An Introduction to Parallel programming with OpenMP*

Tim Mattson

Intel Corp.

Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/ATPESC.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
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP
  - Thread Affinity and Data Locality
  - Thread Private Data

The Loop Worksharing Constructs

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

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

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

Loop construct name:

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

Loop Worksharing Constructs: The schedule clause

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

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

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

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
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP
  - Thread Affinity and Data Locality
  - Thread Private Data

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.

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.

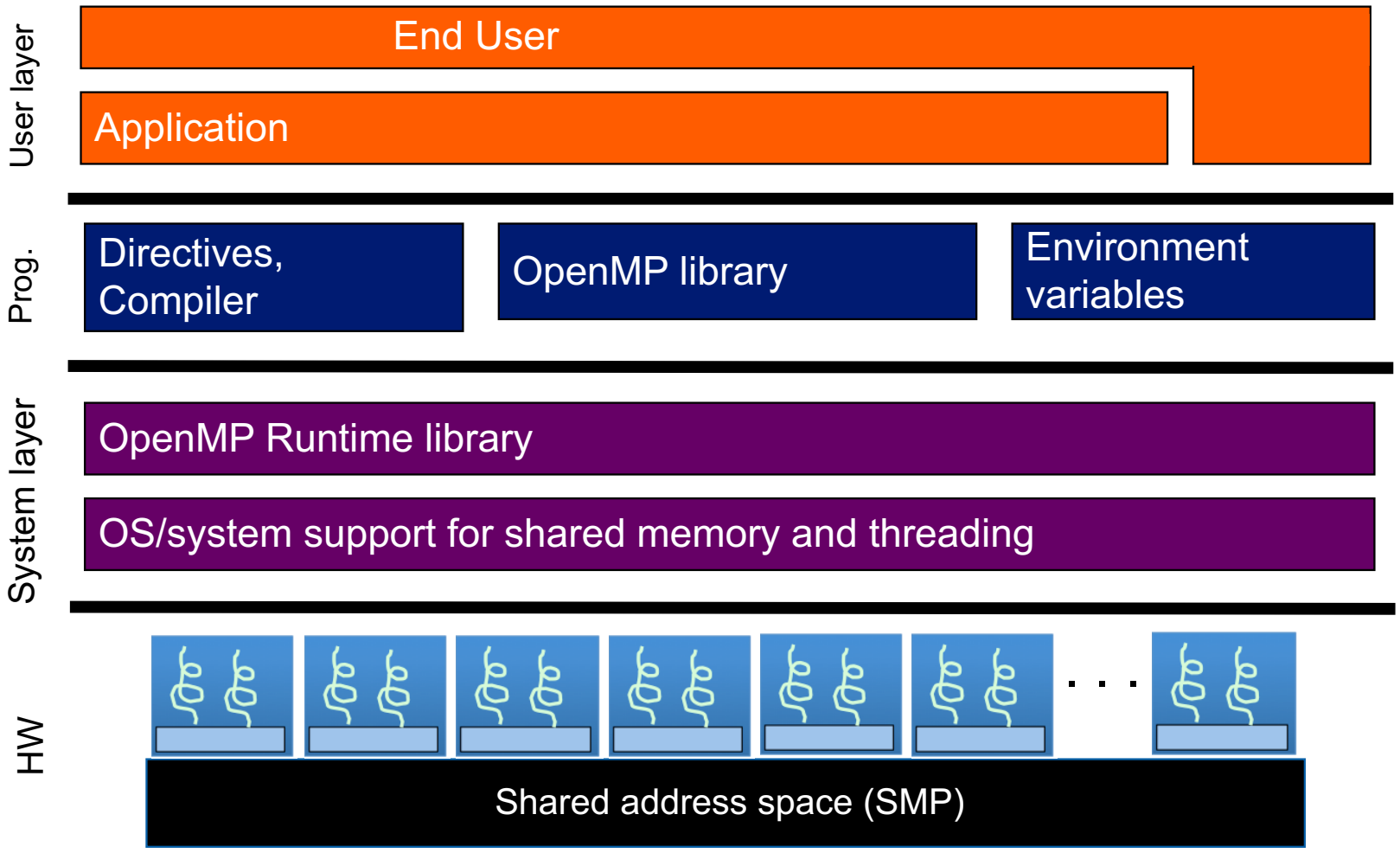
sequential	0.0019 secs
critical	0.079 secs
Locks per bin	0.029 secs
Reduction, replicated histogram array	0.00097 secs

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)

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
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP
  - Thread Affinity and Data Locality
  - Thread Private Data

OpenMP basic definitions: Basic Solution stack



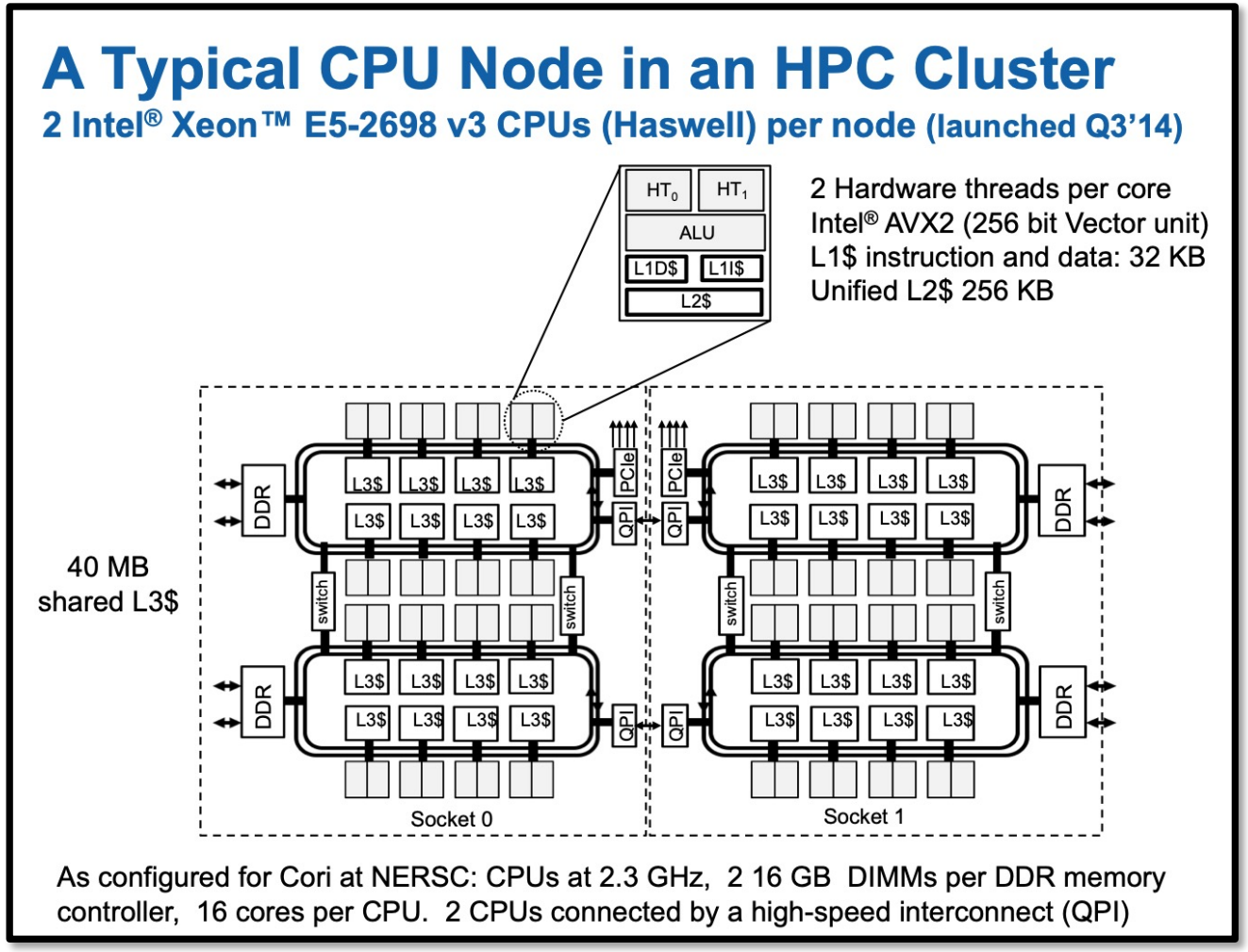
In learning OpenMP, you consider a Symmetric Multiprocessor (SMP) ….
i.e. lots of threads with "**equal cost access**" to memory

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

Does this look like an SMP node to you?

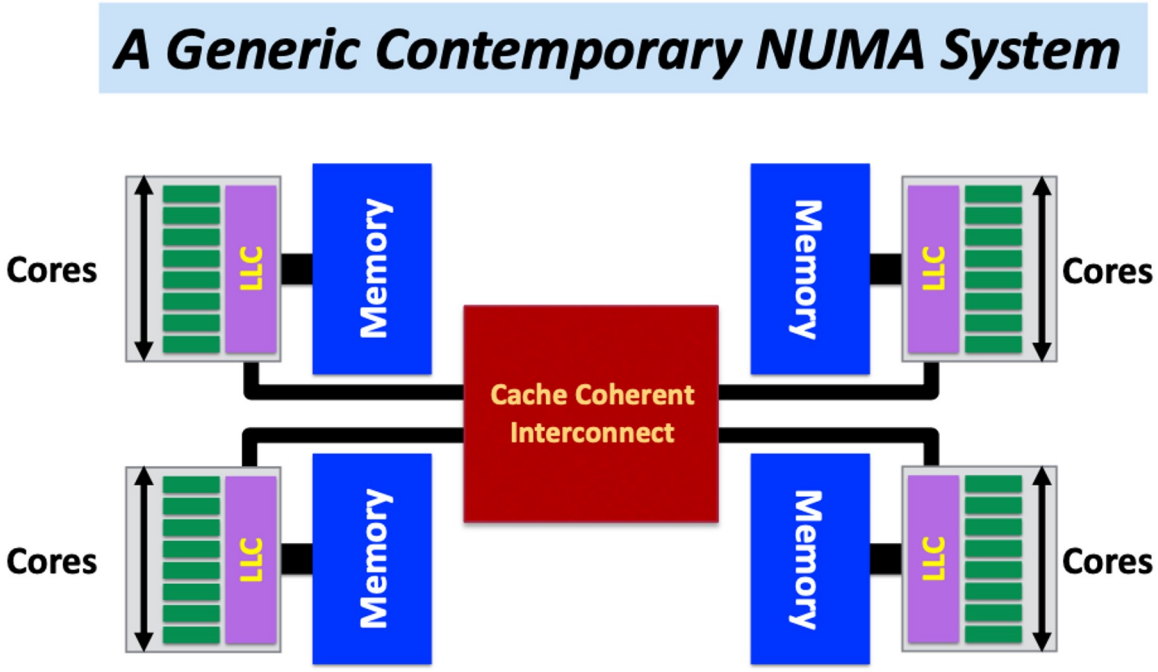
There may be a single address space, but there are multiple levels of non-uniformity to the memory. This is a **N**on-**U**niform **M**emory **A**rchitecture (NUMA)



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

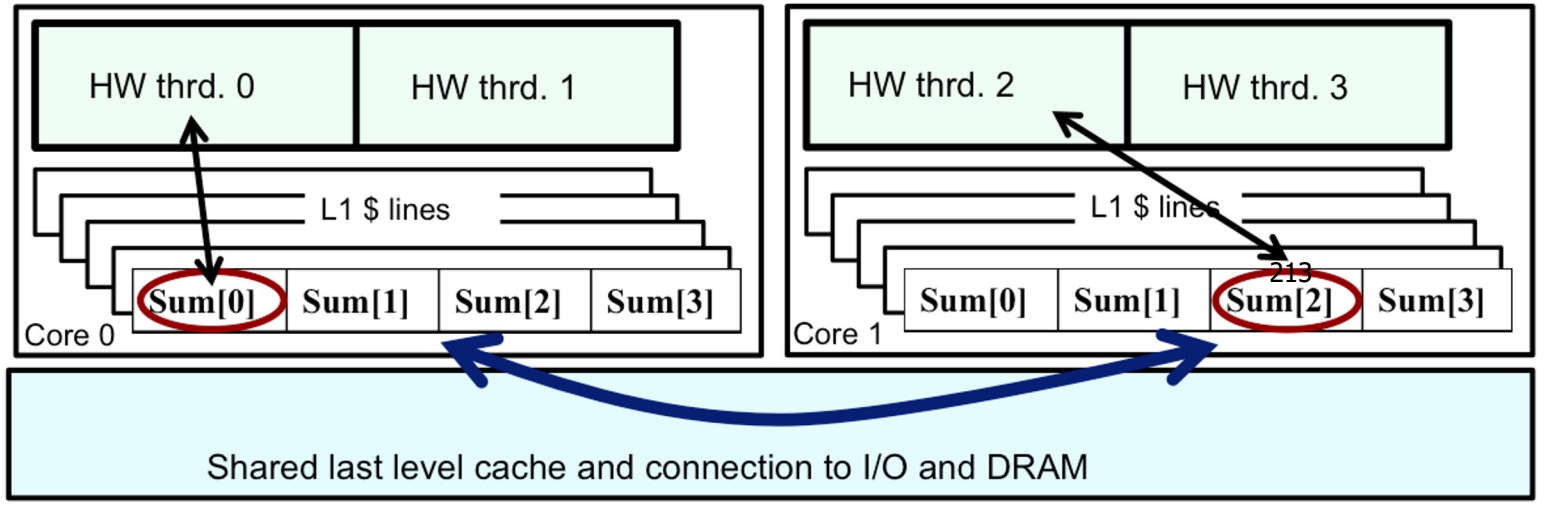
Memory Locality

- 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

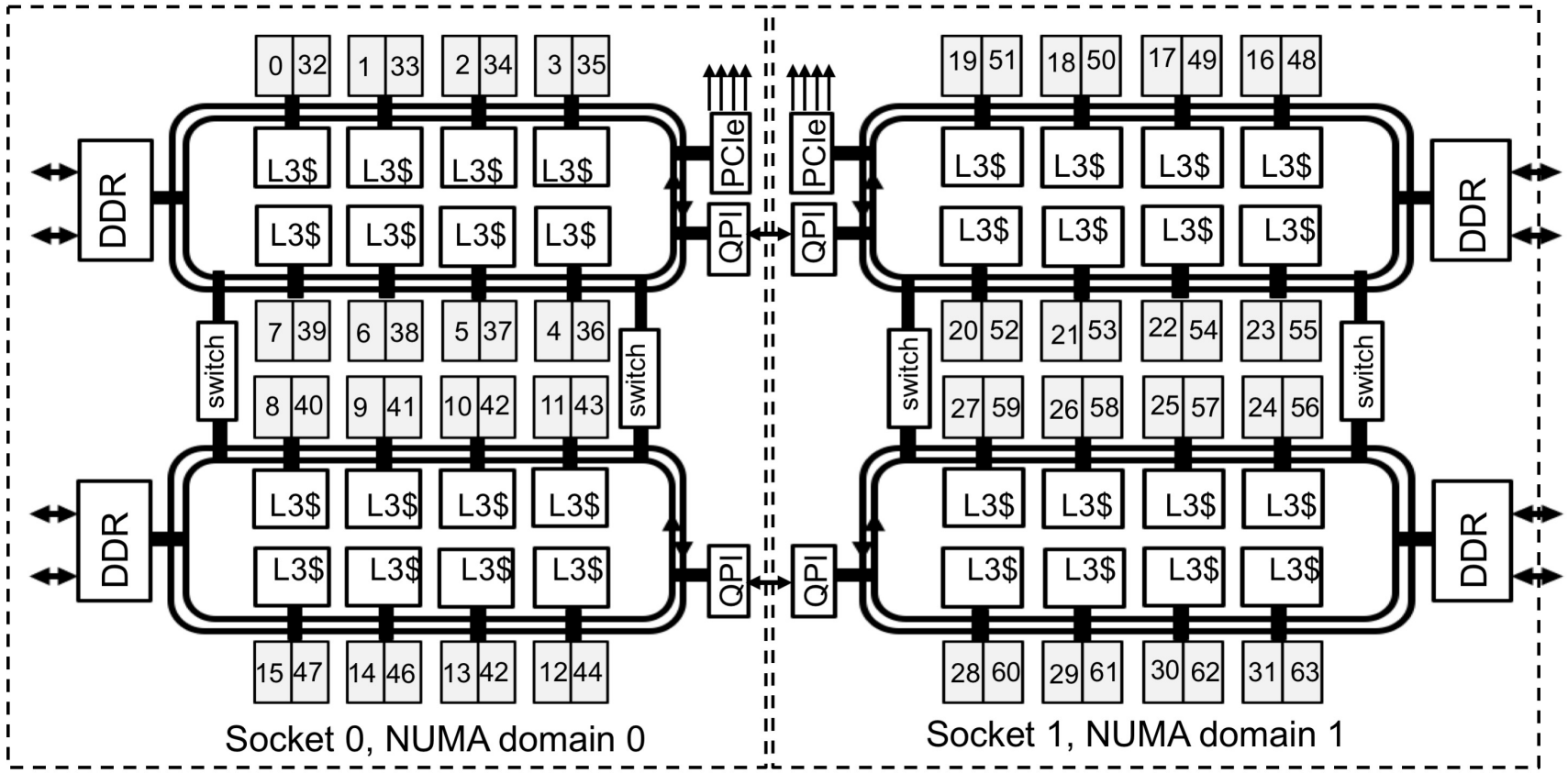
Cache Coherence and False Sharing

- 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

Process / Thread / Memory Affinity (1)

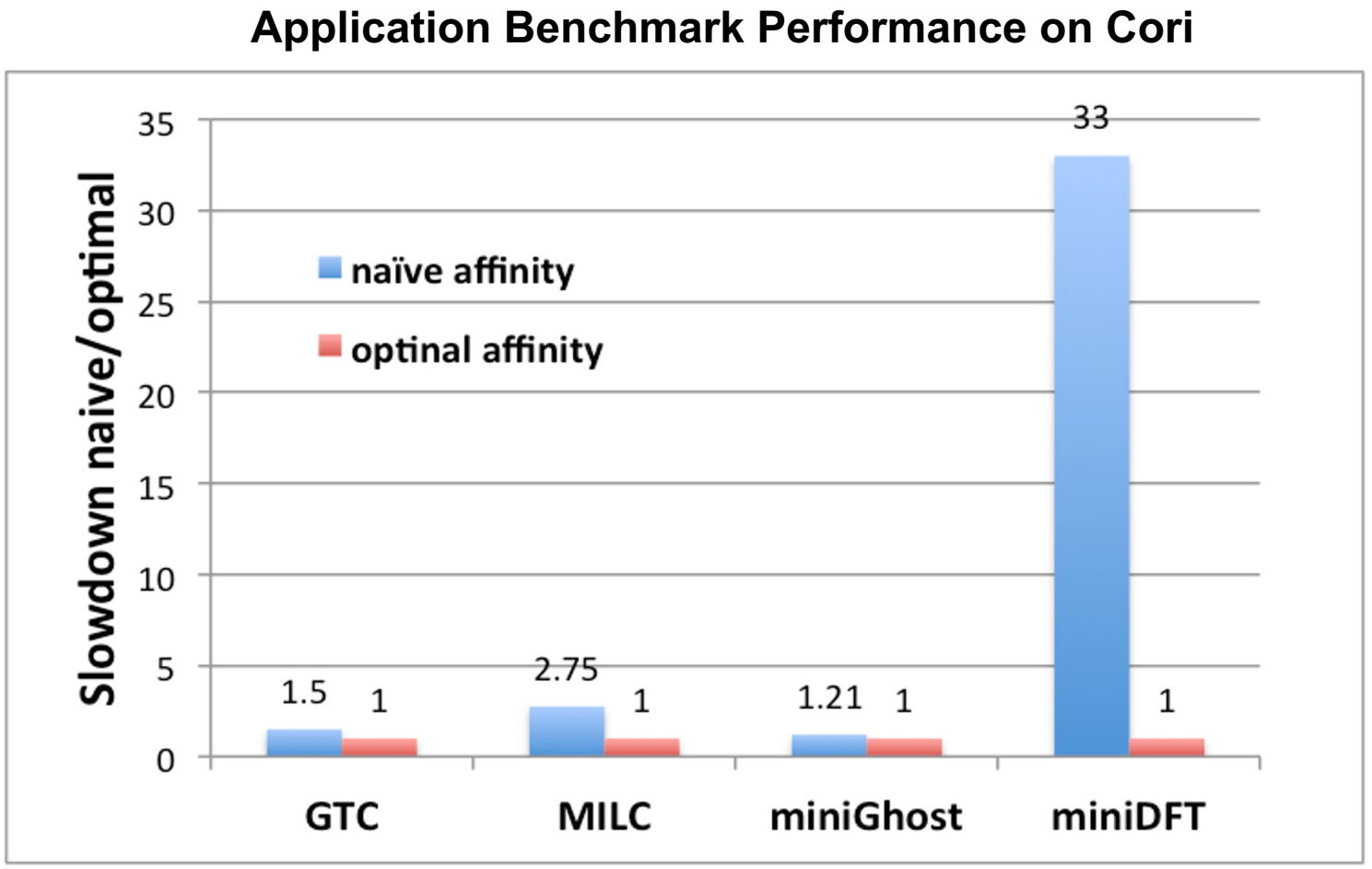
- Process Affinity: also called "CPU pinning", binds processes (MPI tasks, etc.) to a CPU or a range of CPUs on a node
  - It is important to spread MPI ranks evenly onto cores in different NUMA domains
- Thread Affinity: further binding threads to CPUs that are allocated to their parent process
  -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

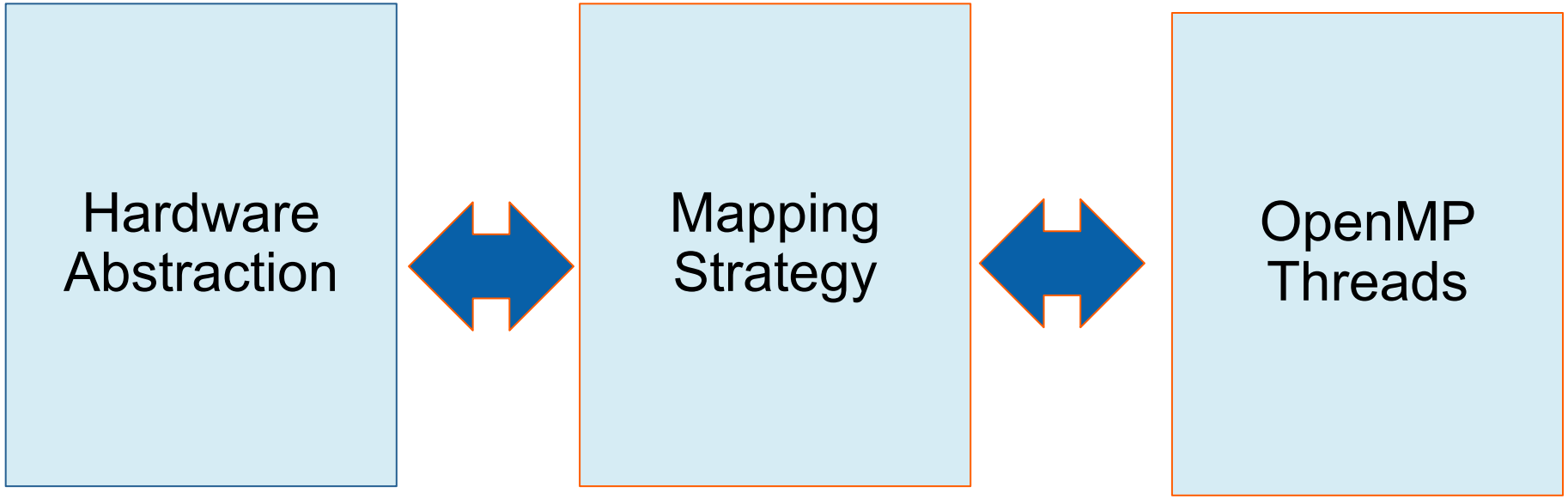
- Correct process, thread and memory affinity is the basis for getting optimal performance. It is also essential for guiding further performance optimizations.

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

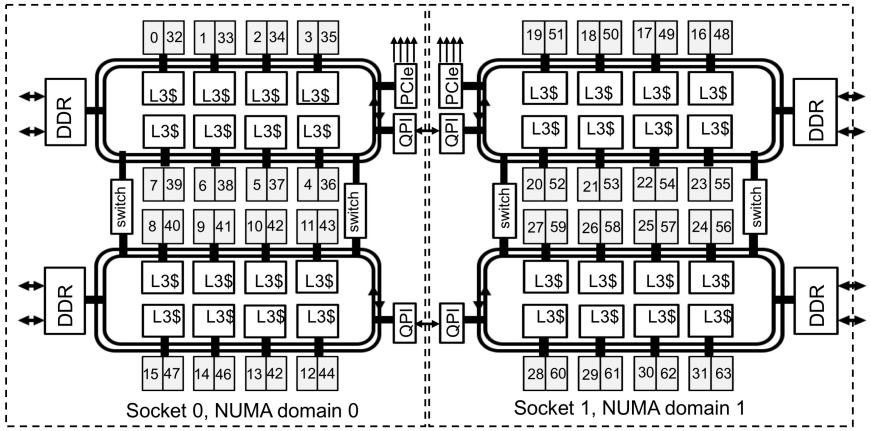
- Control the places where threads are mapped
  - Place threads onto cores to optimize performance
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- 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.

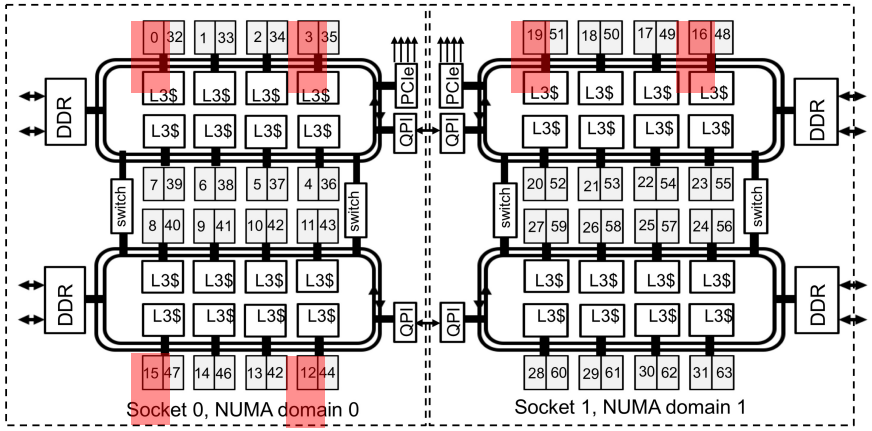


- A place: numbers between { }:

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

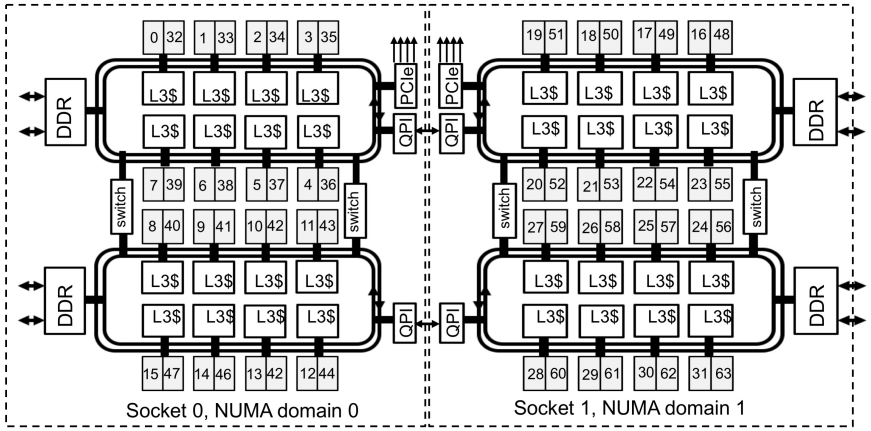
- A place defines where threads can run

```
> 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}

Default Stride is 1

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

- Can define multiple places:

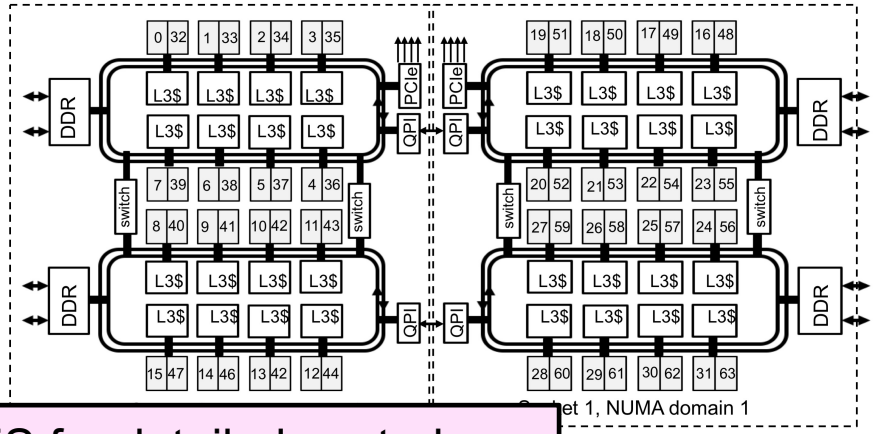
OMP_PLACES="{0,1,2,3},{4,6,8},{9,10,11,12}"

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

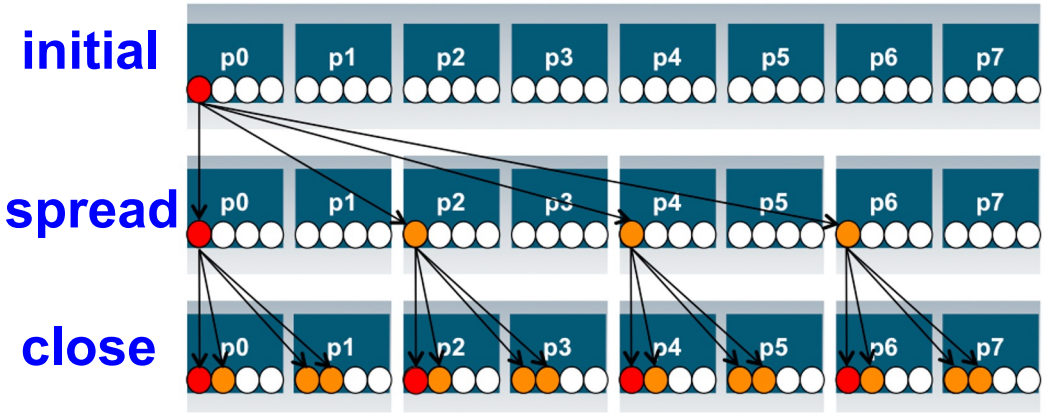
- A combination of OpenMP environment variables and runtime flags are needed for different compilers and different batch schedulers on different systems

```
#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

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

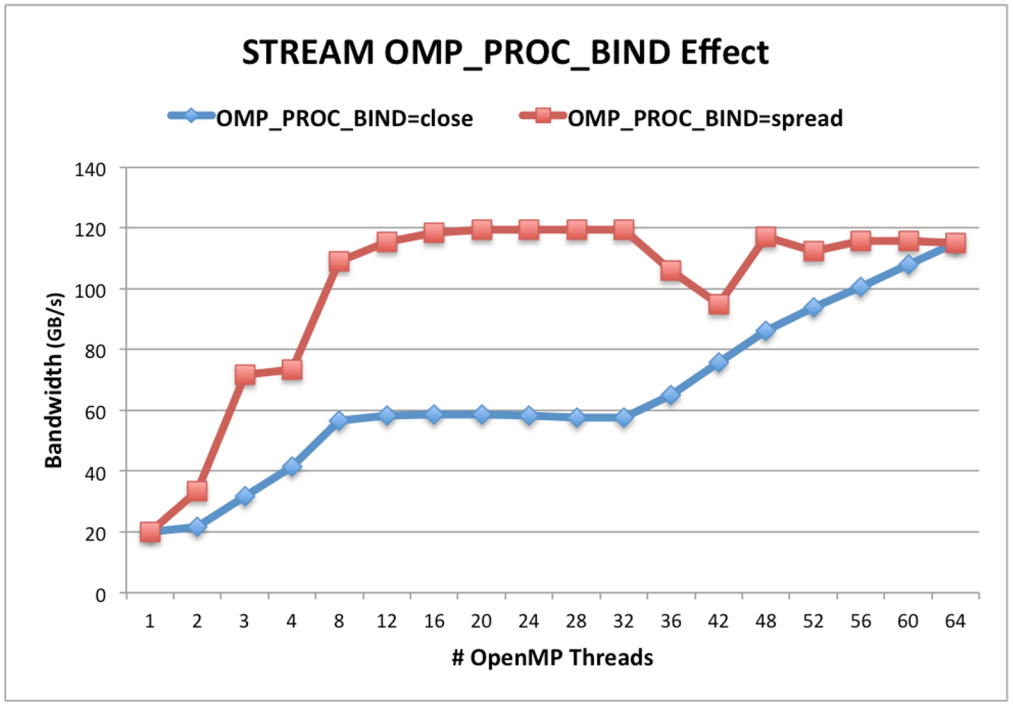
OMP_PROC_BIND Choices for STREAM Benchmark

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



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

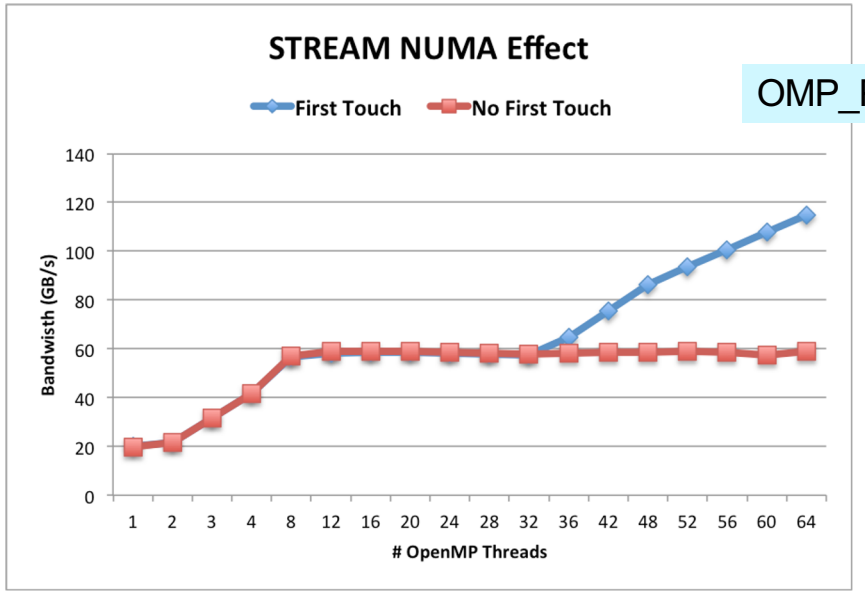
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

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

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);
}
```

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)

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
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP
  - Thread Affinity and Data Locality
  - Thread Private Data

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)

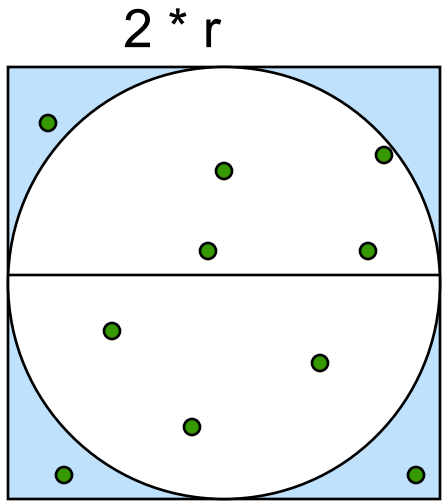
Use threadprivate to create a counter for each thread.

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

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

Exercise: Monte Carlo Calculations

Using random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Exercise: Monte Carlo pi (cont)

- We provide three files for this exercise
  - pi_mc.c: the Monte Carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program.
- Run it multiple times with varying numbers of threads.
- Is the program working correctly? Is there anything wrong?

Parallel Programmers love Monte Carlo algorithms

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
   long i;      long Ncirc = 0;       double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(0,-r, r);  // The circle and square are centered at the origin
   #pragma omp parallel for private (x, y) reduction (+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
      x = random();         y = random();
      if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

Random Numbers: Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

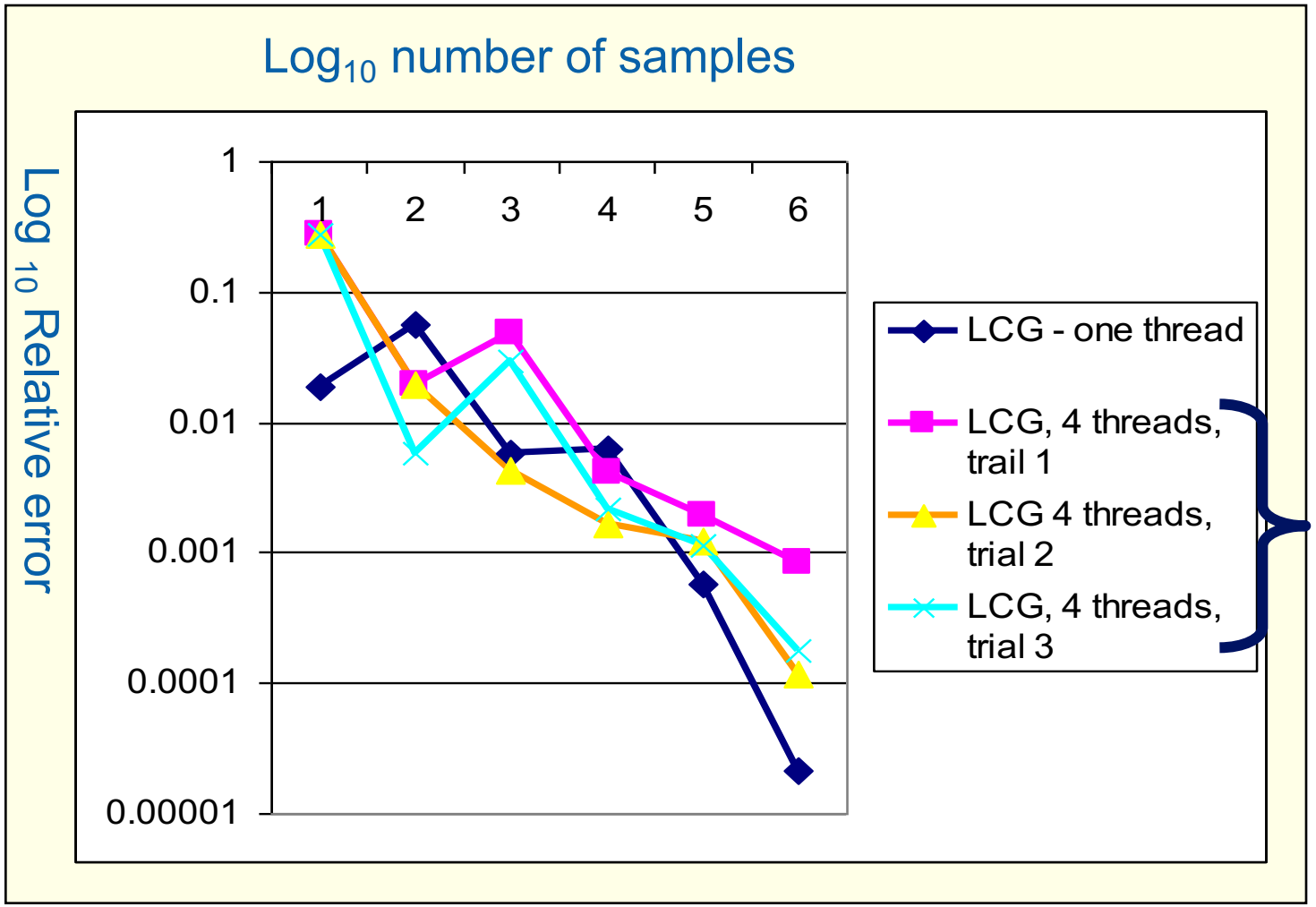
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

Exercise: Monte Carlo pi (cont)

- Create a threadsafe version of the monte carlo pi program
- Do not change the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be thread-safe
- Verify that the program is thread safe by running multiple times for a fixed number of threads.
- Any concerns with the program behavior?

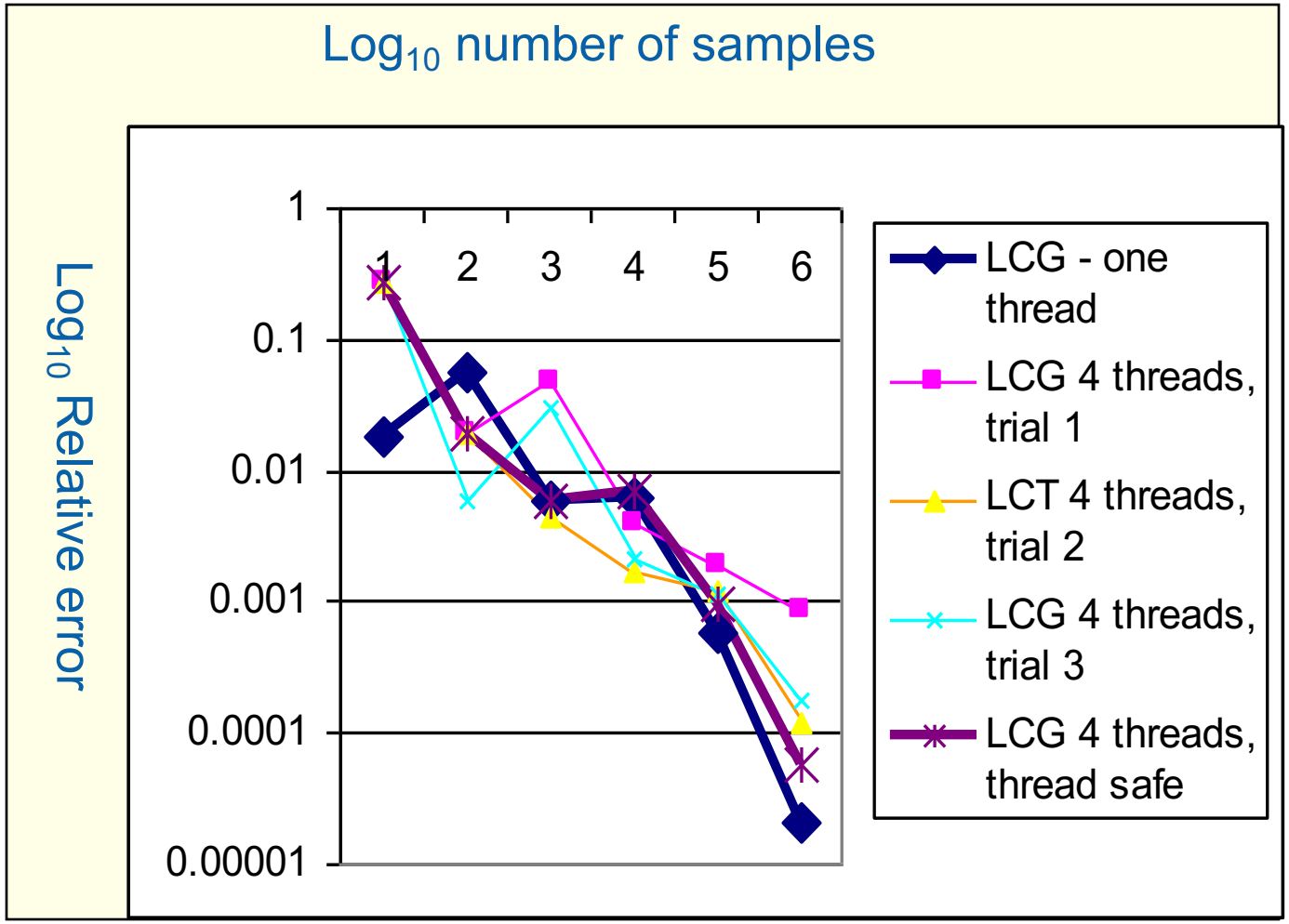
LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread Safe Random Number Generators



Thread safe version gives the same answer each time you run the program.

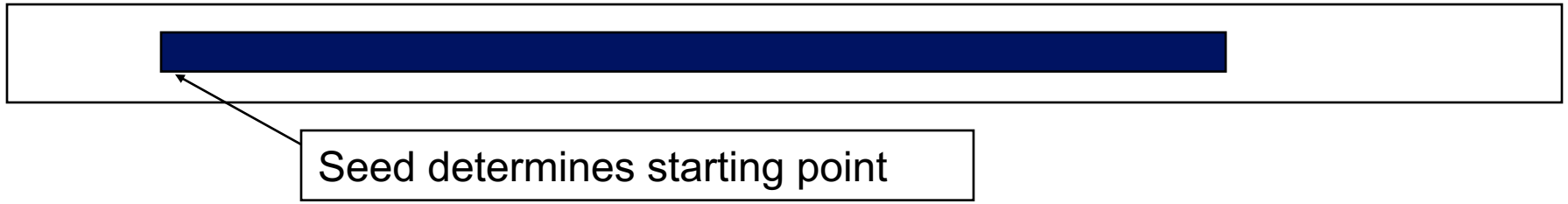
But for large number of samples, its quality is lower than the one thread result!

Why?

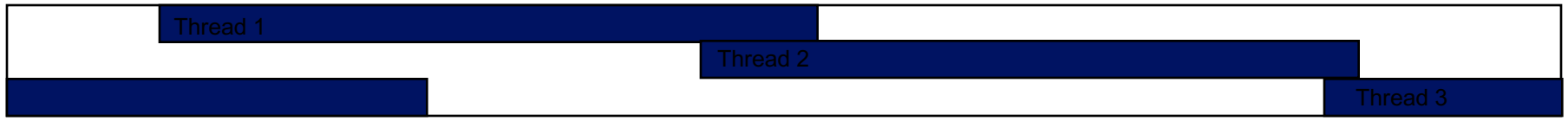
Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range



- Grab arbitrary seeds and you may generate overlapping sequences
  - E.g. three sequences … last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Replicate and Pray
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values "round robin" as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.

If done right, can generate the same sequence regardless of the number of threads …

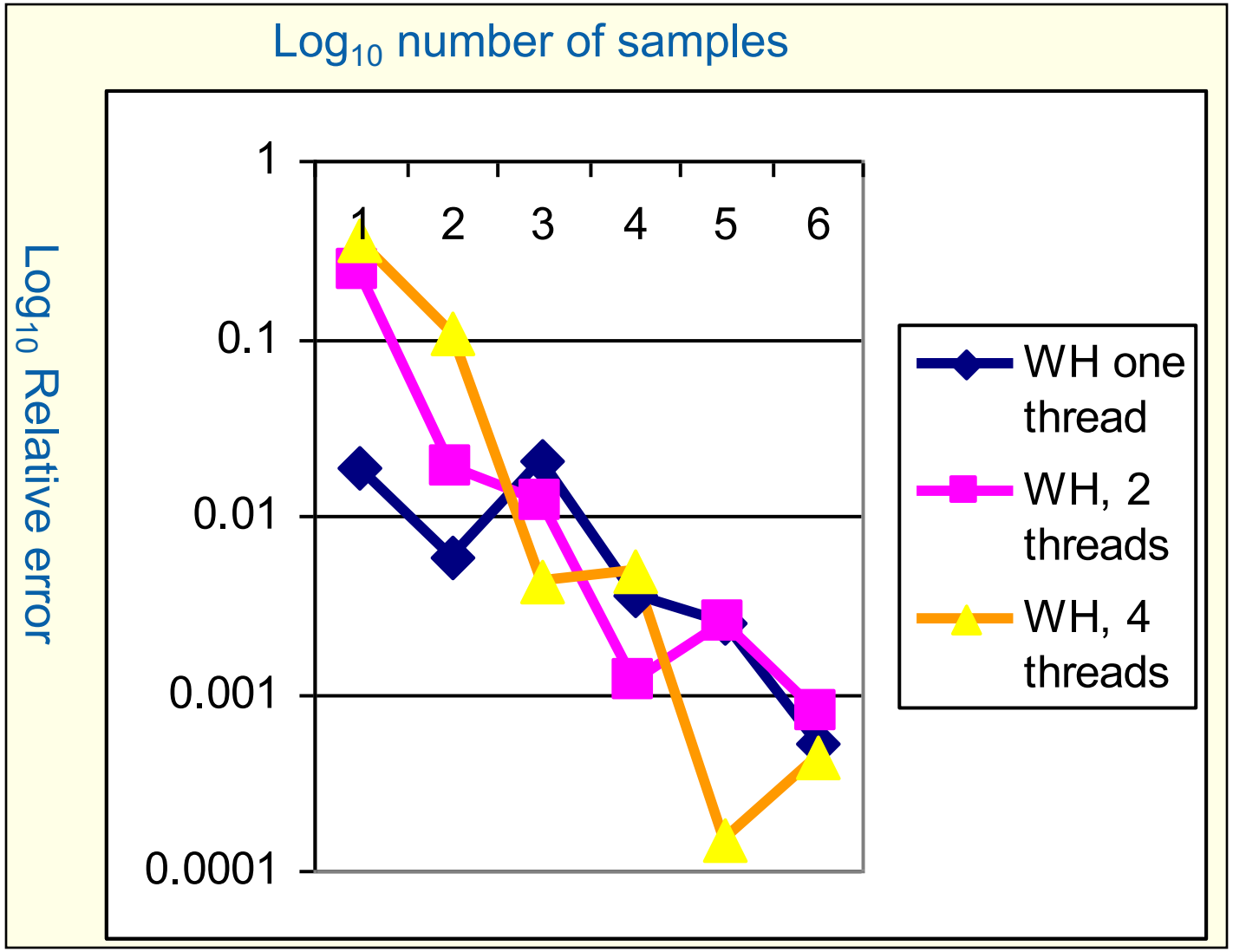
Nice for debugging, but not really needed scientifically.

- Other than "replicate and pray", these are difficult to implement. Be smart … get a math library that does it right.

Intel's Math kernel Library supports a wide range of parallel random number generators.

For an open alternative, the state of the art is the Scalable Parallel Random Number Generators Library (SPRNG): http://www.sprng.org/ from Michael Mascagni's group at Florida State University.

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog Method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the i^{th} number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
    iseed = PMOD/MULTIPLIER;     // just pick a seed
    pseed[0] = iseed;
    mult_n = MULTIPLIER;
    for (i = 1; i < nthreads; ++i)
   {
       iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
       pseed[i] = iseed;
       mult_n = (mult_n * MULTIPLIER) % PMOD;
   }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.