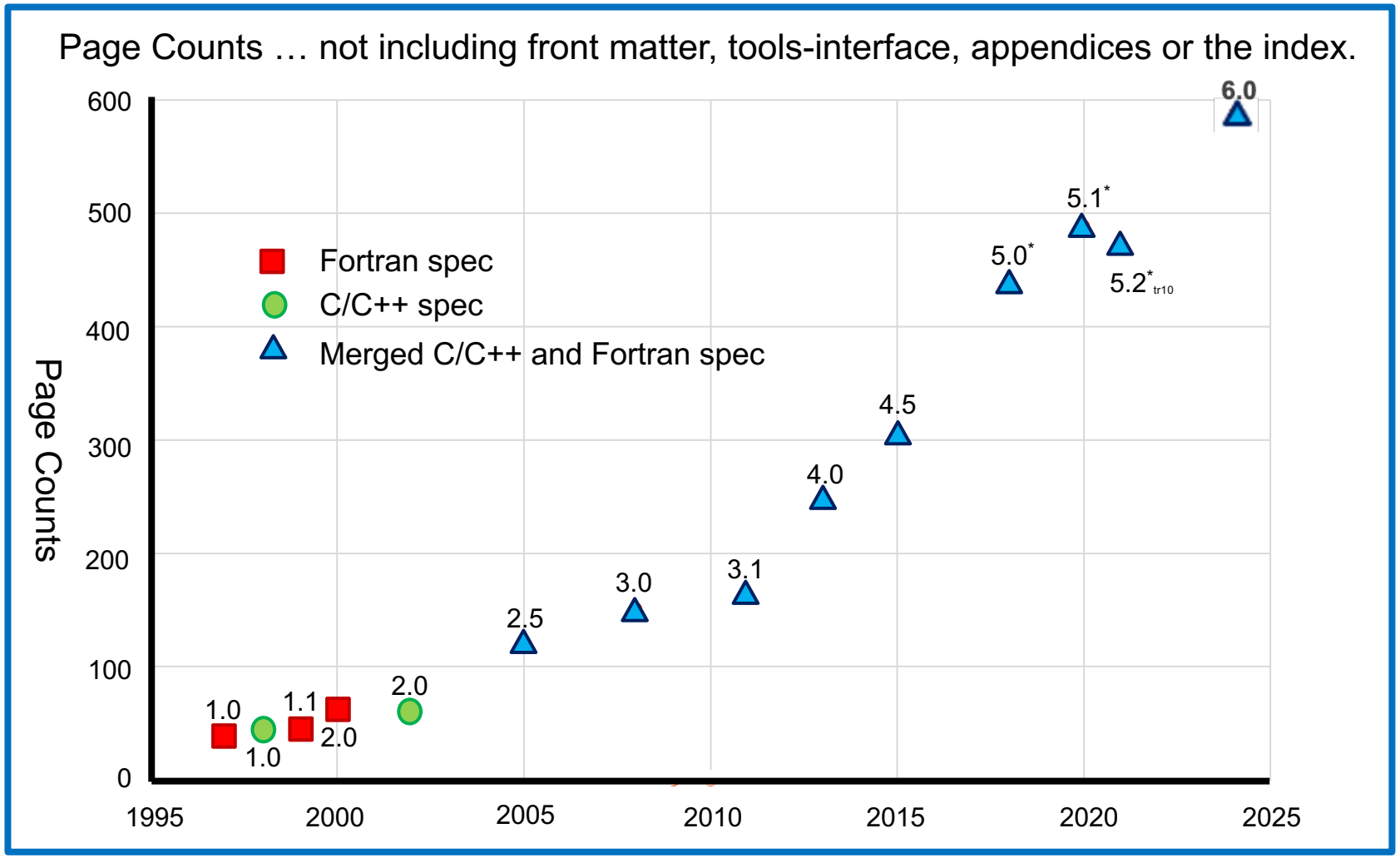
OpenMP: Its more than “parallel for” on SMP systems

Tim Mattson, University of Bristol and Merly.ai

merly.ai

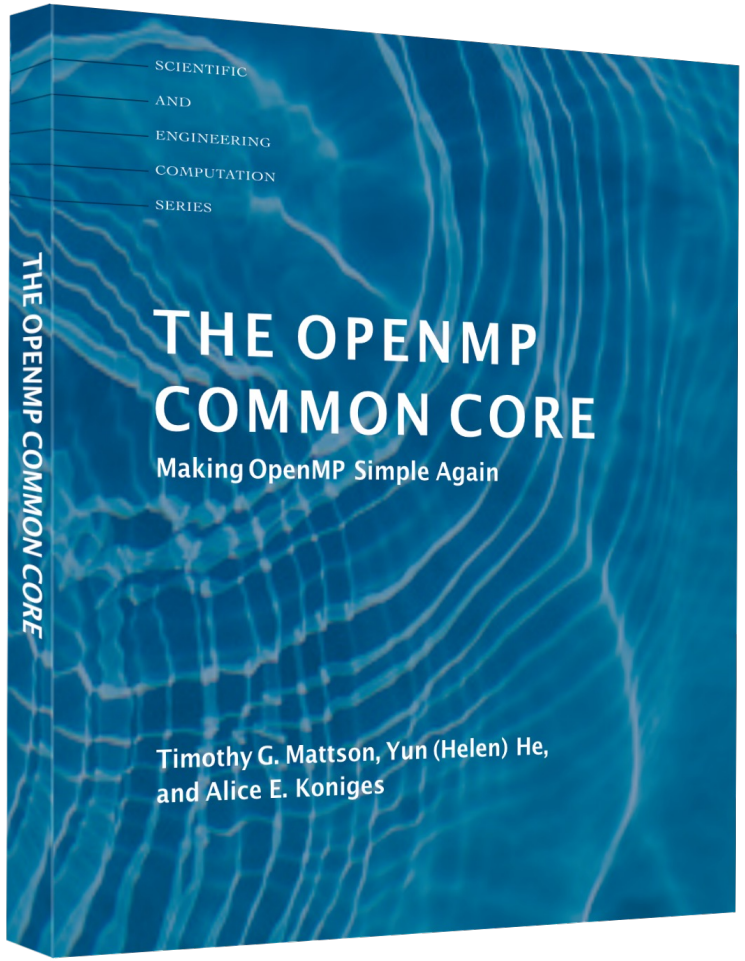
The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers



The OpenMP specification is so long and complex that few (if any) humans understand the full document

The OpenMP Common Core



For many years now, we've been teaching the subset of OpenMP that is most commonly used. We call this the

OpenMP Common Core

We even wrote a book about it.

The list of items in the common core were determined by experience/anecdote … we didn't have hard data to drive the analysis.

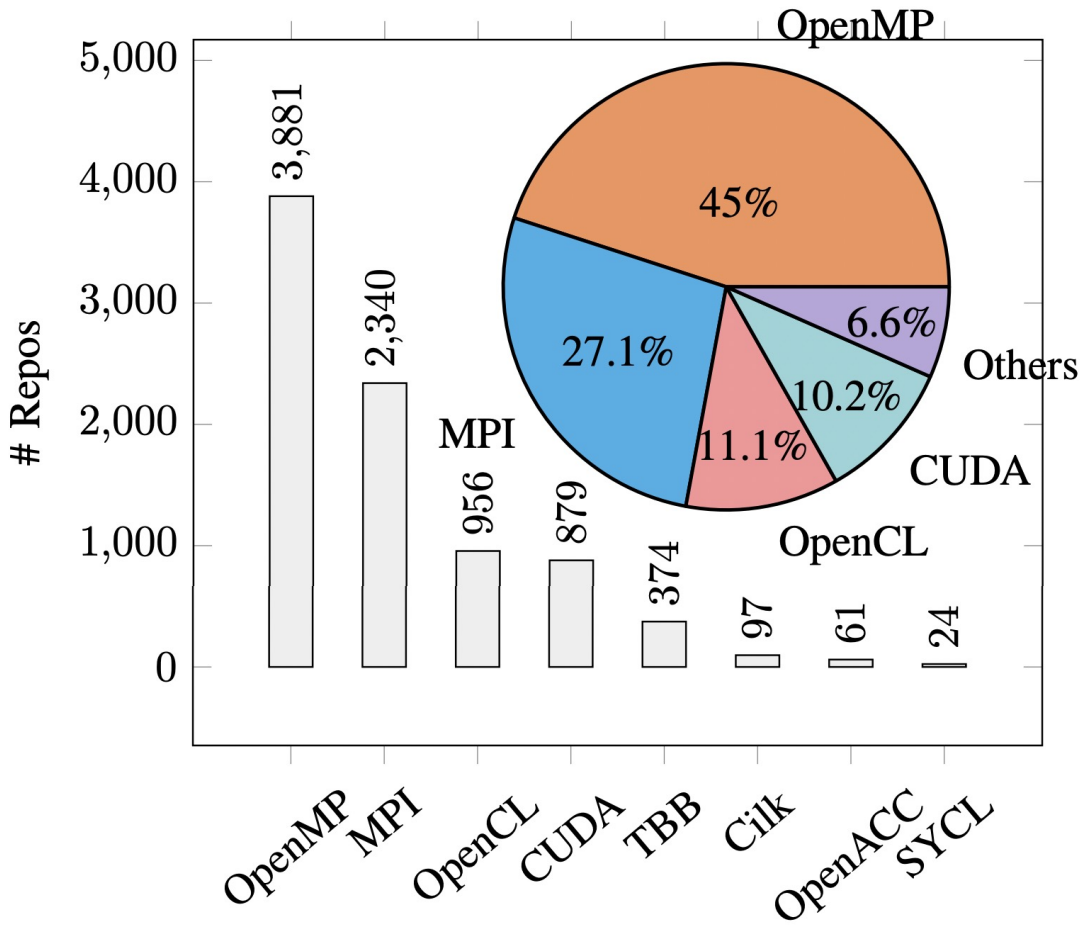
The OpenMP Common Core
#pragma omp parallel
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()
double omp_get_wtime()
setenv OMP_NUM_THREADS N
#pragma omp barrier #pragma omp critical
#pragma omp for #pragma omp parallel for
reduction(op:list)
schedule (static [,chunk]) schedule(dynamic [,chunk])
shared(list), private(list), firstprivate(list)
default(none)
nowait
#pragma omp single
#pragma omp task #pragma omp taskwait

HPCorpus data set for training LLM models for parallel programming

Scan all C, C++ and Fortran codes from github with “last updated” dates between 2012 and mid 2023

	Repos	Size(GB)	Files (#)	Functions (#)
C	144,522	46.23	4,552,736	87,817,591
C++	150,481	26.16	4,735,196	68,233,984
Fortran	3,683	0.68	138,552	359,272

HPCorpus Let us assess usage of HPC programming models … OpenMP is number 1!!!



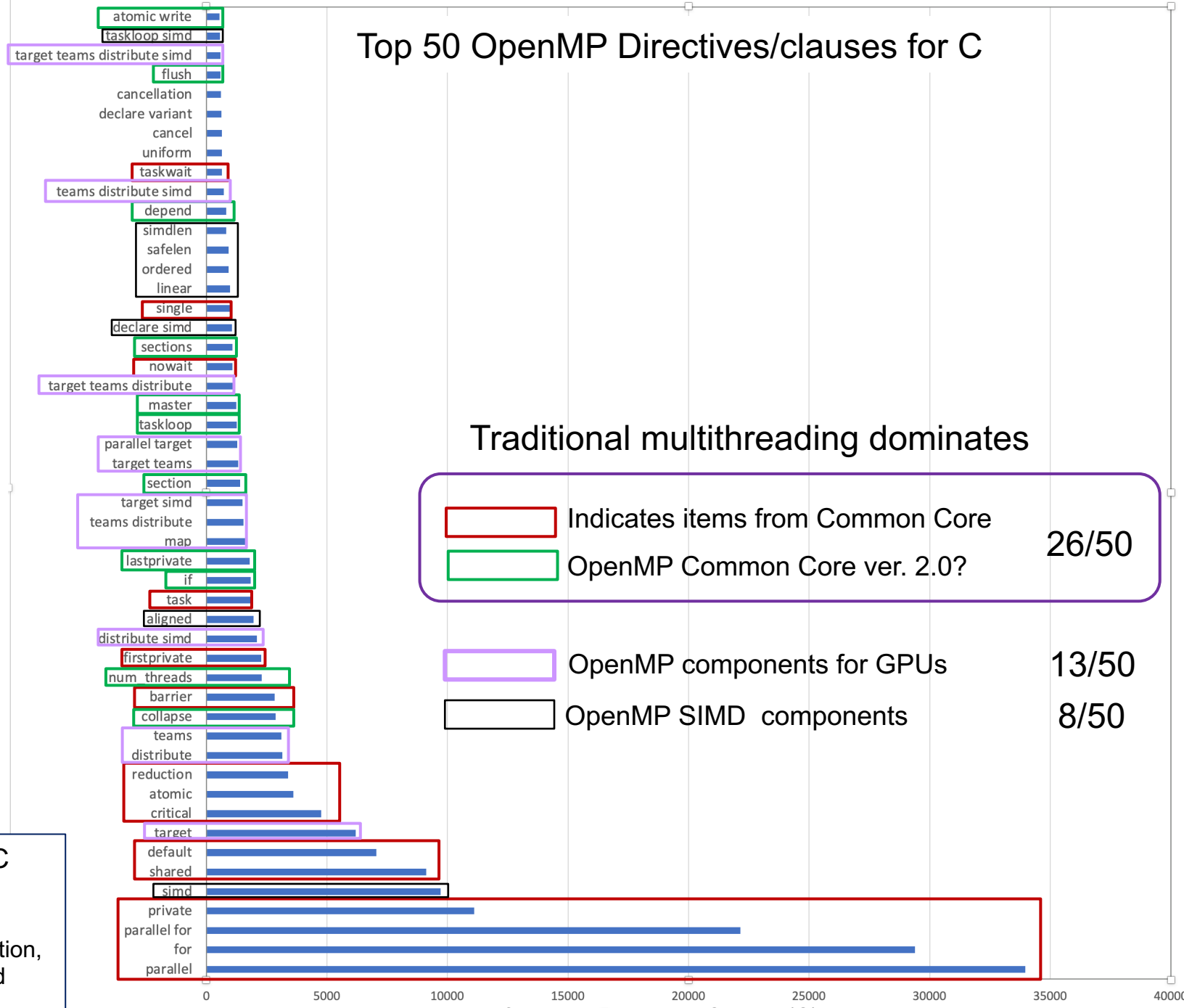
Aggregate numbers over all repositories from 2013 to 2023

Note: since we did not collect files with .cu or .cuf suffixes, we may have undercounted CUDA usage in HPCorpus.

What are people actually using from OpenMP

With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

This data was constructed by summing up counts for different directives and clauses across time from 2013 to the middle of 2023.



HPCorpus … a data set created by scraping ”all” HPC codes from github written in C, C++ and Fortran.

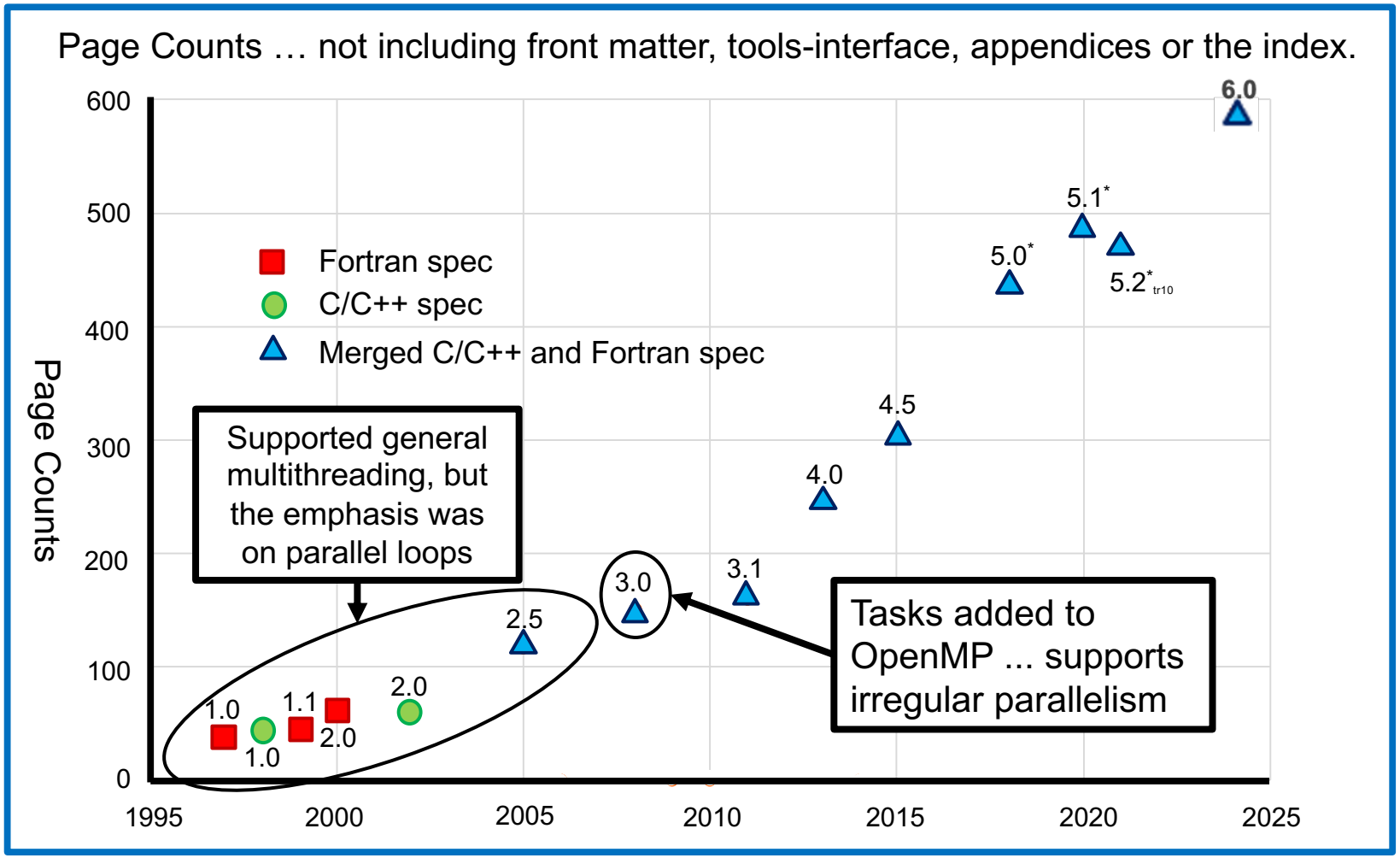
Quantifying OpenMP: Statistical Insights into Usage and Adoption, Tal Kadosh, Niranjan Hasabnis, Tim Mattson, Yuval Pinter, and Gal Oren, IEEE HPEC 2023

Outline

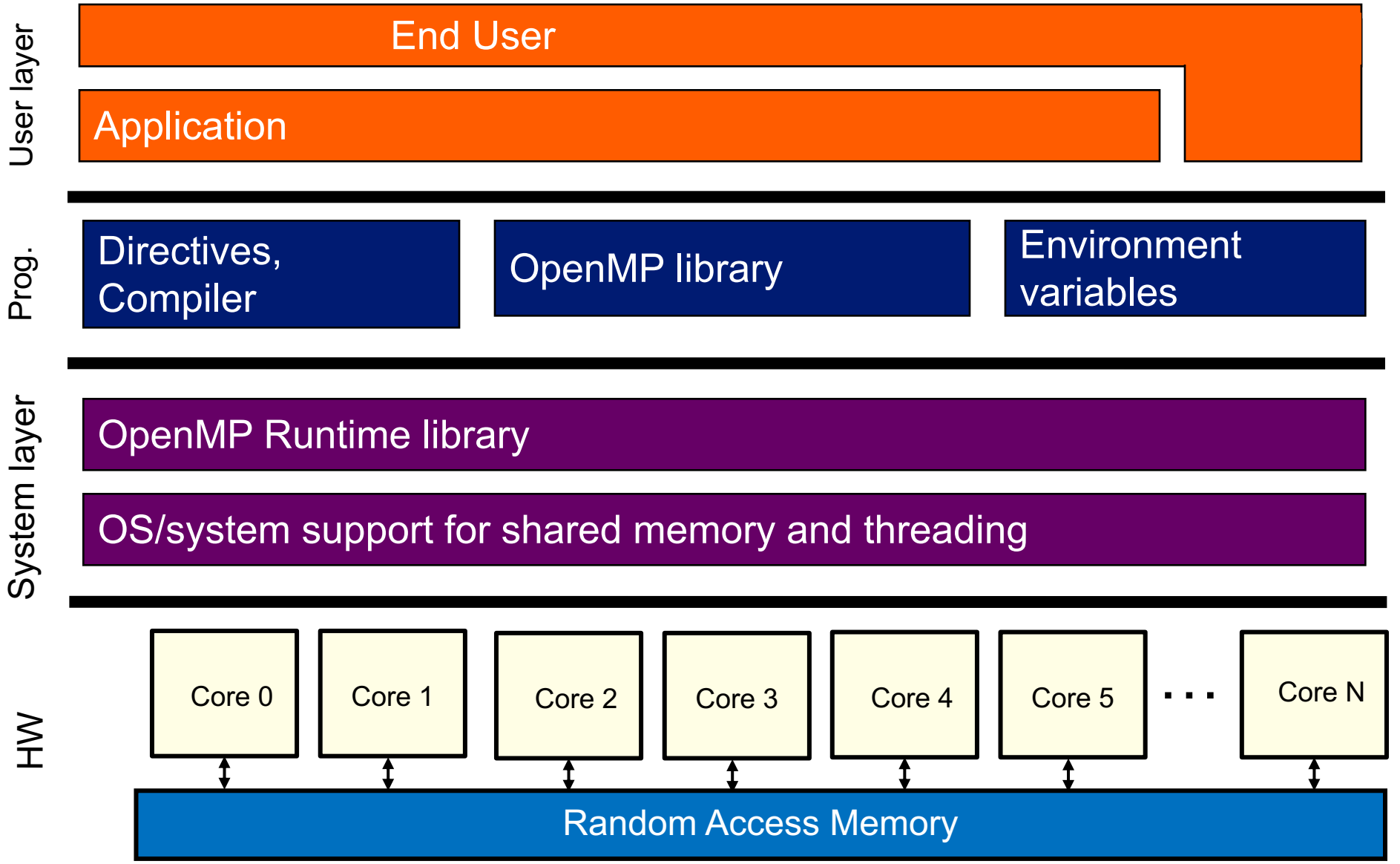
- OpenMP and Real Hardware
- Optimizing code for NUMA systems
- GPU Programming and OpenMP
- Synchronization and the OpenMP memory model
- Random numbers … an example of how to build threadsafe libraries

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers



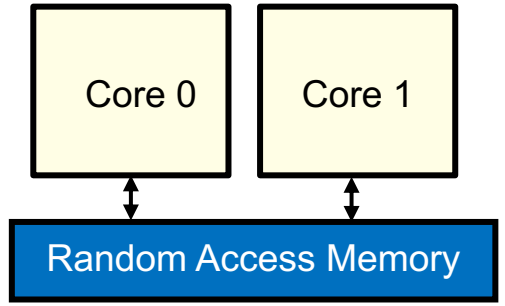
OpenMP Basic Definitions: Basic Solution Stack

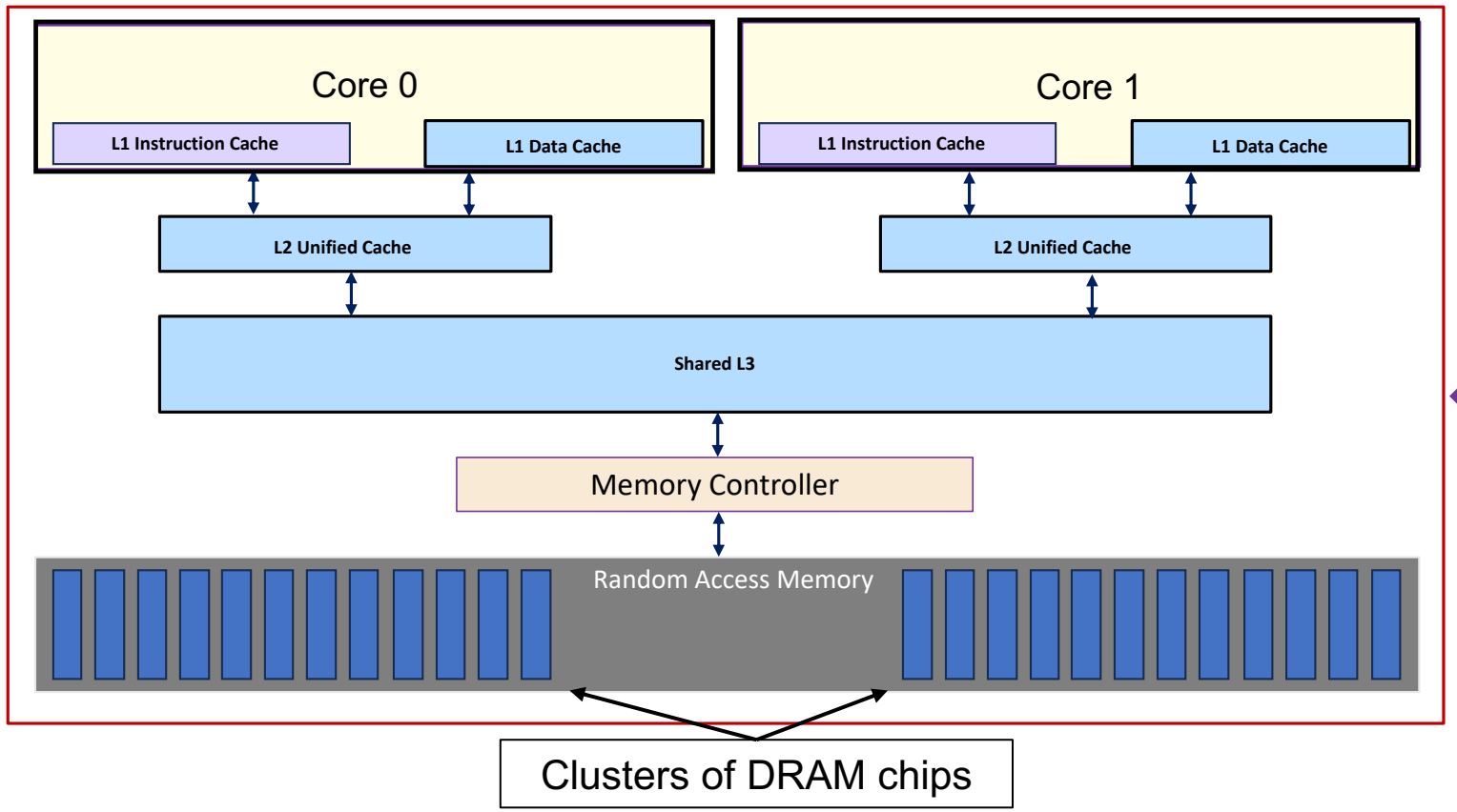


A Symmetric Multiprocessor Case with a shared address space ….
Lots of cores treated equally by the OS and with “equal cost access” to memory

SMP is a lie … The memory is Non-Uniform

We like to draw pictures like this





But reality is much more complex

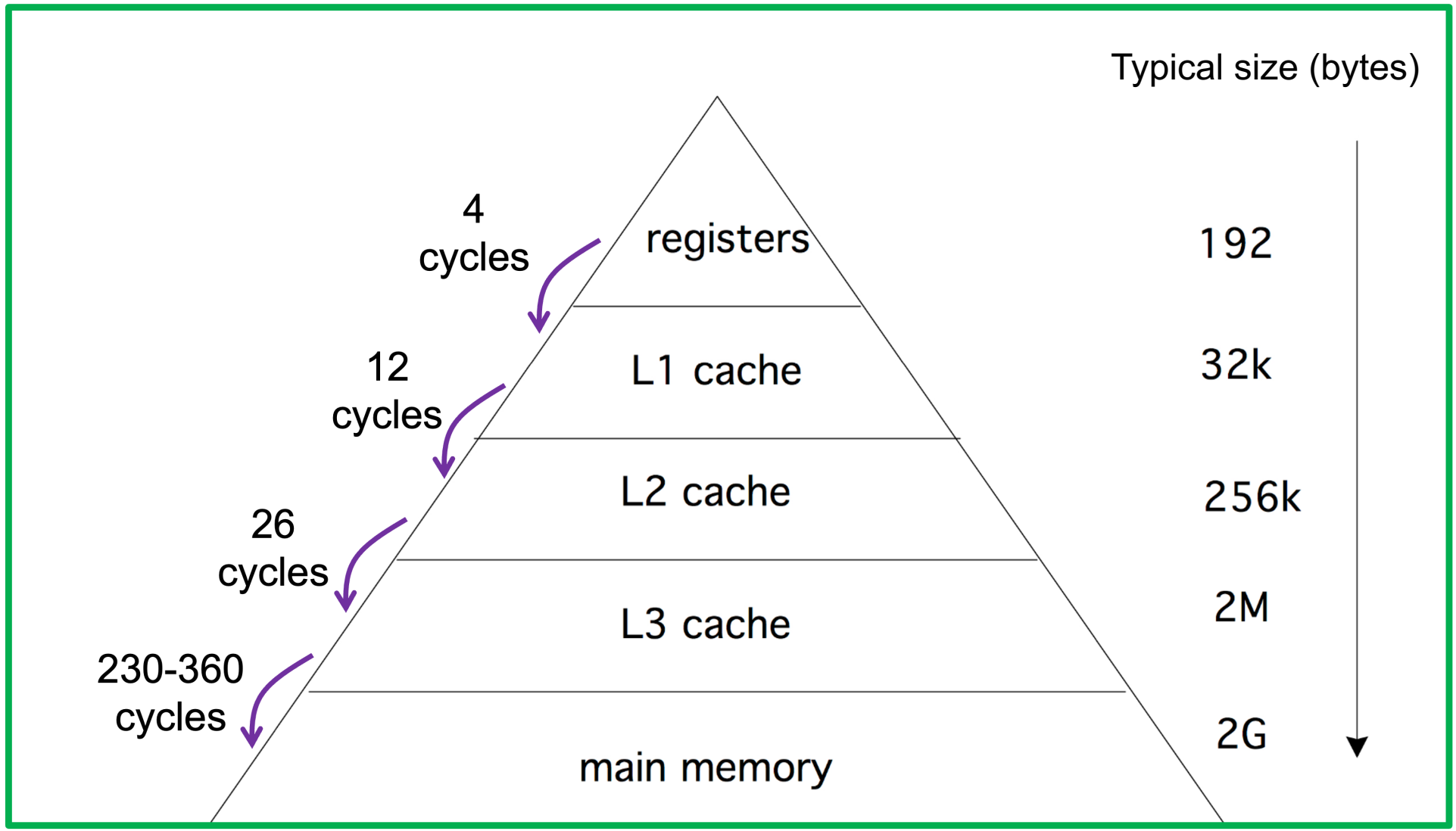
SMP: Symmetric Multiprossesor

DRAM: Dynamic Random Access Memory

L1, L2, L3 pronounced Level 1, Level 2, Level 3. Also written sometimes as $L1, $L2, $L3

L2 is called a unified cache since it holds both instructions and data

Latencies across the Memory Hierarchy



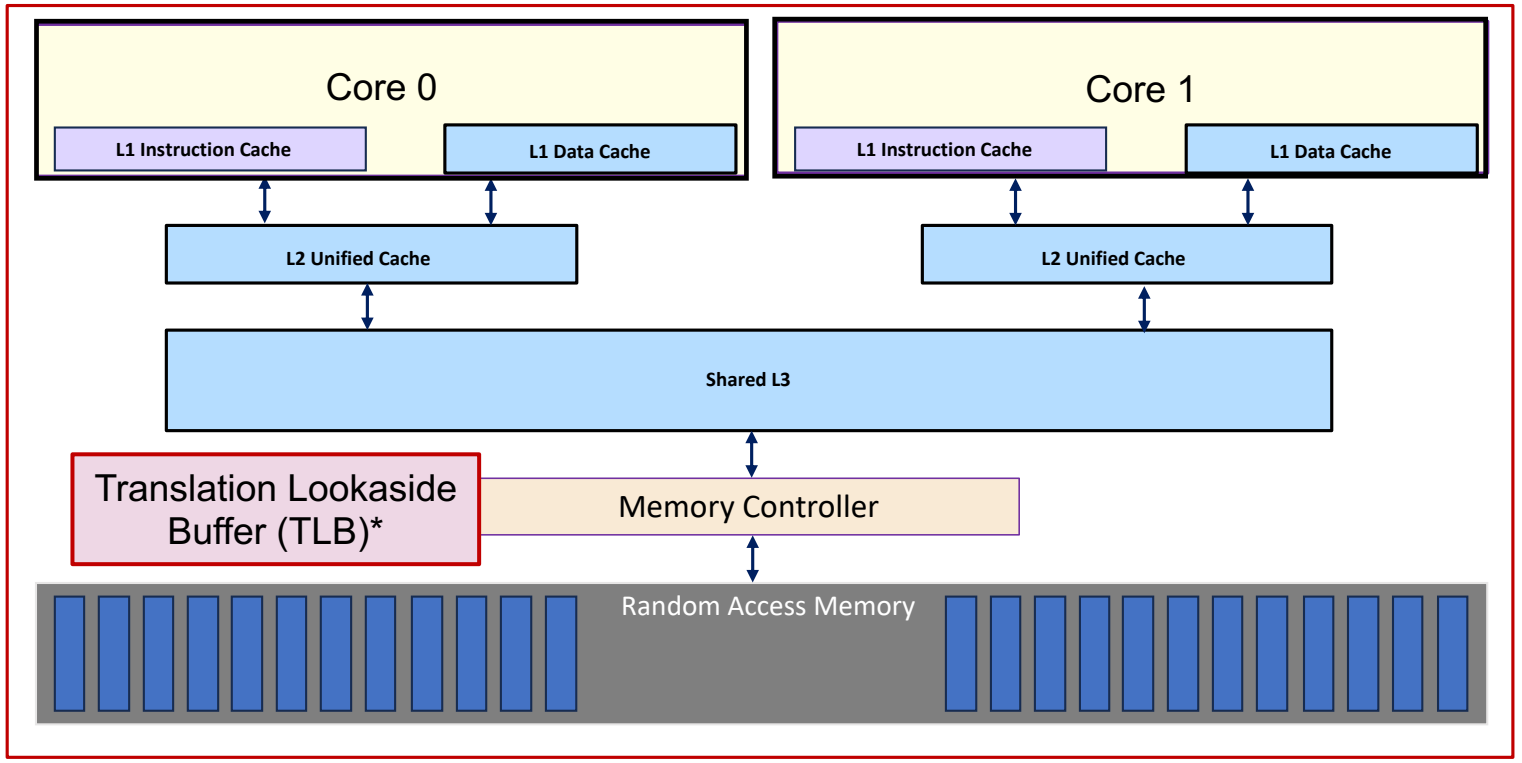
Source: Victor Eijkhout – Processor Architecture – Fall'2023

Memory Hierarchies are even more complex: the TLB

- A 64-bit address space can address 16 exabytes of memory. Clearly, that exceeds even the largest RAMs.
- A systems maps the full address space (virtual memory) onto pages of physical memory.
- The **TLB (Translation Lookaside Buffer)** implements virtual memory and caches the mapping from virtual addresses to physical addresses

  - If the entry is in the TLB page table, a small overhead of reading the entry and computed the physical address is incurred.

  - If the entry is not in the page table, a page fault exception will be raised. Requires expensive OS operations and stalls the CPU.



*Note: we show just one simple configuration of the TLB where it is associated with the memory controller. The TLB can be hierarchical with separate TLB tied to the cache hierarchy and different TLBs for data and instructions

Consider the “simple” Matrix Transpose

- Copy the transpose of A into a second matrix B.

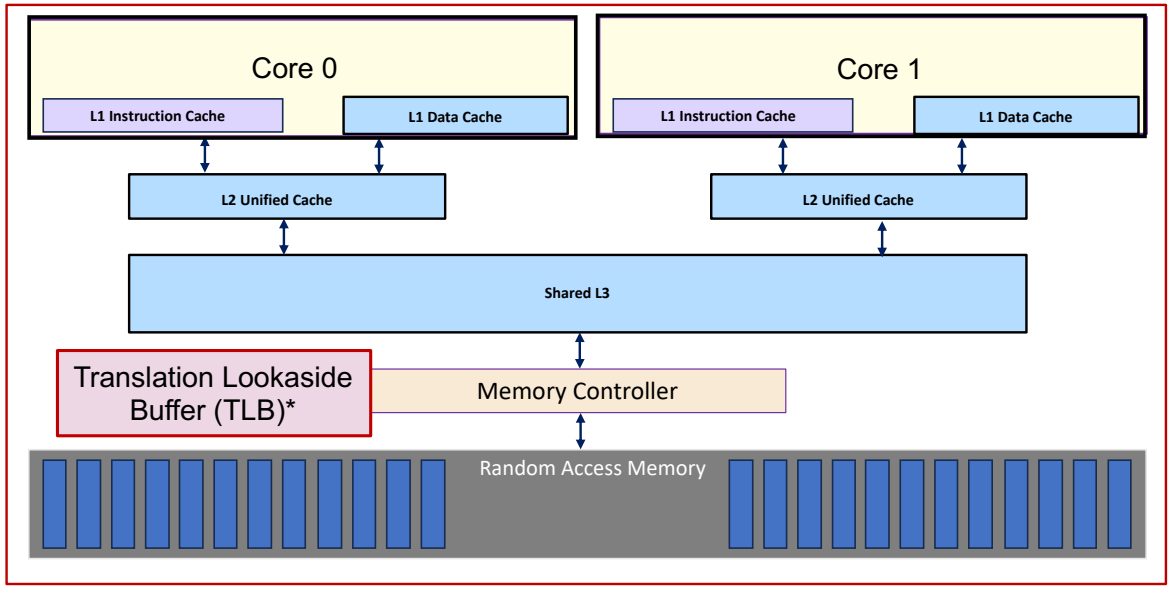
$A \in R^{NxN}$ $B \in R^{NxN}$

```
for (i=0;i<N; i++) {
    for (j=0;j<N;j++) {
        B[i+N*j] = A[j+N*i];
    }
}
```

column j of B

Row i of A

- Consider this operation and how it interacts with the TLB (Translation Lookaside Buffer).



For large N, as you march across addresses of A and B, you span multiple pages in memory … causes multiple page faults

% getconf PAGESIZE → returns the default page size on your system. On my Apple laptop the page size is 16Kbytes

Optimizing Matrix Transpose for the TLB

- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
    for (it=i; it<MIN(N,i+tile_size); it++){
  for (j=0; j<N; j+=tile_size) {
    for (jt=j; jt<MIN(N,j+tile_size);jt++){
            B[it+N*jt] = A[jt+N*it];
        }
      }
   }
}
```

Step 1: split the “i” and “j” loops into two … loop over tiles of a fixed size

Optimizing Matrix Transpose for the TLB

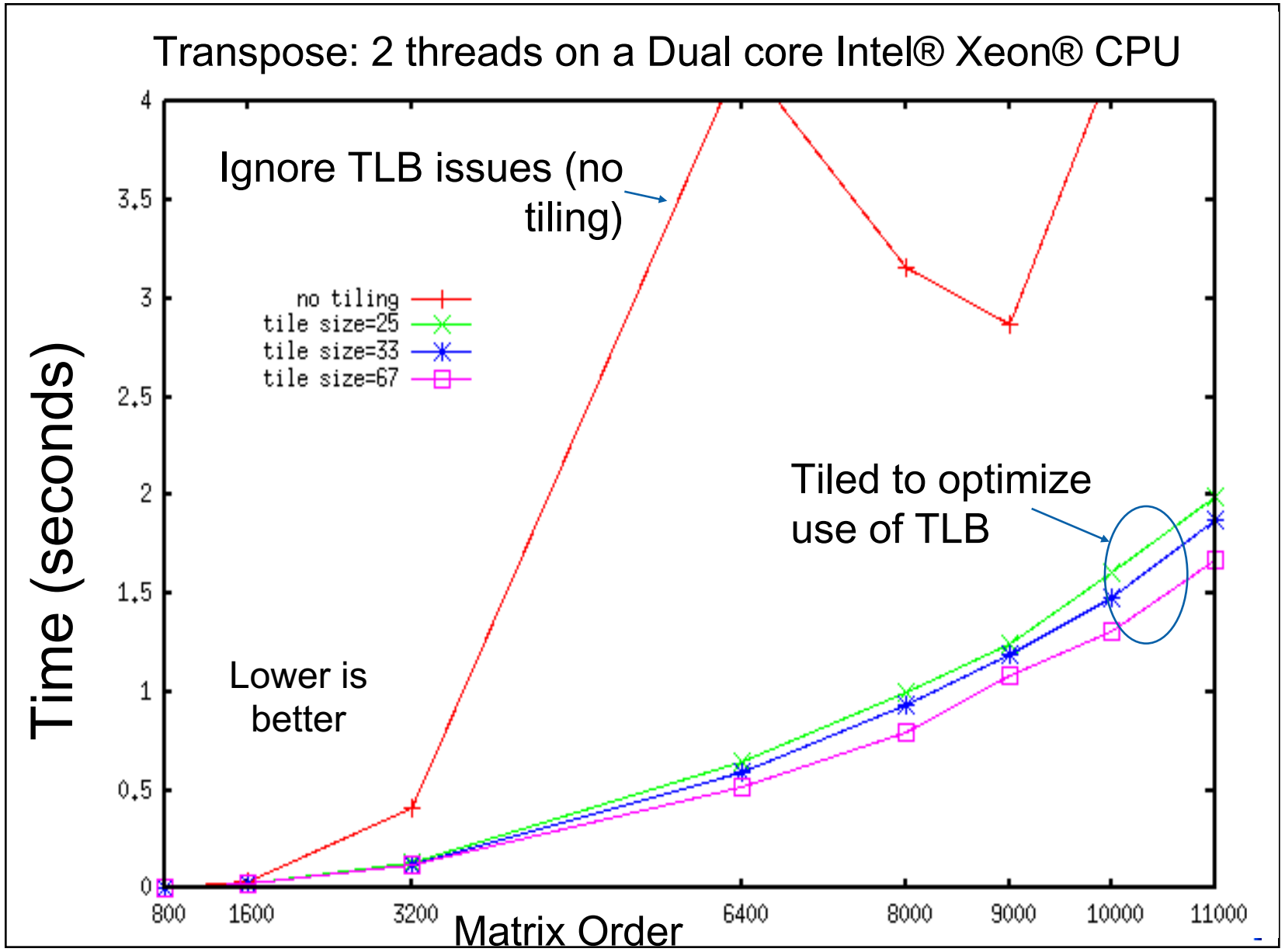
- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
  for (j=0; j<N; j+=tile_size) {
      for (it=i; it<MIN(N,i+tile_size); it++){
          for (jt=j; jt<MIN(N,j+tile_size);jt++){
              B[it+N*jt] = A[jt+N*it];
          }
      }
  }
}
```

Step 2: rearrange the loops. Move the loops over the tiles to the innermost loop-nest

The result … you grab a tile, transpose that tile, then go to the next tile

Do you need to worry about the TLB?



Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel

Optimizing Matrix Transpose for the TLB

- OpenMP version 5.1 added a construct to do this sort of tiling

```
for (i=0; i<N; i+=tile_size) {
  for (j=0; j<N; j+=tile_size) {
     for (it=i; it<MIN(N,i+tile_size); it++){
        for (jt=j; jt<MIN(N,j+tile_size);jt++){
           B[it+N*jt] = A[jt+N*it];
        }
     }
  }
}
```

1st loop (i) tile size

2nd loop (i) tile size

```
#pragma omp tile size(tile_size, tile_size)
for (i=0;i<N; i++) {
    for (j=0;j<N;j++) {
        B[i+N*j] = A[j+N*i];
    }
  }
```

OpenMP allows for partial tiles, so (N%tile_size) can be non-zero

Optimizing Matrix Transpose for the TLB

- OpenMP version 5.1 added a construct to do this sort of tiling

```
for (i=0; i<N; i+=tile_size) {
  for (j=0; j<N; j+=tile_size) {
     for (it=i; it<MIN(N,i+tile_size); it++){
        for (jt=j; jt<MIN(N,j+tile_size);jt++){
           B[it+N*jt] = A[jt+N*it];
        }
     }
  }
}
```

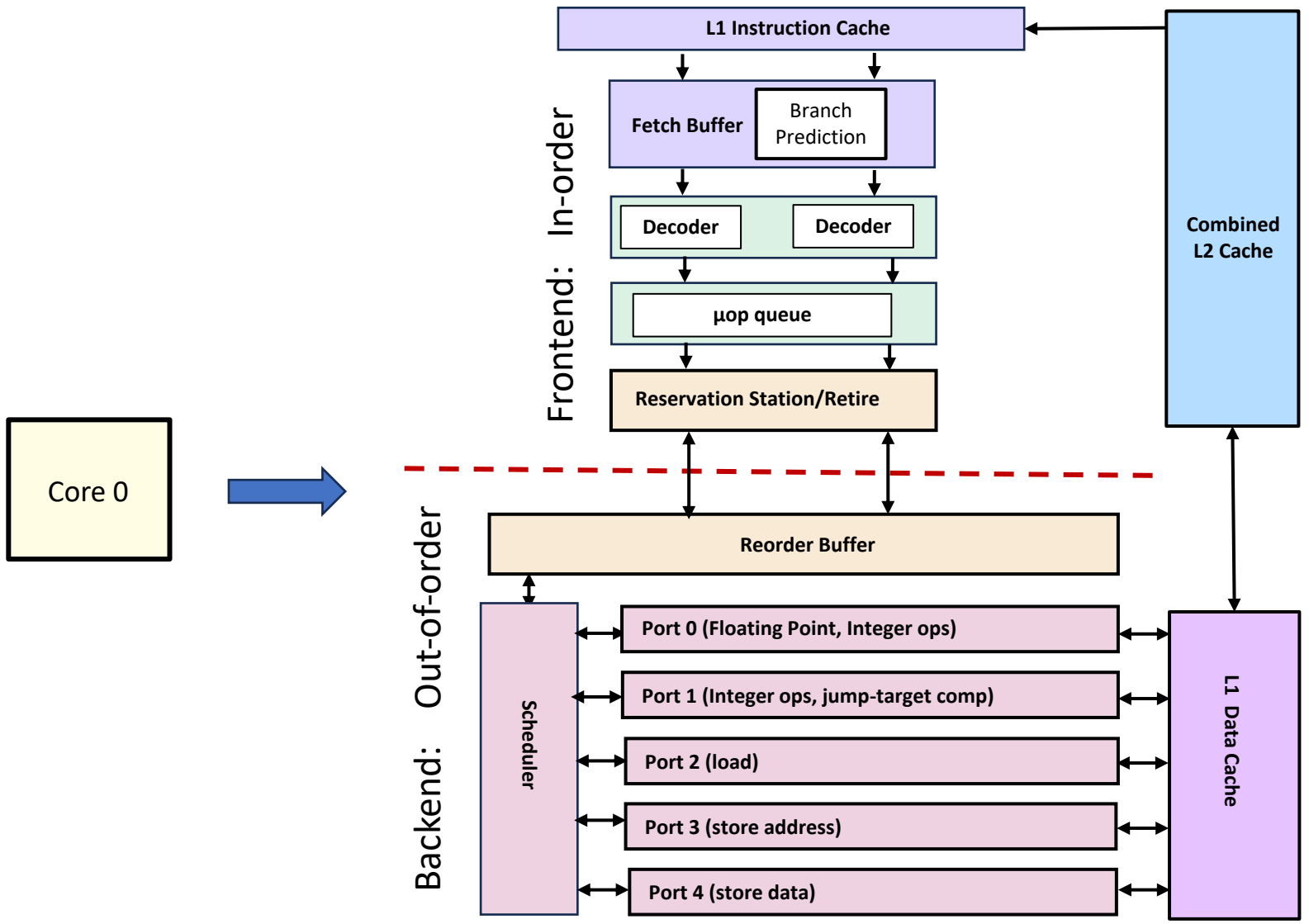
The parallel loop is applied to the outer “tiled” loop

```
#pragma omp parallel for
#pragma omp tile size(tile_size, tile_size)
for (i=0;i<N; i++) {
    for (j=0;j<N;j++) {
        B[i+N*j] = A[j+N*i];
    }
  }
```

OpenMP allows for partial tiles, so (N%tile_size) can be non-zero

gcc-14 doesn’t support the tile construct yet. It takes them a while to catch up with the OpenMP specifications. Clang support since version 17

Once you get past Memory Hierarchy issues, consider the complexity of the cores



Frontend processes instructions for execution.

Maps them into micro-operations (**µop**) and manages them (and the register file) so they retire "in order"

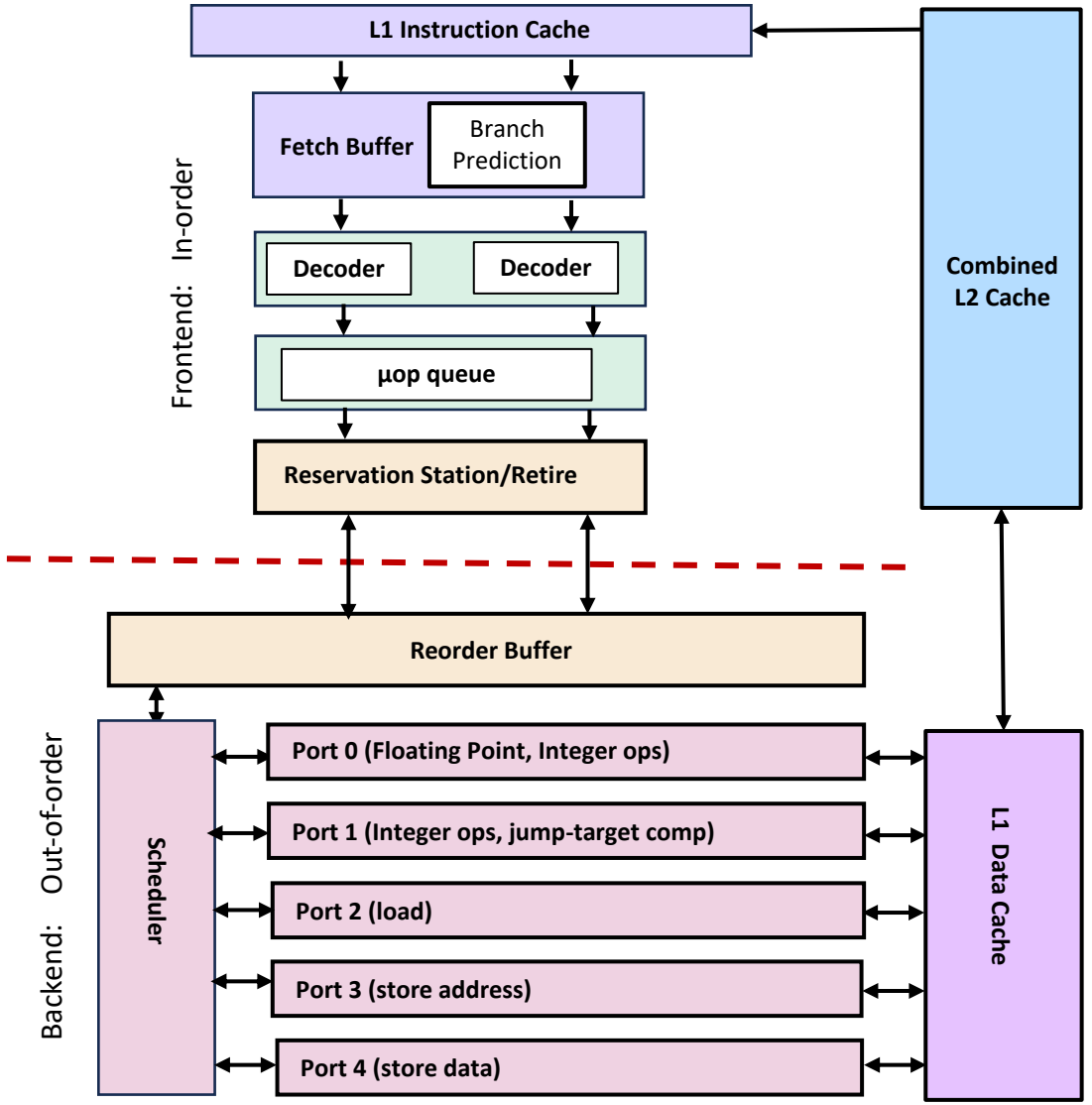
Backend executes decoded instructions.

Instructions are pipelined and include multiple ports and SIMD execution units.

Execution is "out of order" to give the scheduler maximum flexibility to maximize instruction-level parallelism

And its even worse ….

A single frontend typically can’t generate enough decoded instructions to keep all the resource in the backend busy.

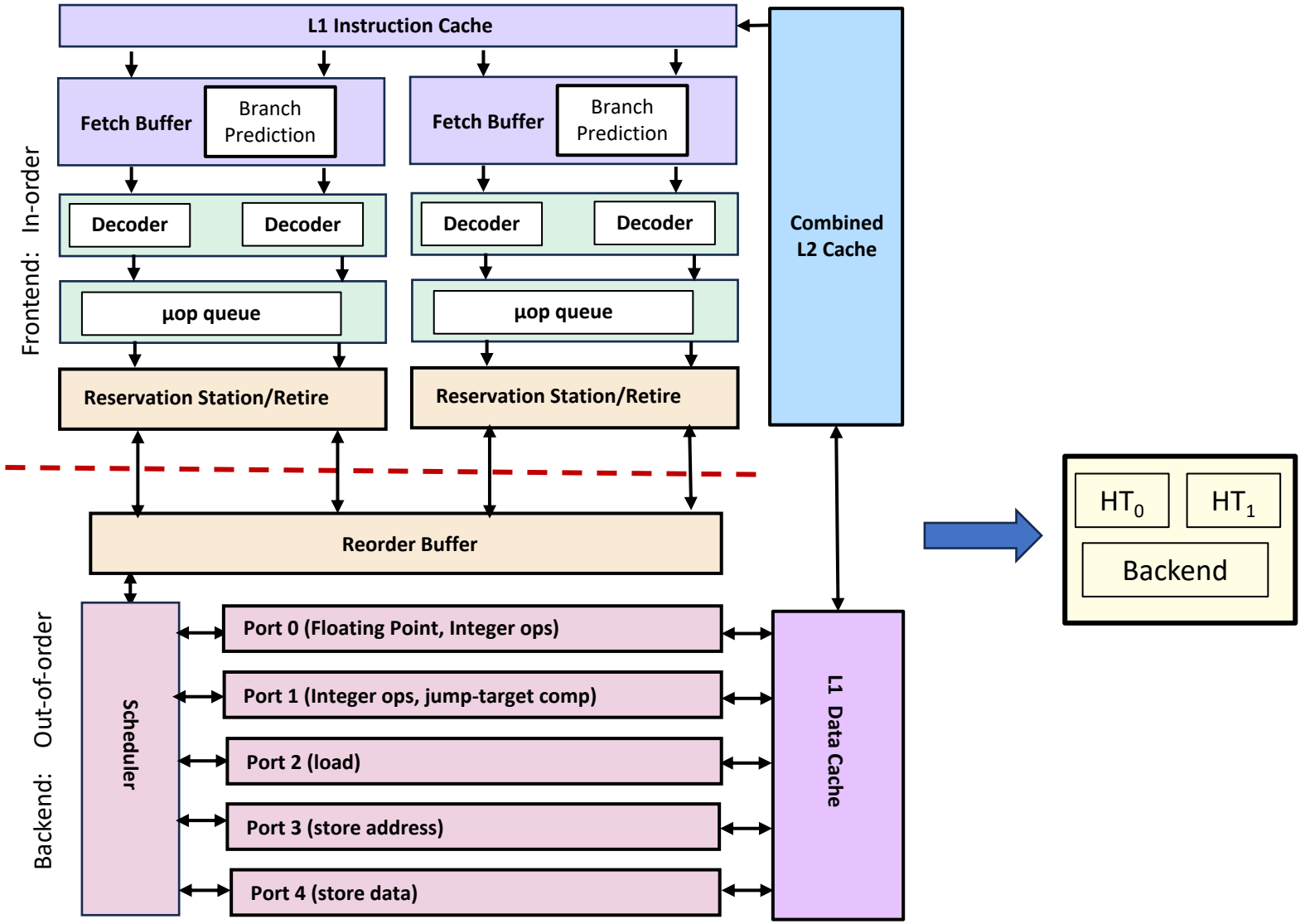


And its even worse ….

So, we have multiple frontends (in this case, 2) feeding a single back end.

These appear to the OS as an additional core … or a virtual core.

They maintain state of a thread in hardware so these are often called **Hardware threads.**



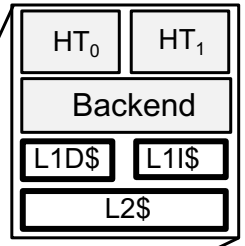
HT: hardware thread

And then we put all this into a real two-socket node

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

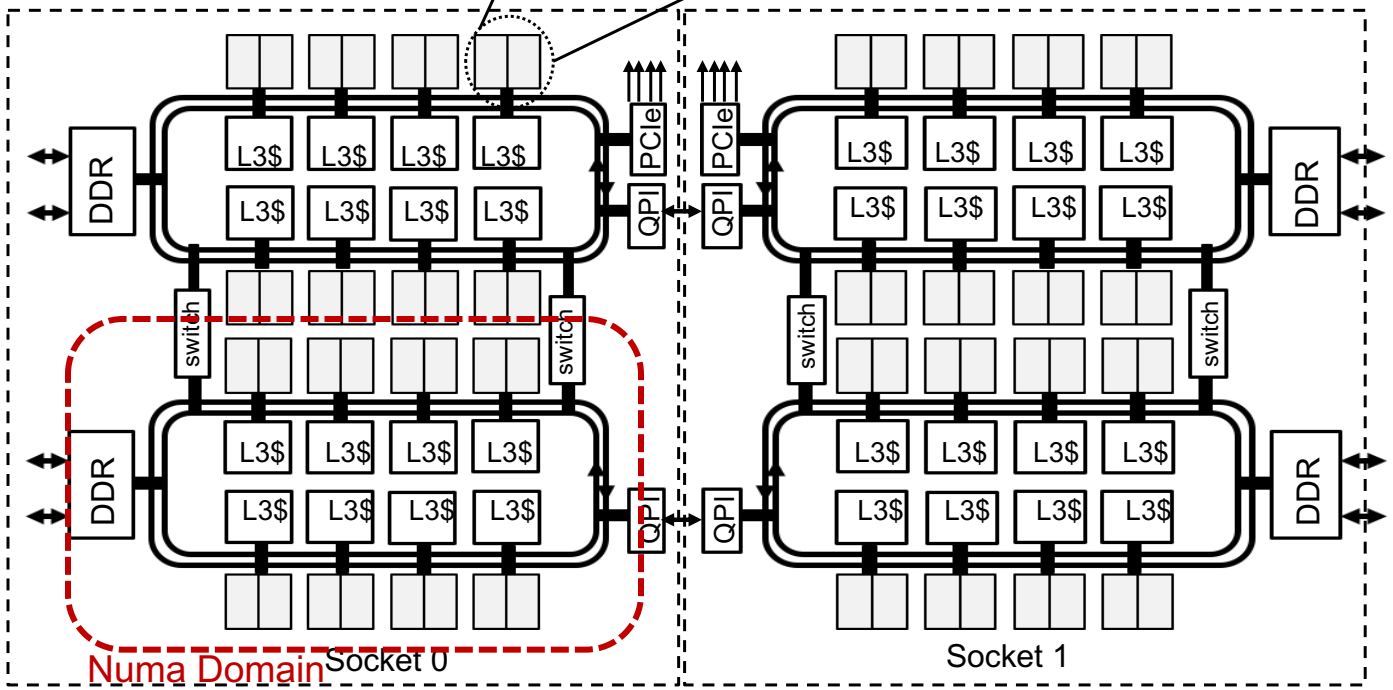
4 blocks of 8 core units connected by an on-chip-network with a DDR memory controller.

Each block is a NUMA domain since memory access from a core to its "own" DDR is less expensive.



2 Hardware threads (HT) per core
Intel® AVX2 (256 bit Vector unit)
L1$ instruction and data: 32 KB
Unified L2$ 256 KB

40 MB shared L3$



DDR: Double Data Rate memory controller

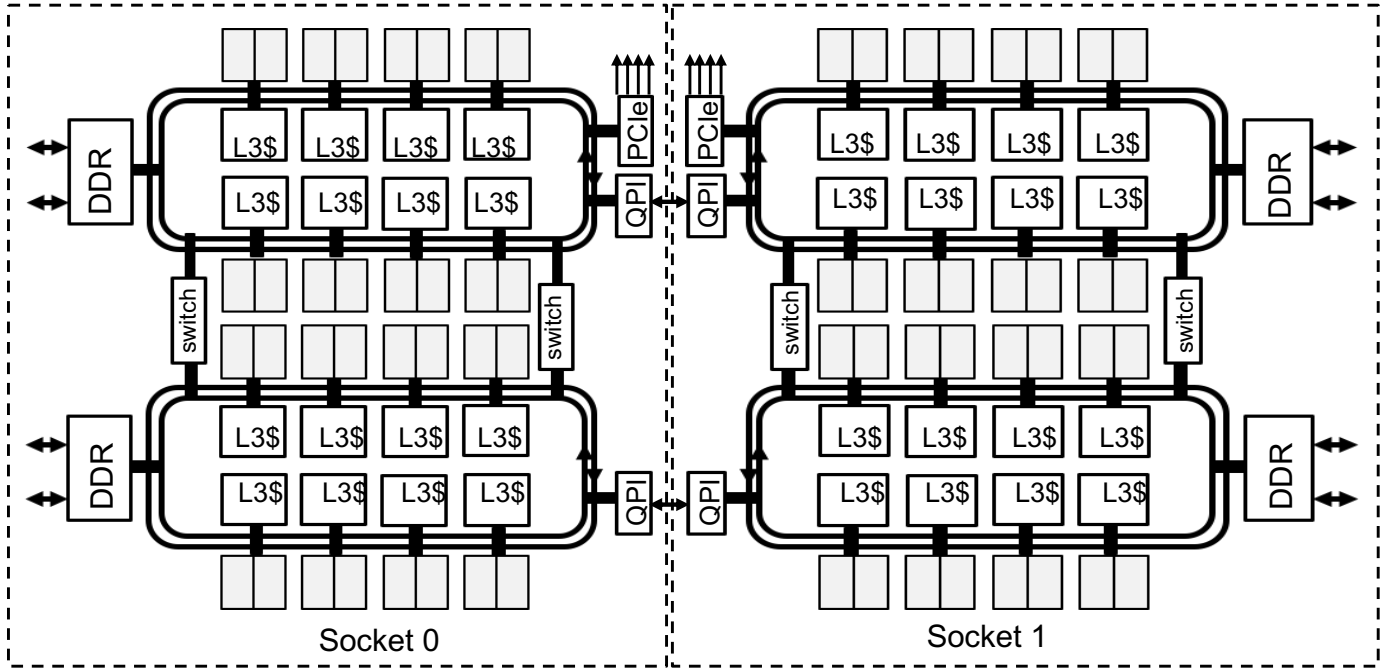
PCIe is the connection from the CPU to other devices in a node.

QPI: Quick Path Interconnect. A coherent interconnect between CPUs. Makes it easy to build multi-CPU nodes.

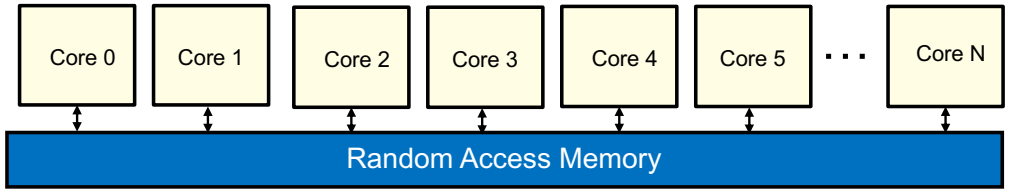
As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

All systems today are nonuniform memory architectures (NUMA)

Given all this complexity in real hardware ….



It's amazing our SMP model works at all ….

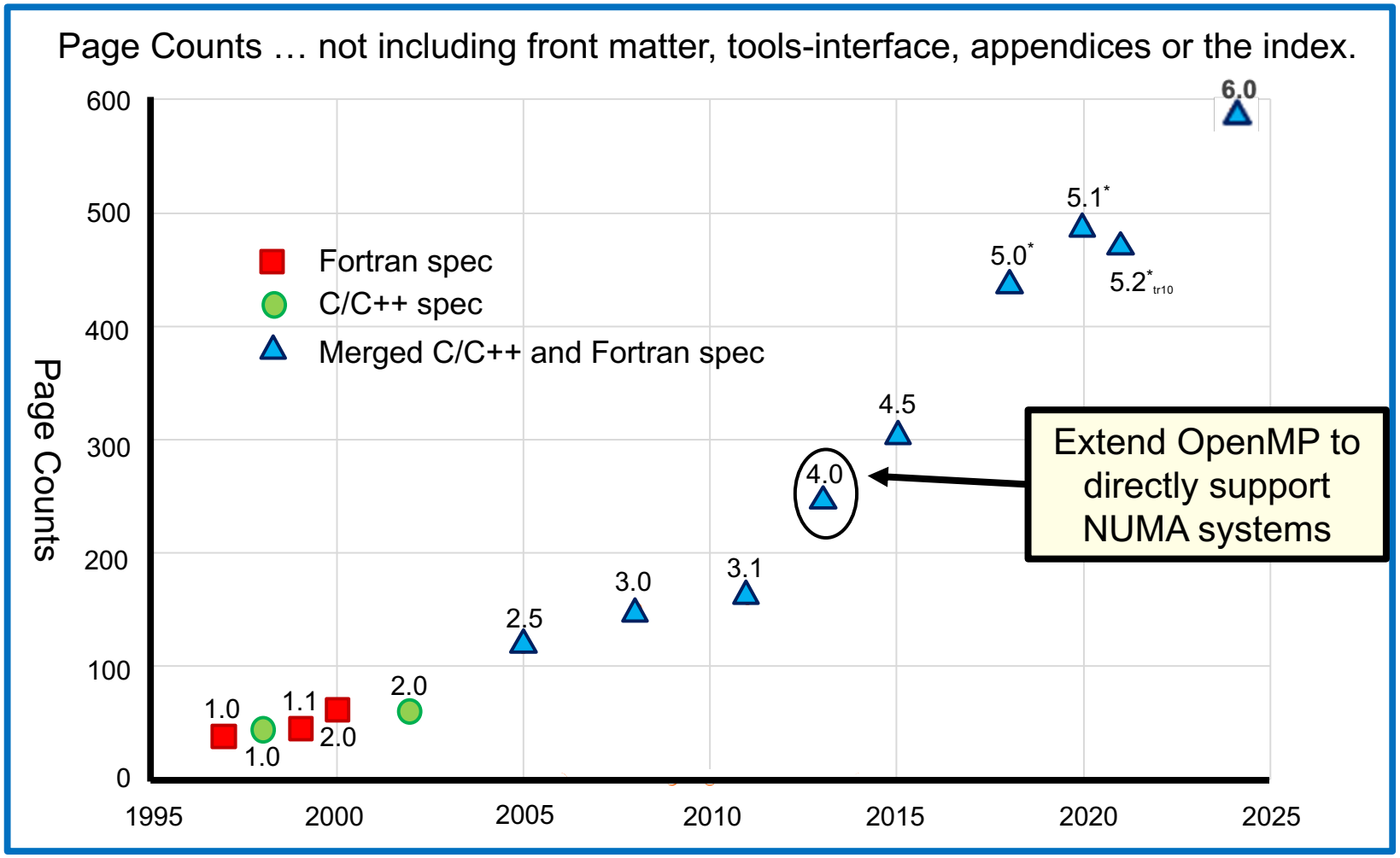


What do you do when you need more performance than you can get by pretending the system is a Symmetric Multiprocessor (SMP)?

Dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers



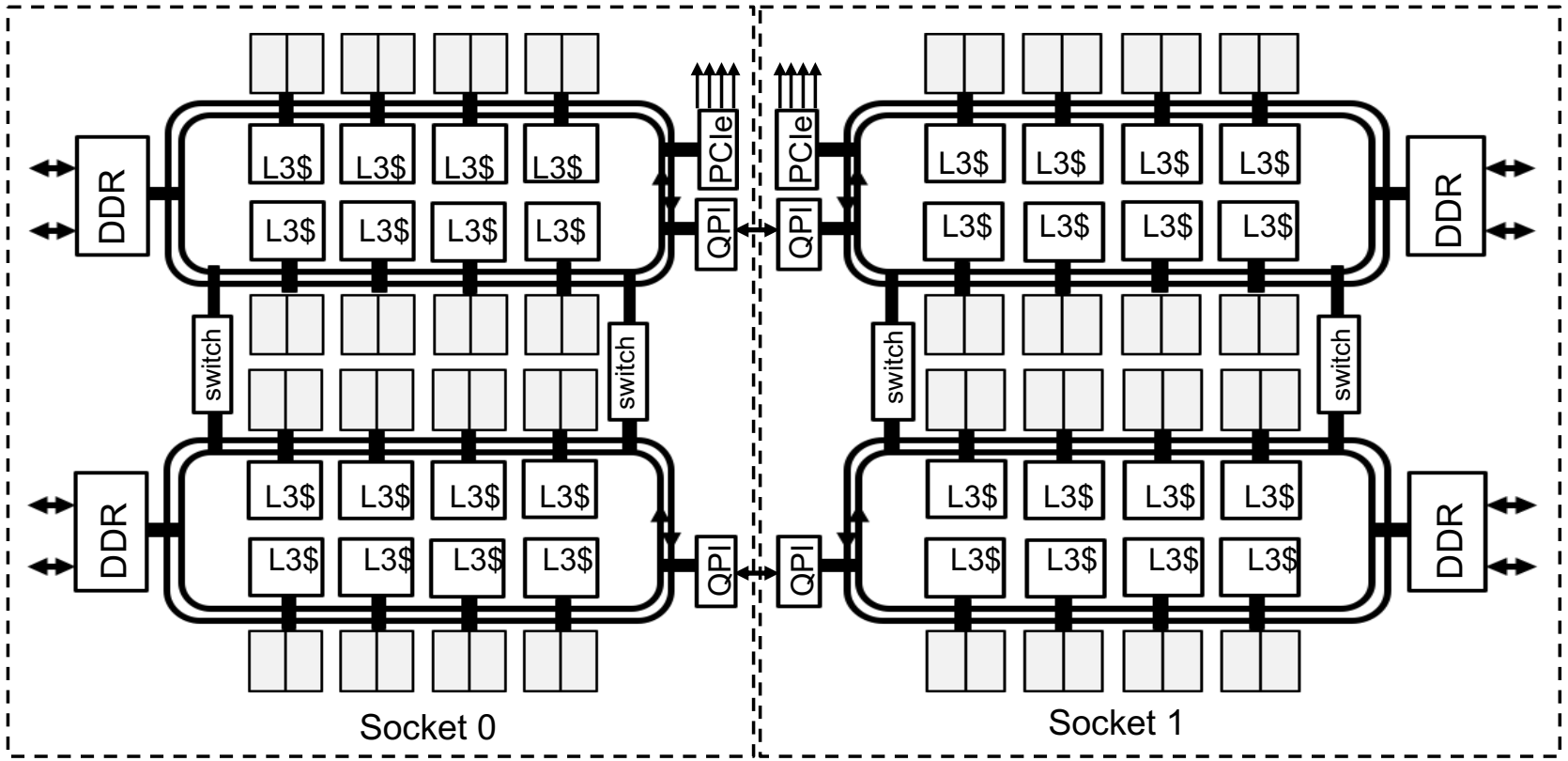
Outline

- OpenMP and Real Hardware
- Optimizing code for NUMA systems
- GPU Programming and OpenMP
- Synchronization and the OpenMP memory model
- Random numbers … an example of how to build threadsafe libraries

The following slides on NUMA systems make heavy use of content from Yun (Helen) He of NERSC, Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP ARB)

NUMA Systems: You must optimize code for their complex memory subsystems

- A floating-point operation takes O(~1 ns).
  - L1 Cache ~1.5 ns
  - L2 Cache reference ~5 ns
  - L3 Cache reference ~25 ns
  - Near memory DRAM access ~100ns
  - Near memory DRAM access ~100ns
  - Far memory DRAM access ~200 ns
- The key to performance is to minimize memory movement …. get the memory movement right and the “rest” is easy

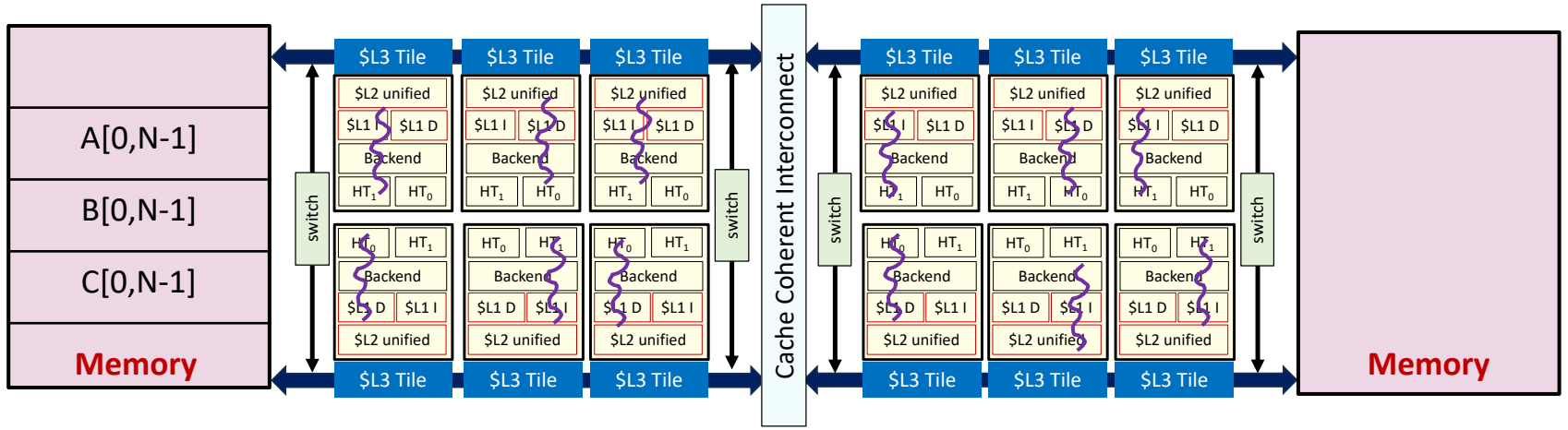


ns: nanosecond

Dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

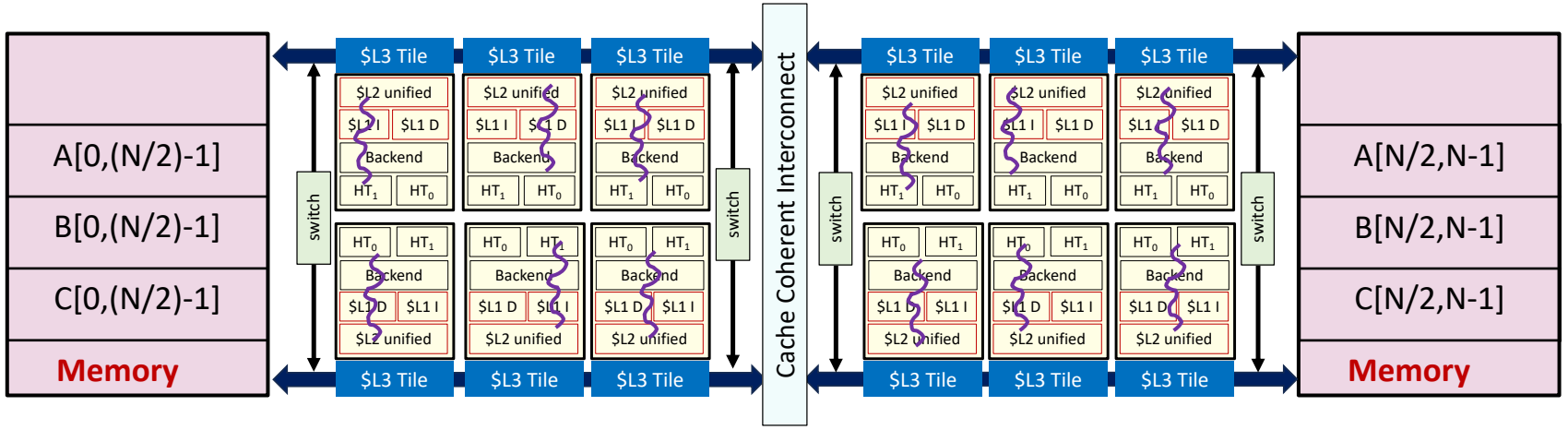
Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores



3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s



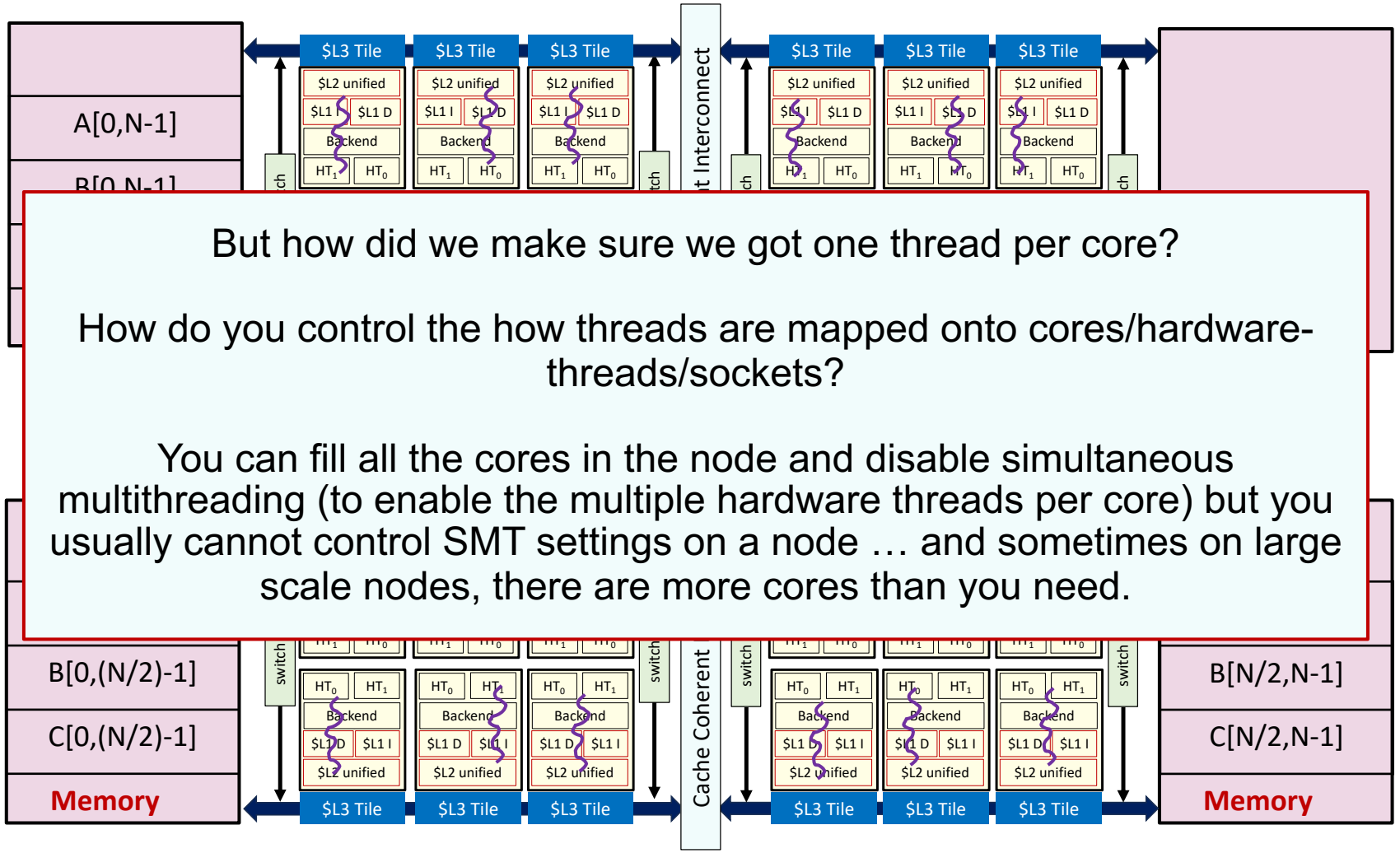
Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores



But how did we make sure we got one thread per core?

How do you control the how threads are mapped onto cores/hardware-threads/sockets?

You can fill all the cores in the node and disable simultaneous multithreading (to enable the multiple hardware threads per core) but you usually cannot control SMT settings on a node … and sometimes on large scale nodes, there are more cores than you need.

3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s

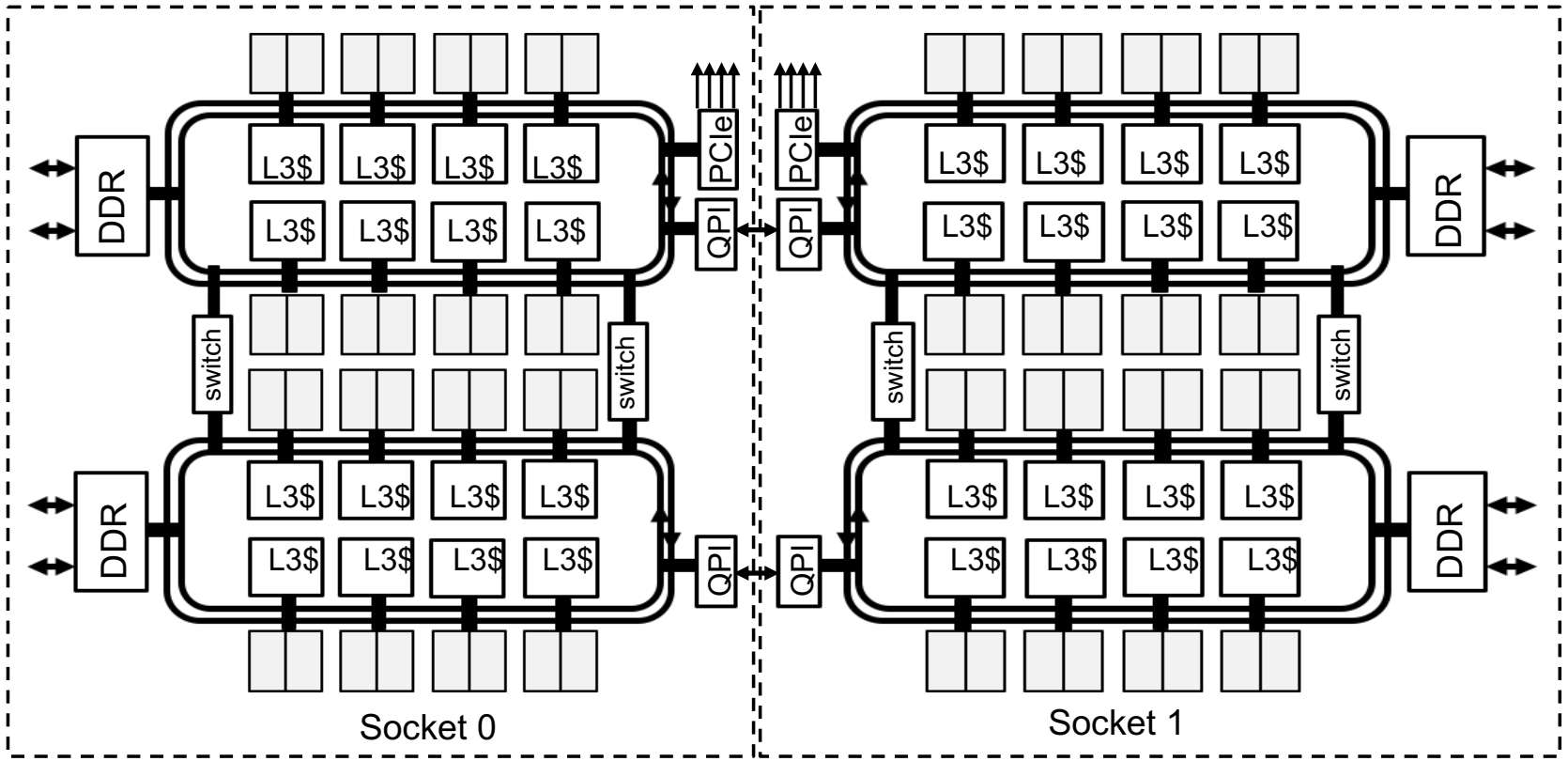
Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

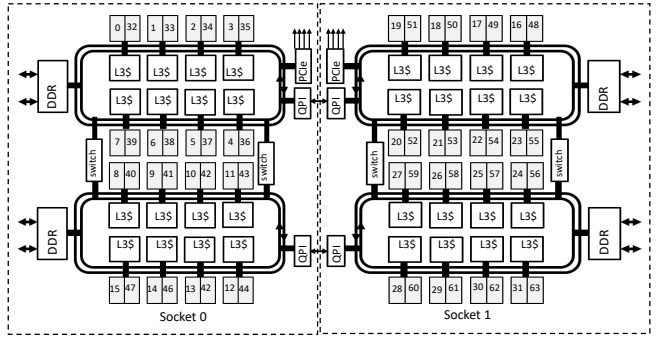
NUMA nodes and the places we can put threads

- OpenMP defines the concept of places on a NUMA node where threads can execute.
- The idea is to map the OS defined virtual cores onto places visible to OpenMP for threads assignment
- The first step is to understand the OS defined virtual cores (also known as virtual processing units or PUs)



Discover the OS view of virtual cores

- Portable Hardware Locality tools …. hwloc-ls, lstopo, Numactl and others depending on the system. Generates text or graphical output depending on how the tools are configured on your system.



Machine (126GB total)

NUMANode P#0 (63GB)

Package P#0

L3 (40MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)
L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)
Core P#0	Core P#1	Core P#2	Core P#3	Core P#4	Core P#5	Core P#6	Core P#7	Core P#8	Core P#9	Core P#10	Core P#11	Core P#12	Core P#13	Core P#14	Core P#15
PU P#0	PU P#1	PU P#2	PU P#3	PU P#4	PU P#5	PU P#6	PU P#7	PU P#8	PU P#9	PU P#10	PU P#11	PU P#12	PU P#13	PU P#14	PU P#15
PU P#32	PU P#33	PU P#34	PU P#35	PU P#36	PU P#37	PU P#38	PU P#39	PU P#40	PU P#41	PU P#42	PU P#43	PU P#44	PU P#45	PU P#46	PU P#47

NUMANode P#1 (63GB)

Package P#1

L3 (40MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)
L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)
Core P#0	Core P#1	Core P#2	Core P#3	Core P#4	Core P#5	Core P#6	Core P#7	Core P#8	Core P#9	Core P#10	Core P#11	Core P#12	Core P#13	Core P#14	Core P#15
PU P#16	PU P#17	PU P#18	PU P#19	PU P#20	PU P#21	PU P#22	PU P#23	PU P#24	PU P#25	PU P#26	PU P#27	PU P#28	PU P#29	PU P#30	PU P#31
PU P#48	PU P#49	PU P#50	PU P#51	PU P#52	PU P#53	PU P#54	PU P#55	PU P#56	PU P#57	PU P#58	PU P#59	PU P#60	PU P#61	PU P#62	PU P#63

Graphical output for a dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

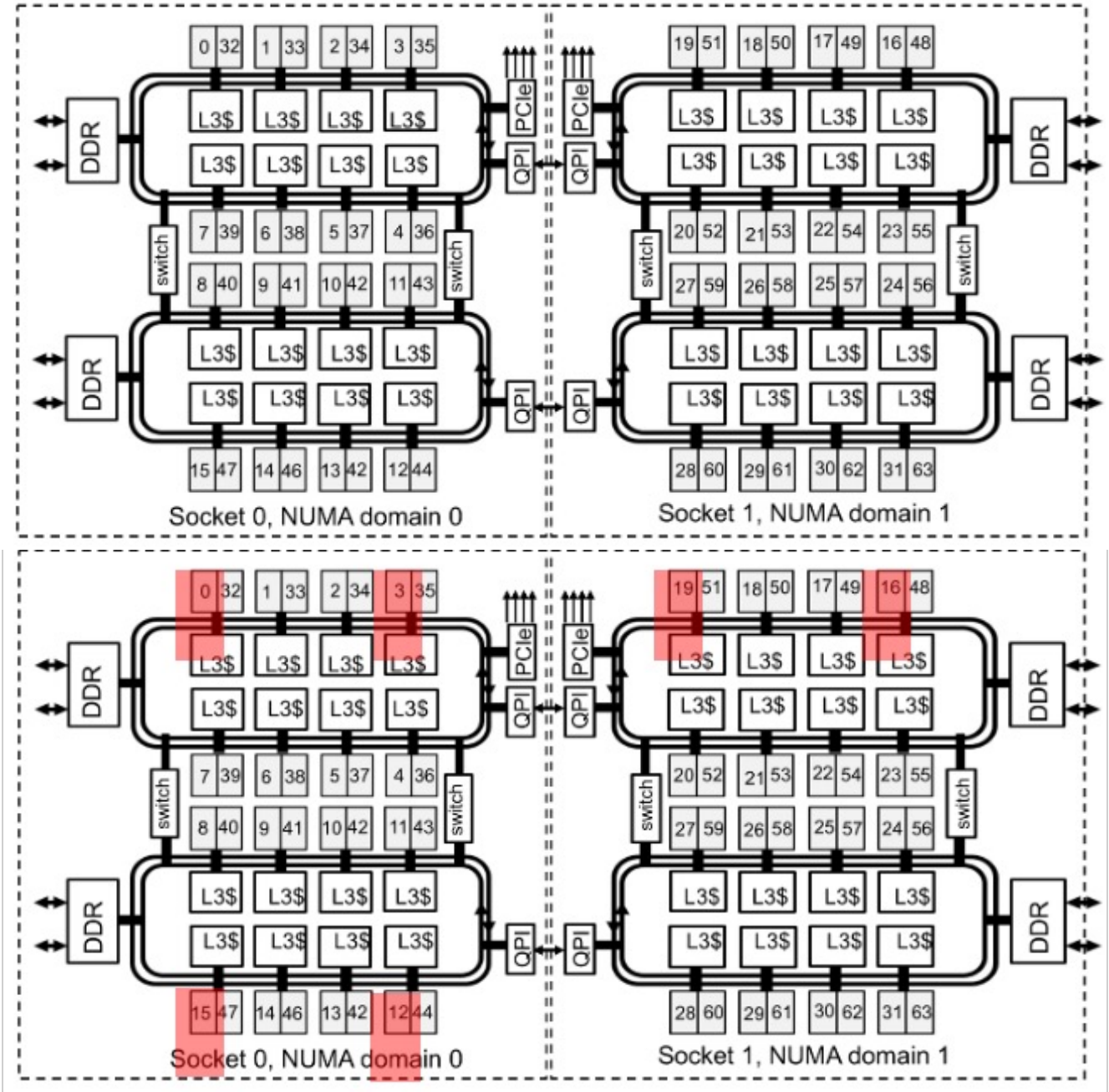
PU: processor unit. The smallest physical execution unit that hwloc recognizes.

Based on content from Yun (Helen) He from NERSC)

Using OMP_PLACES to select where to run code

- After using a tool to learn the logical core IDs (processor units or PUs) use environment variables to define where threads can be scheduled to execute.

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

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

}
```



Using OMP_PLACES to select where to run code

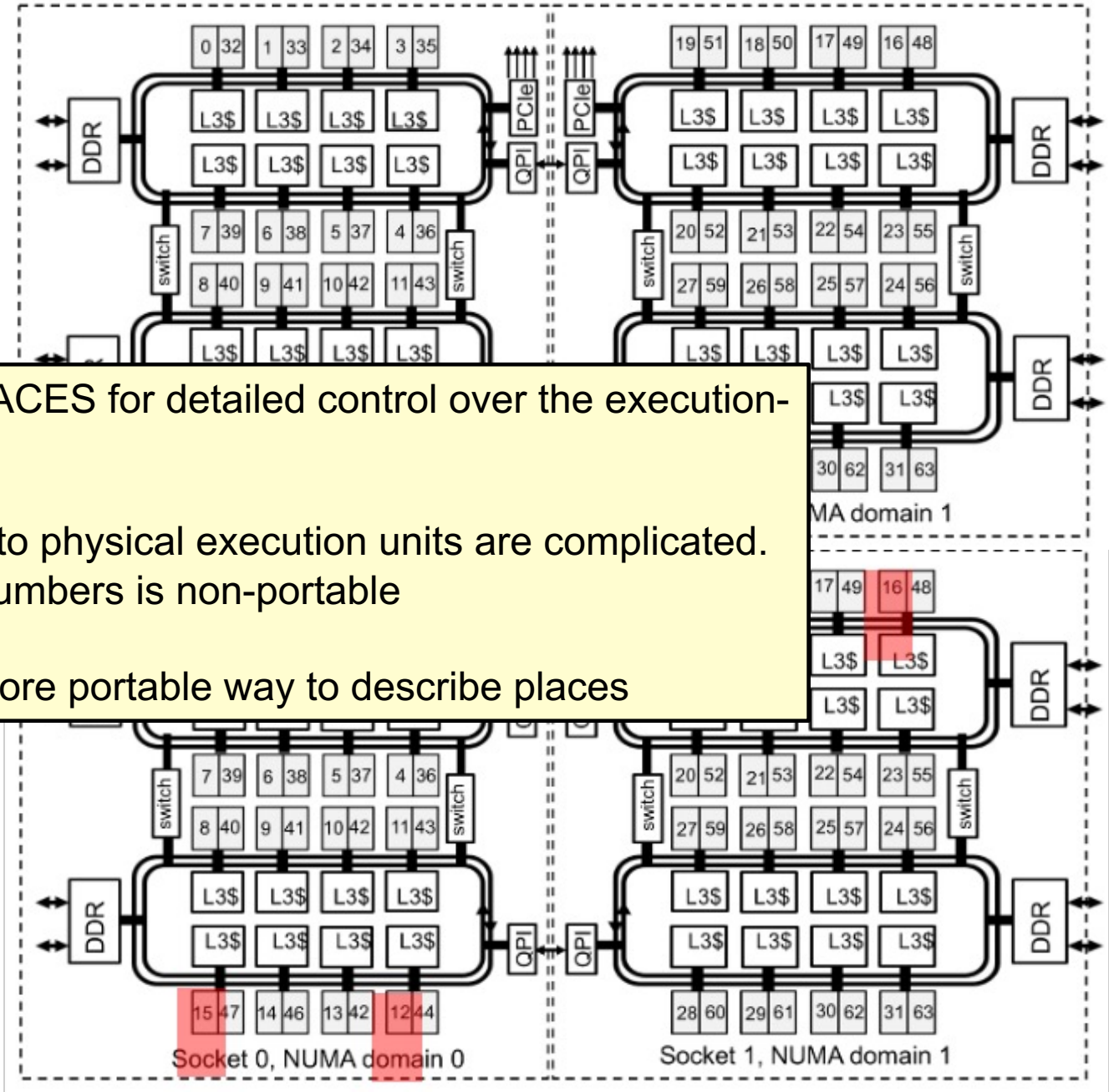
- After using a tool to learn the logical core IDs (processor units or PUs) use environment variables to define where threads can be scheduled to execute.

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

#pragma omp parall
{
   // do a bunch of c

}
```

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



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 (which may have 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

Thread Affinity … mapping threads to places

Thread affinity to places: OMP_PROC_BIND

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

Examples:

```
export OMP_PROC_BIND=spread
```

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

Thread affinity to places: OMP_PROC_BIND

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

Example … using clauses on a parallel construct:

```
#pragma omp parallel num_threads(4) proc_bind(spread)
```

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

Examples: OMP_PROC_BIND

Consider 4 cores total, 2 hardware threads 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	
	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1
	PU 0	PU 1	PU 2	PU 3	PU 4	PU 5	PU 6	PU 7
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1
	PU 0	PU 1	PU 2	PU 3	PU 4	PU 5	PU 6	PU 7
Thread	0		1		2		3	

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

We define places explicitly with the IDs of the OS virtual cores (the PUs).

We do not control where the initial thread is placed. We will assume it is placed on HT1 or Core 0.

For this example, we have 4 place partitions.

OMP_PLACES={0,1},{2,3},{4,5},{6,7}

With close, threads placed in consecutive locations

With spread, threads placed in first place of each partition

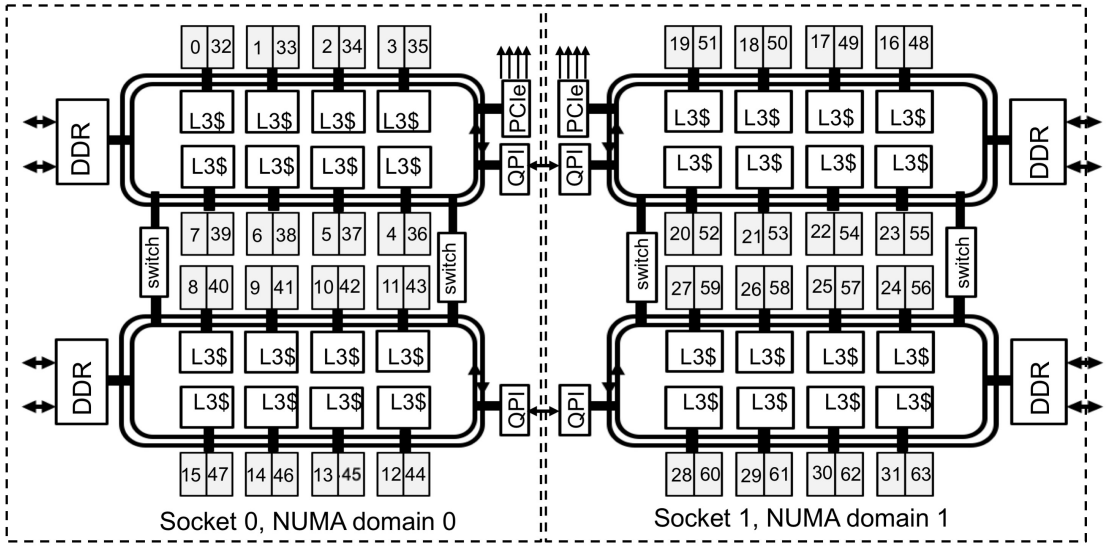
PU: processor unit. The smallest physical execution unit that hwloc recognizes.

OMP_PROC_BIND Choices for STREAM Benchmark

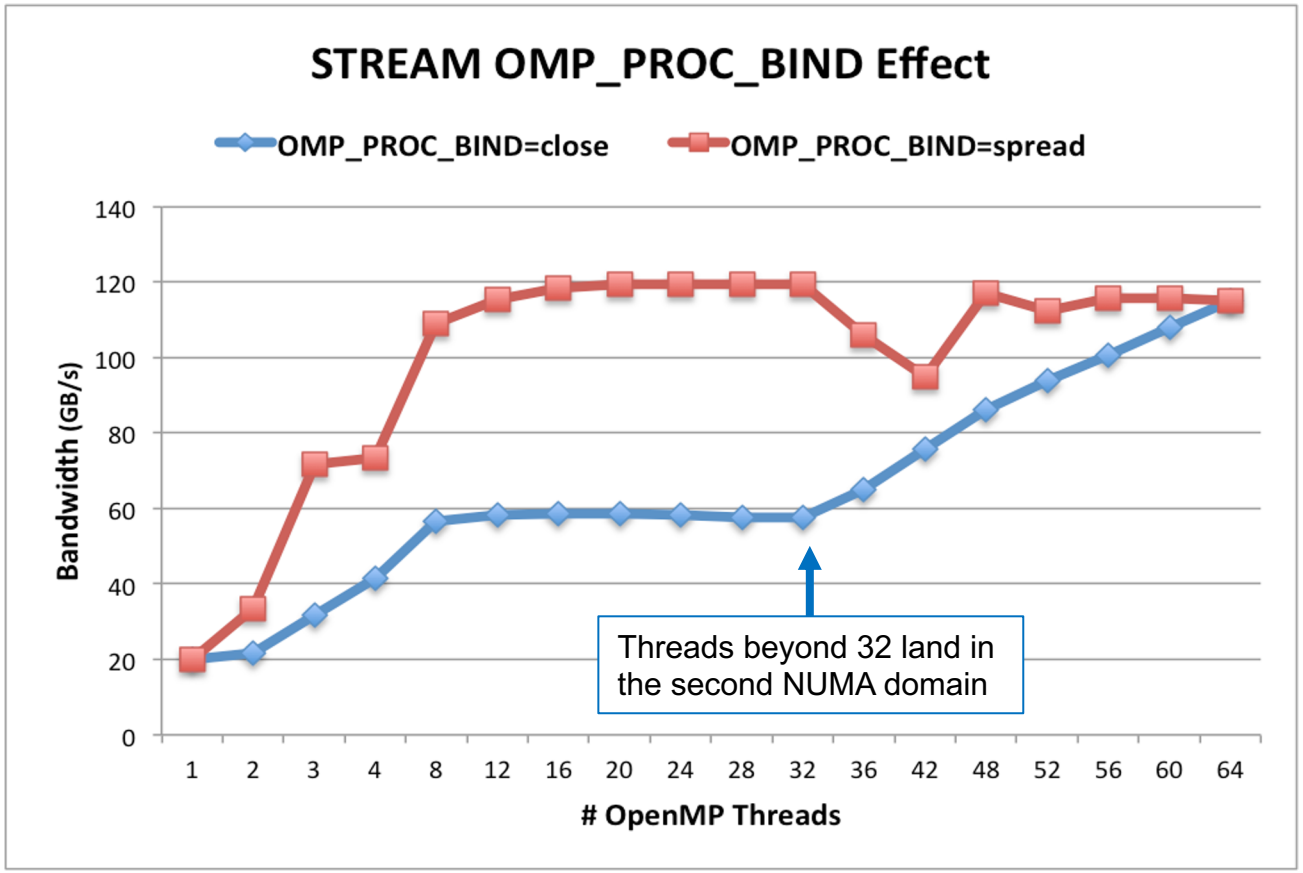
OMP_NUM_THREADS=32
OMP_PLACES=threads

OMP_PROC_BIND=close
Threads 0 to 31 bind to cores (0,32),(1,33),(2,34),…(15,47). All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread
Threads 0 to 31 bind to cores 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.



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



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

Based on content from Yun (Helen) He from NERSC)

Aligning memory to threads … First touch

Memory Affinity: Exploiting "First Touch" page mapping policy

Step 1.1 Initialization by primary thread only

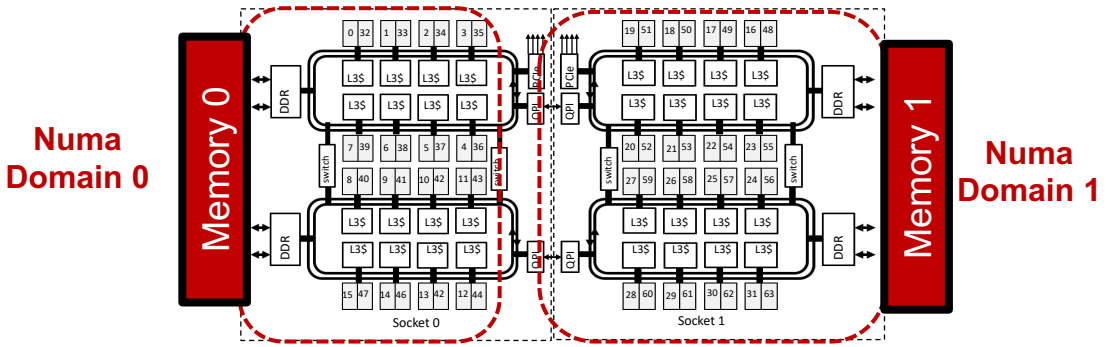
```
for (j=0; j<VectorSize; j++) {
      a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

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

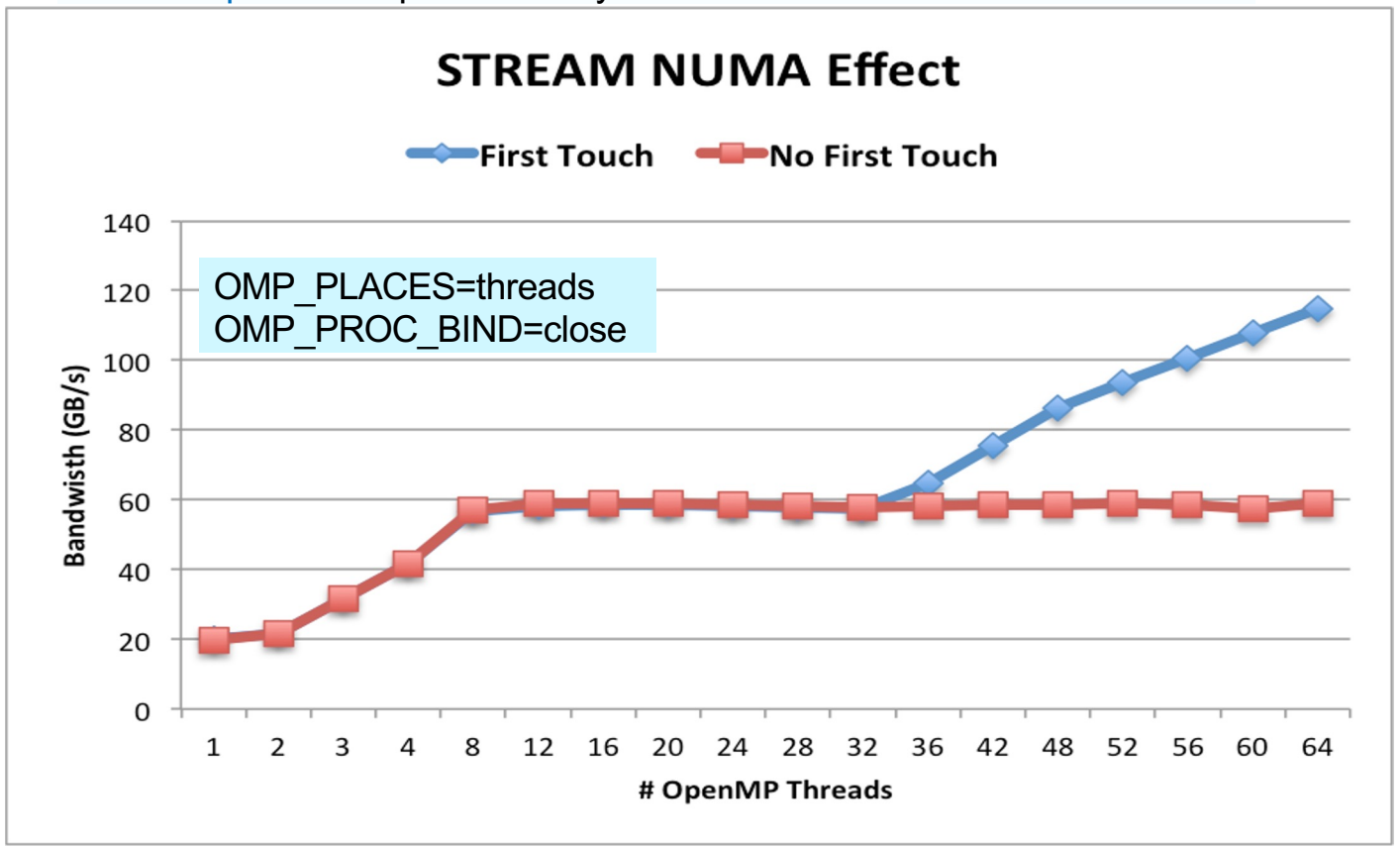
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++)
      a[j]=b[j]+d*c[j];
```



A dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

- The OS maps pages of memory based on a **first touch** policy.
- Hence, Affinity to memory is not defined when memory is allocated … it is defined when the memory is initialized.
- The result is memory is local to the thread which initializes it.

Red: step 1.1 + step 2. Memory from Numa Domain 0 only
Blue: step 1.2 + step 2. Memory used from both NUMA domains



Based on content from Yun (Helen) He from NERSC)

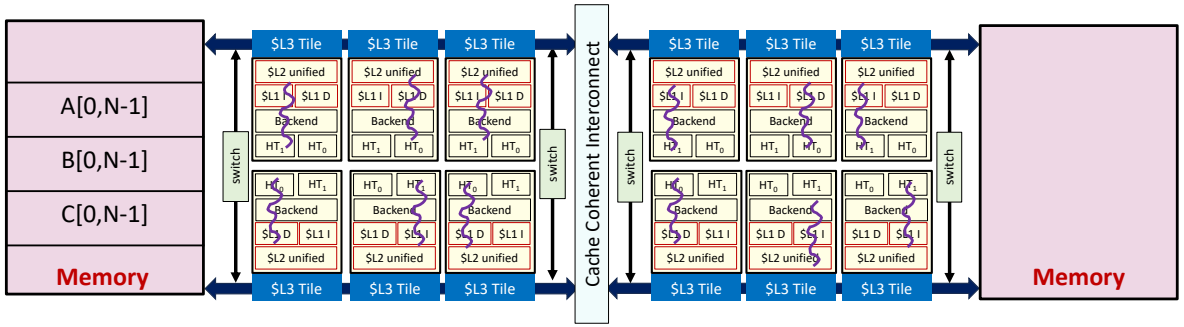
Example: working with the First Touch Policy

Rember this slide?

Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores

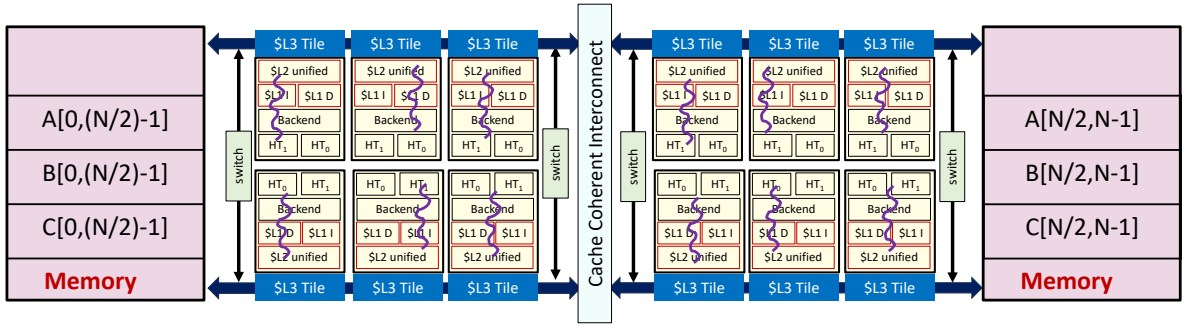
Arrays A, B, and C initialized on primary thread



3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s

Arrays A, B, and C initialized in parallel



Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

But its not just any “in parallel”. You want to initialize the arrays with the same “parallel for schedule” that will be used when the threads do the computations with A, B, and C

Nested parallelism

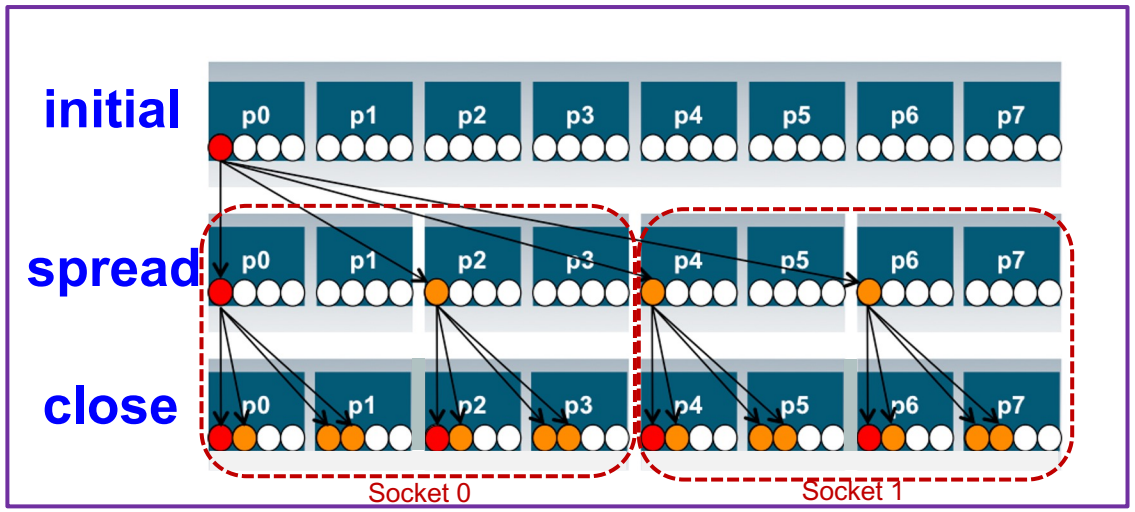
Process and Thread Affinity in Nested OpenMP

Consider a program with nested parallel regions

```
#pragma omp parallel
    #pragma omp parallel
```

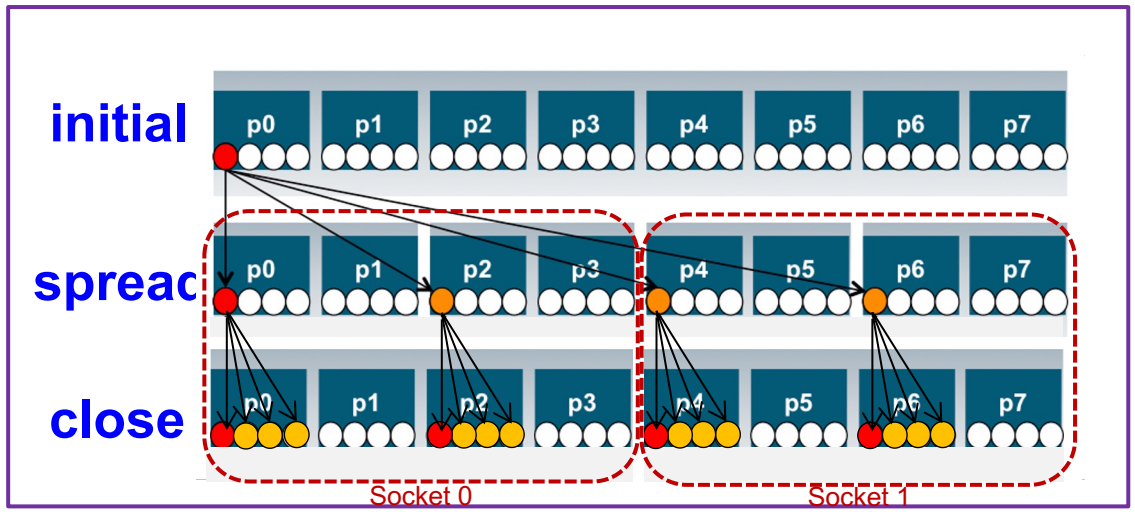
Running on a system with 2 sockets, 4 cores per socket, 4 hardware-threads per core

```
export OMP_MAX_ACTIVE_LEVELS=2
export OMP_NUM_THREADS=4,4
export OMP_PLACES=cores
export OMP_PROC_BIND=spread,close
./a.out
```



Cyclic distribution between “close” cores

```
export OMP_MAX_ACTIVE_LEVELS=2
export OMP_NUM_THREADS=4,4
export OMP_PLACES=threads
export OMP_PROC_BIND=spread,close
./a.out
```



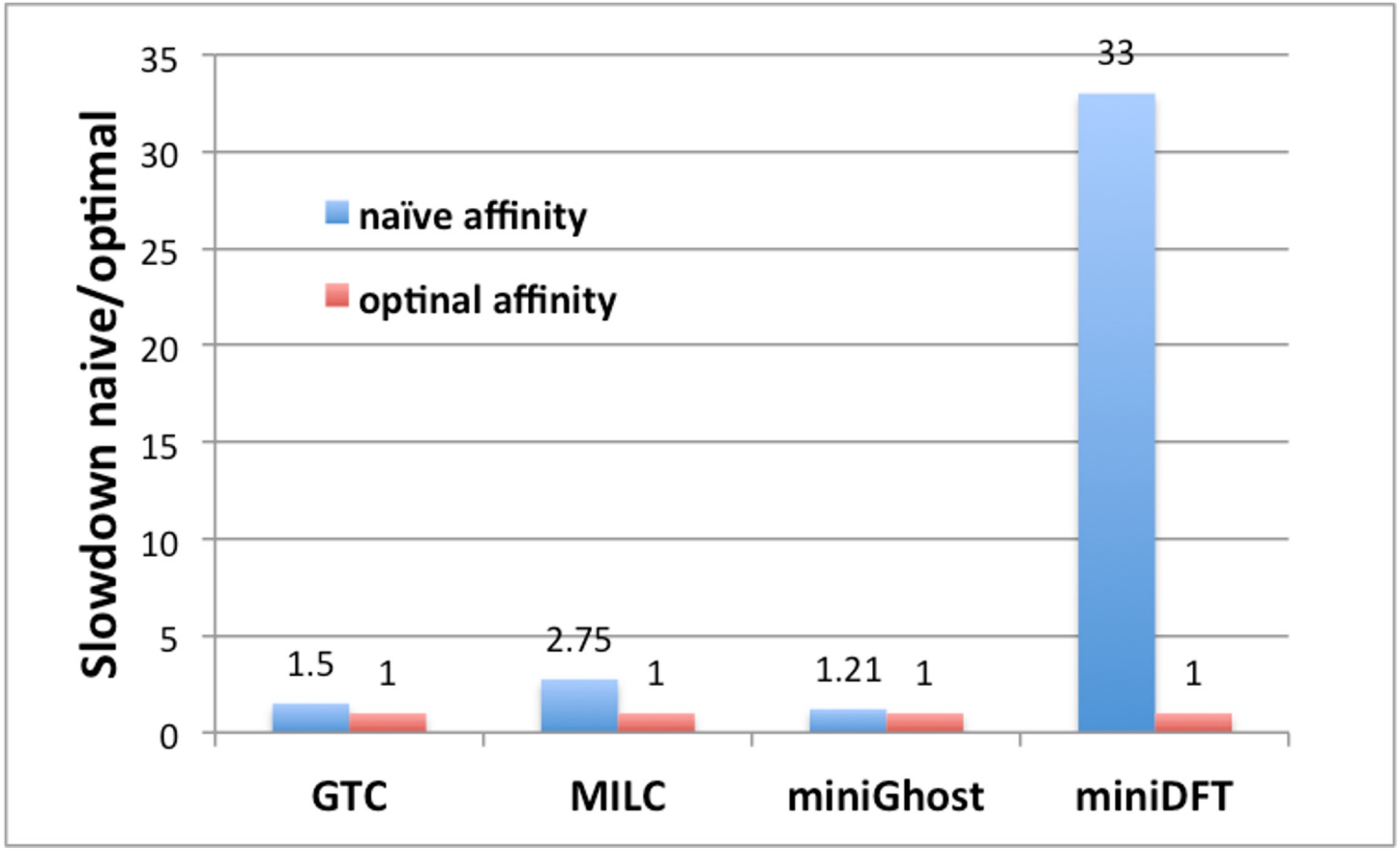
Distribution across four hardware threads

Based on content from Yun (Helen) He from NERSC)

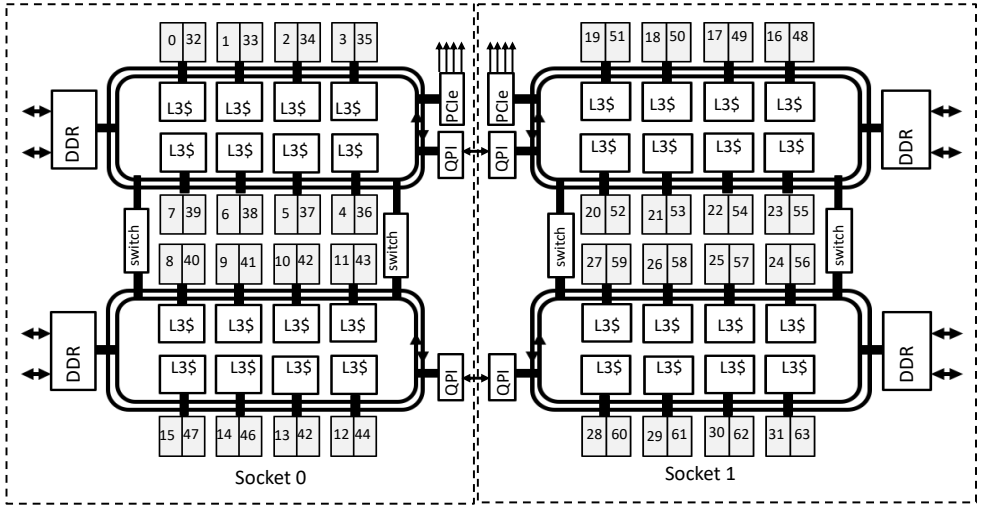
Wrapping up our discussion of taking NUMA features of a system into account in your multithreaded programs …

Getting the affinity right can have serious impacts on performance

Application Benchmark Performance for a number of benchmarks at NERSC



Lower is better



Results running on the Cori system at NERSE which has dual Socket nodes with Intel® Xeon™ E5-2698v3 CPUs

Based on content from Yun (Helen) He from NERSC (National Energy Research Supercomputing Center)

Finding the best strategy for thread affinity

- Experiment to find the best combinations of OMP_PLACES and OMP_PROC_BIND.
  - Using the environment variables makes it easy to try many options

- The best approach depends on the system but also on the features of an application
  - Putting threads for apart … on different sockets
    - May improve aggregate memory bandwidth available to an application
    - May improve combined cache size for the application
    - May increase synchronization overhead
  - Putting threads close together … on adjacent cores that may share some caches
    - May reduce synchronization overhead
    - May decrease memory bandwidth and total cache size

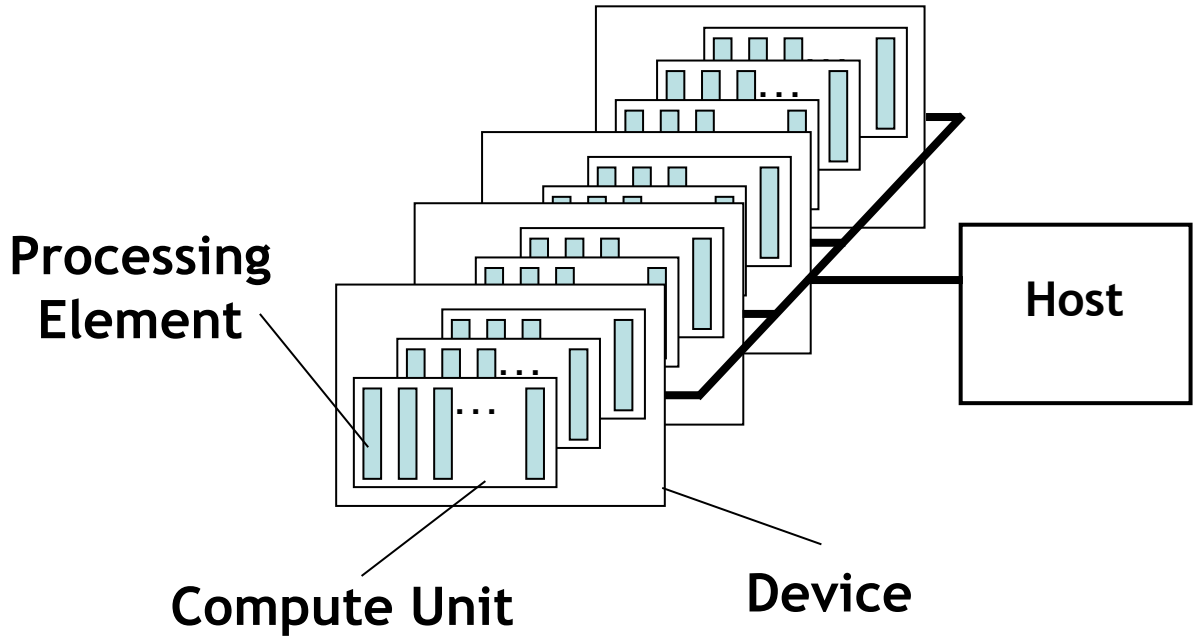
- Vendors have their own constructs for controlling NUMA features of a system.
  - Avoid vendor-specific constructs if you can … use portable OMP_PLACES and OMP_PROC_BIND

Outline

- OpenMP and Real Hardware
- Optimizing code for NUMA systems
- GPU Programming and OpenMP
- Synchronization and the OpenMP memory model
- Random numbers … an example of how to build threadsafe libraries

Introduction to GPU programming

A Generic Host/Device Platform Model



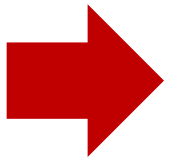
- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***

- Memory divided into ***host memory*** and ***device memory***

The "BIG idea" Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```



Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

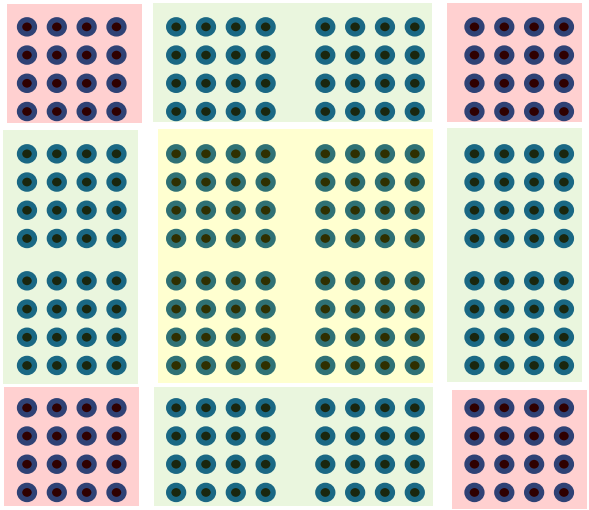
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Write kernel code for the scalar work-items

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< dimGrid,dimBlock>>> (a, b, c, N);
}
```

This is CUDA code

2. Map work-items onto an N dim index space.

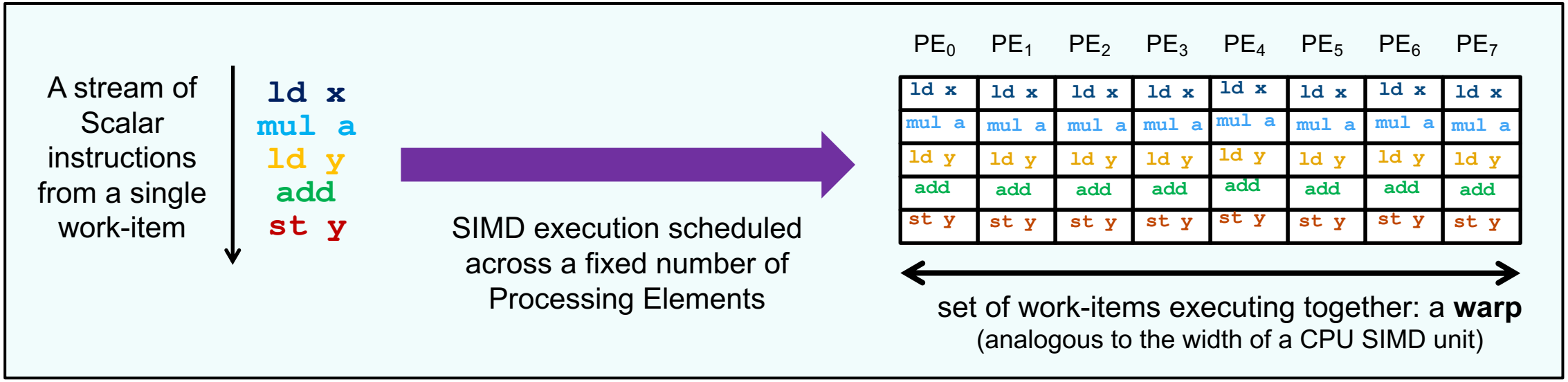


3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

SIMT: One instruction stream maps onto many Processing Elements

- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware

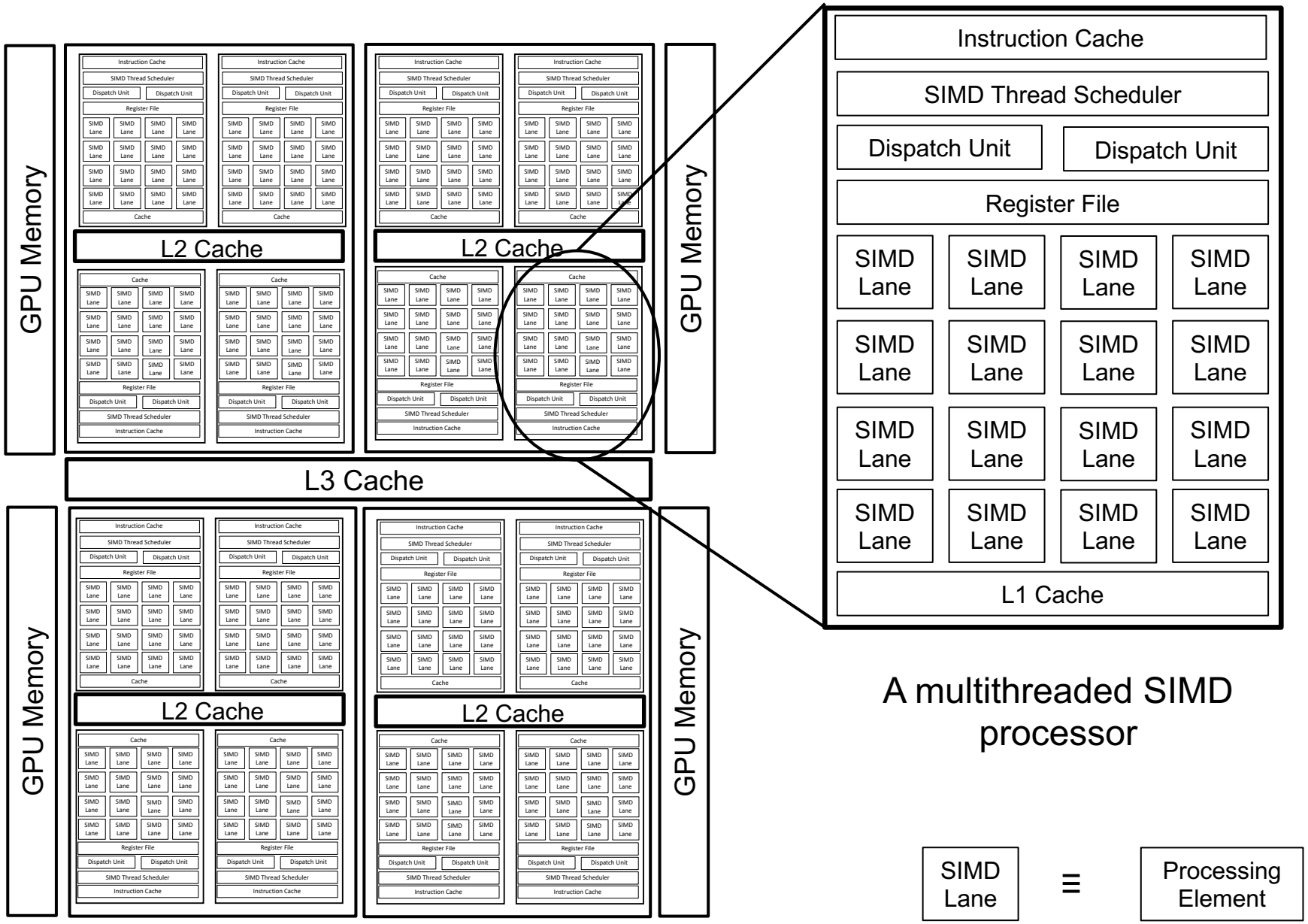


GPU nomenclature is really messed up. (sorry about that … we tried to unify around OpenCL but failed).

Instruction stream at finest grain	**Work-item**, CUDA Thread
Blocks for scheduling work-items	**work-group**, thread block
Execution width for work-items	Subgroup, **warp**
Finest grained processing element (PE) in a GPU	SIMD Lane, **Processing Element**, CUDA Core
Block of PEs driven by a single Instruction sequencer	multithreaded SIMD processor, compute unit, **Streaming multiprocessor**

These names are particularly awful since they conflict with established names from CPU Computing.

A Generic GPU (following Hennessey and Patterson)



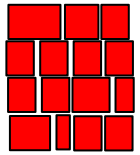
A multithreaded SIMD processor

SIMD Lane ≡ Processing Element ≡ CUDA Core

Computer Architecture: A Quantitative Approach, John L. Hennessy and David A. Patterson.

Executing a program on CPUs and GPUs

Work decomposed into blocks



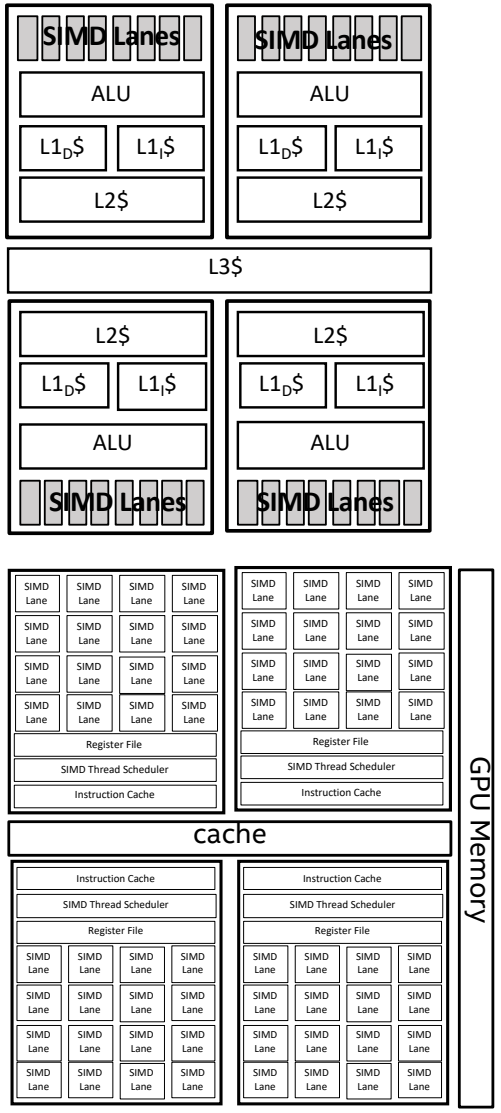
For a CPU

For a GPU

Program defines work

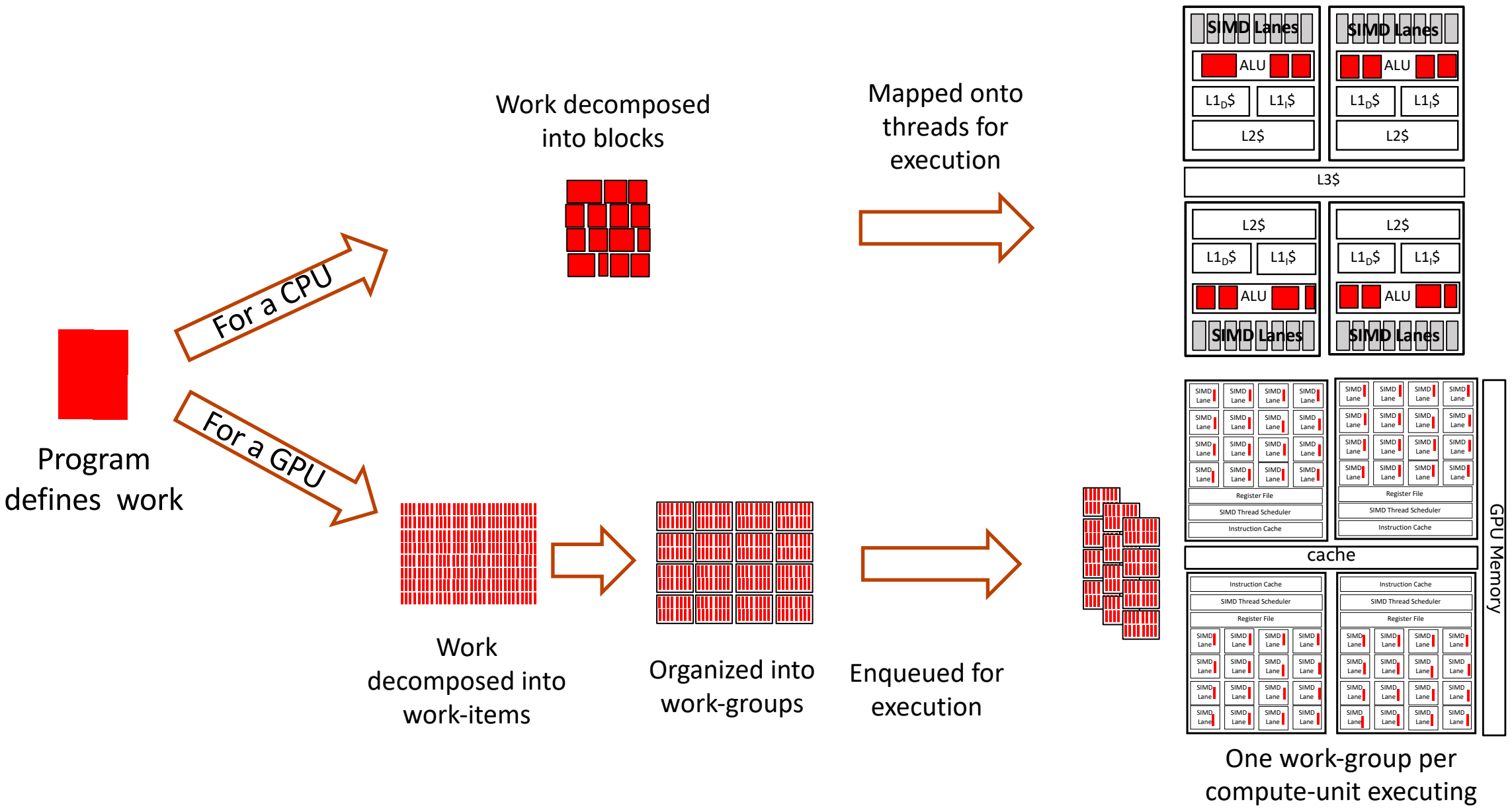
Work decomposed into work-items

Organized into work-groups



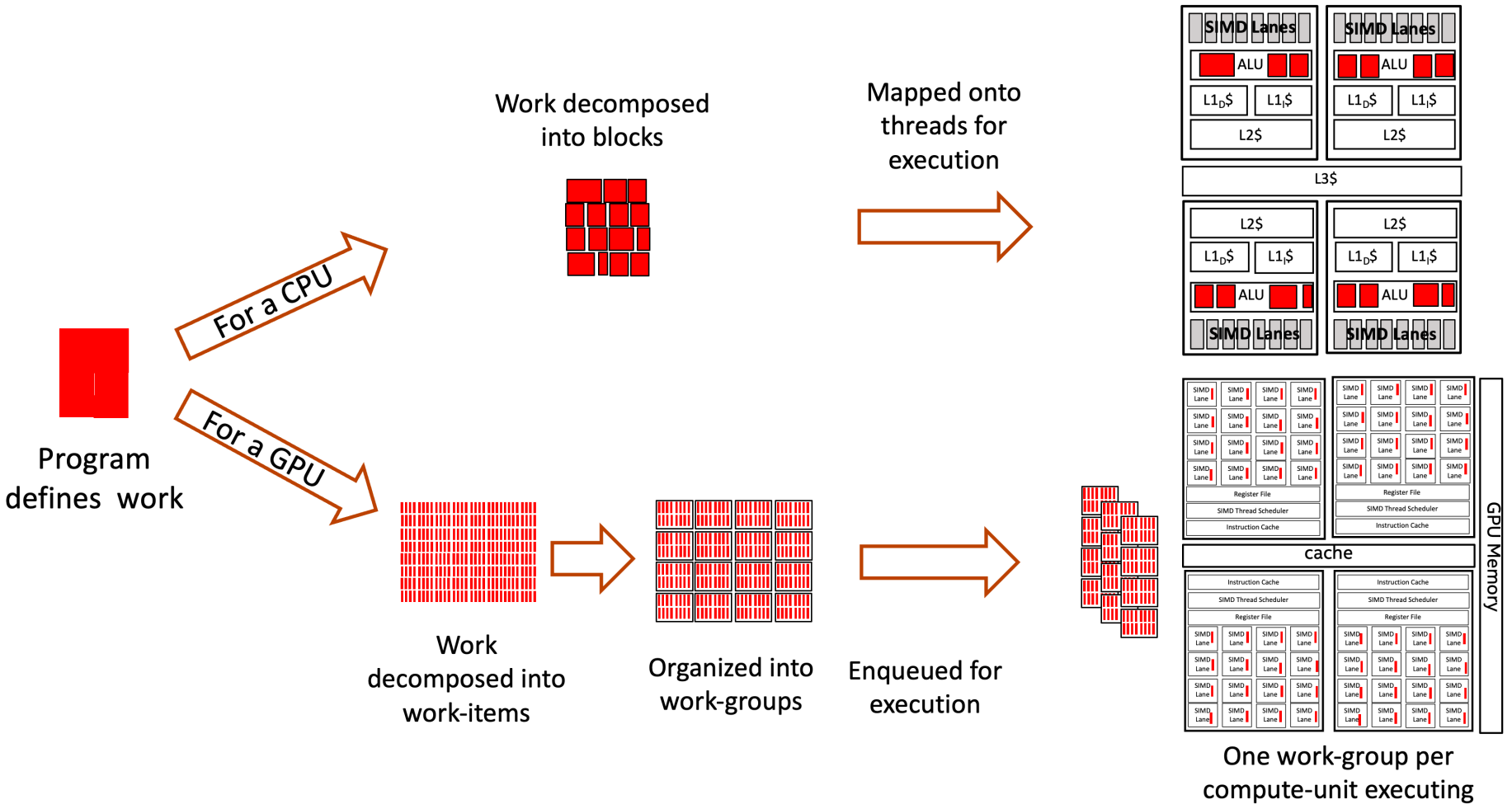
One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

Executing a program on CPUs and GPUs

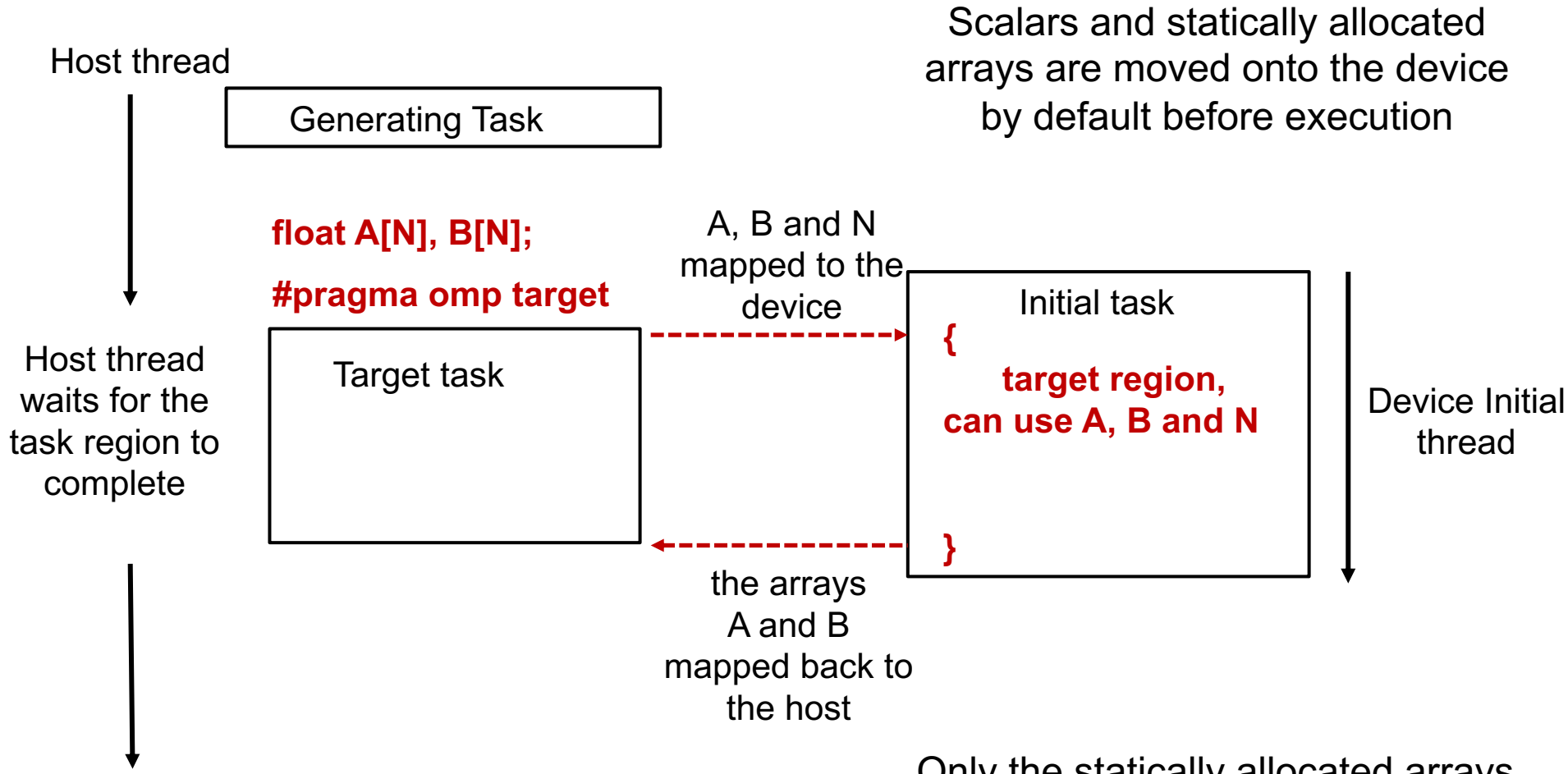


For a CPU, the threads are all active and able to make forward progress.

For a GPU, any given work-group might be in the queue waiting to execute.

Programming a GPU with OpenMP

Running code on the GPU:
The target construct and default data movement



Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Default Data Sharing: example

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {

    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

1. Variables created in host memory.

2. Scalar **N** and stack arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Now let's run code in parallel on the device

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {
    #pragma omp loop
    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

The loop construct tells the compiler:

"this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn't contain any OpenMP constructs. So do whatever you can to make the code run fast"

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to "do it". This is new for OpenMP

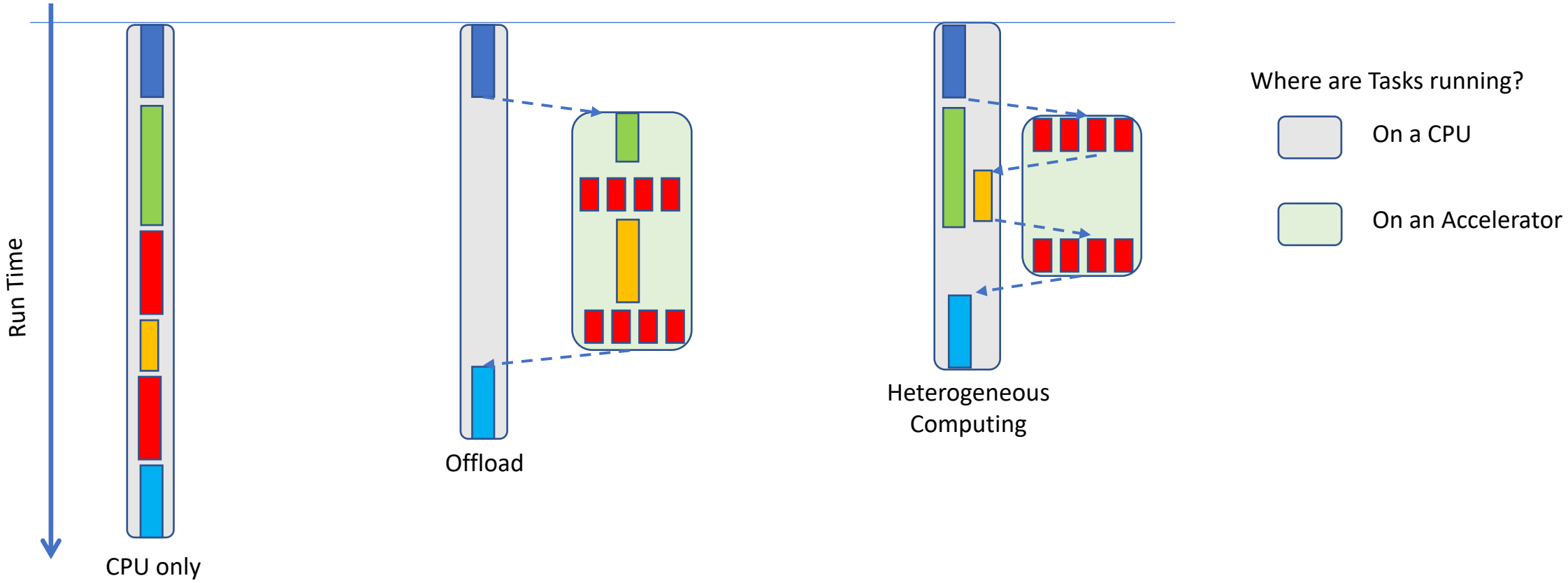
Solution: Simple vector add in OpenMP on GPU

```
int main()
{
    float a[N], b[N], c[N], res[N];
    int err=0;

   // fill the arrays
   #pragma omp parallel for
   for (int i=0; i<N; i++){
      a[i] = (float)i;
      b[i] = 2.0*(float)i;
      c[i] = 0.0;
      res[i] = i + 2*i;
   }

   // add two vectors
   #pragma omp target
   #pragma omp loop
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

   // test results
   #pragma omp parallel for reduction(+:err)
   for(int i=0;i<N;i++){
      float val = c[i] - res[i];
      val = val*val;
      if(val>TOL) err++;
   }
   printf("vectors added with %d errors\n", err);
   return 0;
}
```

No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense

- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



5-point stencil: the heat program

- The heat equation models changes in temperature over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

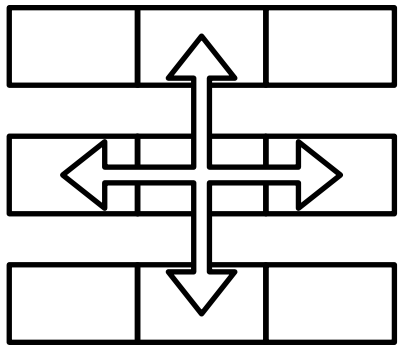
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Serial CPU code

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
 const double r2 = 1.0 - 4.0*r;

 // malloc and initialize u_tmp and u (code not shown)

 for (int t = 0; t < nsteps; ++t) {

    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
 }
}
```

Loop over time steps

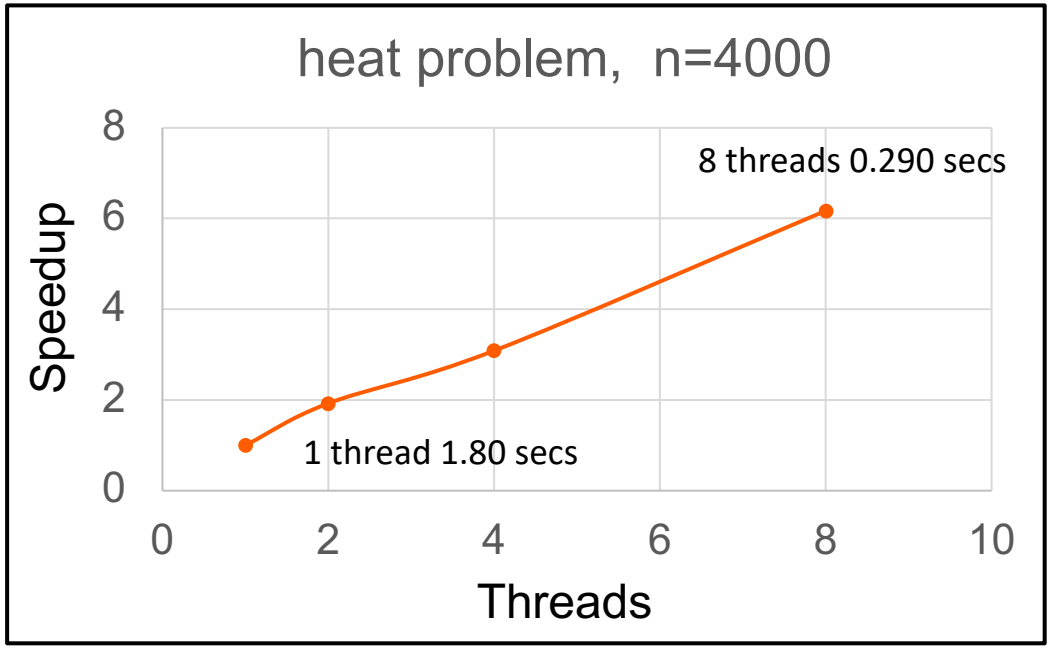
Loop over NxN spatial domain

Update the 5-point stencil. Boundary conditions on the edges of the domain are fixed at zero.

Heat diffusion problem: 5-point stencil code

Parallel CPU code, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {

   #pragma omp parallel for collapse(2)
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```



Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

```
GPU Solver time = 1.40 secs
```

This isn't much better than the runtime for a single CPU (1.8 secs) and worse than 8 cores on a CPU (0.29 secs).

Why is the performance so bad?

```
NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c
```

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
   }
   // Pointer swap to get ready for next step
   tmp = u;
   u = u_tmp;
   u_tmp = tmp;
}
}
```

With a runtime of 1.4 secs (worse than the CPU time) we see that Data Movement dominates performance.

At the beginning of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes to the device

We need to create a **data region** on the GPU that is distinct from the target region.

That way, we can keep the data on the device between target constructs

At the end of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes from the device

Target enter/exit data constructs

- Create a data region on the target device (a device data environment) with two standalone directives:

```
#pragma omp target enter data map(…)
#pragma omp target exit data map(…)
```

- The **target enter data** maps variables to the device data environment.

- The **target exit data** unmaps variables from the device data environment.

- Once created, subsequent **target** regions inherit the existing data environment.

Target enter/exit data example

```
void init_array(int *A, int *B, int N) {
   for (int i = 0; i < N; ++i) { A[i] = i;  B[i]=2*I;}

  #pragma omp target enter data map(to: A[0:N], B[0:N])
}

int main(void) {

   int N = 1024;
   int *A = malloc(sizeof(int) * N);
   int *B = malloc(sizeof(int) * N);
   init_array(A, B, N);

   #pragma omp target
   #pragma omp loop
   for (int i = 0; i < N; ++i)
      A[i] = A[i] * B[i];

 #pragma omp target exit data map(from: A[0:N])
}
```

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;
  // malloc and initialize u_tmp and u (code not shown)
  #pragma omp target enter data map(to: u[0:n*n], u_tmp[0:n*n])

  for (int t = 0; t < nsteps; ++t) {
     #pragma omp target
     #pragma omp loop
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]          +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
       }
     }
     // Pointer swap to get ready for next step
     tmp = u;
     u = u_tmp;
     u_tmp = tmp;
  }
 #pragma omp target exit data map(from: u[0:n*n])
}
```

Create a data region and map indicated data on entry

GPU Solver time* = **0.42** secs

This is a general principal … if you want performance, you must optimize data movement.

*includes time for target enter/exit data

Exit the data region and map indicated data

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c

Heat diffusion problem: 5-point stencil code

Parallel CPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)


for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
     }
     // Pointer swap for  next step
     tmp = u;
     u = u_tmp;
     u_tmp = tmp;
  }

}
```

This is the ij loop order.

Let's optimize the CPU code as well

CPU

Num threads	ij loop order
1	1.512849
2	0.776229
4	0.400822
8	0.227317

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

Heat diffusion problem: 5-point stencil code

Parallel CPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)


for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for
    for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap for  next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

}
```

This is the ji loop order. Swap these loops to get the ij order.

Make j the outermost loop so adjacent loop iterations access adjacent memory locations.

CPU

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

This is particularly important on a GPU … you want memory coalesced with the GPUs processing elements (PE) … i.e., elements of u accessed by PE_i should be adjacent to the elements of u accessed by PE_{i+1}

Heat diffusion problem: 5-point stencil code

Parallel CPU and GPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)
#pragma omp target enter data map(to: u[0:n*n], u_tmp[0:

for (int t = 0; t < nsteps; ++t) {
    #pragma omp target
    #pragma omp loop
    for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
 #pragma omp target exit data map(from: u[0:n*n])
}
```

This is the ji loop order.

Memory coalescence is important for CPUs and GPUs.

Note: collapse(2) did not help on the GPU or the CPU

CPU

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

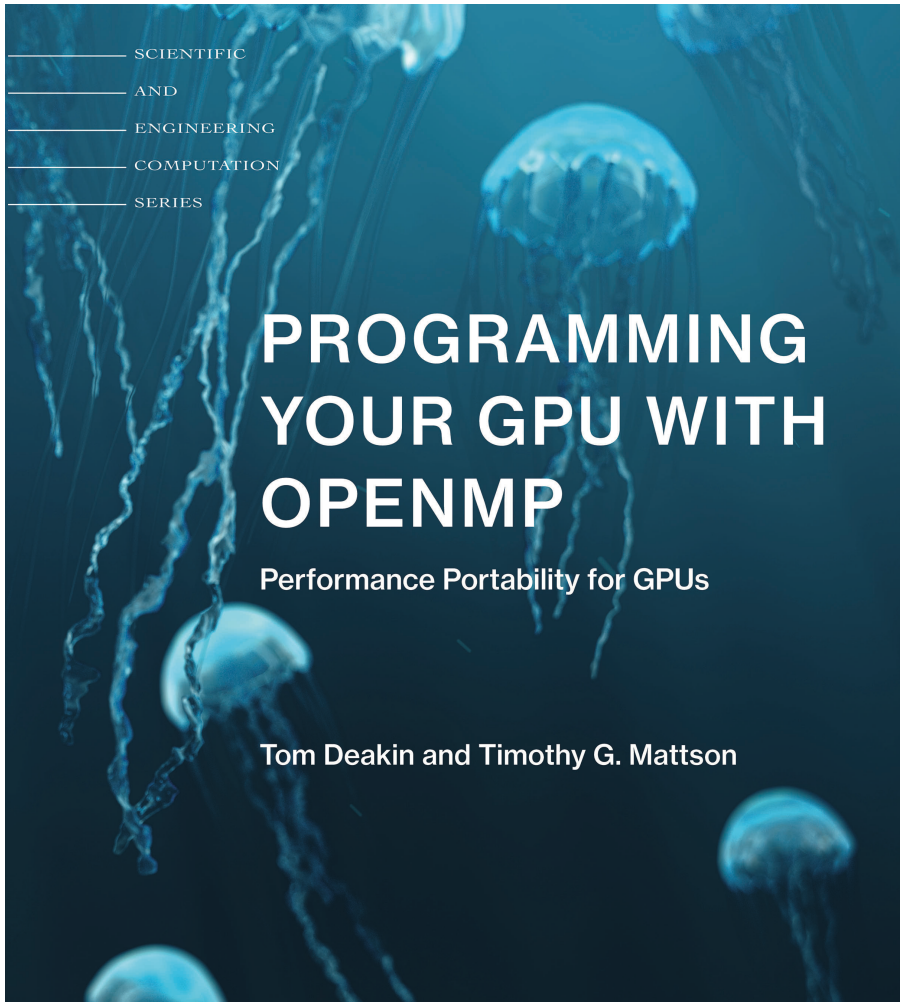
GPU

ij without timing enter and exit data	ij loop order	ji without timing enter and exit data	ji loop order
0.056830	0.417887	0.020123	0.358905

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler. nvc -fast -mp=gpu heat.c

GPU programming with OpenMP

- There is much more … which you can learn about from our book
  - Loop is a descriptive construct … you leave all the details to the runtime. Always start with Loop plus enter-data/exit-data since often that is all you need
  - OpenMP includes constructs for detailed control of the GPU so you can do programing akin to that with CUDA. I do not recommend this. You maximize portability if you let the runtime system handle mapping code onto hardware details for you. But if you want to control local memories, you may have no choice.
  - The interop constructs let you call functions native to a particular GPU (such as BLAS) from inside the OpenMP program. They are a bit complicated to work with. See our book to learn more.



Learn all the details of GPU programming with OpenMP (up to version 5.2) . Released in November 2023

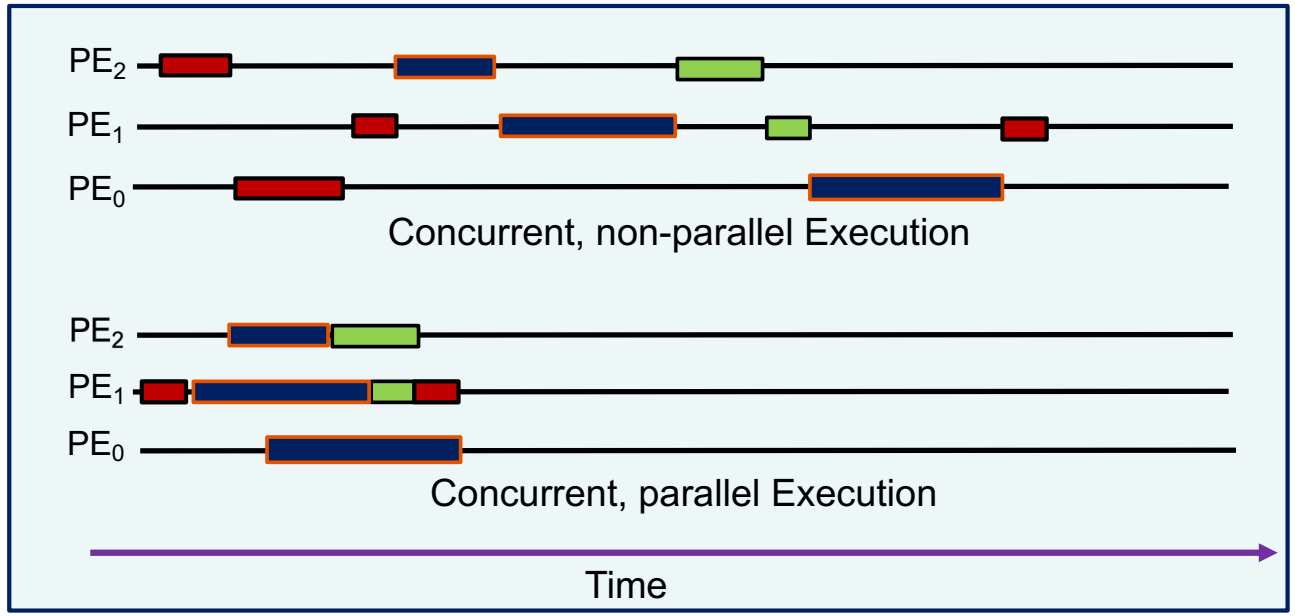
Outline

- OpenMP and Real Hardware
- Optimizing code for NUMA systems
- GPU Programming and OpenMP
- Synchronization and the OpenMP memory model
- Random numbers … an example of how to build threadsafe libraries

Concurrency vs. Parallelism

- Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.

- Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.



With Multithreading systems such as OpenMP, threads execute concurrently.
Synchronization is used to create and ordering constraint in the otherwise unordered execution of threads

PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Histogram: a running example to explore synchronization

- Our histogram program 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.
- We will parallelize the filling of the histogram in a way to ensure the program is race free and gets the same result as the sequential program.

```
for(i=0;i<NBUCKETS; i++){
     hist[i] = 0;
}


for(i=0;i<NVALS;i++){
    ival = (int)  x[i];
    hist[ival]++;
}
```

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 with locks: mutual exclusion but more control

- 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

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

Free-up storage when done.

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

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)

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

Atomics are far more important than just a lightweight way to enforce mutual exclusion.

Atomics are essential when you need to work with the details of the OpenMP Memory model.

And this comes up when you consider pairwise synchronization

Consider two threads: a producer/consumer pair

One thread **produces** a result that a different thread **consumes**

```
#include <stdio.h>
#include <omp.h>
#define COUNT 1000000
int main()
{
  int answer = 0, flag= 0,err=0;
  for (int i=0; i<COUNT; i++) {
    flag = 0;  answer=0;
    #pragma omp parallel shared(flag,answer) num_threads(2)
    {
      int id = omp_get_thread_num();
      if (id == 0) {
         answer = 42;
         flag = 1;
      }
      else if (id == 1){
         while (flag == 0)  { }
         if(answer!=42) err++;
      }
    }
  }
  return 0;
}
```

Thread zero produces the answer and then sets a flag to communicate the answer to another thread

Thread one “spins" in a while loop until the flag is non-zero which indicates that answer is available.

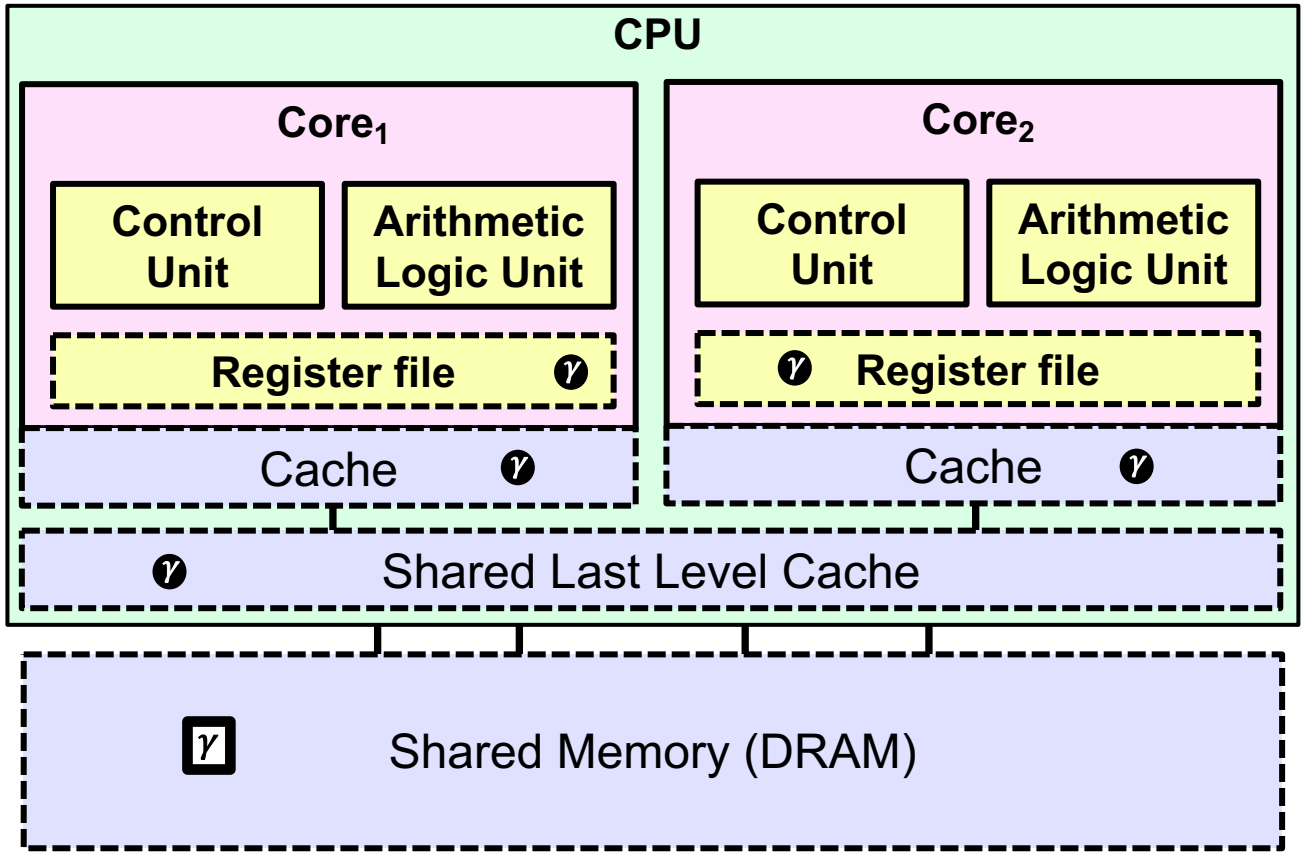
In the jargon of concurrent programming, this is called a “spin lock”

Put this in a file sync.c and compile as: **gcc –fopenmp –O3 sync.c**

The program went through a few loop iterations and then hangs …. Why?

Memory Models …

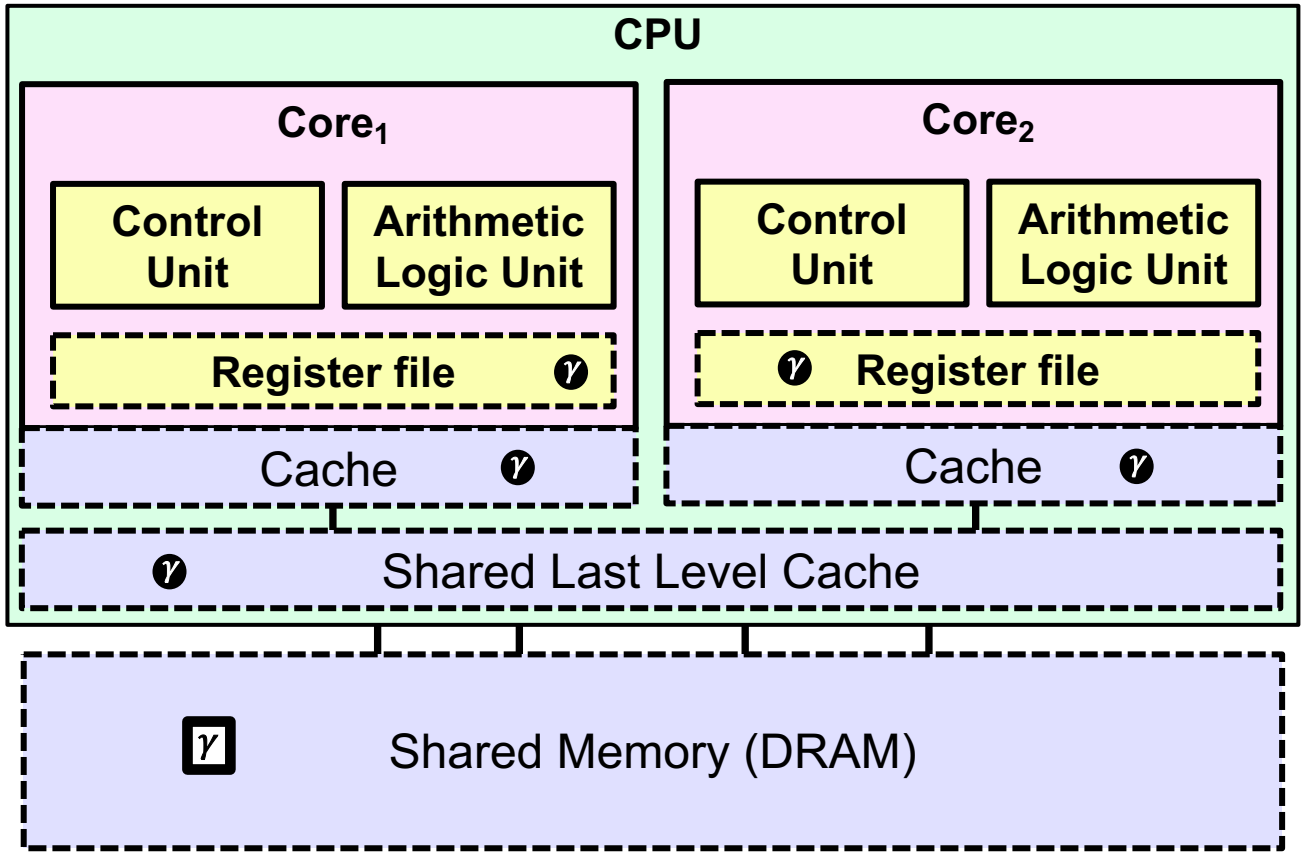
- A shared address space is a region of memory visible to the team of threads … multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different values.



- Which value of γ is the one a thread should see at any point in a computation?

Memory Models …

- A shared address space is a region of memory visible to the team of threads … multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different values.



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

- Which value of γ is the one a thread should see at any point in a computation?

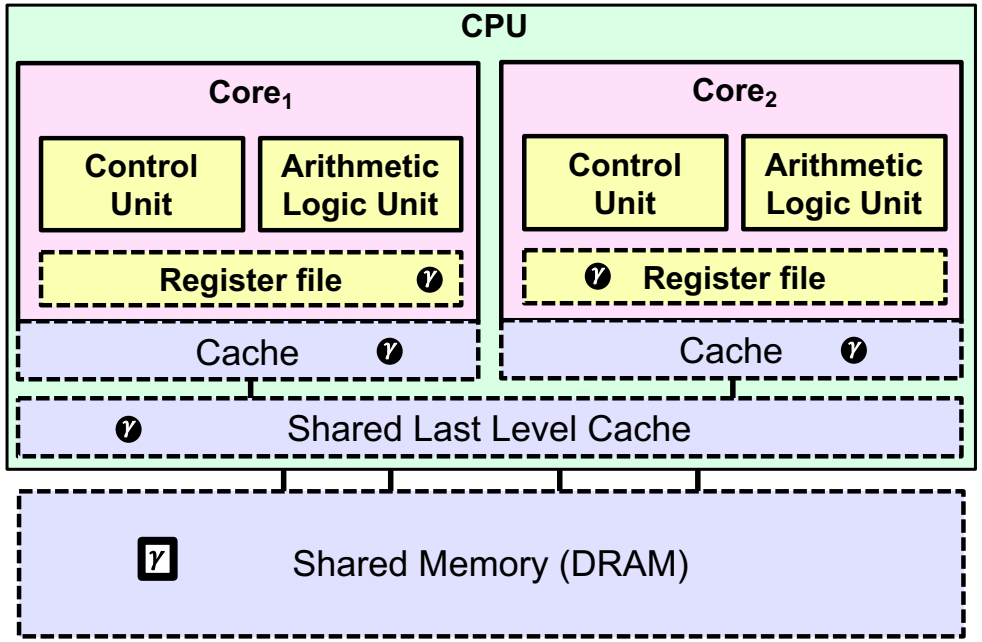
Memory Models …

- The fundamental issue is how do the values of variables across the memory hierarchy interact with the statements executed by two or more threads?
- Two options:

1. **Sequential Consistency**

- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.

- **All threads see the same interleaved order of loads and stores**



2. **Relaxed Consistency**

- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.

- **Threads may see different orders of loads and stores**

Most (if not all) multithreading programming models assume **relaxed consistency**. Maintaining sequential consistency across the full program-execution adds too much synchronization overhead.

Why did this program fail?

Two issues:

(1) Can **flag** = 1 while **answer** = 0?

(2) Can thread 1 fail to see updates to **flag**?

```
#include <stdio.h>
#include <omp.h>
#define COUNT 1000000
int main()
{
  int answer = 0, flag= 0,err=0;
  for (int i=0; i<COUNT; i++) {
    flag = 0;  answer=0;
    #pragma omp parallel shared(flag,answer) num_threads(2)
    {
      int id = omp_get_thread_num();
      if (id == 0) {
         answer = 42;
         flag = 1;
      }
      else if (id == 1){
         while (flag == 0)  { }
         if(answer!=42) err++;
      }
    }
  }
  return 0;
}
```

The compiler can reorder statements, so **flag** is set to 1 before **answer** is set to **42**

Thread 1 can load **flag** from the register file. It may not even go to cache (let alone memory) to see an updated value.

Regardless of how the compiler orders stores to **answer** and **flag**, thread 1 may see a different order than thread 0

Why did this program fail?

Two issues:

(1) Can **flag** = 1 while **answer** = 0?

(2) Can thread 1 fail to see updates to **flag**?

```
#include <stdio.h>
#include <omp.h>
#define COUNT 1000000
int main()
{
  int answer = 0, flag= 0,err=0;
  for (int i=0; i<COUNT; i++) {
    flag = 0;  answer=0;
    #pragma omp parallel shared(flag,answer) num_threads(2)
    {
      int id = omp_get_thread_num();
      if (id == 0) {
         answer = 42;
         flag = 1;
      }
      else if (id == 1){
         while (flag == 0)  { }
         if(answer!=42) err++;
      }
    }
  }
  return 0;
}
```

We need to enforce ordering constraints between the concurrent threads … we need to consider the memory model and put the right synchronization constructs in place.

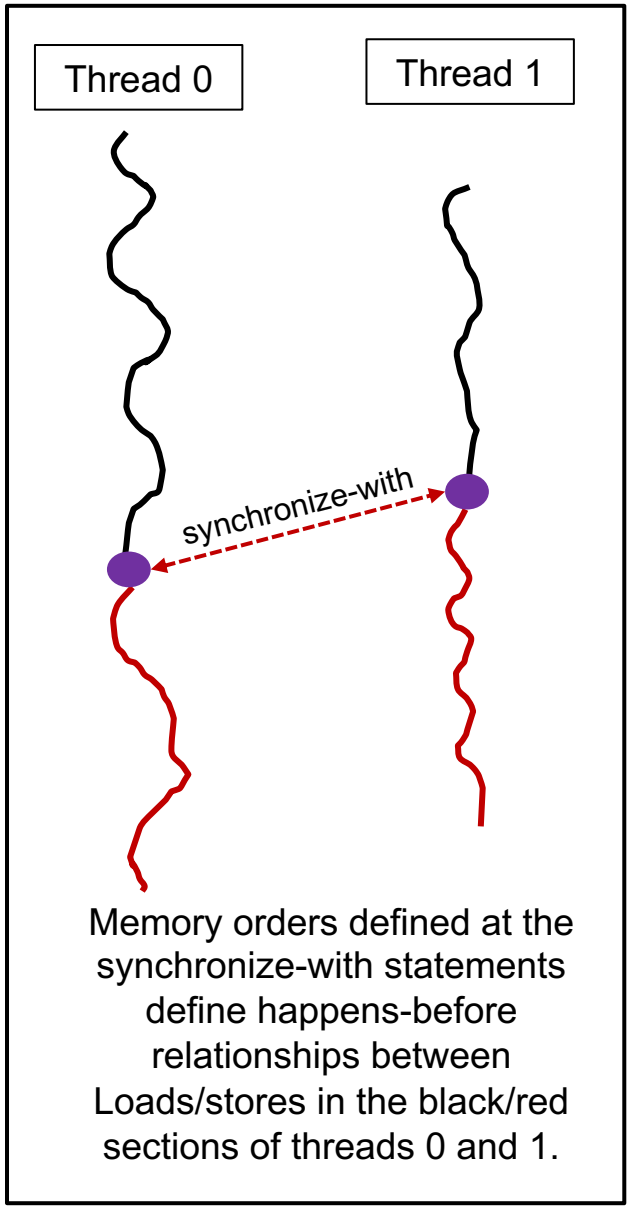
The compiler can reorder statements, so **flag** is set to 1 before **answer** is set to **42**

Thread 1 can load **flag** from the register file. It may not even go to cache (let alone memory) to see an updated value.

Regardless of how the compiler orders stores to **answer** and **flag**, thread 1 may see a different order than thread 0

Memory Models: *Happens-before* and *synchronized-with* relations

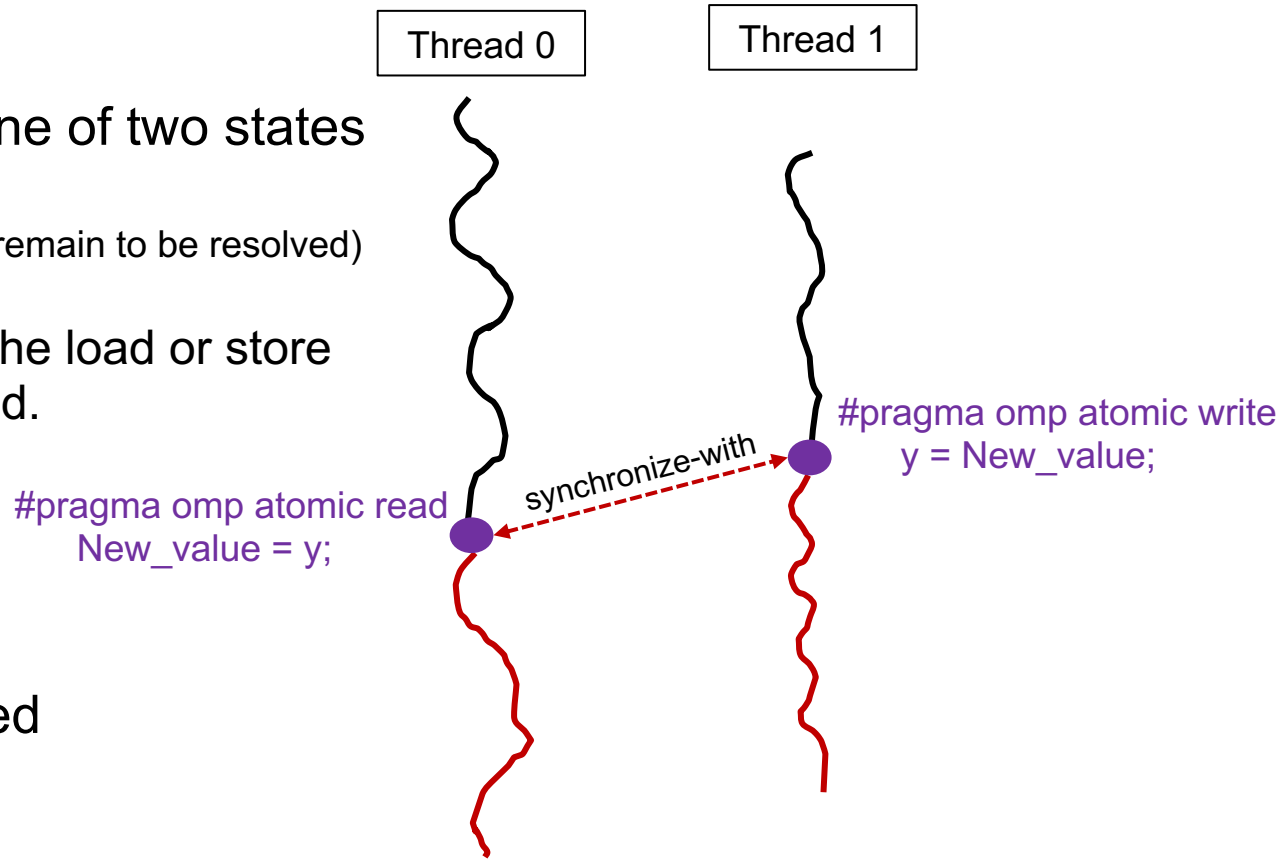
- Single thread execution:
  - Program order ... Loads and stores appear to occur in the order defined by the program's semantics. If you can't observe it, however, compilers can reorder instructions to maximize performance.
- Multithreaded execution … concurrency in action
  - The compiler doesn't understand instruction-ordering across threads … loads/stores to shared memory across threads can expose ambiguous orders of loads and stores
  - Instructions between threads are unordered except when specific ordering constraints are imposed, i.e., **synchronization**.
  - Synchronization lets us force that some instructions **happens-before** other instructions
- Two parts to synchronization:
  - A **synchronize-with** relationship exists at statements in 2 or more threads at which memory order constraints can be established.
  - **Memory order**: defines the view of loads/stores on either side of a synchronized-with operations.



Memory orders defined at the synchronize-with statements define happens-before relationships between Loads/stores in the black/red sections of threads 0 and 1.

Atomic Operations and Synchronized-with

- An atomic operation can only be observed in one of two states
  - The operation has not happened yet
  - The operation has happened and is complete (no side-effects remain to be resolved)
- For example, on an atomic load or store operation, the load or store has happened and is complete, or it has not occurred.



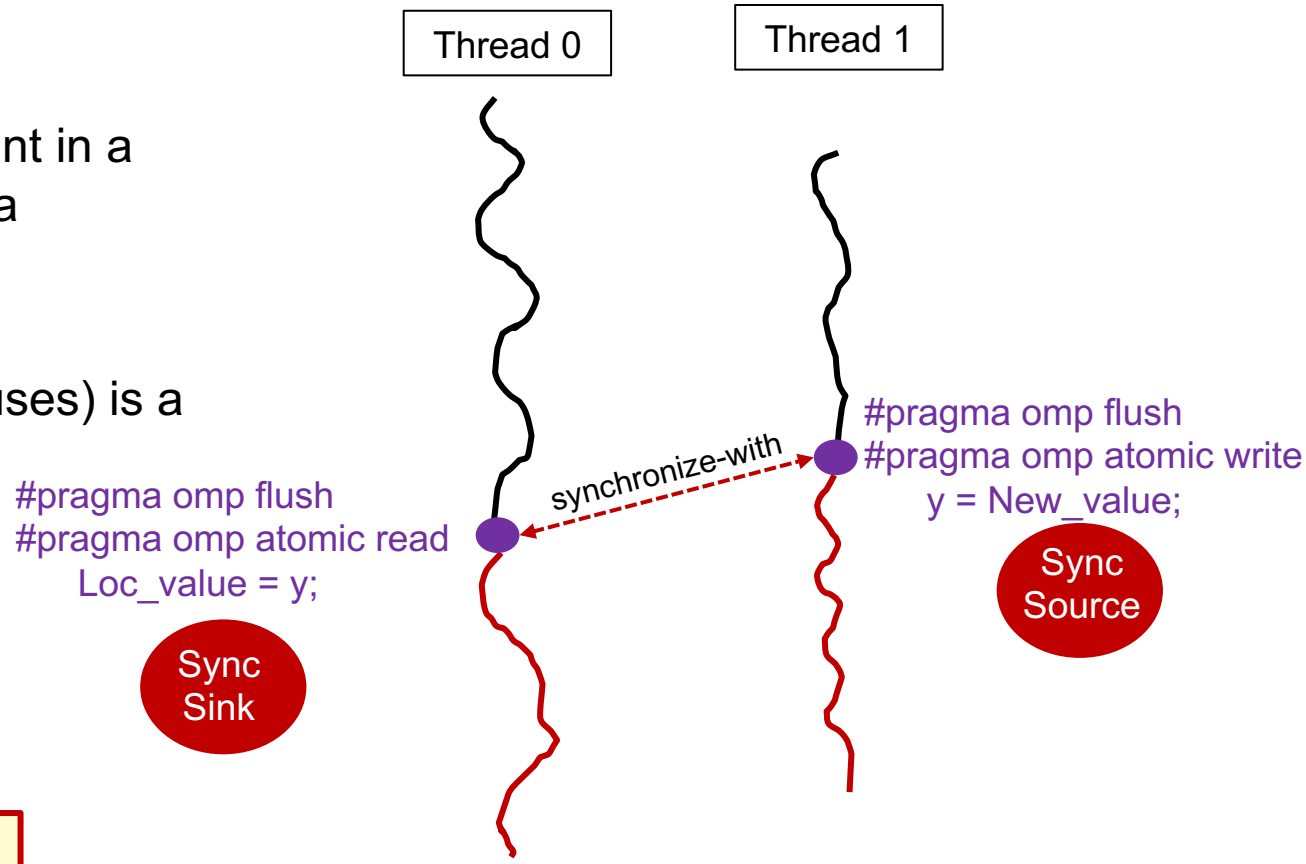
- A **synchronized-with** relationship is established between a pair of atomic operations.
- The variables involved are visible to the programmer (such as with atomic constructs) or the variables are internal to a high level synchronization construct (barrier, critical, locks, etc).

Memory orders

- Memory orders establish which loads and stores can be moved around synchronized-with relations.
- The key construct is **flush**. … flush defines a point in a program at which a thread is guaranteed to see a consistent view of memory.
- The default case for flush (i.e., no additional clauses) is a **strong flush**:
  - Previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred

A strong flush on its own does NOT define a synchronization point. The flush only addresses memory orders.

To synchronize threads, you need a synchronized-with relation which in this case, comes from an atomic write paired with an atomic read



Memory orders defined at the synchronize-with statements define happens-before relationships between Loads/stores in the black/red sections of threads 0 and 1.

Black operations on Thread 1 happen-before Red operations on thread 0.

Memory orders

- Memory orders establish which loads and stores can be moved around synchronized-with relations.
- The strong flush by itself is expensive as it impacts all the shared variables visible to a thread.
- There are more focused forms of memory order
  The 2 most fundamental memory orders are:

read-acquire

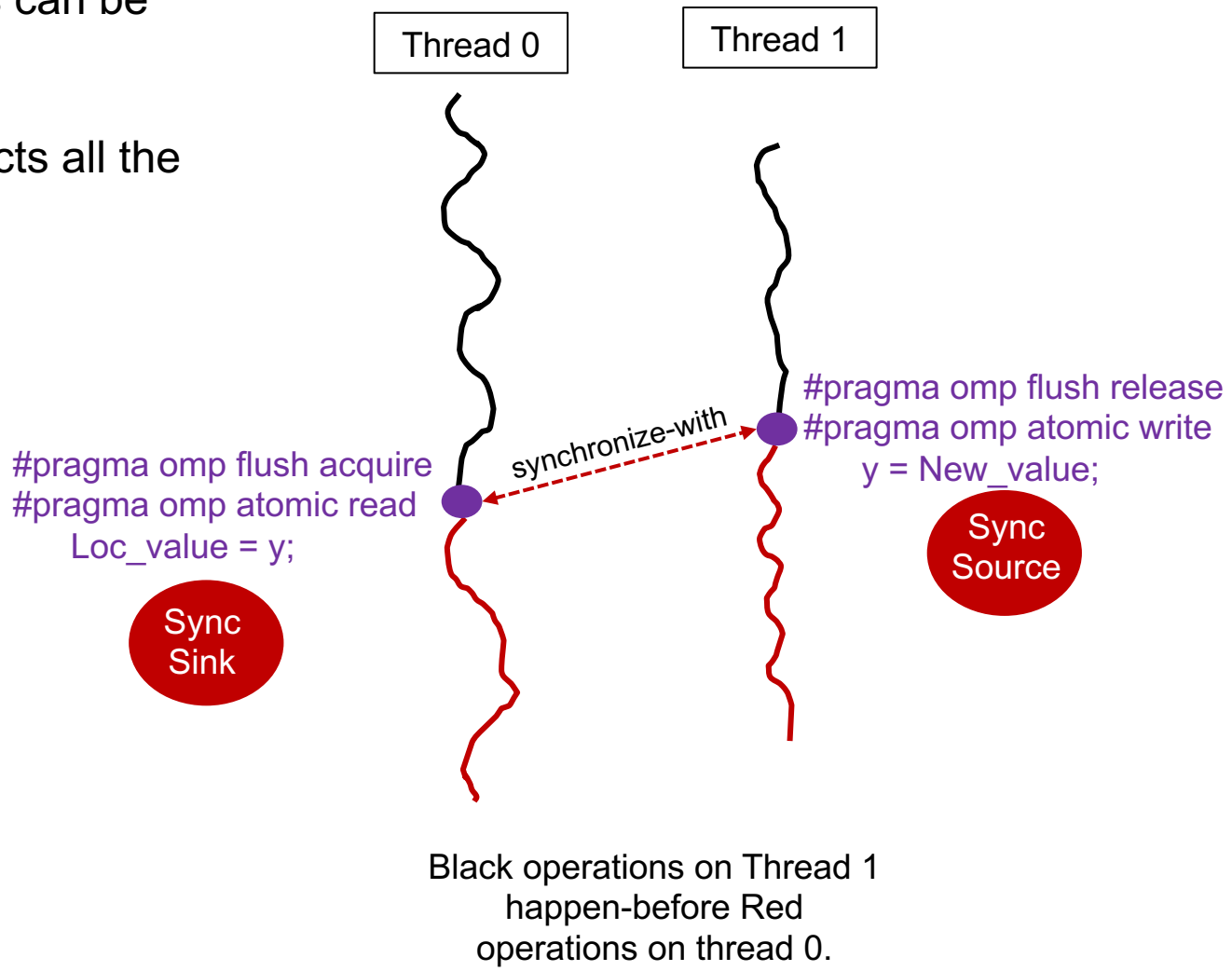
all memory operations stay below the line

- Acquire: Reads/writes that follow the read-with-acquire cannot happen-before the read-with-acquire operation.

all memory operations stay above the line

write-release

- Release: Reads/Writes prior to the write-with-release must happen-before the write-with-release.



Black operations on Thread 1 happen-before Red operations on thread 0.

Memory orders

- Memory orders establish which loads and stores can be moved around synchronized-with relations.
- The strong flush by itself is expensive as it impacts all the shared variables visible to a thread.
- There are more focused forms of memory order
  The 2 most fundamental memory orders are:

read-acquire

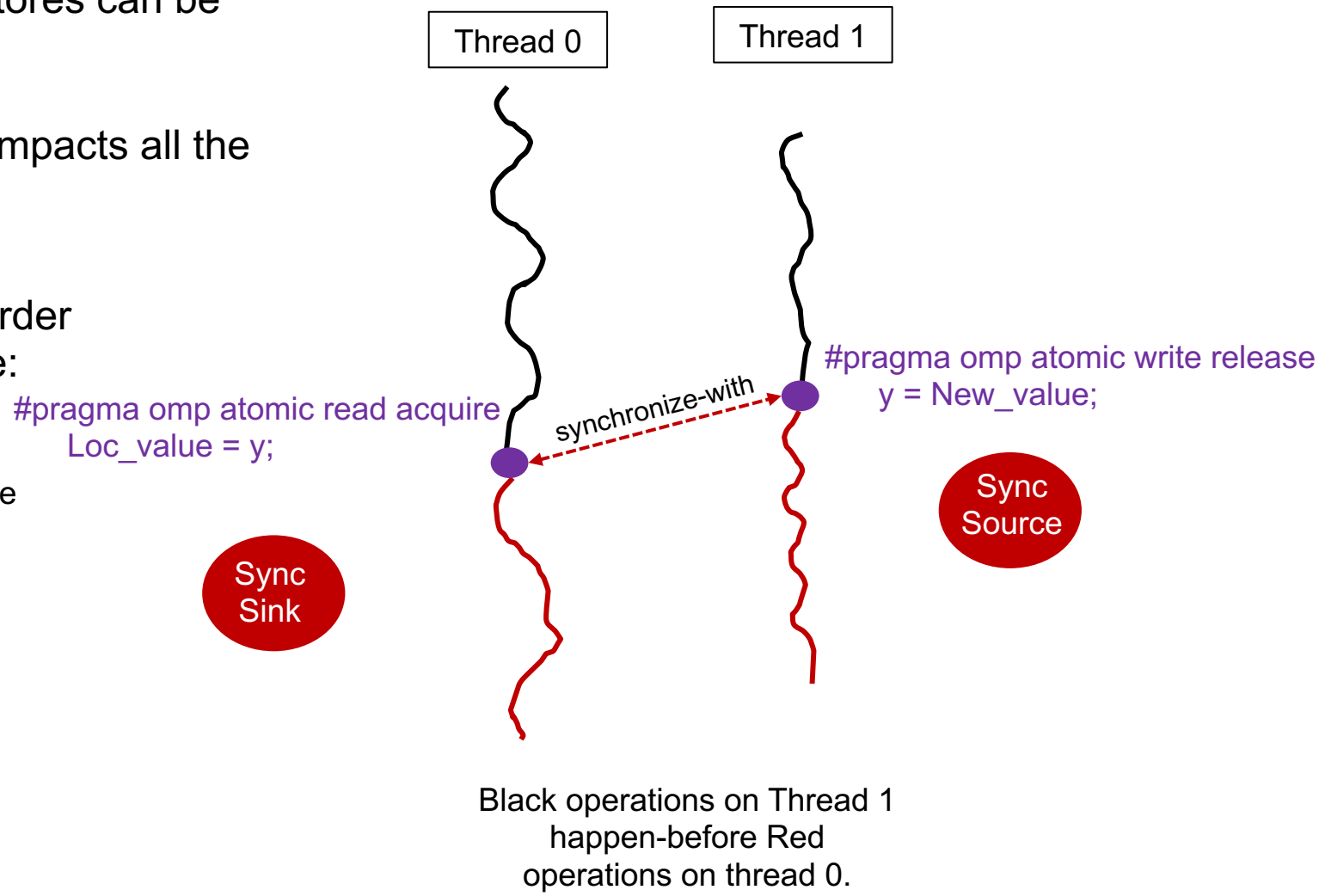
all memory operations stay below the line

- Acquire: Reads/writes that follow the read-with-acquire cannot happen-before the read-with-acquire operation.

all memory operations stay above the line

write-release

- Release: Reads/Writes prior to the write-with-release must happen-before the write-with-release.



Black operations on Thread 1 happen-before Red operations on thread 0.

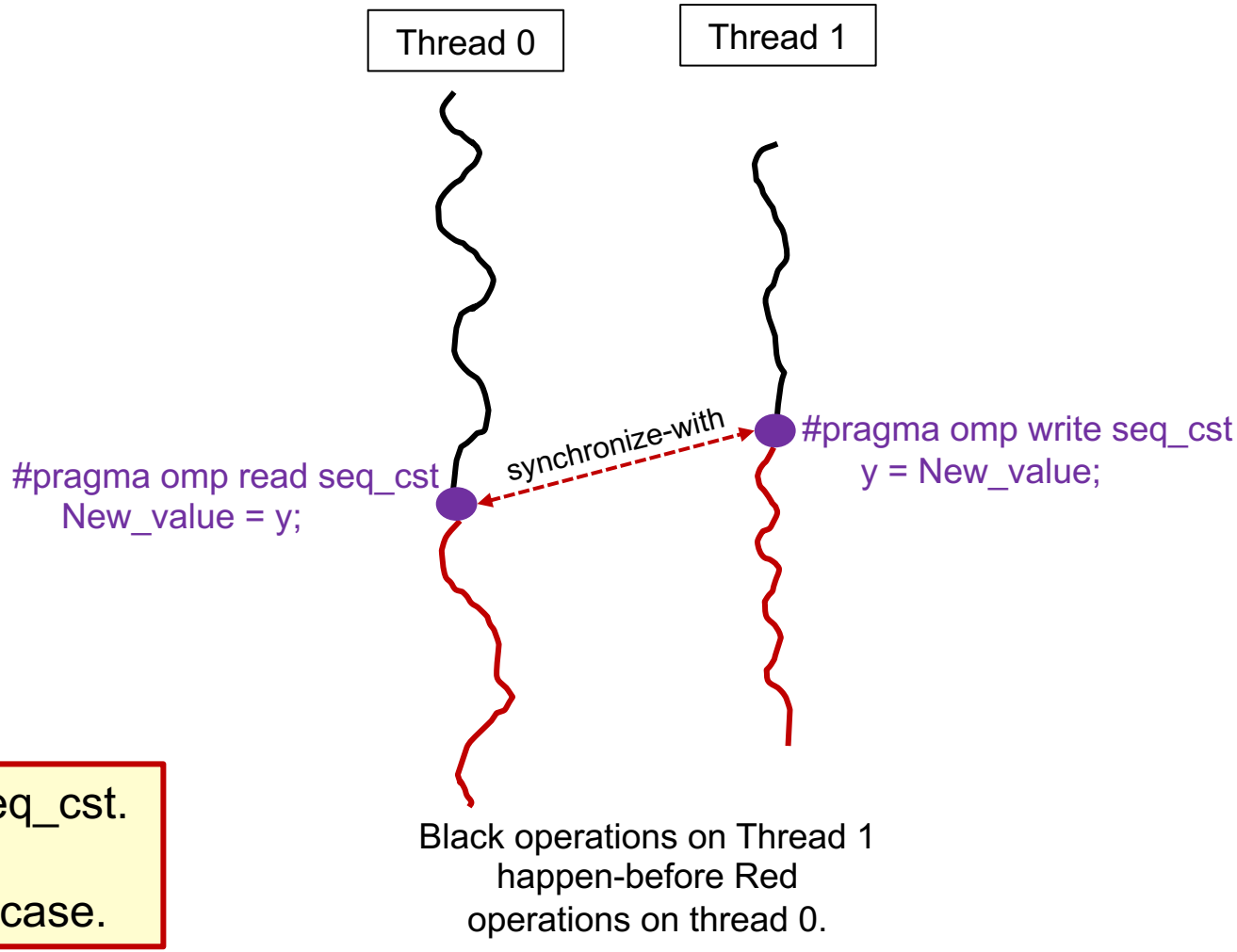
We can combine the flush and the atomic constructs

producer/consumer program correctly synchronized

```
#include <stdio.h>
#include <omp.h>
#define COUNT 1000000
int main()
{
  int answer = 0, flag= 0,err=0;
  #pragma omp parallel shared(flag,answer) num_threads(2)
  {
    int id = omp_get_thread_num();
    if (id == 0) {
       answer = 42;
       #pragma omp atomic write release
           flag = 1;
    }
    else if (id == 1){
       int fetch = 0;
       while (fetch == 0)  {
         #pragma omp atomic read acquire
           fetch = flag;
       }
       if(answer!=42) err++;
    }
  }
  return 0;
}
```

Other Memory orders in OpenMP

- Other OpenMP memory orders
  - **acq_rel**: Applies acquire and release memory order constraints at a single point in a program's execution.
  - **seq_cst**: sequential consistency. All data accessible to a thread are written to memory, subsequent writes are set to load from memory (akin to the strong flush)



The most important memory order to use is seq_cst.

It can be more expensive, but it is the safest case.

Keep it simple … let OpenMP take care of Flushes for you

- A flush operation is implied by OpenMP constructs …
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

This has not been a detailed discussion of the full OpenMP memory model. The goal was to explain how memory models work and to understand the subset of features people commonly use.

- OpenMP programs that:
  - Do not use non-sequentially consistent atomic constructs;
  - Do not rely on the accuracy of a false result from omp_test_lock and omp_test_nest_lock; and
  - Correctly avoid data races

… behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flushes in such programs are redundant.

WARNING:
If you find yourself 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.

Outline

- OpenMP and Real Hardware
- Optimizing code for NUMA systems
- GPU Programming and OpenMP
- Synchronization and the OpenMP memory model
- Random numbers … an example of how to build threadsafe libraries

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);
}
```

Data Copying: Copyin

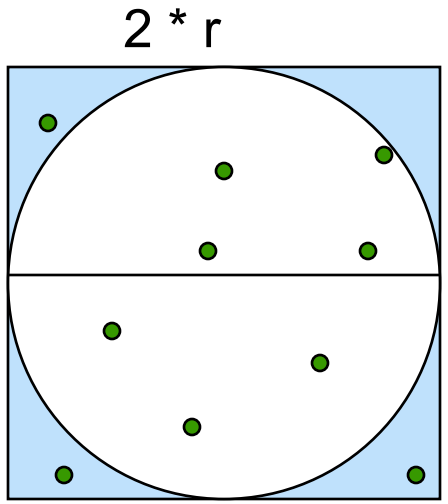
You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

!$ Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialized
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

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;
```

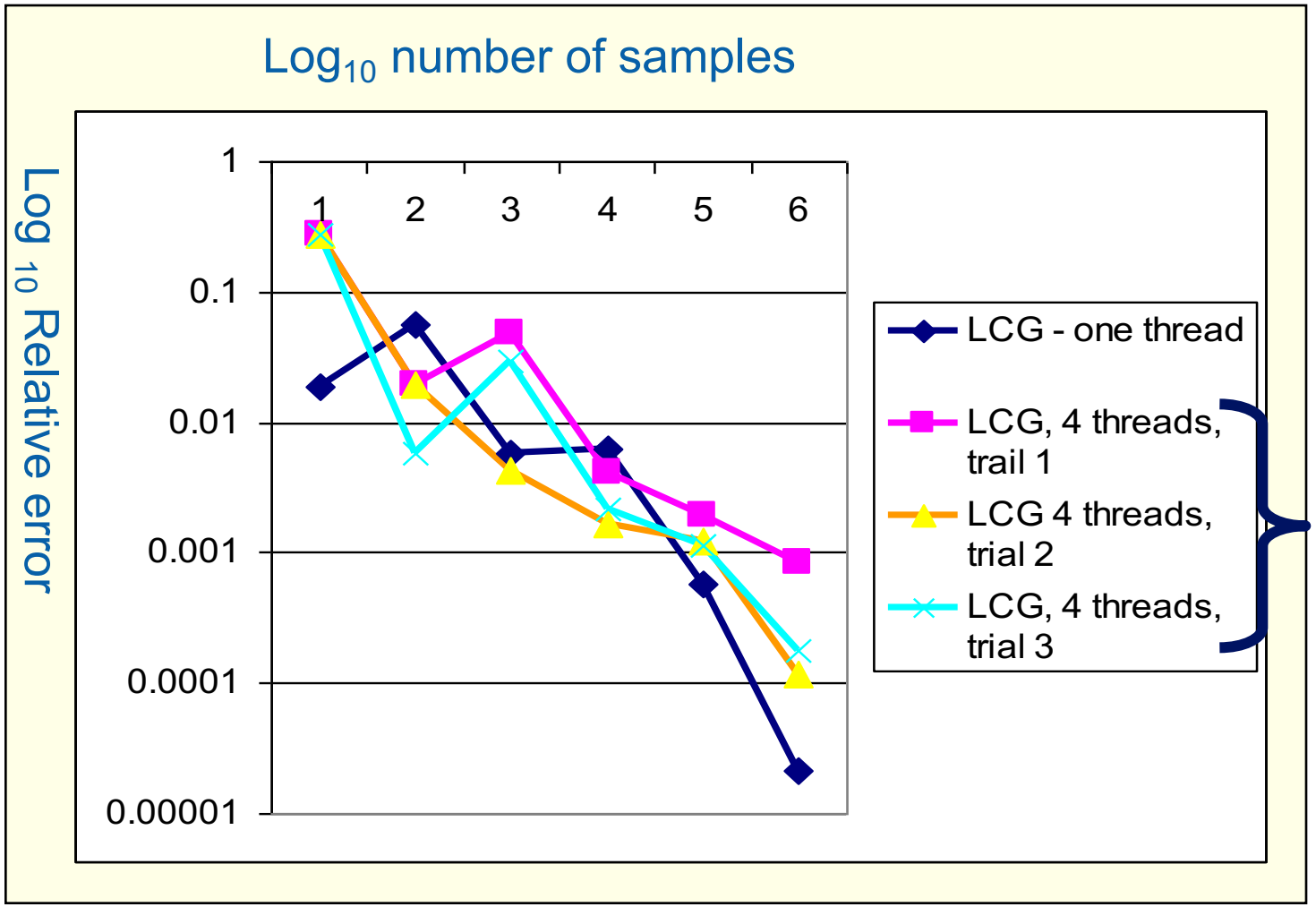
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

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?

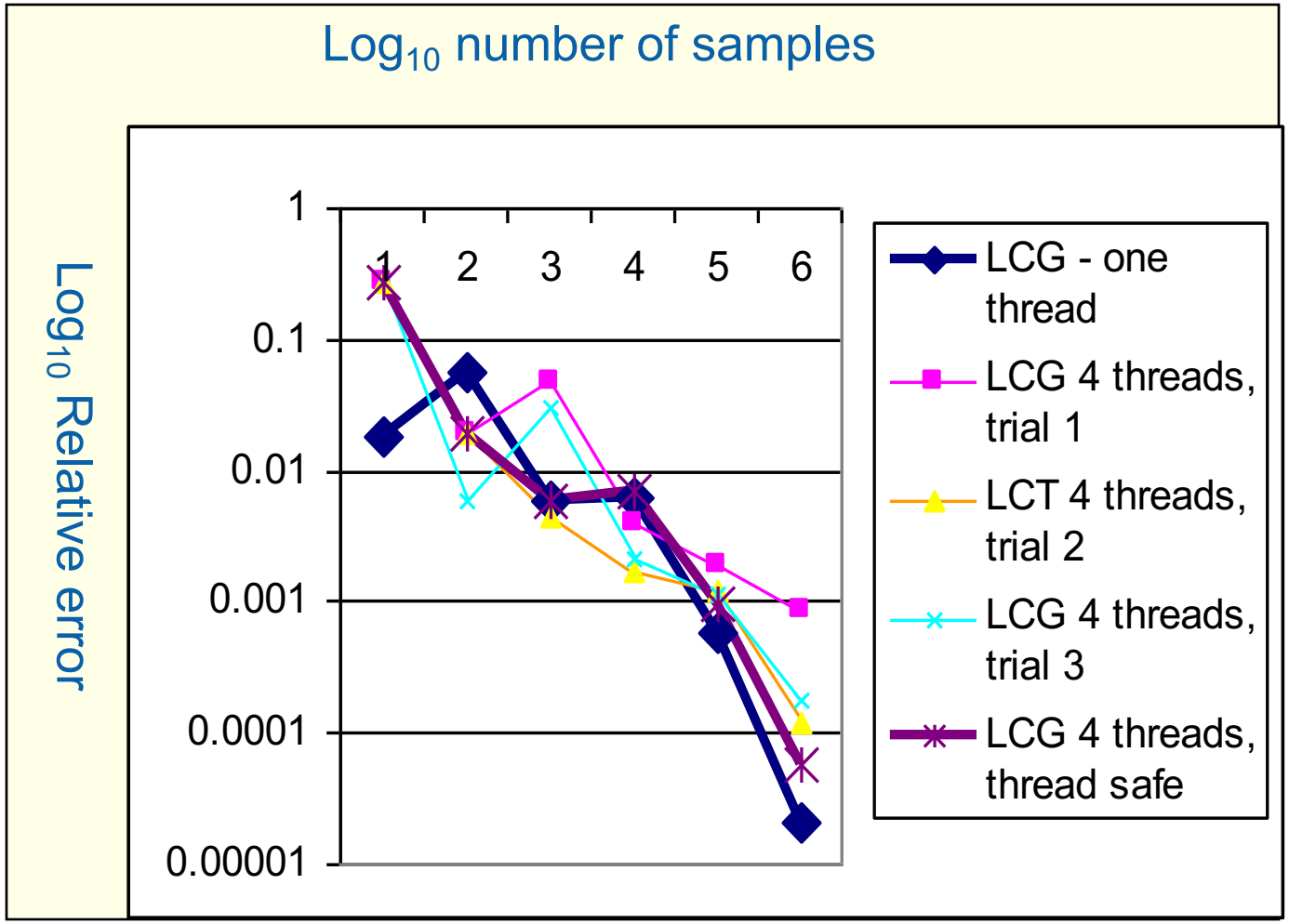
LCG code: threadsafe version

```
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.

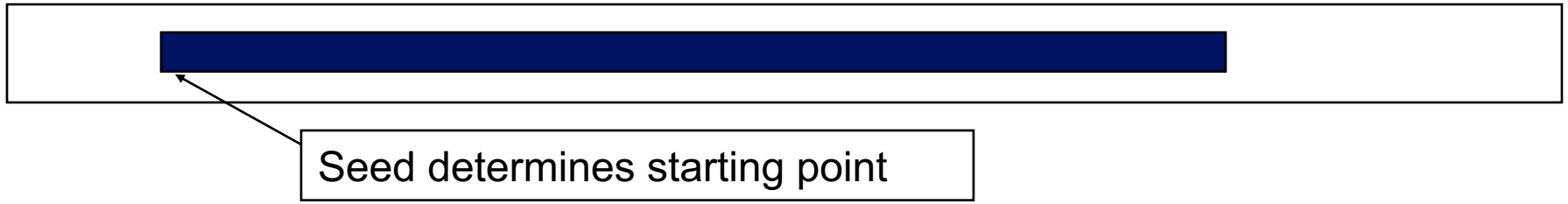
But for large number of samples, its quality is lower than the one thread result!

Why?

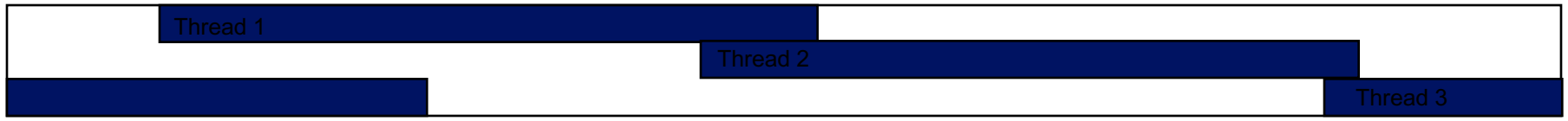
Pseudo Random Sequences

- 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.

MKL Random Number Generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Initialize a stream or pseudo random numbers

Select type of RNG and set seed

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

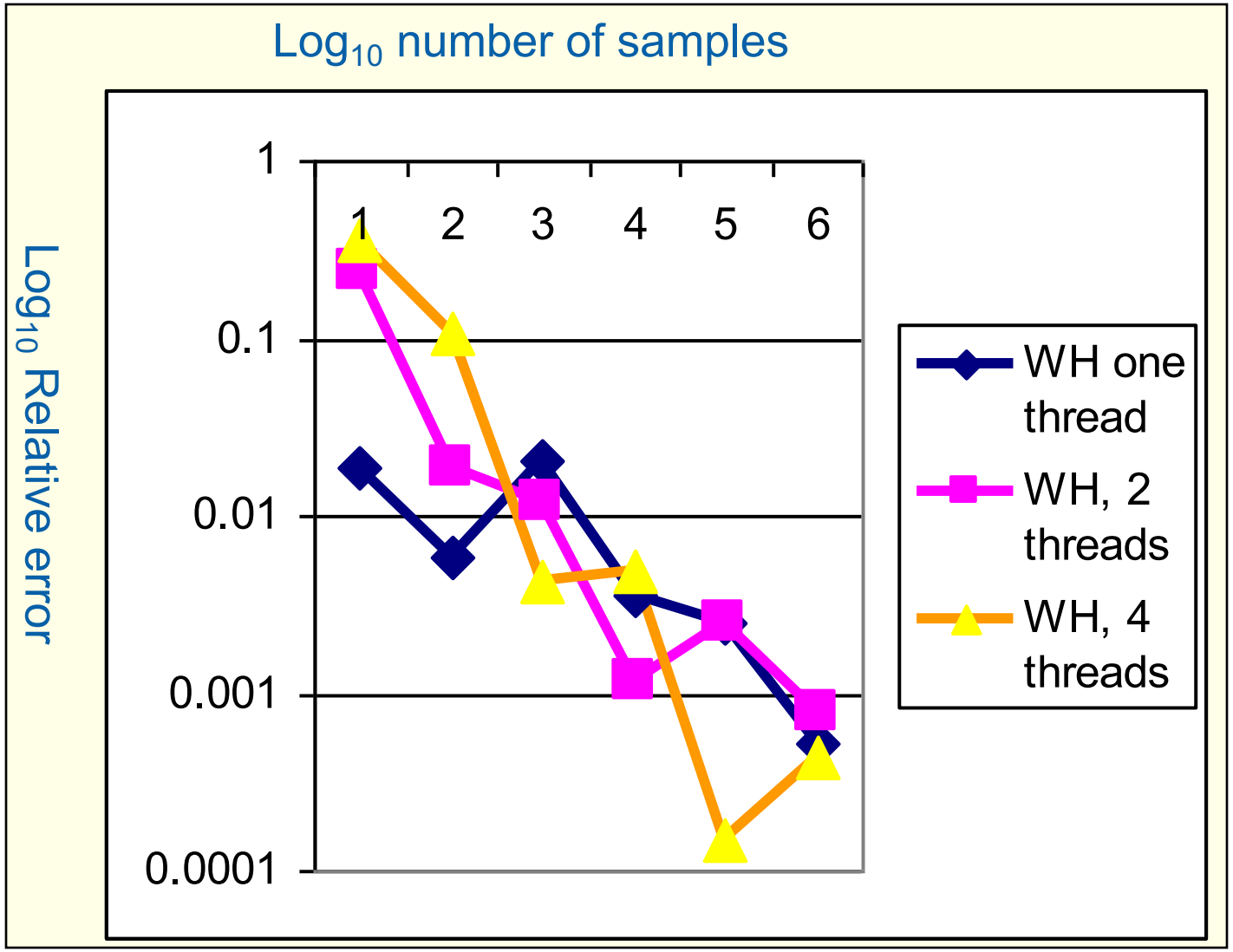
Delete the stream when you are done

Wichmann-Hill Generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                                …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

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.

This is all we have time for concerning OpenMP beyond the Common Core

Conclusion/Summary

- OpenMP is the most popular parallel programming model in use today thanks to the fact that multicore systems are ubiquitous.
  - By far, most people just use the common-core (primarily parallel for) and pretend their systems are SMP.

- In reality, all processors are NUMA systems.
  - Remember the TLB and the need to block even simple memory bound workflows
  - Thread placement and affinity are crucial for getting the best performance.
  - Initialize data on the same processing elements that will do the computing (i.e., consider first touch)

- GPUs are powerful and fun to program …. But please do it portably
  - OpenMP lets you write GPU code that can run ”anywhere”
  - You need to use all the parallelism available on a node. Only OpenMP gives you portable programming that mixes CPU and GPU parallelism.

- Most programmers assume a simple memory model with collective synchronization only.
  - To cover the full range of synchronization problems and to build concurrent data structures, you need to understand OpenMP’s detailed memory model and how to use atomics to build custom synchronization protocols