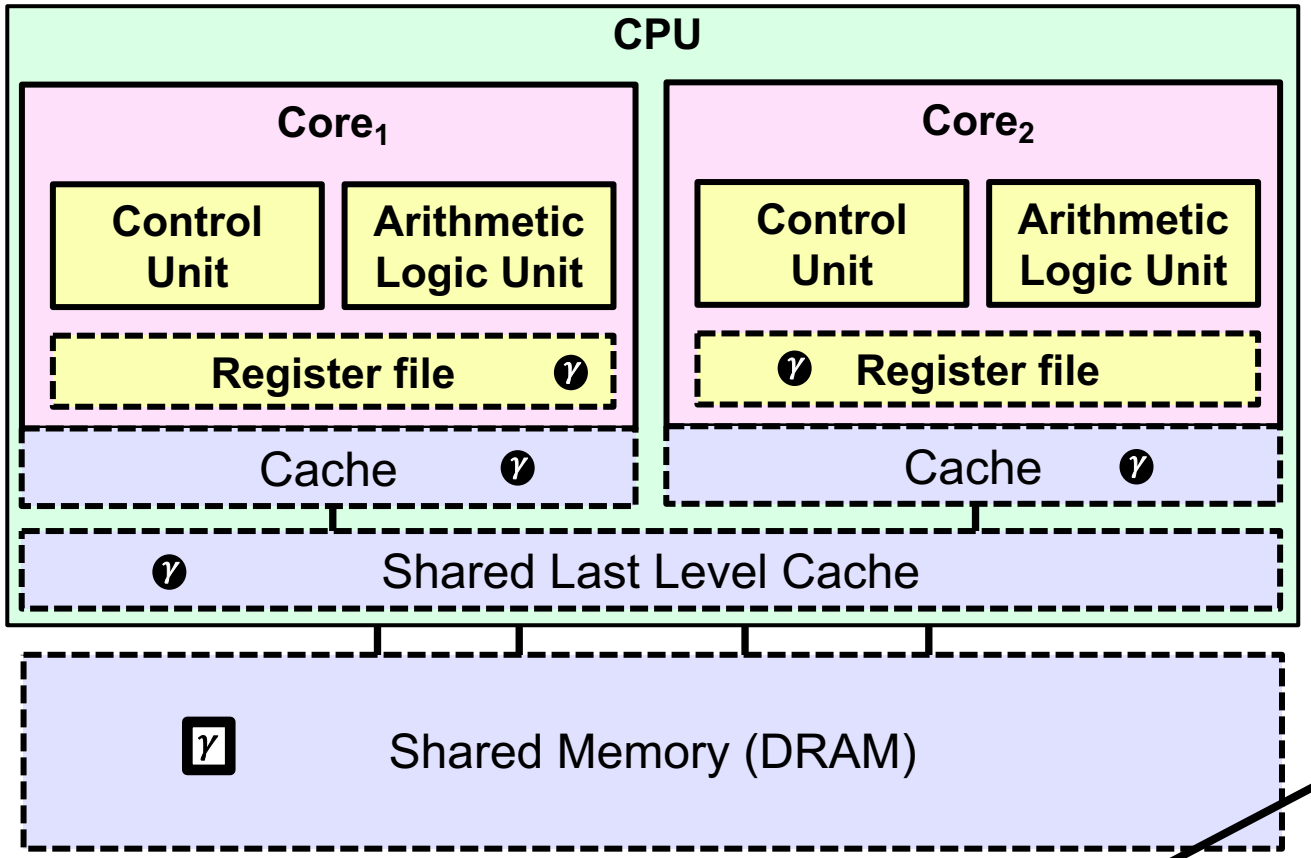
- Programming models for Multithreading support shared memory.
- All threads share an address space … but consider the variable γ



- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

Memory Models …

- Programming models for Multithreading support shared memory.
- All threads share an address space … but consider the variable γ



A memory consistency model (or "memory model" for short) provides the rules needed to answer this question.

- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

OpenMP and Relaxed Consistency

- Most (if not all) multithreading programming models (including OpenMP) supports a **relaxed-consistency** memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent only at certain points in the program
  - The operation that enforces consistency is called the **flush operation***

*Note: in OpenMP 5.0 the name for the flush described here was changed to a ”strong flush”. This was done so we could distinguish the traditional OpenMP flush (the strong flush) from the new synchronizing flushes (acquire flush and release flush).

Flush Operation

- Defines a sequence point at which a thread enforces a consistent view of memory.

- For variables visible to other threads and associated with the flush operation (the **flush-set**)
  - The compiler can't move loads/stores of the flush-set around a flush:
    - All previous read/writes of the flush-set by this thread have completed
    - No subsequent read/writes of the flush-set by this thread have occurred
  - Variables in the flush set are moved from temporary storage to shared memory.
  - Reads of variables in the flush set following the flush are loaded from shared memory.

IMPORTANT POINT: The flush makes the calling threads temporary view match the view in shared memory. Flush by itself does not force synchronization.

Memory Consistency: Flush Example

- Flush forces data to be updated in memory so other threads see the most recent value

Flush without a list: flush set is all thread visible variables

Flush with a list: flush set is the list of variables

```
double A;
A = compute();
#pragma omp flush(A)
   // flush to memory to make sure other
   //  threads can pick up the right value
```

Note: OpenMP's flush is analogous to a fence in other shared memory APIs

What is the BIG DEAL with Flush?

- Compilers routinely reorder instructions implementing a program
  - Can better exploit the functional units, keep the machine busy, hide memory latencies, etc.
- Compilers generally cannot move instructions:
  - Past a barrier
  - Past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Warning: the flush operation (a strong flush) does not actually synchronize different threads. It just ensures that a thread’s variables are made consistent with main memory

Flush and Synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

  ….

  (but not on entry to worksharing regions)

WARNING:
If you find your self wanting to write code with explicit flushes, stop and get help. It is very difficult to manage flushes on your own. Even experts often get them wrong.

This is why we defined OpenMP constructs to automatically apply flushes most places where you really need them.

Example: prod_cons.c

- Parallelize a producer/consumer program
  - One thread produces values that another thread consumes.

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *) malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf secs, The sum is %lf \n",runtime,sum);
}
```

  - Often used with a stream of produced values to implement "pipeline parallelism"
  - The key is to implement pairwise synchronization between threads

Pairwise Synchronization in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When needed, you have to build it yourself.
- Pairwise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise: Producer/Consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);

        flag = 1;

      }
      #pragma omp section
      {

        while (flag == 0){

        }

        sum = Sum_array(N, A);
      }
    }
}
```

Put the flushes in the right places to make this program race-free.

Do you need any other synchronization constructs to make this work?

Solution (try 1): Producer/Consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
      }
      #pragma omp section
      {
        #pragma omp flush (flag)
        while (flag == 0){
            #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

This program works with the x86 memory model (loads and stores use relaxed atomics), but it technically has a race … on the store and later load of flag

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

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

Atomics and Synchronization Flags

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {  fill_rand(N, A);
          #pragma omp flush
          #pragma omp atomic write
                  flag = 1;
          #pragma omp flush (flag)
       }
       #pragma omp section
       {  while (1){
             #pragma omp flush(flag)
             #pragma omp atomic read
                    flg_tmp= flag;
              if (flg_tmp==1) break;
          }
          #pragma omp flush
          sum = Sum_array(N, A);
       }
     }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict

Still painful and error prone due to all of the flushes that are required

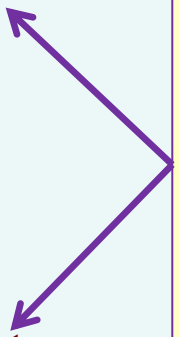
OpenMP 4.0 Atomic: Sequential consistency

- Sequential consistency:
  - The order of loads and stores in a race-free program appear in some interleaved order and all threads in the team see this same order.
- OpenMP 4.0 added an optional clause to atomics
  - #pragma omp atomic [read | write | update | capture] [**seq_cst**]
- In more pragmatic terms:
  - If the seq_cst clause is included, OpenMP adds a flush without an argument list to the atomic operation so you don't need to.
- In terms of the C++'11 memory model:
  - Use of the seq_cst clause makes atomics follow the sequentially consistent memory order.
  - Leaving off the seq_cst clause makes the atomics relaxed.

Advice to programmers: save yourself a world of hurt … let OpenMP take care of your flushes for you whenever possible … use seq_cst

Atomics and Synchronization Flags (4.0)

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {  fill_rand(N, A);

          #pragma omp atomic write seq_cst
                 flag = 1;

       }
       #pragma omp section
       {  while (1){

             #pragma omp atomic read seq_cst
                   flg_tmp= flag;
              if (flg_tmp==1) break;
           }

           sum = Sum_array(N, A);
       }
     }
}
```

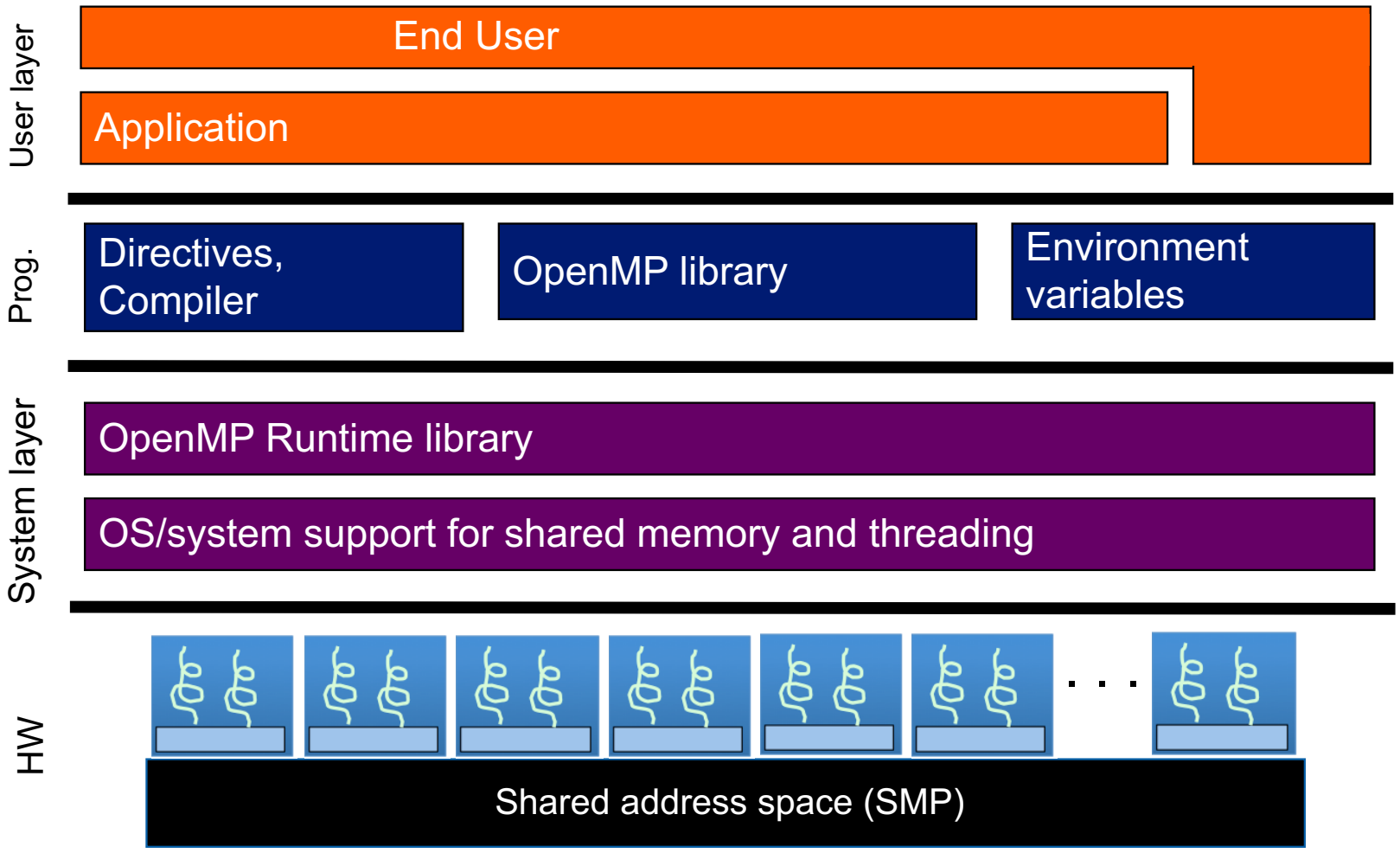


This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict – and you do not use any explicit flush constructs (OpenMP does them for you)

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

OpenMP basic definitions: Basic Solution stack



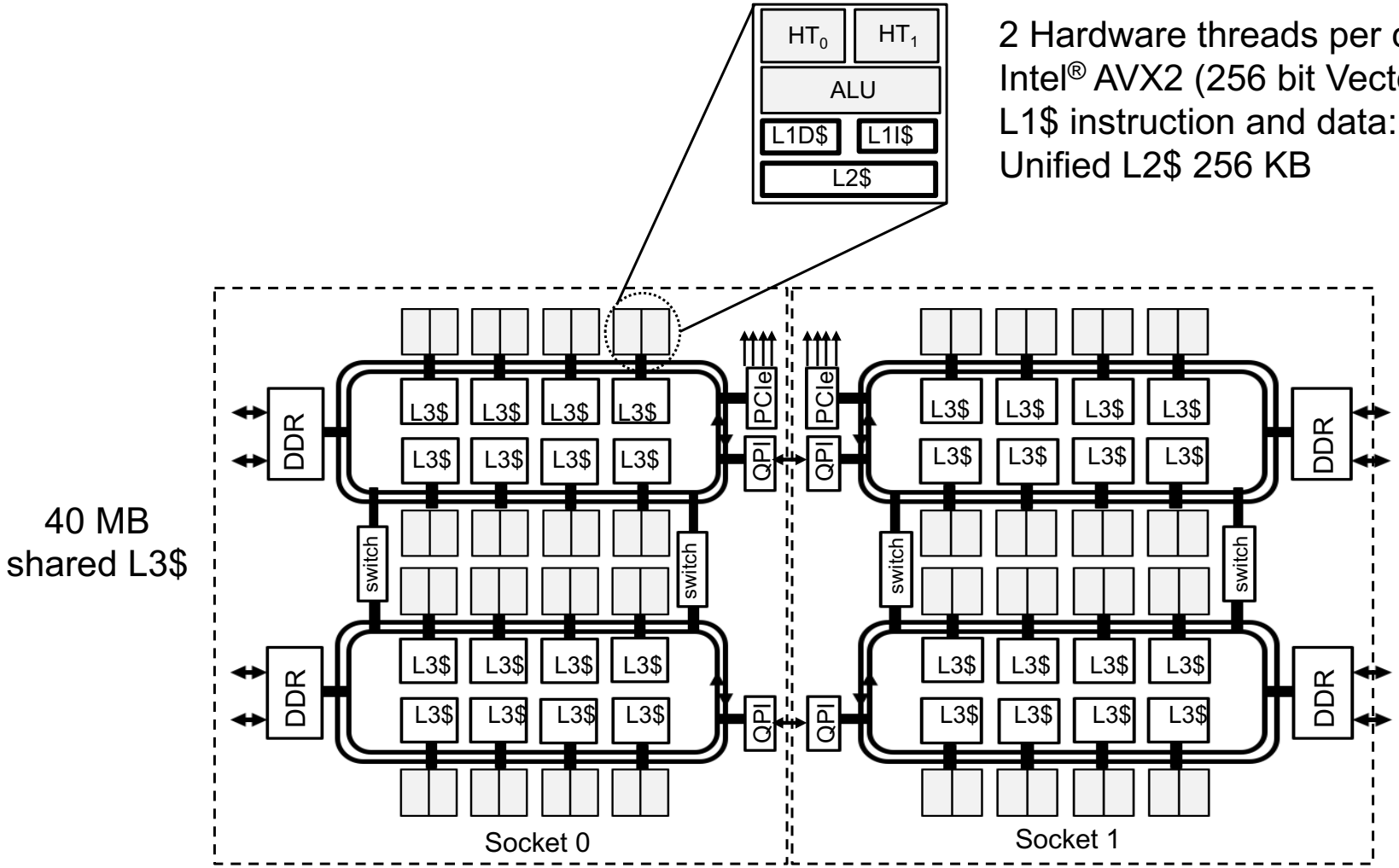
In learning OpenMP, you consider a Symmetric Multiprocessor (SMP) ….
i.e. lots of threads with "**equal cost access**" to memory

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

A Typical CPU Node in an HPC System

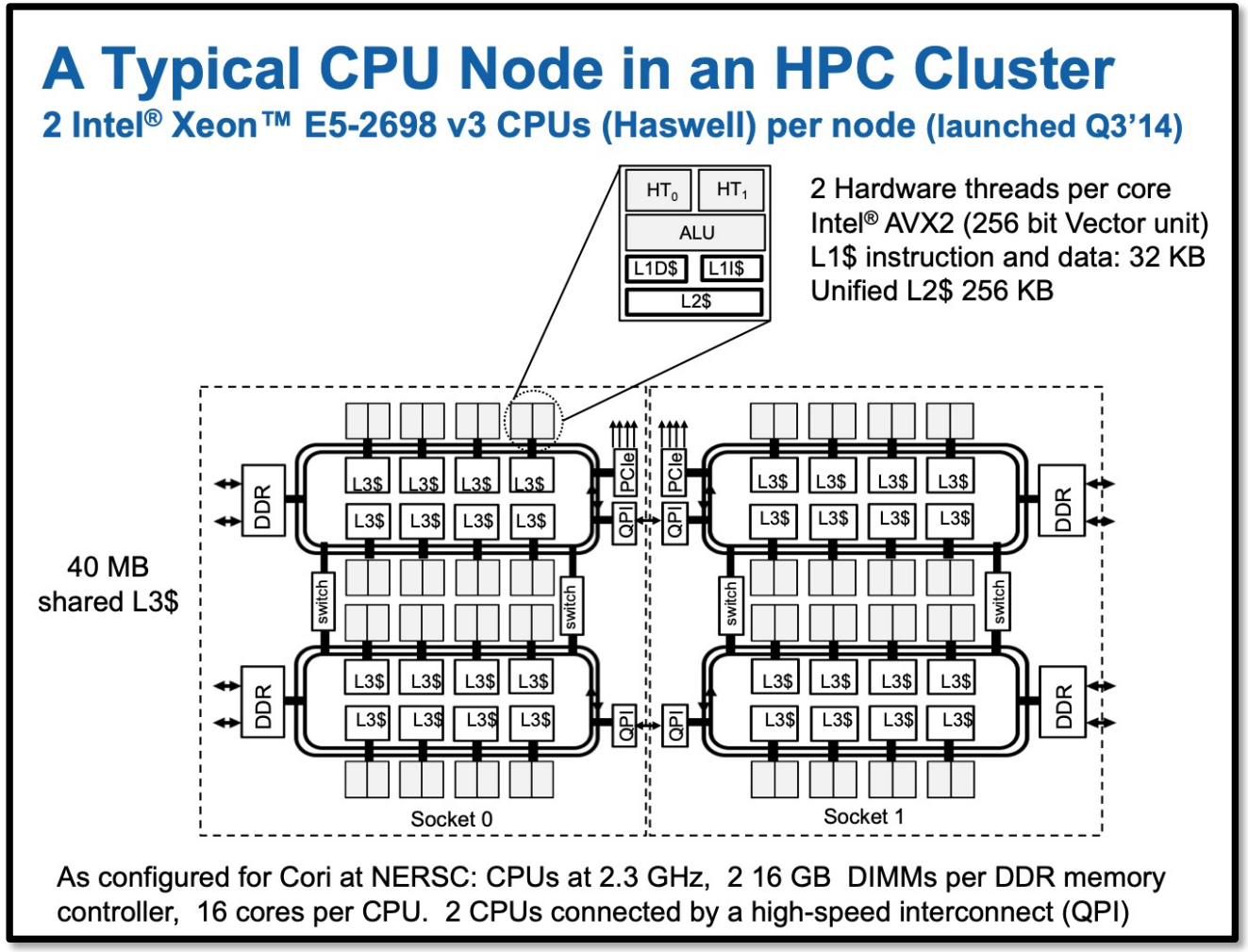
2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)



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?

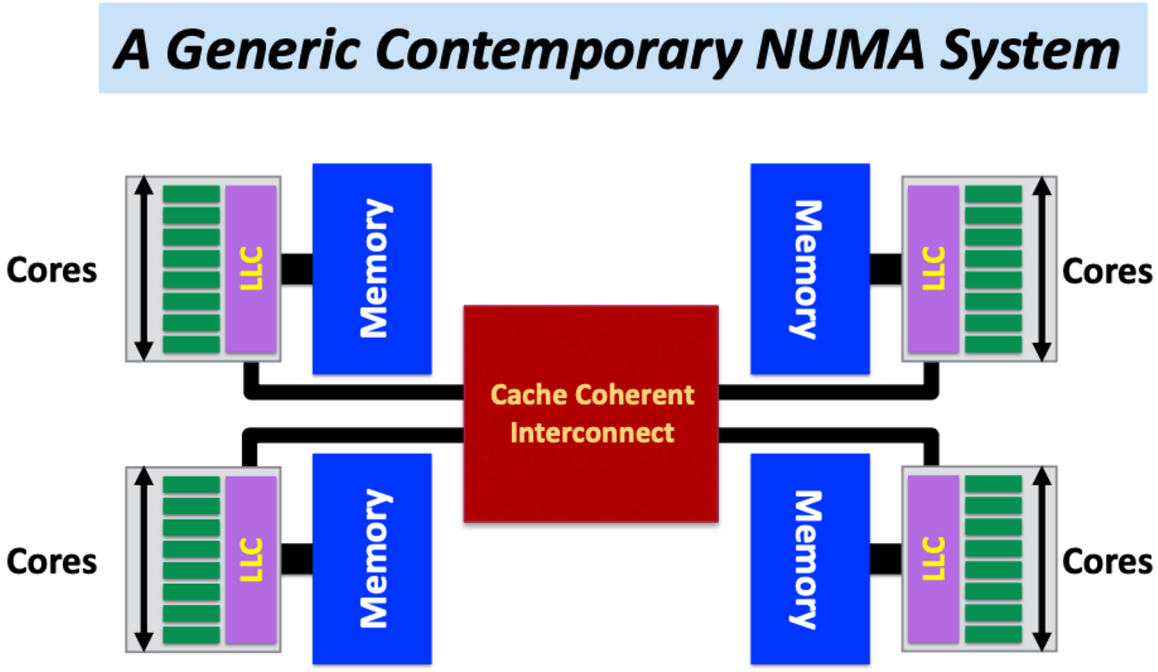
There may be a single address space, but there are multiple levels of non-uniformity to the memory. This is a **N**on-**U**niform **M**emory **A**rchitecture (NUMA)



Even a single CPU is properly considered a NUMA architecture

NUMA Systems

- Most systems today are Non-Uniform Memory Access (NUMA)
- 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

Also possible on-package high bandwidth memory on the node

Diagram courtesy Ruud van der Pas

Memory Locality

- Most systems today are Non-Uniform Memory Access (NUMA)
- Example, the Intel® Xeon Phi™ processor

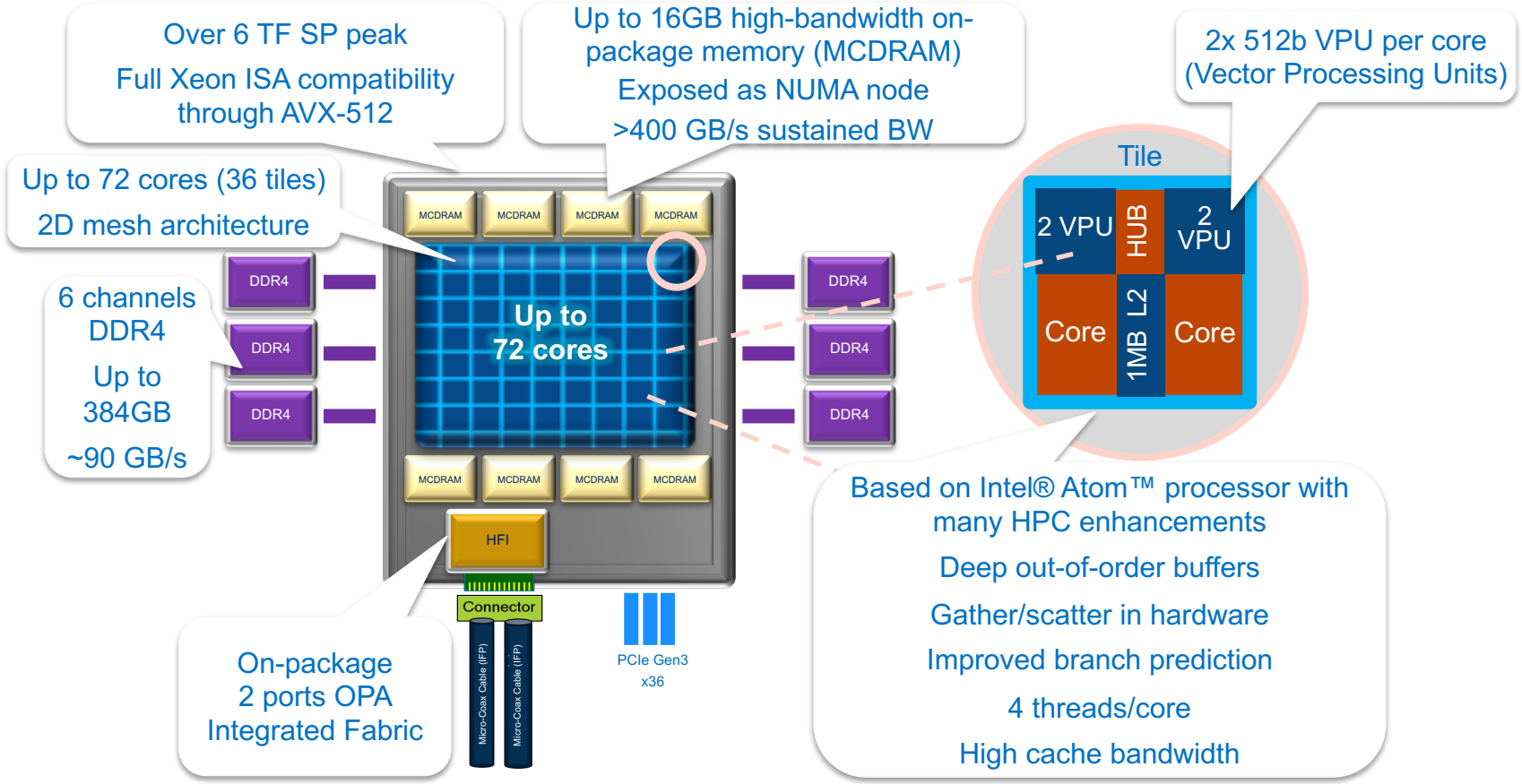


Diagram is for conceptual purposes only and only illustrates a CPU and memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

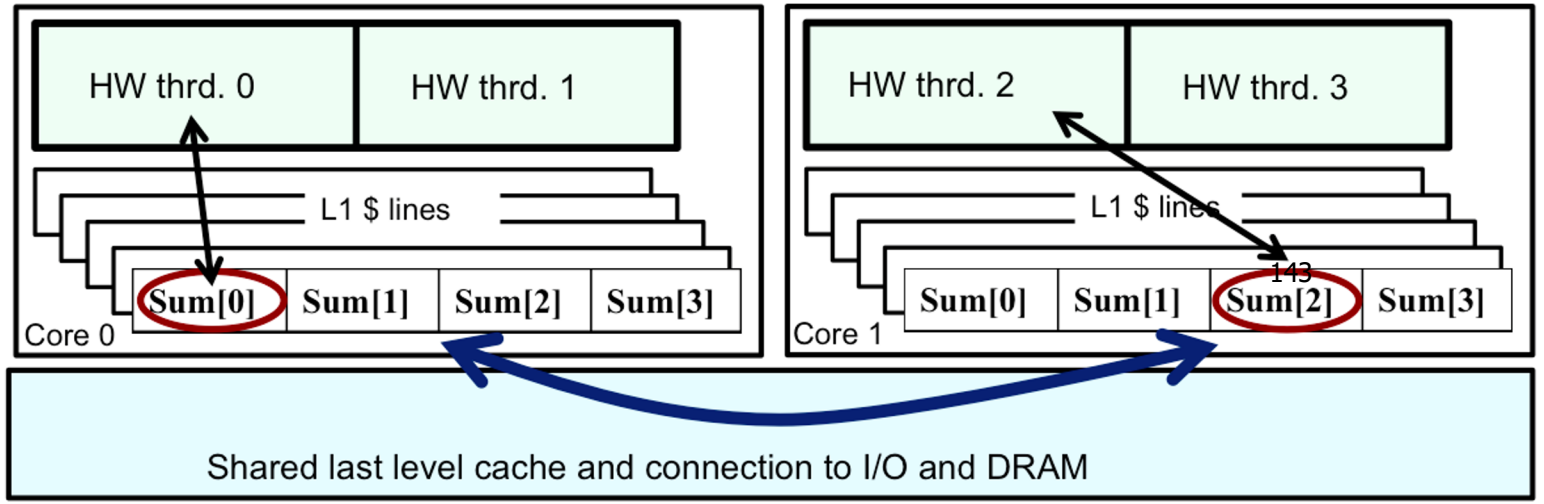
Memory Locality

- Memory access in different NUMA domains are different
  - Accessing memory in remote NUMA is slower than accessing memory in local NUMA
  - Accessing High Bandwidth Memory on KNL* is faster than DDR
- OpenMP does not explicitly map data across shared memories
- Memory locality is important since it impacts both memory and intra-node performance

*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 Ghz …
the “bootable” version that sits in a socket, not a co-processor

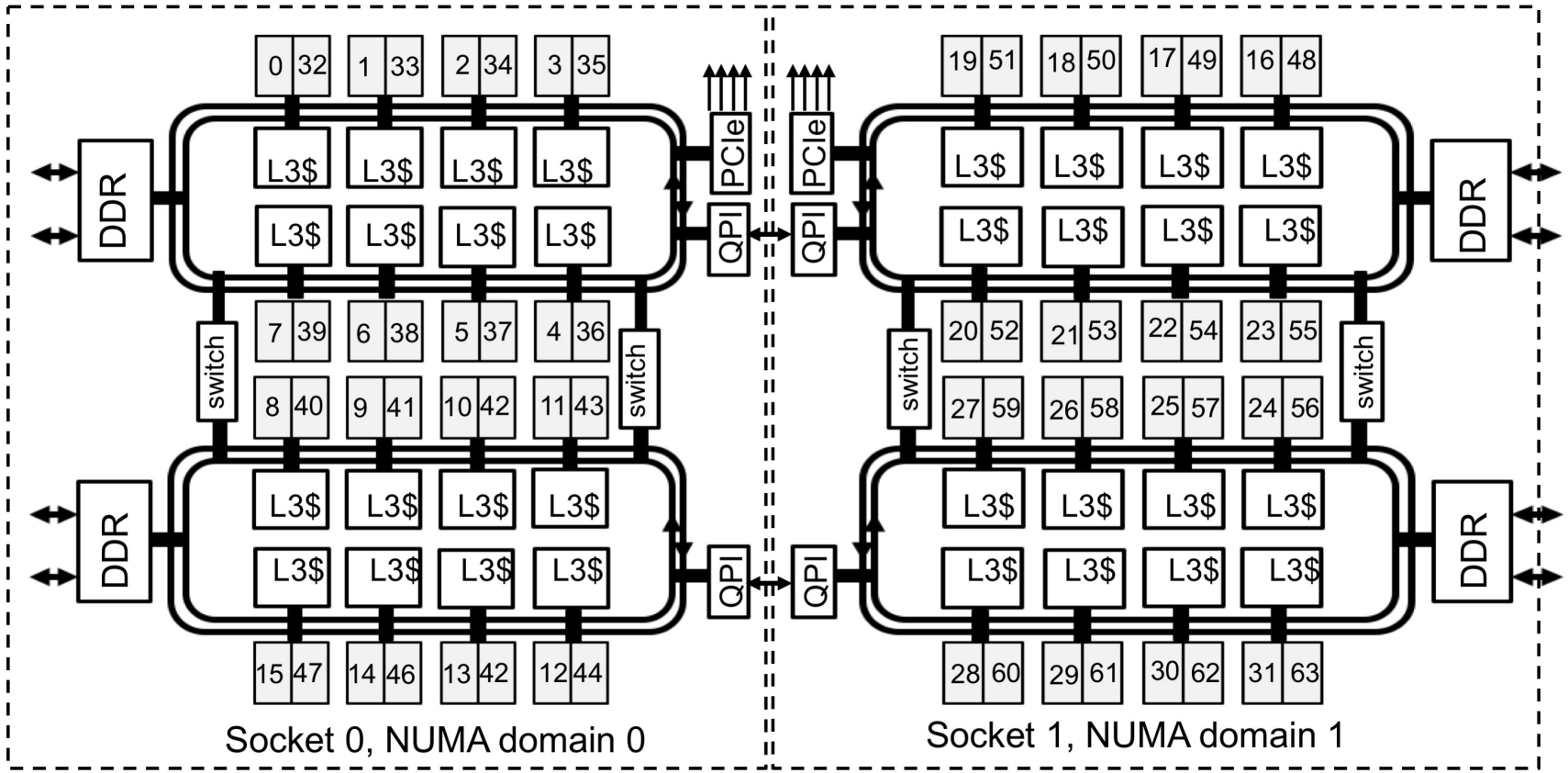
Cache Coherence and False Sharing

- ccNUMA node: cache-coherence NUMA node.
- Data from memory are accessed via cache lines.
- Multiple threads hold local copies of the same (global) data in their caches. Cache coherence ensures the local copy to be consistent with the global data.
- Main copy needs to be updated when a thread writes to local copy.
- Writes to same cache line from different threads is called false sharing or cache thrashing, since it needs to be done in serial. Use atomic or critical or private variables to avoid race condition.



Exploring your NUMA world: NUMACTL

- numactl shows you how the OS processor-numbers map onto the physical cores of the chip:



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

Tool to Check NUMA Node Information: numactl

- **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
```

Shows relative costs …. In this case, there's a factor of two in the cost of the local (on CPU) DRAM vs going to the other socket

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

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")

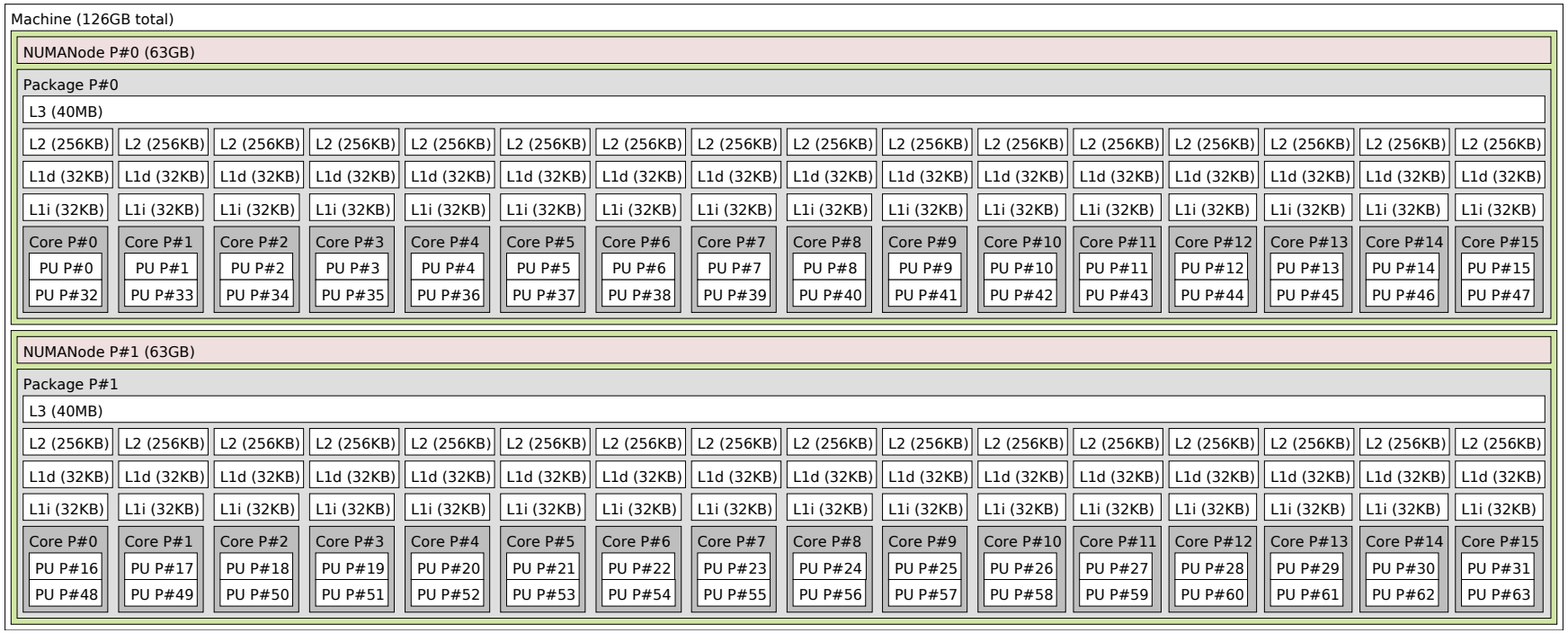
- CPU binding example:
  - % numactl --cpunodebind 0,1 ./code.exe

    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

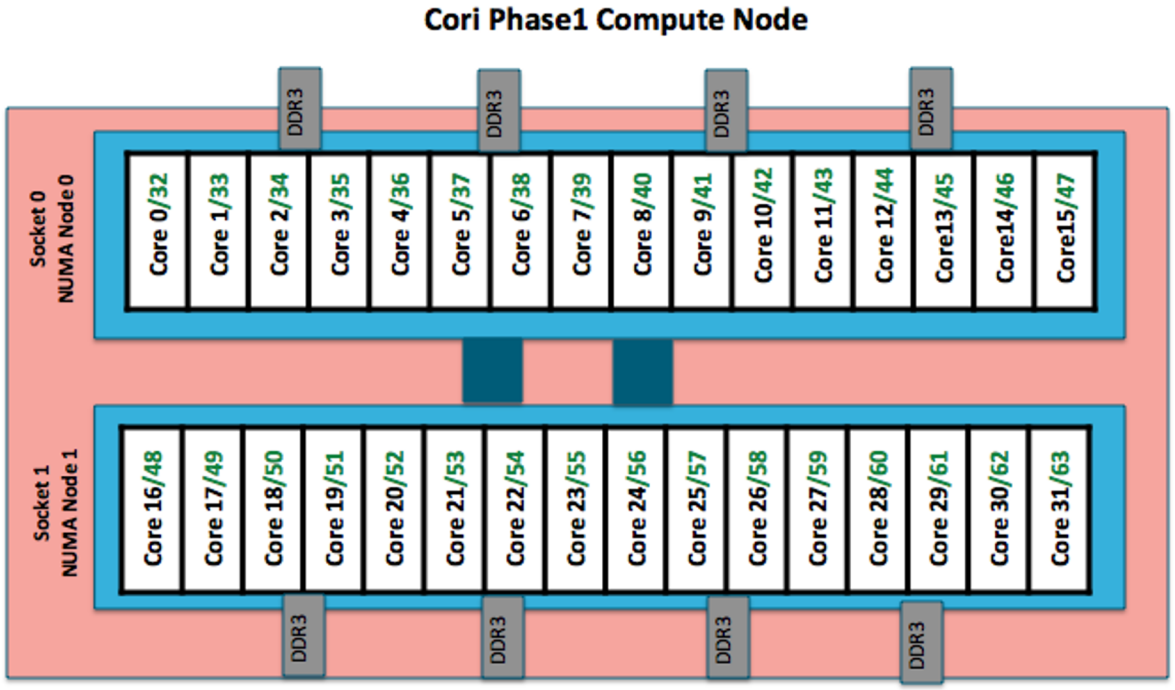
Tools to Check Node Information: hwloc

- 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

Find Processor Info on a Mac Laptop

```
$ sysctl -n machdep.cpu.brand_string
   Intel(R) Core(TM) i7-8569U CPU @ 2.80GHz

$ system_profiler |grep Processor
      …
      Processor Name: Quad-Core Intel Core i7
      Processor Speed: 2.8 GHz
      Number of Processors: 1
      …
```

Exercise: Node Information

- Characterize the processor/memory layout of your system

- Try on a Cori login node, a Cori Haswell and a Cori KNL node, and find out the differences

Process / Thread / Memory Affinity (1)

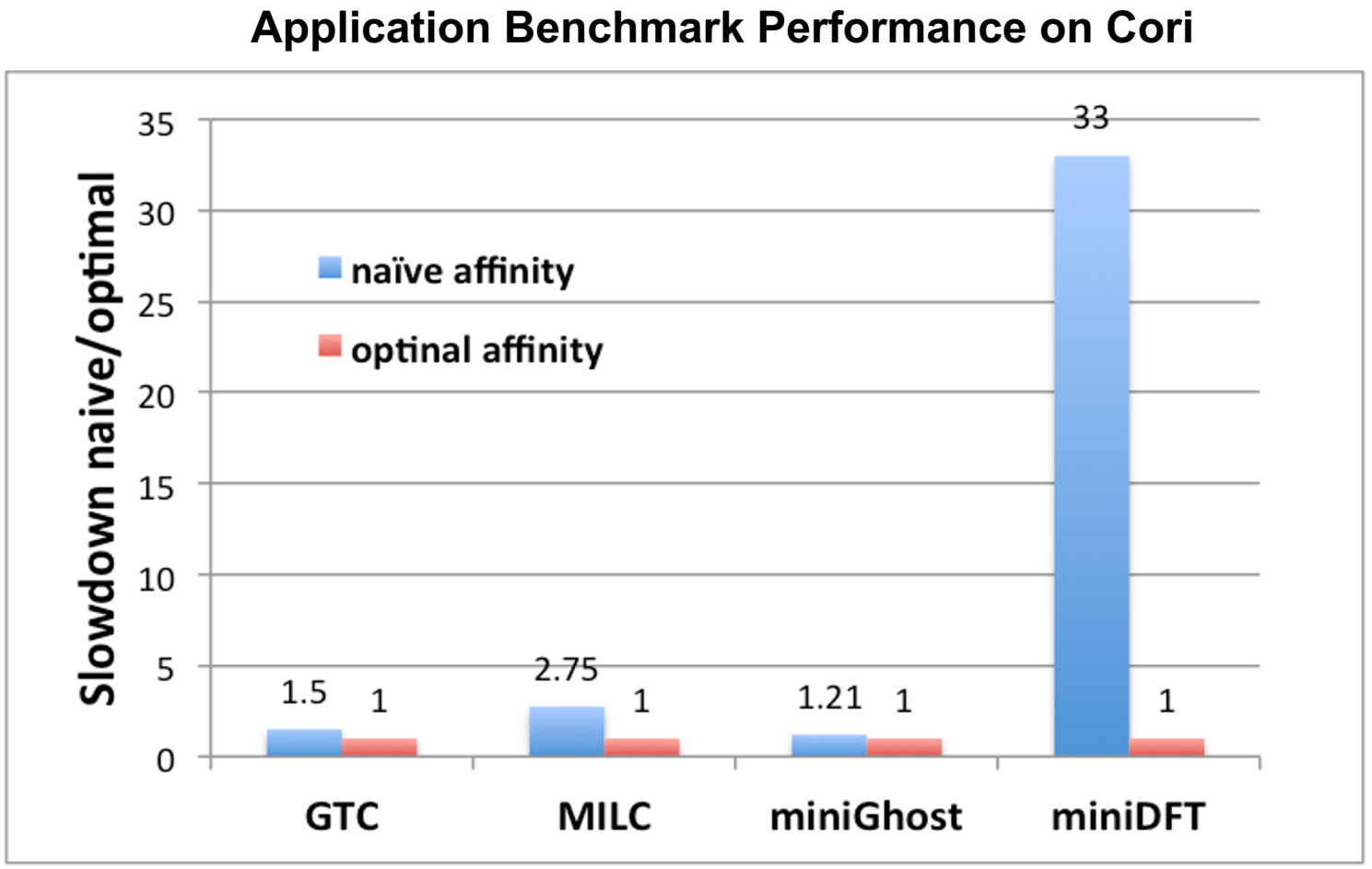
- Process Affinity: also called "CPU pinning", binds processes (MPI tasks, etc.) to a CPU or a range of CPUs on a node
  - It is important to spread MPI ranks evenly onto cores in different NUMA domains
- Thread Affinity: further binding threads to CPUs that are allocated to their parent process
  - 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 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
- 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

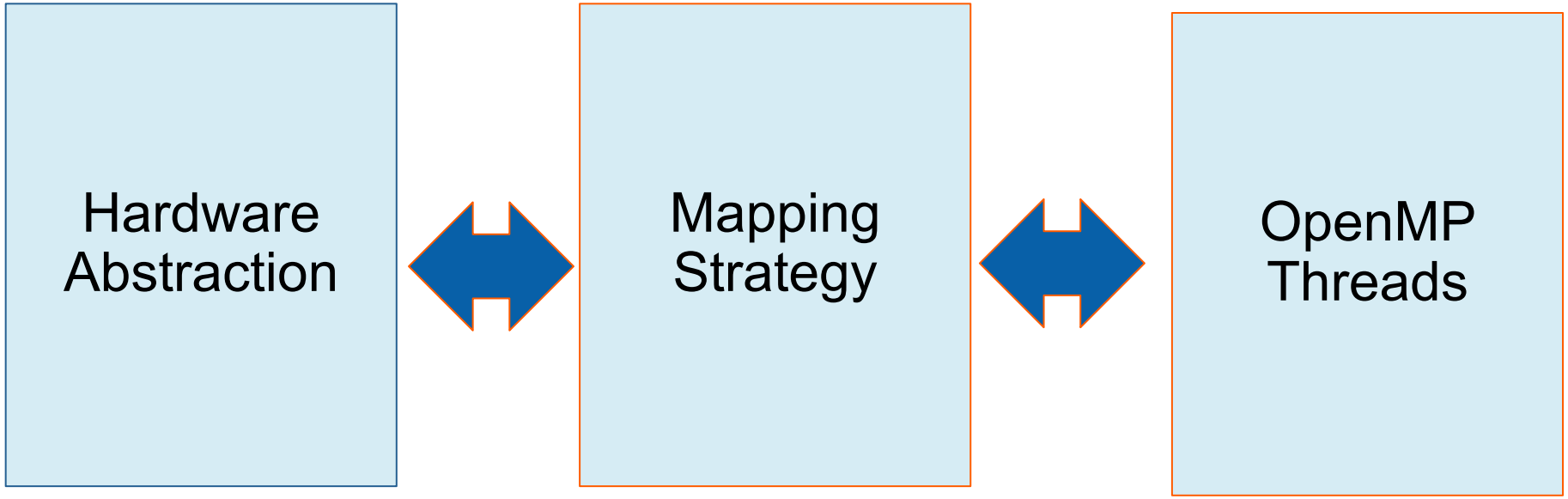
- Correct process, thread and memory affinity is the basis for getting optimal performance. It is also essential for guiding further performance optimizations.

Naïve *vs.* Optimal Affinity



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

Writing NUMA-aware OpenMP Code

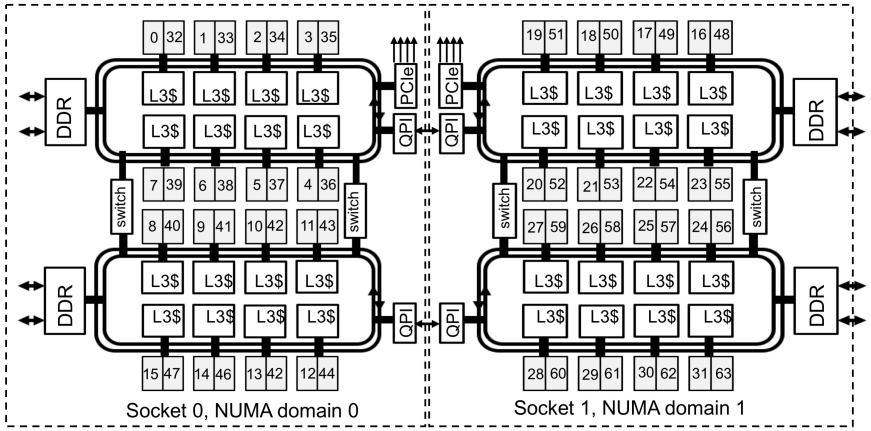
- Control the places where threads are mapped
  - Place threads onto cores to optimize performance
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To Optimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

- Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

The Concept of Places

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

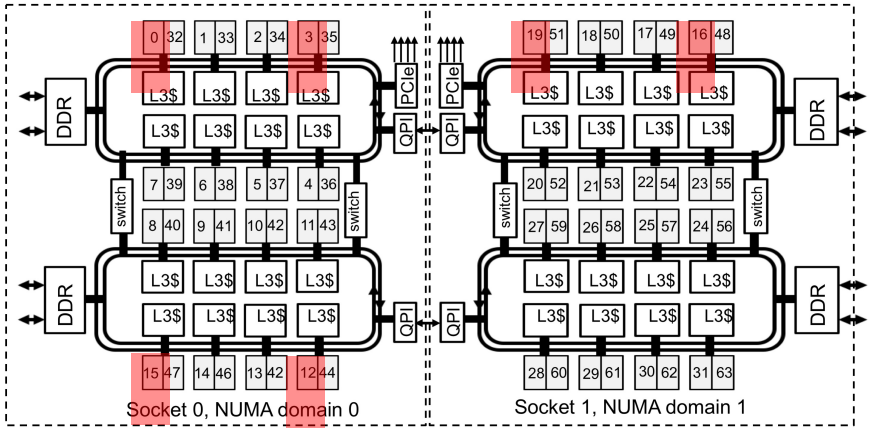


- A place: numbers between { }:

  export OMP_PLACES="{0,1,2,3}"

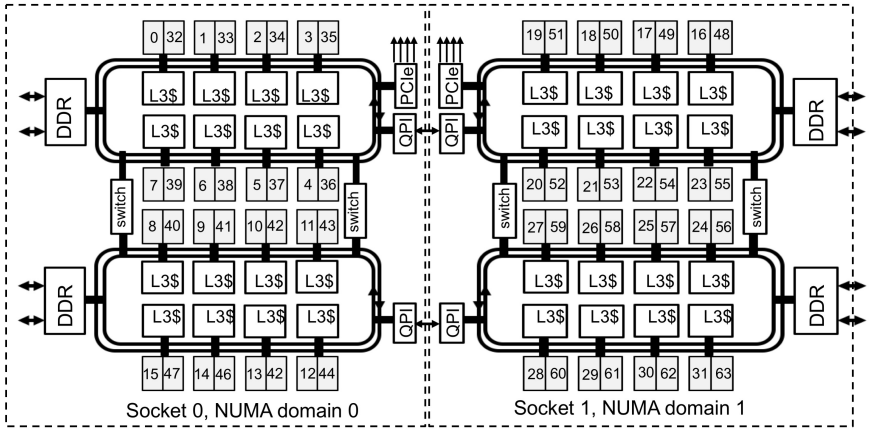
- A place defines where threads can run

```
> export OMP_PLACES "{0, 3, 15, 12, 19,  16, 28, 31}"
> export NUM_THREADS= 6

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

}
```



The Concept of Places

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



- Set with an environment variable:

  export OMP_PLACES="{0,1,2,3}"

- Can also specify with {lower-bound:length:stride}

  OMP_PLACES="{0,1,2,3}" → OMP_PLACES="{0:4:1}" → OMP_PACES="{0:4}"

Default Stride is 1

- Can define multiple places:

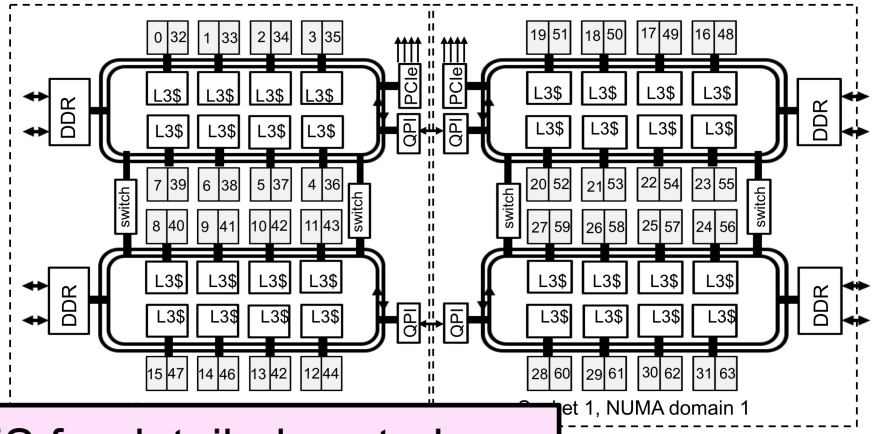
  OMP_PLACES="{0,1,2,3},{4,6,8},{9,10,11,12}"

  OMP_PLACES="{0,4},{4,3:2},{9:4}"

These are equivalent

The Concept of Places

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



- Set w
  expo
- Can a
  OMP_ … 1
  … 4}”
- Can

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

- The rules for mapping onto physical execution units are complicated.
- PLACES expressed as numbers is non-portable

There has to be an easier and more portable way to describe places

OMP_PLACES=“{0,1,2,3},{4,6,8},{9,10,11,12}”

OMP_PLACES=“{0,4},{4,3:2},{9:4}”

These are equivalent

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

Writing NUMA-aware OpenMP Code

- Control the places where threads are mapped
  - Place threads onto cores to optimize performance
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To Optimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

- Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

Mapping Strategy: 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
  - primary*: bind threads to the same place as the master thread
- The values **primary***, **close,** and **spread** imply the value **true**

Examples:

```
export OMP_PROC_BIND=spread
export OMP_PROC_BIND=spread,close (for nested levels)
```

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

Mapping Strategy: OMP_PROC_BIND (2)

- Put threads far apart (spread) may improve aggregated memory bandwidth and available cache size for your application, but may also increase synchronization overhead

- Put threads “close” have the reverse impact as “spread”

Mapping Strategy: 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

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()

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

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
```

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

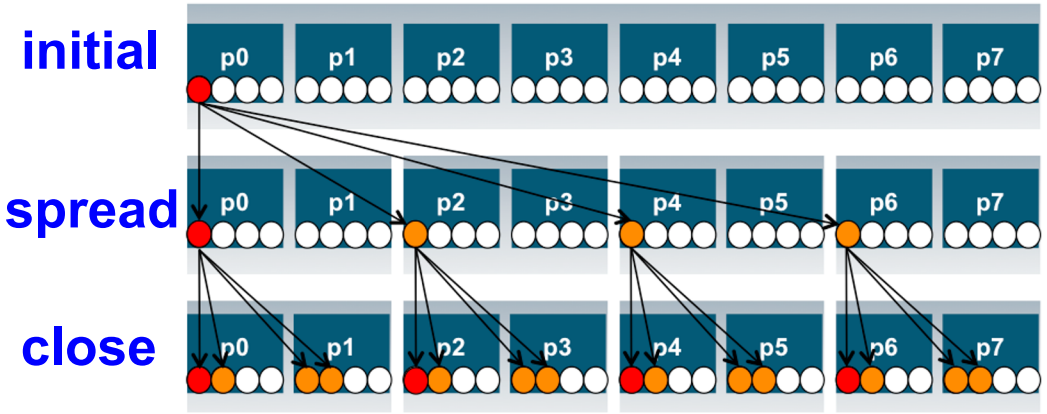
- A combination of OpenMP environment variables and runtime flags are needed for different compilers and different batch schedulers on different systems

```
#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

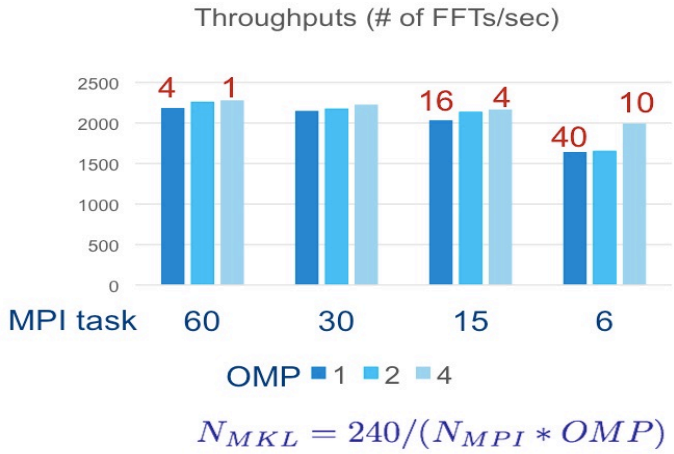
When to Use Nested OpenMP

- Beneficial to use nested OpenMP to allow more fine-grained thread parallelism
- Some application teams are exploring with nested OpenMP to allow more fine-grained thread parallelism
  - Hybrid MPI/OpenMP not using node fully packed
  - Top level OpenMP loop does not use all available threads
  - Multiple levels of OpenMP loops are not easily collapsed
  - Certain computational intensive kernels could use more threads
  - MKL can use extra cores with nested OpenMP
- Nested level can be arbitrarily deep

Use Multiple Threads in MKL

- By Default, in OpenMP parallel regions, only 1 thread will be used for MKL calls.
  - MKL_DYNAMICS is true by default
- Nested OpenMP can be used to enable multiple threads for MKL calls. Treat MKL as a nested inner OpenMP region.
- Sample settings

```
export OMP_NESTED=true
export OMP_PLACES=cores
export OMP_PROC_BIND=sprad,close
export OMP_NUM_THREADS=6,4
export MKL_DYNAMICS=false
export OMP_MAX_ACTIVE_LEVELS=2
```



FFT3D on KNC, Ng=64^3 *example*

Courtesy of Jeongnim Kim, Intel

*KNC: Intel® Xeon Phi™ processor (Knights Corner) … the first generation co-processor version of the chip.

Exercise: Affinity Verification

- Run the “Hello World” code, use OMP_DISPLAY_AFFINITY to observe affinity status

- Change thread binding and number of threads and see how affinity status changes

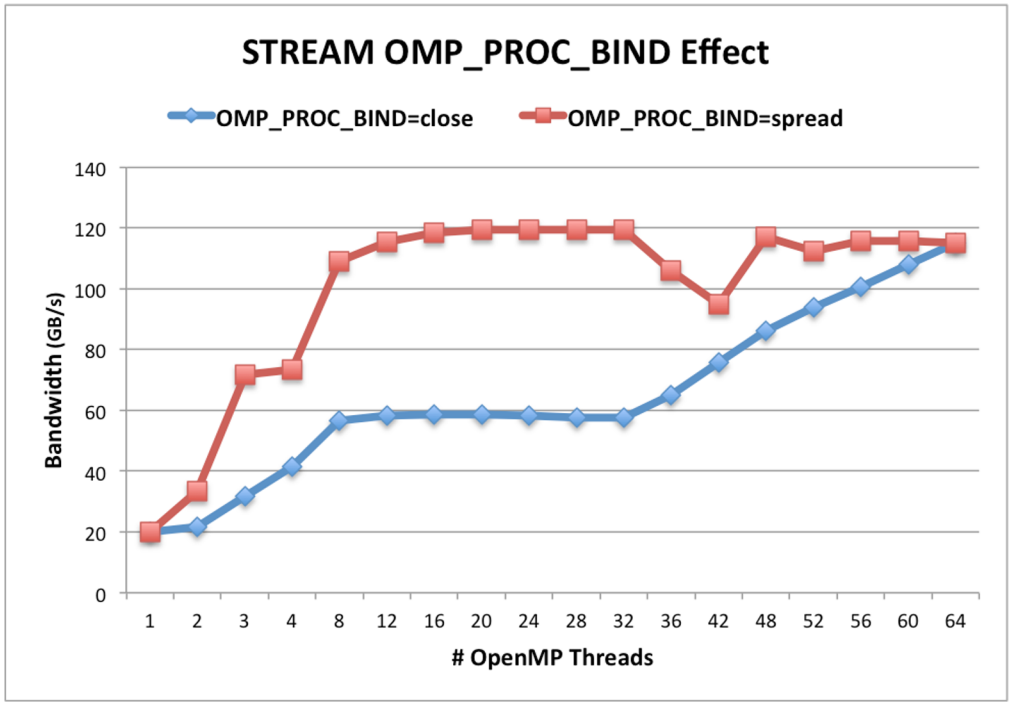
OMP_PROC_BIND Choices for STREAM Benchmark

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



Exercise: STREAM Benchmark

- Use the STREAM benchmark code: C/affinity/stream.c
  - Sample batch script: “run_stream_sample.sh”

    % sbatch <job_script>
  - STREAM memory bandwidth results: check “Best Rate” for “Triad” in the output
  - Experiment with different OMP_NUM_THREADS, OMP_PROC_BIND, and OMP_PLACES, and OMP_DISPLAY_AFFINITY settings to check thread affinity output and performance result
  - Run with 8, 16, 32, 48, 64 threads, and OMP_PROC_BIND=spread or close
- Compare your results with the previous STREAM plot

Writing NUMA-aware OpenMP Code

- Control the places where threads are mapped
  - Place threads onto cores to optimize performance
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To Optimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

- Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

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

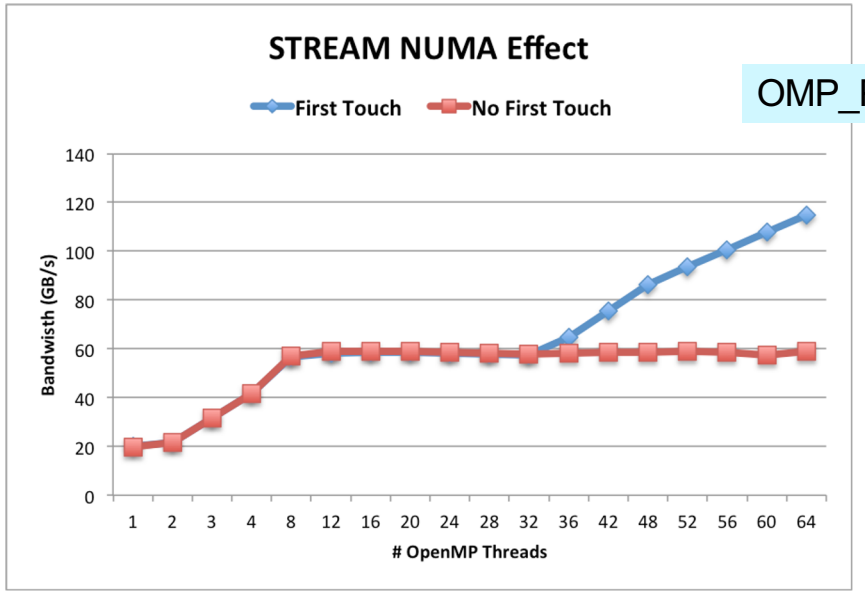
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

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

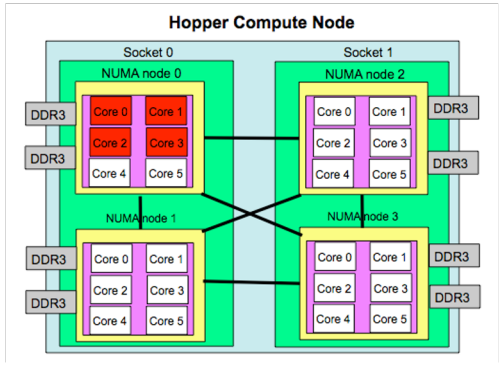
“Perfect Touch” is Hard

- Hard to do “perfect touch” for real applications
- General recommendation: use number of threads fewer than number of CPUs per NUMA domain
- In the previous example, there are 16 cores (32 CPUs) per NUMA domain. Sample run options:
  - 2 MPI tasks, 1 MPI task per NUMA domain, with 32 OpenMP threads (if using hyperthreads) or 16 OpenMP threads (if not using hyperthreads) per MPI task
  - 4 MPI tasks, 2 MPI tasks per NUMA domain, with 16 OpenMP threads (if using hyperthreads) or 8 OpenMP threads (if not using hyperthreads) per MPI task
  - …

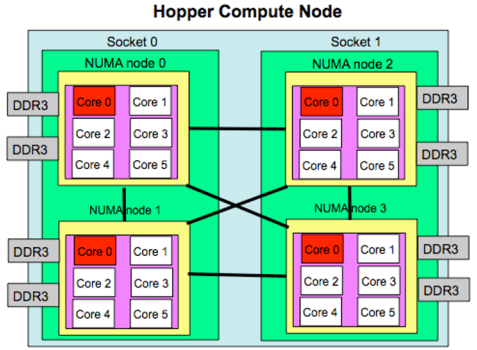
MPI Process Affinity Example: aprun "-S" Option

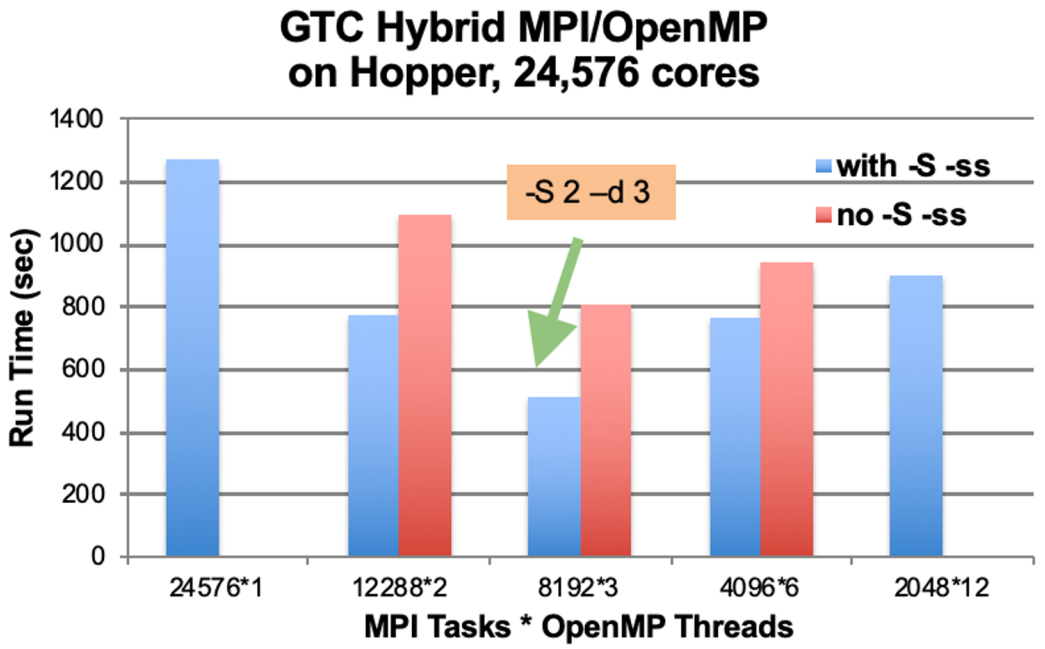
- Important to spread MPI ranks evenly onto different NUMA nodes
- Use the "-S" option: specify #MPI_tasks per NUMA domain
- The example below was from an XE6 system (NERSC Hopper)

aprun –n 4 –d 6



aprun –n 4 **–S 1**–d 6





Exercise: Importance of First Touch

- Do the same STREAM experiments with the no first touch code: “stream_nft.c” to understand the impact of first touch
  - Experiment with different OMP_NUM_THREADS, OMP_PROC_BIND, and OMP_PLACES, and OMP_DISPLAY_AFFINITY settings to check thread affinity output and performance result
  - Run with 8, 16, 32, 48, 64 threads, and OMP_PROC_BIND=spread or close

- Compare your results with the previous STREAM plot

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
}
```

Memory Allocators (in OpenMP 5.0)

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

- Support versatile types of memory available on current and future systems: DDR, High-Bandwidth Memory (HBM), non-volatile memory, constant memory
- Memory allocators define types of memory that variables can be allocated to, such as large capacity, low latency, cgroup, thread local, etc.

Using Memory Allocators

```
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N];
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // use default OMP_ALLOCATOR

    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(omp_low_lat_mem_alloc:a)
    {
        some_parallel_code();
    }
    omp_free(p);
}
```

A NUMA Case study

Benchmarking … I Must Control Everything!

- Goal: To compare different programming systems applied to the same problem:
  - We must control everything we can to make sure any observed differences are due to the different programming systems.

- We need to know exactly which cores we are using and how thread IDs map onto cores … so we can understand data detailed memory movement and make sure it's the same between the different test cases.

Step 1: Know Your System

- My system did not have numactl or Hwloc. So I went with my third option …. lscpu (note: I'm only showing a subset of the actual output):

```
$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           46 bits physical, 48 bits virtual
CPU(s):                  72
On-line CPU(s) list:     0-71
Thread(s) per core:      2
Core(s) per socket:      18
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   63
Model name:              Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
Stepping:                2
CPU MHz:                 1197.539
CPU max MHz:             3600.0000
CPU min MHz:             1200.0000
L1d cache:               1.1 MiB
L1i cache:               1.1 MiB
L2 cache:                9 MiB
L3 cache:                90 MiB
NUMA node0 CPU(s):       0-17,36-53
NUMA node1 CPU(s):       18-35,54-71
```

SMT enabled … two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

Note: a HW thread is a CPU (or core) as far as the OS is concerned. These two lines show you the numbering of these "cores".

Step 1: Know Your System

- My system did not have numactl or Hwloc. So I went with my third option …. lscpu (note: I'm only showing a subset of the actual output):

```
$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           46 bits physical, 48 bits virtual
CPU(s):                  72
On-line CPU(s) list:     0-71
Thread(s) per core:      2
Core(s) per socket:      18
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   63
Model name:              Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
Stepping:                2
CPU MHz:                 1197.539
CPU max MHz:             3600.0000
CPU min MHz:             1200.0000
L1d cache:               1.1 MiB
L1i cache:               1.1 MiB
L2 cache:                9 MiB
L3 cache:                90 MiB
NUMA node0 CPU(s):       0-17,36-53
NUMA node1 CPU(s):       18-35,54-71
```

SMT enabled … two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

The numbering of these "cores" (in 2 sockets).

0/36	1/37	2/38	3/39	4/40	5/41	6/42	7/43	8/44
9/45	10/46	11/47	12/48	13/49	14/50	15/51	16/52	17/53
18/54	19/55	20/56	21/57	22/58	23/59	24/60	25/61	26/62
27/63	28/64	29/65	30/66	31/67	32/68	33/69	34/70	35/71

Setup a Runscript (so you can reproduce the computations later)

```
#! /usr/bin/env bash
#  Run script for DGEMM with C and OpenMP

# Define shared parameters for the calculations we will run
BLOCK=0
ORDER=1000
ITERS=5

# setup environment for the intel compilers
source /opt/intel/compilers_and_libraries_2020.4.304/linux/bin/compilervars.sh -arch intel64

# Setup display of mapping from OpenMP threads to "hardware" threads.
export OMP_DISPLAY_AFFINITY=true
export OMP_AFFINITY_FORMAT="Thrd Lev=%3L, thrd_num=%5n,  thrd_aff=%15A"

# Enable explicit affinity control.
export OMP_PLACES="{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10},{11},{12},{13},{14},{15},{16}"
export OMP_PROC_BIND=close

./dgemm 8  $ITERS $ORDER $BLOCK
./dgemm 16 $ITERS $ORDER $BLOCK
```

A NUMA Case Study: Results

```
Parallel Research Kernels version 2.17
OpenMP Dense matrix-matrix multiplication
Thrd Lev=1  , thrd_num=0    ,  thrd_aff=0
Thrd Lev=1  , thrd_num=4    ,  thrd_aff=4
Thrd Lev=1  , thrd_num=3    ,  thrd_aff=3
Thrd Lev=1  , thrd_num=5    ,  thrd_aff=5
Thrd Lev=1  , thrd_num=1    ,  thrd_aff=1
Thrd Lev=1  , thrd_num=2    ,  thrd_aff=2
Thrd Lev=1  , thrd_num=6    ,  thrd_aff=6
Thrd Lev=1  , thrd_num=7    ,  thrd_aff=7
Matrix order         = 1000
Number of threads     = 8
Rate : 21650.601956  +/- 1589.413250 MFlops/s
```

Notice the one-to-one mapping of thread ID onto hardware thread.

Normally, this is going too far, but for benchmarking, this is a handy trick.

```
Parallel Research Kernels version 2.17
OpenMP Dense matrix-matrix multiplication
Thrd Lev=1  , thrd_num=0    ,  thrd_aff=0
Thrd Lev=1  , thrd_num=13   ,  thrd_aff=13
Thrd Lev=1  , thrd_num=4    ,  thrd_aff=4
Thrd Lev=1  , thrd_num=11   ,  thrd_aff=11
Thrd Lev=1  , thrd_num=10   ,  thrd_aff=10
Thrd Lev=1  , thrd_num=8    ,  thrd_aff=8
Thrd Lev=1  , thrd_num=9    ,  thrd_aff=9
Thrd Lev=1  , thrd_num=1    ,  thrd_aff=1
Thrd Lev=1  , thrd_num=3    ,  thrd_aff=3
Thrd Lev=1  , thrd_num=2    ,  thrd_aff=2
Thrd Lev=1  , thrd_num=12   ,  thrd_aff=12
Thrd Lev=1  , thrd_num=7    ,  thrd_aff=7
Thrd Lev=1  , thrd_num=6    ,  thrd_aff=6
Thrd Lev=1  , thrd_num=5    ,  thrd_aff=5
Thrd Lev=1  , thrd_num=14   ,  thrd_aff=14
Thrd Lev=1  , thrd_num=15   ,  thrd_aff=15
Matrix order         = 1000
Number of threads     = 16
Rate : 38765.867067  +/- 3303.460980 MFlops/s
```

Obtain Optimal Affinity on Cori KNL Example

KNL Compute Nodes

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.

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

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)

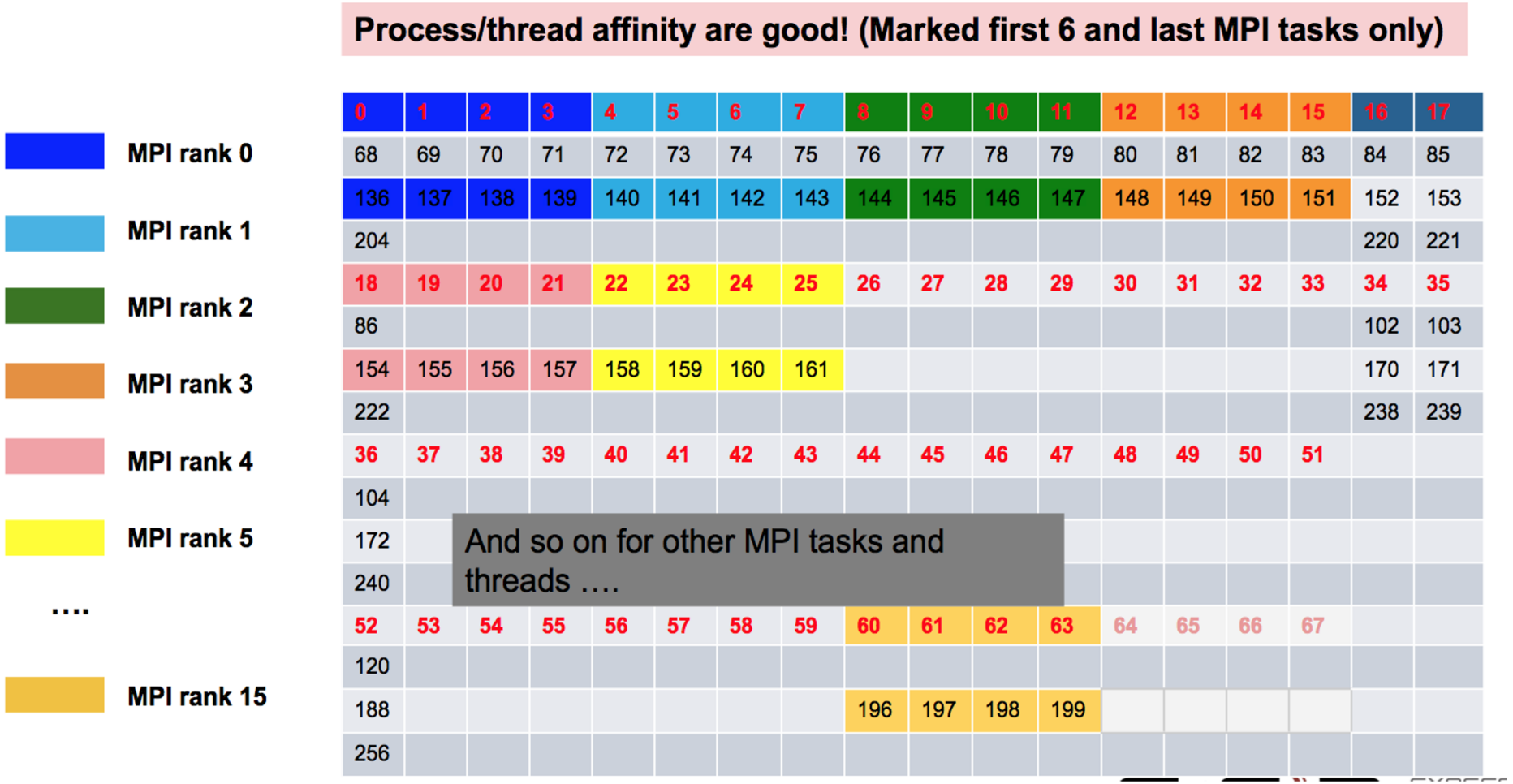
Example mpirun or srun Commands: Fix the Problem

- 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!



Intel KNL Quad,Flat Node Example

Cori KNL quad,flat node example
68 cores (272 CPUs)

% 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 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237
238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267
268 269 270 271
node 0 size: 96723 MB
node 0 free: 93924 MB
node 1 cpus:
node 1 size: 16157 MB
node 1 free: 16088 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

- The quad,flat mode has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory).
- And NUMA node 1 has MCDRAM (high bandwidth memory).

Essential Runtime Settings for KNL MCDRAM Memory Affinity

- In quad, cache mode, no special setting is needed to use MCDRAM
- In quad,flat mode, using quad,flat as an example
  - NUMA node 1 is MCDRAM
- Enforced memory mapping to MCDRAM
  - If using >16 GB, malloc will fail
  - Use "**numactl -m 1 ./myapp**" as the executable (instead of "./myapp")
- Preferred memory mapping to MCDRAM
  - If using >16 GB, malloc will spill to DDR
  - Use "**numactl -p 1 ./myapp**" as the executable (instead of "./myapp")

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)