

OpenMP: It's more than “parallel for” on SMP systems



Tim Mattson, Human Learning Group

An brief aside about the origins of parallel programming models

The Bad Old days ...

Parallel programming environments in the 90's

ABCPL	CORRELATE	GLU	Mentat	Parafraise2	
ACE	CPS	GUARD	Legion	Paralation	pC++
ACT++	CRL	HASL.	Meta Chaos	Parallel-C++	SCHEDULE
Active messages	CSP	Haskell	Midway	Parallaxis	SciTL
Adl	Cthreads	HPC++	Millipede	ParC	POET
Adsmith	CUMULVS	JAVAR.	CparPar	ParLib++	SDDA.
ADDAP	DAGGER	MIRUS	Mirage	ParLin	SHMEM
AFAPI	DAPPLE	HPC	MpC	Parmacs	SIMPLE
ALWAN	Data Parallel C	HPF	MOSIX	Parti	Sina
AM	DC++	IMPACT	Modula-P	pC	SISAL.
AMDC	DCE++	ISIS.	Modula-2*	pC++	distributed smalltalk
AppLeS	DDD	JAVAR	Multipol	PCN	SMI.
Amoeba	DICE.	JADE	MPI	PCP:	SONiC
ARTS	DIPC	Java RMI	MPC++	PH	Split-C.
Athapascan-0b	DOLIB	javaPG	Munin	PEACE	SR
Aurora	DOME	JavaSpace	Nano-Threads	PCU	Sthreads
Automap	DOSMOS.	JIDL	NESL	PET	Strand.
bb_threads	DRL	Joyce	NetClasses++	PETSc	SUIF.
Blaze	DSM-Threads	Khoros	Nexus	PENNY	Synergy
BSP	Ease .	Karma	Nimrod	Phosphorus	Telegrphos
BlockComm	ECO	KOAN/Fortran-S	NOW	POET.	SuperPascal
C*.	Eiffel	LAM	Objective Linda	Polaris	TCGMSG.
"C* in C	Eilean	Lilac	Occam	POOMA	Threads.h++.
C**	Emerald	Linda	Omega	POOL-T	TreadMarks
CarlOS	EPL	JADA	OpenMP	PRESTO	TRAPPER
Cashmere	Excalibur	WWWinda	Orca	P-RIO	uC++
C4	Express	ISETL-Linda	OOF90	Prospero	UNITY
CC++	Falcon	ParLin	P++	Proteus	UC
Chu	Filaments	Eilean	P3L	QPC++	V
Charlotte	FM	P4-Linda	p4-Linda	PVM	ViC*
Charm	FLASH	Glenda	Pablo	PSI	Visifold V-NUS
Charm++	The FORCE	POSYBL	PADE	PSDM	VPE
Cid	Fork	Objective-Linda	PADRE	Quake	Win32 threads
Cilk	Fortran-M	LiPS	Panda	Quark	WinPar
CM-Fortran	FX	Locust	Papers	Quick Threads	WWWinda
Converse	GA	Lparx	AFAPL	Sage++	XENOOPS
Code	GAMMA	Lucid	Para++	SCANDAL	XPC
COOL	Glenda	Maisie	Paradigm	SAM	Zounds
		Manifold			ZPL

A combination of vendors wanting to lock-down applications and the fact that its "fun" to create new programming environments created chaos for application developers

History of MPI

Workstation vendors wanted into the HPC market

PVM was great but didn't support quality SW engineering

Hardware:

By the early 90's, massively parallel processors (MPPs) and the new trend with clusters convinced even the skeptics that the "killer micros" had won.

MPP Vendors

Needed a common foundation to build a parallel SW industry

User Community

Fed-up recoding as they moved between platforms

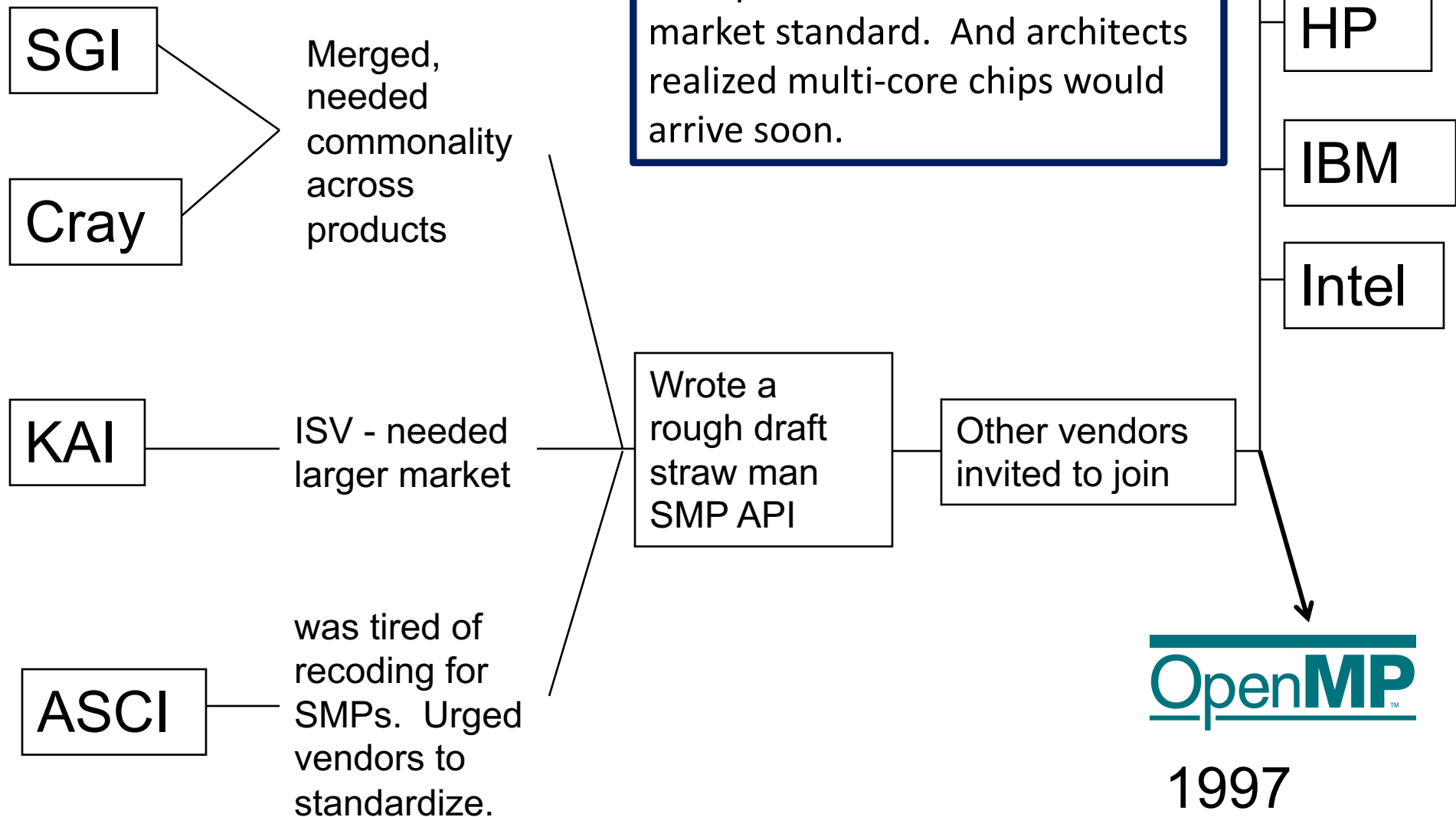
After several years of informal discussions, the MPI forum was created in 1992. A draft specification was presented one year later at SC'93.



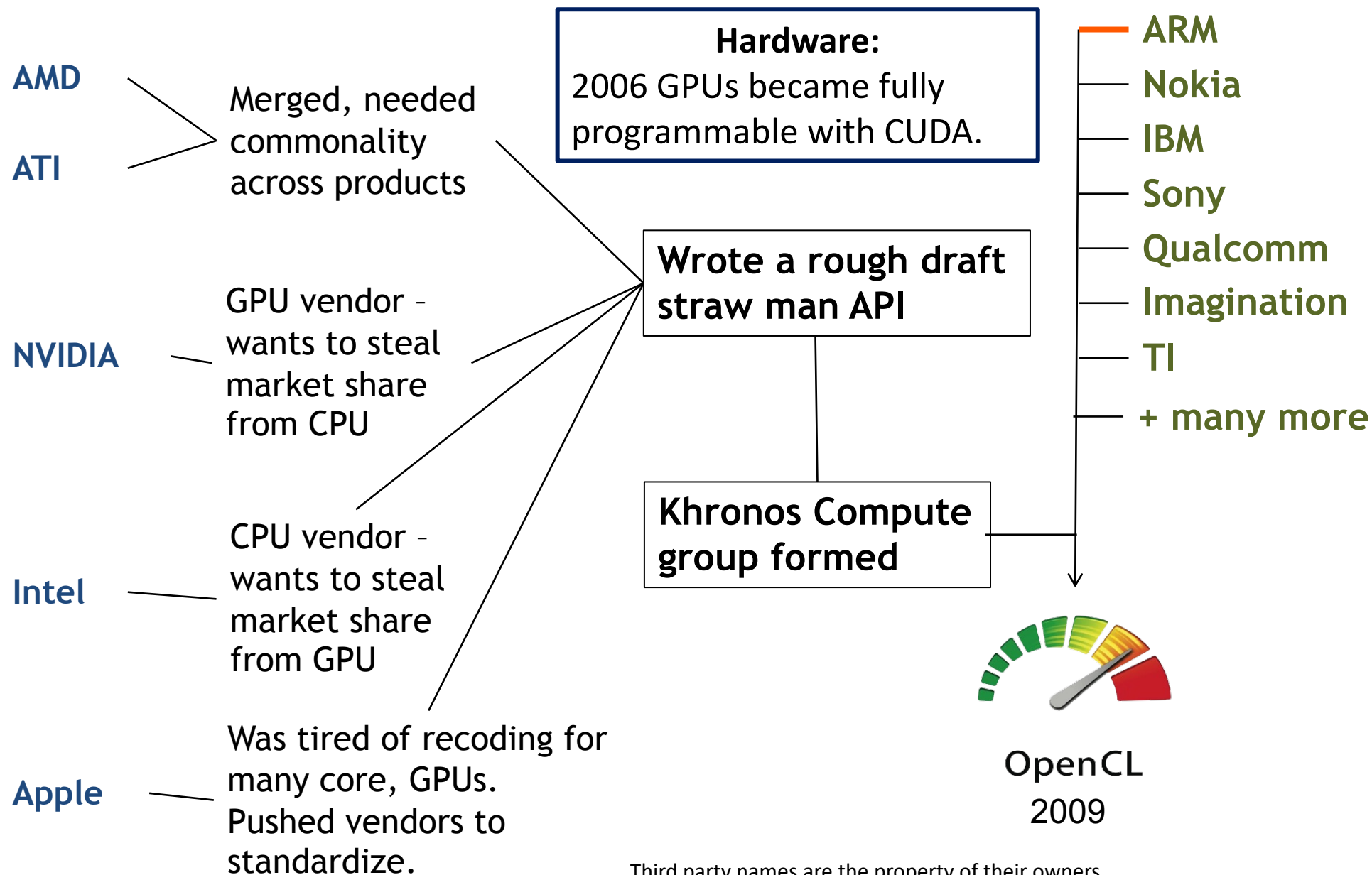
1994

Many of us worked in the MPI forum ... leadership came from the DOE National Labs. In particular, the reference implementation from Bill Gropp and Rusty Lusk of Argonne national lab called MPIch helped us get it right in the 1.0 specification and made sure a working implementation of the standard was available right from the beginning.

History of OpenMP



The origins of OpenCL

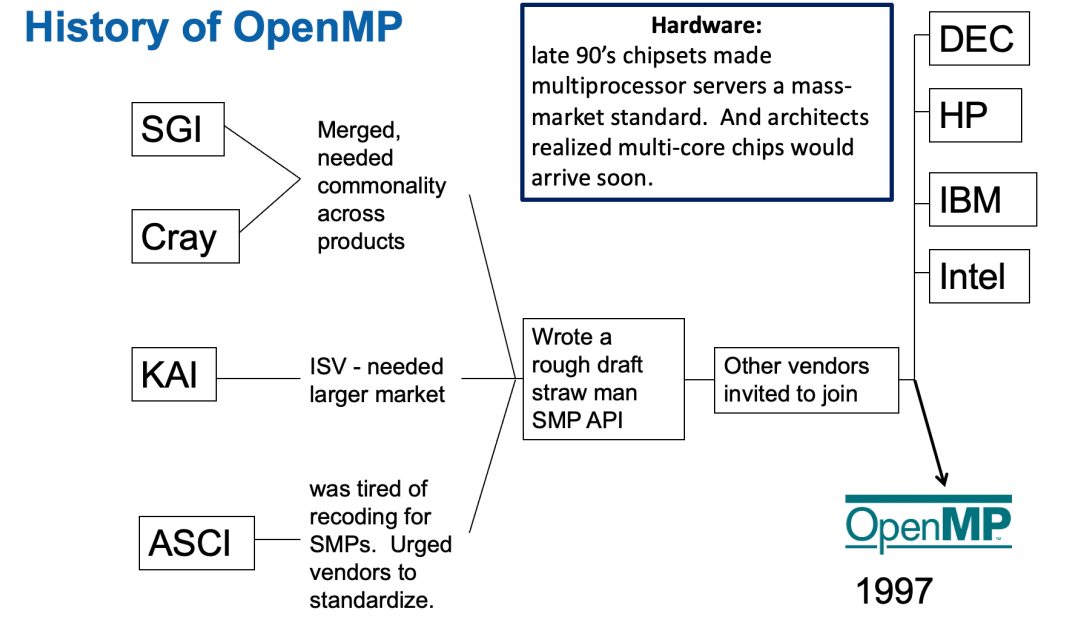


Third party names are the property of their owners.

Key Lesson for you

- Application developers in HPC are the community that really matters You have MUCH MORE power than you think, but only:
 - You are willing to speak with One Voice.
 - You refuse to be trapped in a walled Garden.
 - Vendors will try to divide and conquer you. Hold firm or all is lost!

History of OpenMP



History of MPI

Workstation vendors wanted into the HPC market

PVM was great but didn't support quality SW engineering

Hardware:

By the early 90's, massively parallel processors (MPPs) and the new trend with clusters convinced even the skeptics that the "killer micros" had won.

MPP Vendors

Needed a common foundation to build a parallel SW industry

User Community

Fed-up recoding as they moved between platforms

After several years of informal discussions, the MPI forum was created in 1992. A draft specification was presented one year later at SC'93.



1994

Many of us worked in the MPI forum ... leadership came from the DOE National Labs. In particular, the reference implementation from Bill Gropp and Rusty Lusk of Argonne national lab called MPICH helped us get it right in the 1.0 specification and made sure a working implementation of the standard was available right from the beginning.

The Bad Old days ...

Parallel programming environments in the 90's

ABCPL	CORRELATE	GLU	Mentat	Parafuse2	pC++
ACE	CPS	GUARD	Legion	Paralation	SCHEDULE
ACT++	CRL	HASL	Meta Chaos	Parallel-C++	SciTL
Active messages	CSP	Haskell	Midway	Parallaxis	ScTL
Adl	Cthreads	HPC++	Millipede	ParC	SHMEM
Adsmith	CUMULVS	JAVAR	CparPar	ParLib++	SIMPLE
ADDAP	DAGGER	FORUS	Mirage	ParLin	Sina
AFAP	DAPPLE	HPC	MpC	Parmas	SISAL
ALWAN	Data Parallel C	HPF	MOSIX	Parti	distributed smalltalk
AM	DC++	IMPACT	Modula-P	pC	SML
AMDC	DCE++	ISIS	Modula-2*	pC++	SONIC
AppLeS	DDD	JAVAR	Multipol	PCN	Split-C
Amoeba	DICE	JADE	MPI	PCP	SR
ARTS	DIPC	Java RMI	MPC++	PH	Threads
Athapascan-0b	DOLIB	javaPG	Munin	PEACE	Strand
Aurora	DOME	JavaSpace	Nano-Threads	PCU	SUIF
Automap	DOSMOS	JIDL	JOYCE	PET	Synergy
bb_threads	DRL	Joyce	Khoros	PETSc	SuperPascal
Blaze	DSM-Threads	Khoros		PENNY	TCGMSG
BSP					Threads.h++
BlockComm					TreadMarks
C*					TRAPPER
"C*" in C					
C**					
CarlOS					
Cashmere					
C4					
CC++					
Chu					
Charlotte					
Charm					
Charm++					
Cid					
Cilk					
CM-Fortran					
Converse					
Code					
COOL					

Please don't let this happen again

**EVERYTIME YOU USE CUDA OR
OPENACC**



**GOD KILLS A
KITTEN**

**... Back to our previously scheduled
program**

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



Tim Mattson, University of Bristol

Preliminaries

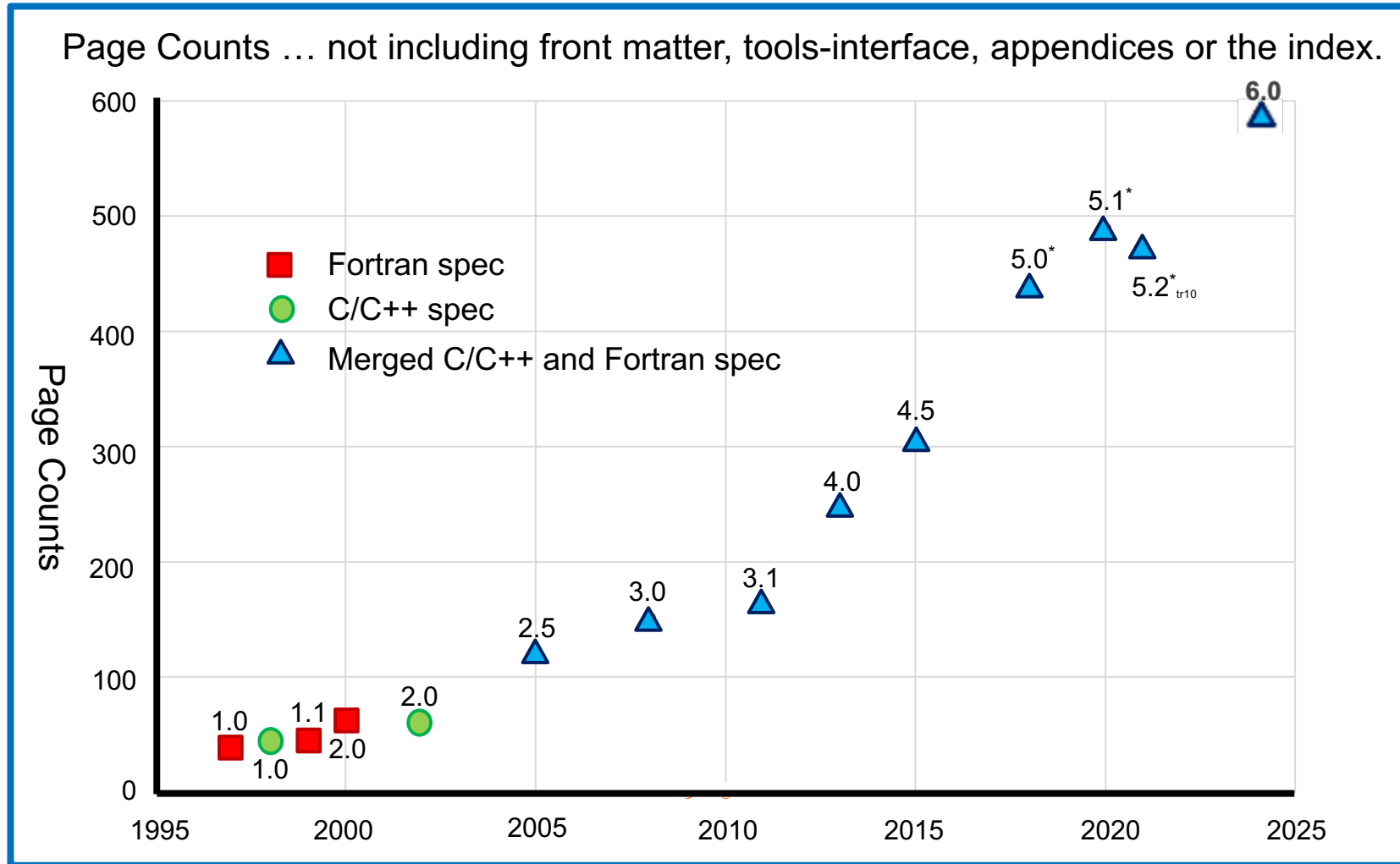
- Our plan for the day .. Active learning!
 - We will mix short lectures with short exercises.
 - You will use your laptop to connect to a multiprocessor server.
- Please follow these simple rules
 - Do the exercises that we assign and then change things around and experiment.
 - Embrace active learning!
 - Don't cheat: Do Not look at the solutions before you complete an exercise ... even if you get really frustrated.

Download tutorial materials onto your laptop:
git clone <https://github.com/tgmattso/ATPESC.git>

- ➡ • A Very Brief OpenMP Review
 - Irregular Parallelism and Tasks
 - OpenMP and Real Hardware: Dealing with NUMA
 - Synchronization and the OpenMP Memory Model
 - GPU Programming with OpenMP
 - Recap

The Growth of Complexity in OpenMP

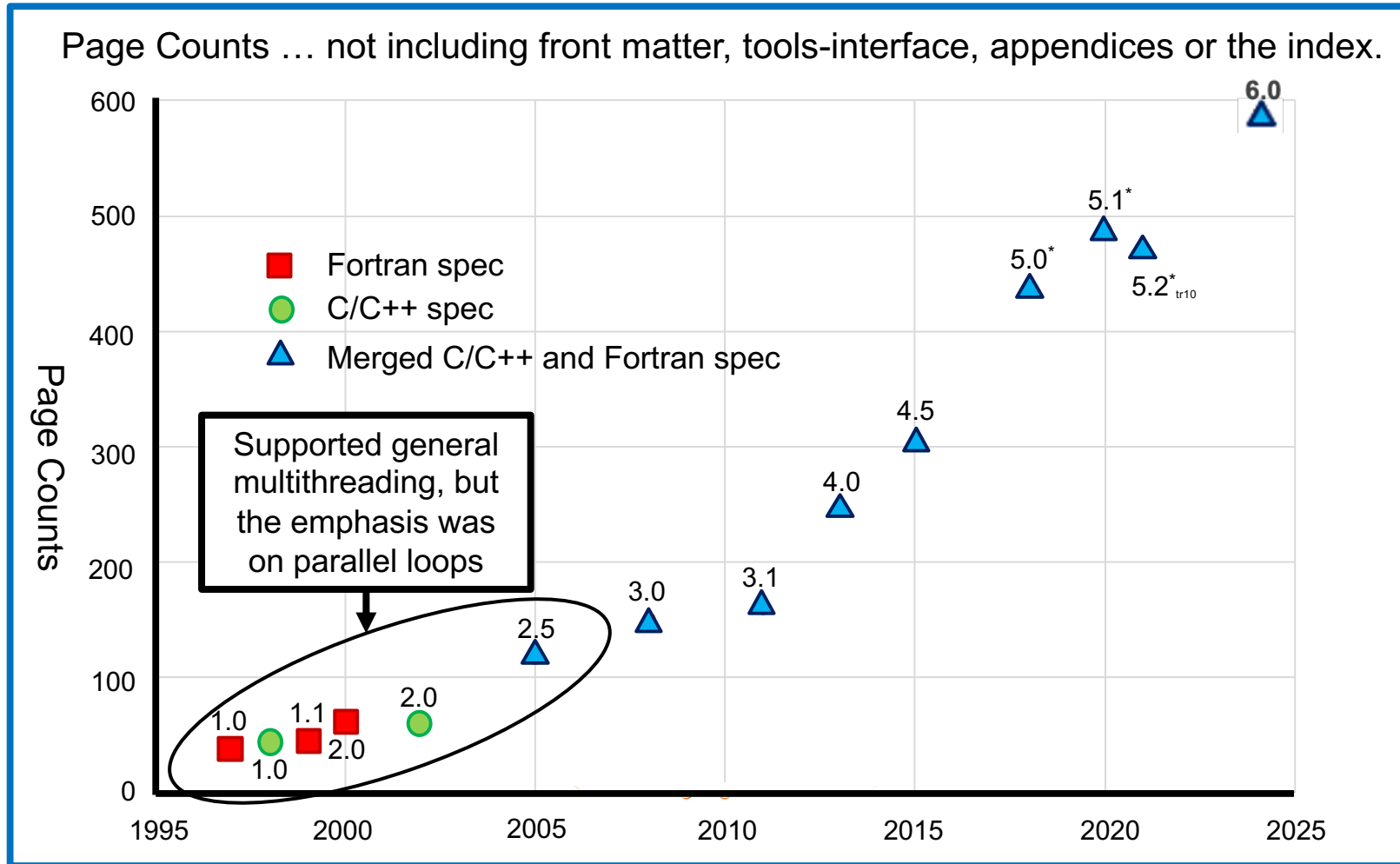
Our goal in 1997 ... A simple interface for application programmers



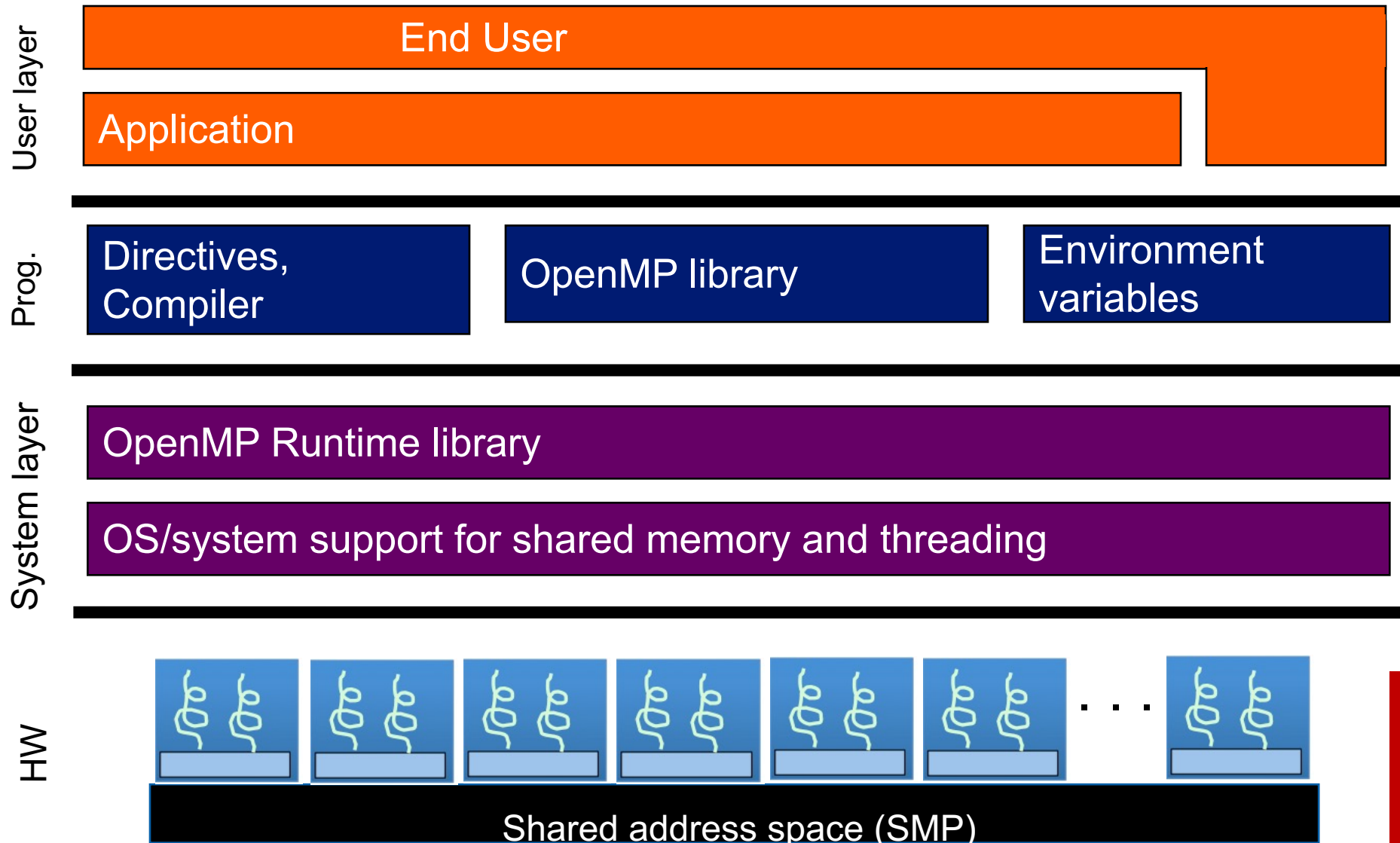
The OpenMP specification is so long and complex that few (if any) humans understand the full document

The Growth of Complexity in OpenMP

Our goal in 1997 ... A simple interface for application programmers



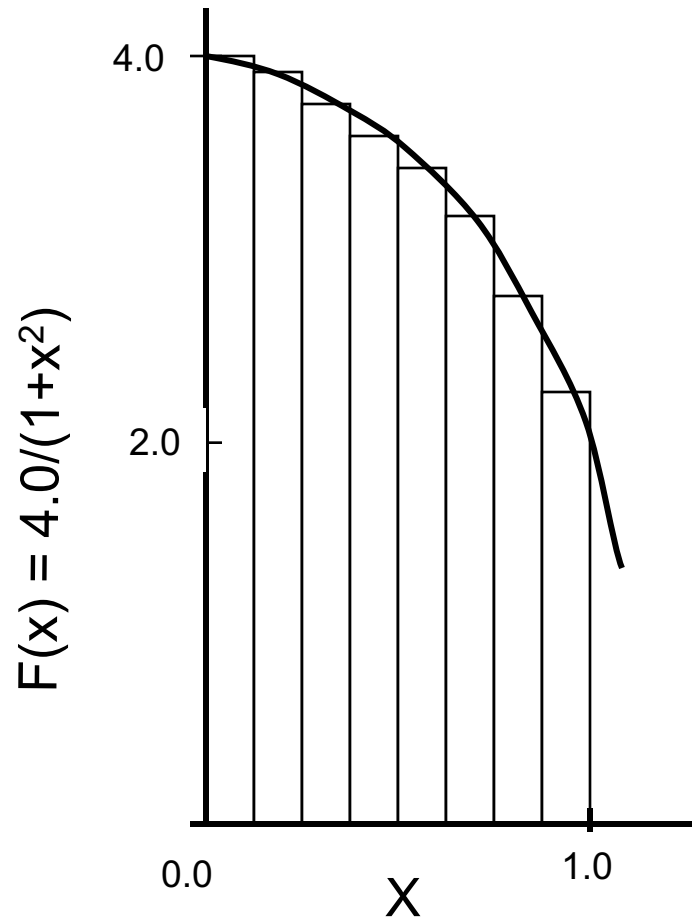
OpenMP Basic Definitions: Basic Solution Stack



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

A tour of OpenMP: Various flavors of the pi program

Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of N rectangles:

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

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

Serial PI Program

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

    step = 1.0/(double) num_steps;

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

Serial PI Program

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

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

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

Exercise: the Parallel Pi Program

- Create a parallel version of the pi program using anything you want from OpenMP:
 - OpenMP directives
 - #pragma omp parallel
 - #pragma omp for
 - #pragma omp barrier
 - #pragma omp critical
 - Runtime library routines
 - #include <omp.h>
 - omp_set_num_threads();
 - int omp_get_num_threads();
 - int omp_get_thread_num();
 - double omp_get_wtime();
 - Common clauses on OpenMP directives
 - private(list), shared(list), firstprivate(list)
 - reduction(+:list)
 - nowait

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

    step = 1.0/(double) num_steps;

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

Splitting up work between threads explicitly: block distribution

```
#include <omp.h>
```

```
static long num_steps = 100000;      double step;
```

```
#define NUM_THREADS 2
```

```
void main ()
```

```
{  int i, nthreads; double pi, sum[NUM_THREADS];
```

```
    step = 1.0/(double) num_steps;
```

```
    omp_set_num_threads(NUM_THREADS);
```

```
    #pragma omp parallel
```

```
    {
```

```
        int i, id, numthrds, istart, iend;
```

```
        double x;
```

```
        id = omp_get_thread_num();
```

```
        numthrds = omp_get_num_threads();
```

```
        istart = id*(num_steps/numthrds );    iend=(id+1)*(num_steps/numthrds);
```

```
        if(id == (numthrds-1)) iend = num_steps;
```

```
        if (id == 0)  nthreads = numthrds;
```

```
        for (i=istart, sum[id]=0.0; i< iend; i++) {
```

```
            x = (i+0.5)*step;
```

```
            sum[id] += 4.0/(1.0+x*x);
```

```
        }
```

```
    }
```

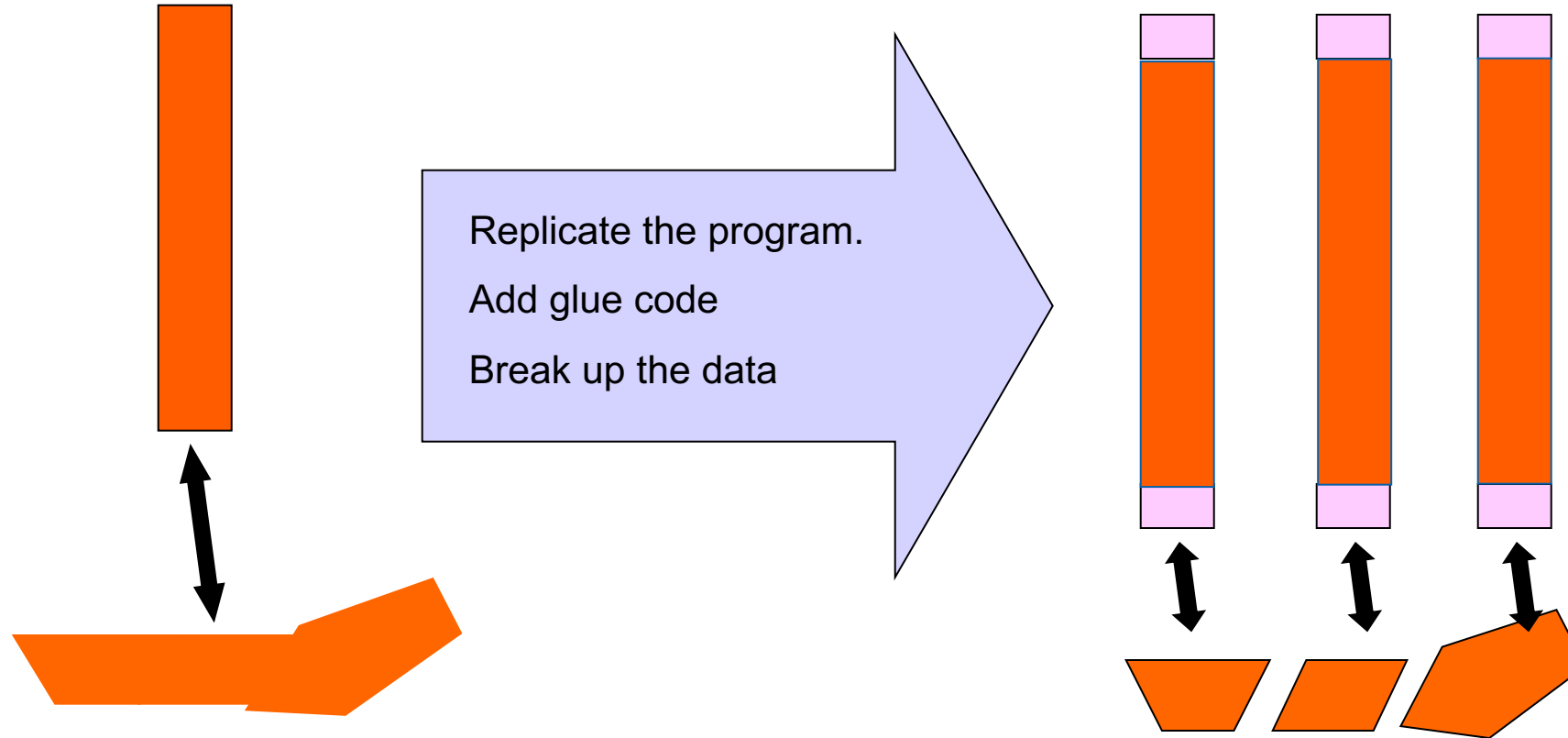
```
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
```

```
}
```

This is a common trick in SPMD algorithms ... it's a **blocked distribution** with one block per thread.

SPMD: Single Program Multiple Data

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



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

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

Splitting up work between threads explicitly: cyclic distribution

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

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

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

Example: PI with a loop and a reduction

```
#include <omp.h>
```

```
void main ()
```

```
{
```

```
    long num_steps = 100000;
```

```
    double pi, sum = 0.0;
```

```
    double step = 1.0/(double) num_steps;
```

```
    #pragma omp parallel for reduction(+:sum)
```

```
    for (int i=0;i< num_steps; i++){
```

```
        double x = (i+0.5)*step;
```

```
        sum = sum + 4.0/(1.0+x*x);
```

```
    }
```

```
    pi = step * sum;
```

```
}
```

Results*: OpenMP and the pi program

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

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

threads	SPMD critical	PI Loop
1	1.87	1.91
2	1.00	1.02
3	0.68	0.80
4	0.53	0.68

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

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

*Intel compiler (icpc) with -O0 on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Array Sections with Reduce

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

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

```
#pragma omp parallel for 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;
```

Loop Worksharing Constructs: The schedule clause

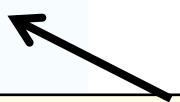
- The schedule clause affects how loop iterations are mapped onto threads
 - **schedule(static [,chunk])**
 - Deal-out blocks of iterations of size “chunk” to each thread.
 - **schedule(dynamic[,chunk])**
 - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
 - **schedule(guided[,chunk])**
 - 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.

Nested Loops

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

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



Number of loops
to be
parallelized,
counting from
the outside

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

Data Sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses (note: *list* is a comma-separated list of variables)
 - **shared(list)**
 - **private(list)**
 - **firstprivate(list)**
- These can be used on **parallel** and **for** constructs ... other than **shared** which can only be used on a **parallel** construct
- Force the programmer to explicitly define storage attributes
 - **default (none)**

default() can only be used
on parallel constructs

Sections Worksharing Construct

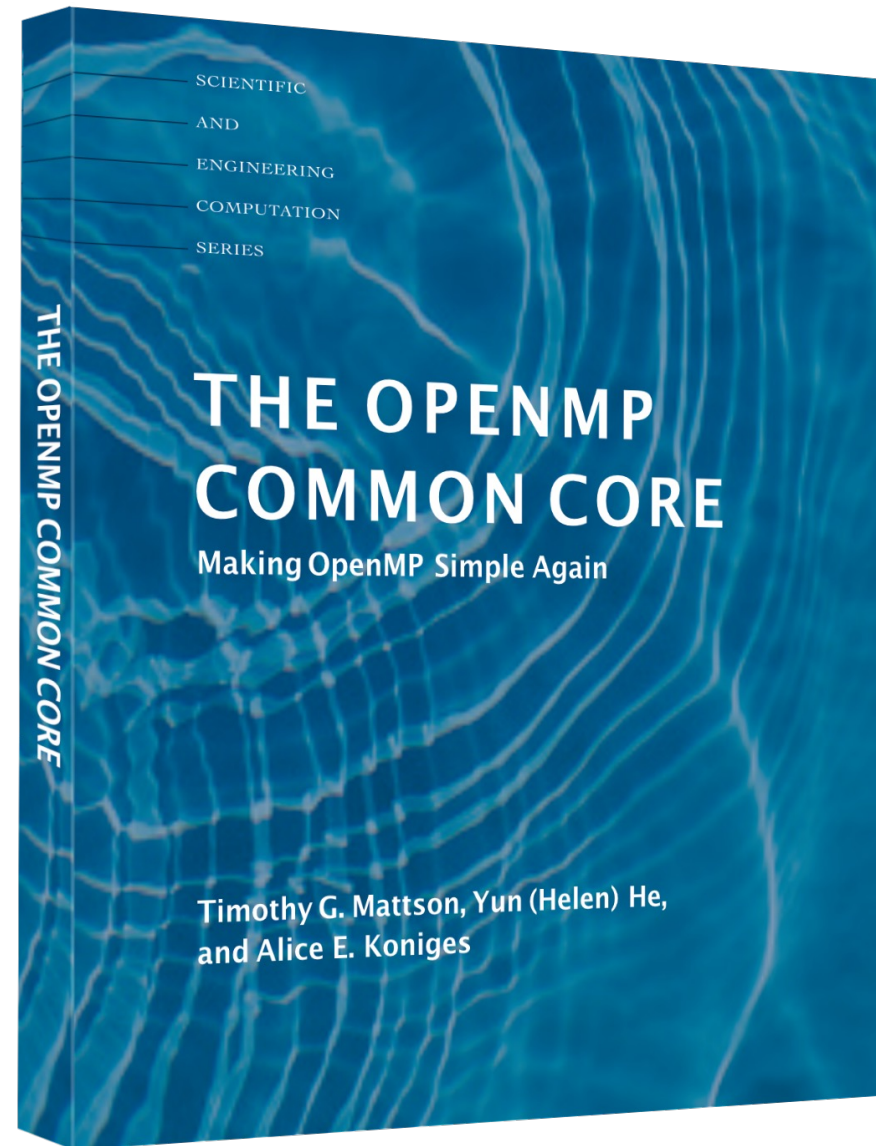
- The *Sections* worksharing construct gives a different structured block to each thread.

```
#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.

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use
- It's geared towards people learning OpenMP, but as one commentator put it ... **everyone at any skill level should read the memory model chapters.**
- Available from MIT Press



- A Very Brief OpenMP Review

- ➔ • Irregular Parallelism and Tasks

**Lets go beyond programs
based on simple parallel
for constructs**

- OpenMP and Real Hardware: Dealing with NUMA
- Synchronization and the OpenMP Memory Model
- GPU Programming with OpenMP
- Recap

Irregular Parallelism

- Let's call a problem “irregular” when one or both of the following hold:
 - Data Structures are sparse or involve indirect memory references
 - Control structures are not basic for-loops
- Example: Traversing Linked lists:

```
p = listhead ;  
while (p) {  
    process(p) ;  
    p=p->next;  
}
```

- Using what we've learned so far, traversing a linked list in parallel using OpenMP is difficult.

Exercise: Traversing linked lists

- Consider the program linked.c
 - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program selecting from the following list of constructs:

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
schedule(static[,chunk]) or schedule(dynamic[,chunk])
private(), firstprivate(), default(none)
```

- Hint: Just worry about the while loop that is timed inside main(). You don't need to make any changes to the “list functions”

Linked Lists with OpenMP (without tasks)

- See the file solutions/linked_notasks.c

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}  
struct node *parr = (struct node*) malloc(count*sizeof(struct node));  
p = head;  
for(i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}  
#pragma omp parallel  
{  
    #pragma omp for schedule(static,1)  
    for(i=0; i<count; i++)  
        processwork(parr[i]);  
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Number of threads	Schedule	
	Default	Static,1
1	48 seconds	45 seconds
2	39 seconds	28 seconds

Linked Lists with OpenMP (without tasks)

- See the file solutions/linked_notasks.c

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}  
struct node *parr = (struct node*) malloc(count*sizeof(struct node));  
p = head;  
for(i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}  
#pragma omp parallel  
{  
    #pragma omp for schedule(static,1)  
    for(i=0; i<count; i++)  
        processwork(parr[i]);  
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Number of threads	Schedule	
	Default	Static,1
1	48 seconds	45 seconds
2	39 seconds	28 seconds

With so much code to add and three passes through the data, this is really ugly.

There has got to be a better way to do this

Solutions from other people

- A much more elegant solution based on the SPMD pattern

```
#pragma omp parallel firstprivate(p)
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int count = 0;
    while(p!=NULL && count < id){
        count ++;
        p = p->next;
    }
    while (p != NULL) {
        processwork(p);
        int count = 0;
        while(p!=NULL && count < nthreads){
            count ++;
            p = p->next;
        }
    }
}
```

Each thread has its own copy of the list pointer set to the head of the list

Each thread advances its list pointer by a number equal to the thread ID

Increment the thread's list pointer by the number of threads

Solutions from other people

- A particularly elegant, SPMD solution

```
#pragma omp parallel firstprivate(p)
{
    int i = 0;
    int thread_num = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    while (p != NULL)
    {
        if(i % num_threads == thread_num)
            processwork(p);
        p = p->next;
        ++i;
    }
}
```

Each thread has its own copy of the list pointer set to the head of the list

Each of the threads traverses the list, but only one thread processes the work for any give node using the modulus of the count with the number of threads

Linked Lists with OpenMP

```
while (p != NULL) {
    p = p->next;
    count++;
}
struct node *parr = (struct node*) malloc(count*sizeof(struct
node));
p = head;
for(i=0; i<count; i++) {
    parr[i] = p;
    p = p->next;
}
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

```
#pragma omp parallel firstprivate(p) default(none)
{
    int i = 0;
    int thread_num = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    while (p != NULL)
    {
        if(i % num_threads == thread_num)
            processwork(p);
        p = p->next;
        ++i;
    }
}
```

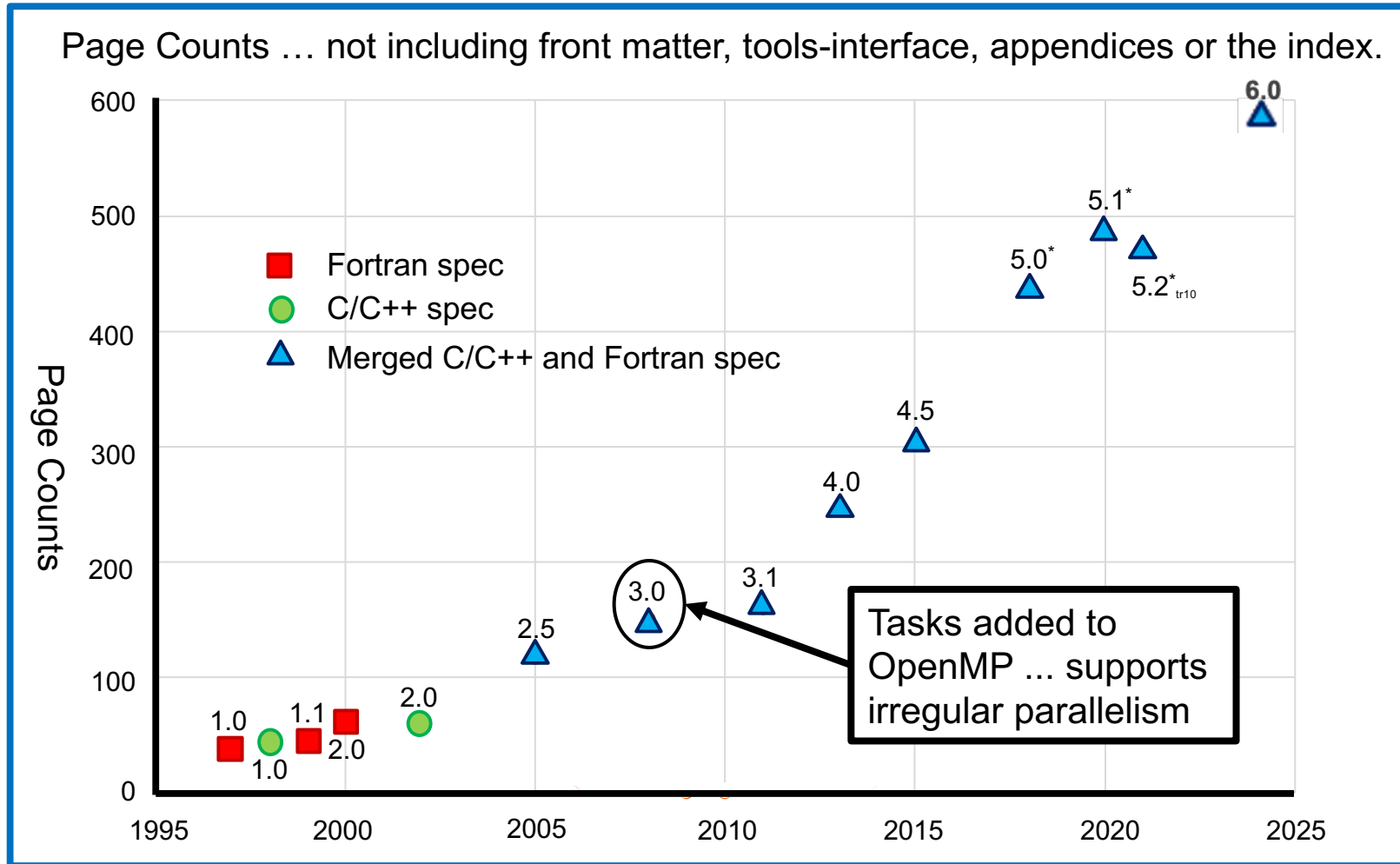
```
#pragma omp parallel firstprivate(p)
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int count = 0;
    while(p!=NULL && count < id){
        count ++;
        p = p->next;
    }
    while (p != NULL) {
        processwork(p);
        int count = 0;
        while(p!=NULL && count < nthreads){
            count ++;
            p = p->next;
        }
    }
}
```

With so much code to add and multiple passes through the data, all of these approaches are kind of ugly.

There has got to be a better way to handle irregular problems

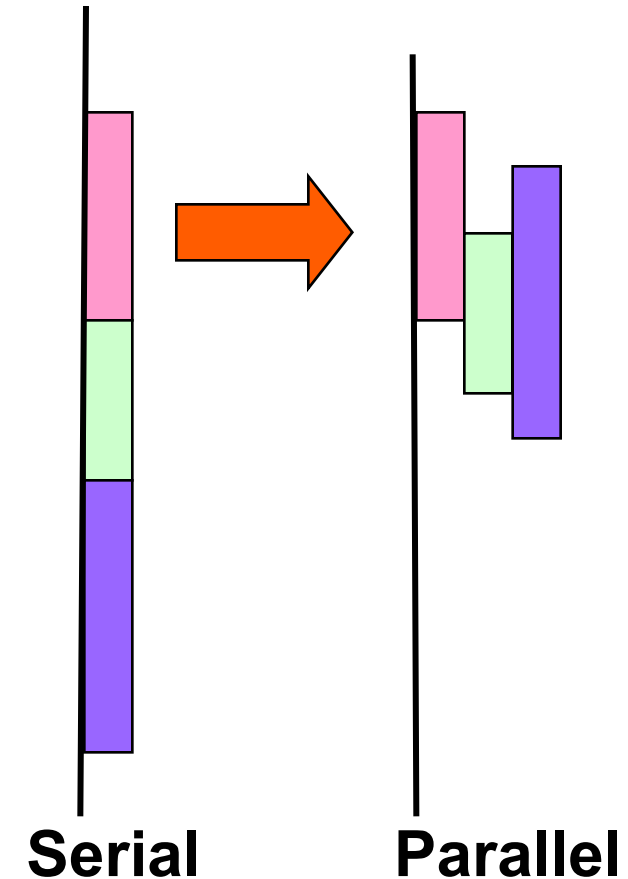
The Growth of Complexity in OpenMP

Our goal in 1997 ... A simple interface for application programmers



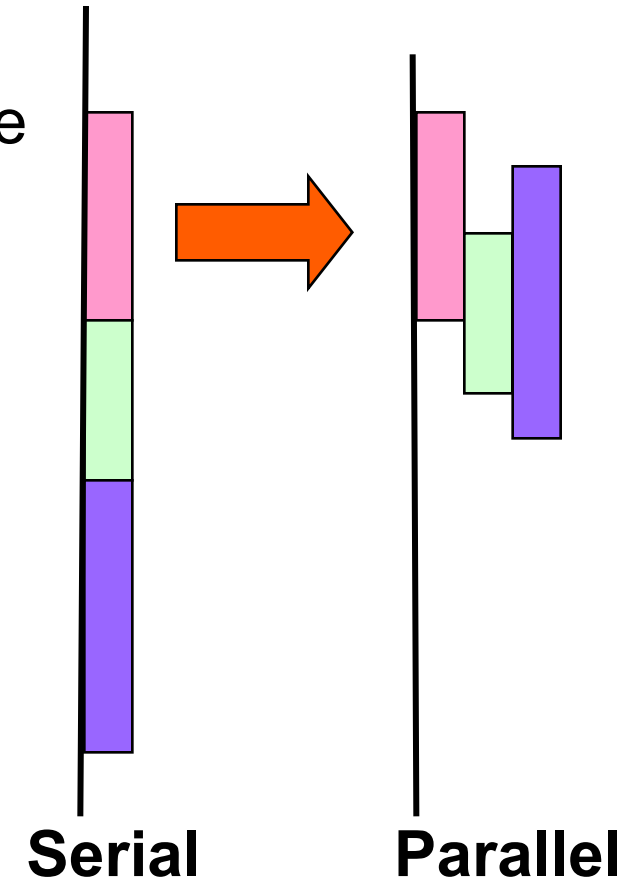
What are Tasks?

- Tasks are independent units of work
- Tasks are composed of:
 - code to execute
 - data to compute with
- Threads are assigned to perform the work of each task.
 - The thread that encounters the task construct may execute the task immediately.
 - The threads may defer execution until later



What are Tasks?

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e., a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

Single Worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the primary* thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {   exchange_boundaries();   }
    do_many_other_things();
}
```

*This used to be called the “master thread”. The term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

Task Directive

`#pragma omp task [clauses]`
structured-block

```
#pragma omp parallel
```

Create some threads

```
{
```

```
    #pragma omp single
```

One Thread
packages tasks

```
{
```

```
    #pragma omp task  
        fred();
```

```
    #pragma omp task  
        daisy();
```

```
    #pragma omp task  
        billy();
```

Tasks executed by
some thread in some
order

```
}
```

All tasks complete before this barrier is released

```
}
```

Data Scoping with Tasks

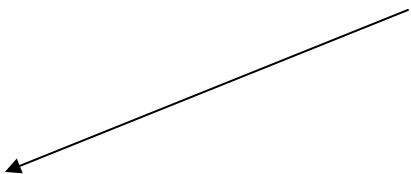
- Variables can be shared, private or firstprivate with respect to a task
- These concepts are a little bit different compared with threads:
 - If a variable is **shared** on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
 - If a variable is **private** on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
 - If a variable is **firstprivate** on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data Scoping Defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
 - Variables that are private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private



Exercise: Traversing linked lists

- Consider the program linked.c
 - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program selecting from the following list of constructs:

```
#pragma omp parallel
#pragma omp single
#pragma omp task
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
private(), firstprivate()
```

- Hint: Just worry about the contents of main(). You don't need to make any changes to the “list functions”

Parallel Linked List Traversal

```
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead ;
        while (p) {
            #pragma omp task firstprivate(p)
            {
                process (p) ;
            }
            p=next (p) ;
        }
    }
}
```

Only one thread
packages tasks

makes a copy of **p**
when the task is
packaged

When/Where are Tasks Complete?

- At thread barriers (explicit or implicit)
 - all tasks generated inside a region must complete at the next barrier encountered by the threads in that region. Common examples:
 - **Tasks generated inside a single construct:** all tasks complete before exiting the barrier on the single.
 - **Tasks generated inside a parallel region:** all tasks complete before exiting the barrier at the end of the parallel region.
- At taskwait directive
 - i.e. Wait until all tasks defined in the current task have completed.
`#pragma omp taskwait`
 - Note: applies only to tasks generated in the current task, not to “descendants” of those tasks.

Example

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma omp taskwait
        #pragma omp task
        billy();
    }
}
```

fred() and **daisy()** must complete before **billy()** starts, but this does not include tasks created inside **fred()** and **daisy()**

All tasks including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma omp taskwait
        #pragma omp task
        billy();
    }
}
```

The barrier at the end of the single is expensive and not needed since you get the barrier at the end of the parallel region.

Use `nowait` to turn it off.

All tasks including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example: Fibonacci numbers

```
int fib (int n)
{
    int x,y;
    if (n < 2) return n;

    x = fib(n-1);
    y = fib (n-2);
    return (x+y);
}
```

```
int main()
{
    int NW = 5000;
    fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(2^n)$ recursive implementation!

Parallel Fibonacci

```
int fib (int n)
{  int x,y;
   if (n < 2) return n;
```

```
  #pragma omp task shared(x)
```

```
    x = fib(n-1);
```

```
  #pragma omp task shared(y)
```

```
    y = fib (n-2);
```

```
  #pragma omp taskwait
```

```
    return (x+y);
```

```
}
```

```
Int main()
```

```
{  int NW = 5000;
```

```
  #pragma omp parallel
```

```
  {
```

```
    #pragma omp single
```

```
      fib(NW);
```

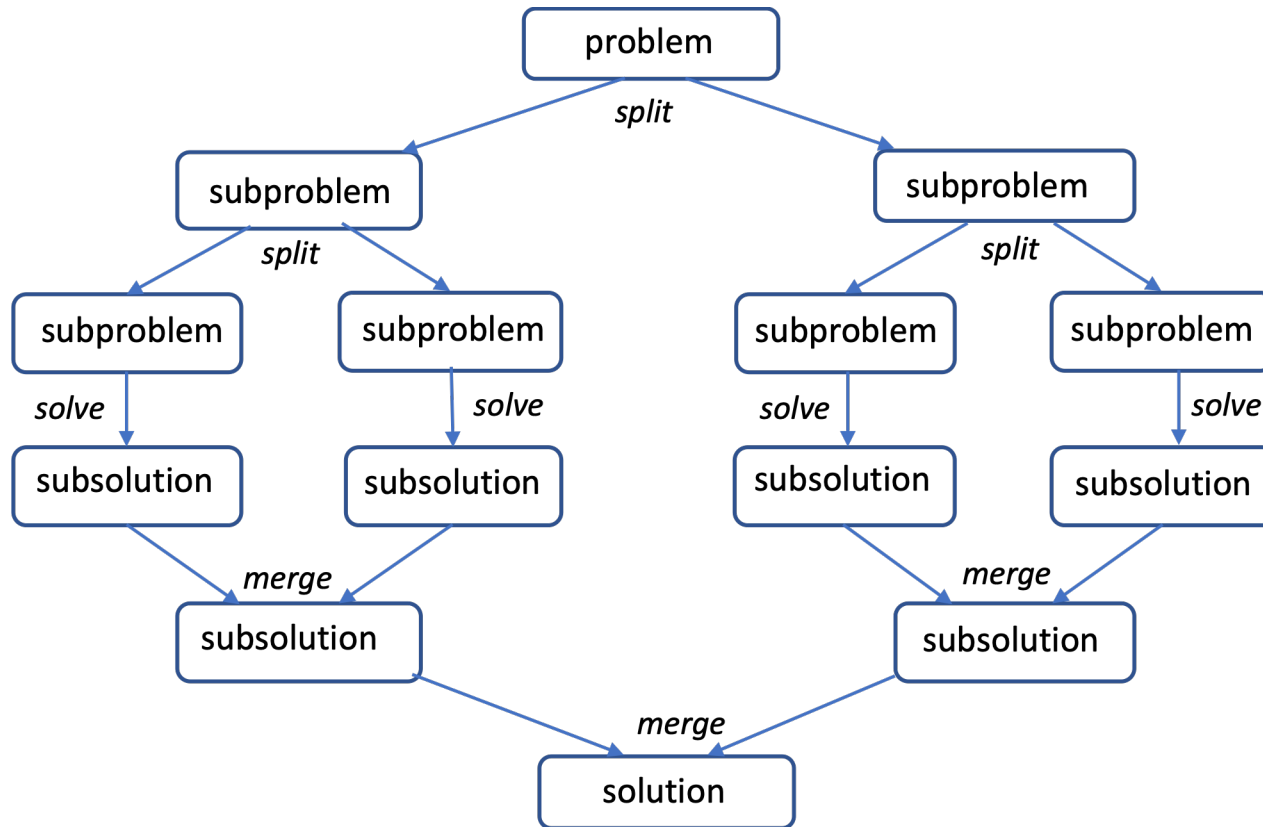
```
  }
```

```
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- **x, y** are local, and so by default they are private to current task
 - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Divide and Conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



- 3 Options for parallelism:
 - Do work as you split into sub-problems
 - Do work only at the leaves
 - Do work as you recombine

Exercise: PI with tasks

- Go back to the original pi.c program
 - Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

- Hint: first create a recursive pi program and verify that it works. Think about the computation you want to do at the leaves. If you go all the way down to one iteration per leaf-node, won't you just swamp the system with tasks?

Program: OpenMP tasks

```
include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK 10000000
double pi_comp(int Nstart,int Nfinish,double step)
{ int i,iblk;
  double x, sum = 0.0,sum1, sum2;
  if (Nfinish-Nstart < MIN_BLK){
    for (i=Nstart;i< Nfinish; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  }
  else{
    iblk = Nfinish-Nstart;
    #pragma omp task shared(sum1)
      sum1 = pi_comp(Nstart,      Nfinish-iblk/2,step);
    #pragma omp task shared(sum2)
      sum2 = pi_comp(Nfinish-iblk/2, Nfinish,      step);
    #pragma omp taskwait
      sum = sum1 + sum2;
  }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
  step = 1.0/(double) num_steps;
  #pragma omp parallel
  {
    #pragma omp single
      sum =
        pi_comp(0,num_steps,step);
  }
  pi = step * sum;
}
```

Results*: OpenMP and the pi program

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

threads	SPMD critical	PI Loop	Pi tasks
1	1.87	1.91	1.87
2	1.00	1.02	1.00
3	0.68	0.80	0.76
4	0.53	0.68	0.52

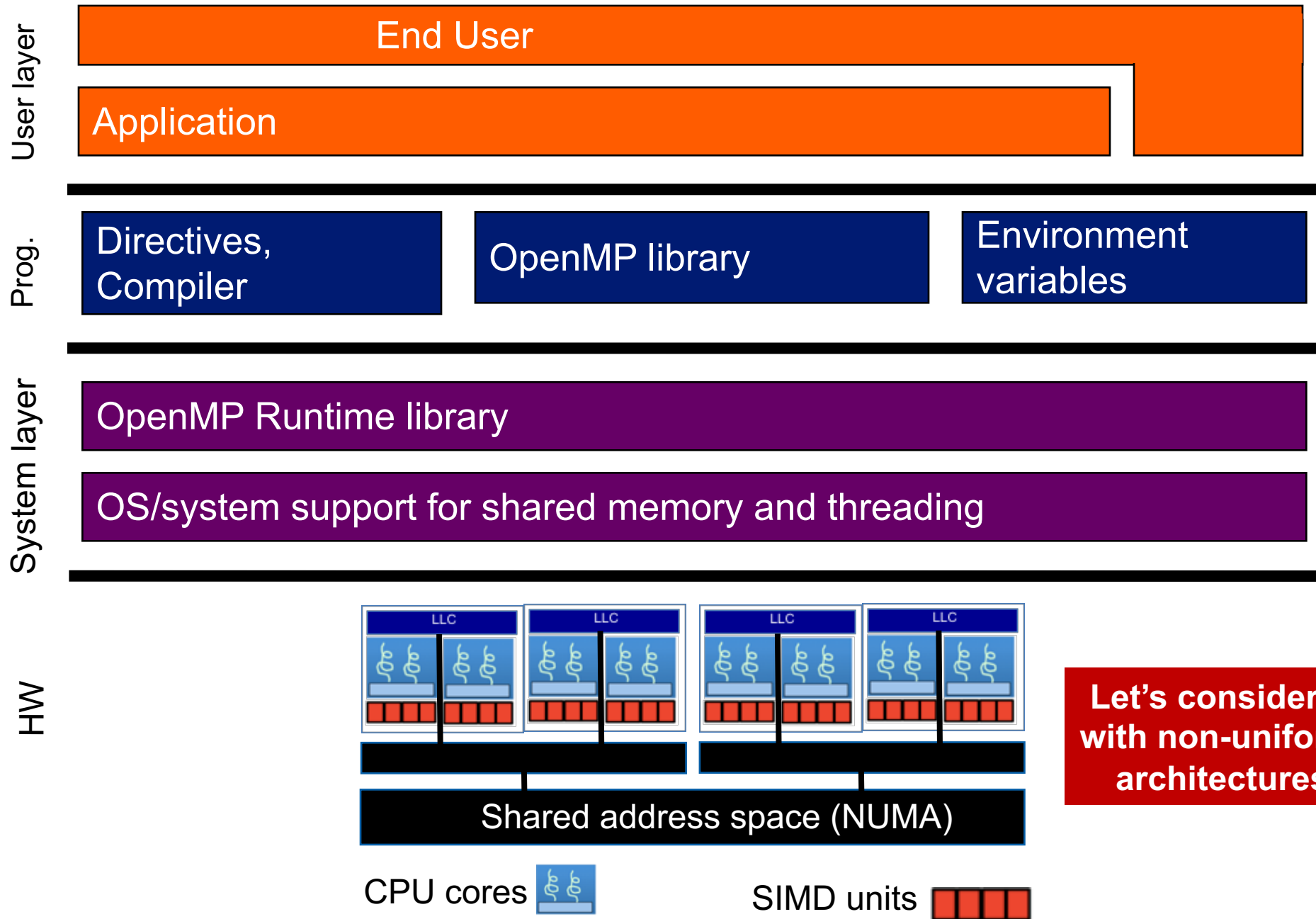
Using Tasks

- Don't use tasks for things already well supported by OpenMP
 - e.g. standard do/for loops
 - the overhead of using tasks is greater
- Don't expect miracles from the runtime
 - best results usually obtained where the user controls the number and granularity of tasks

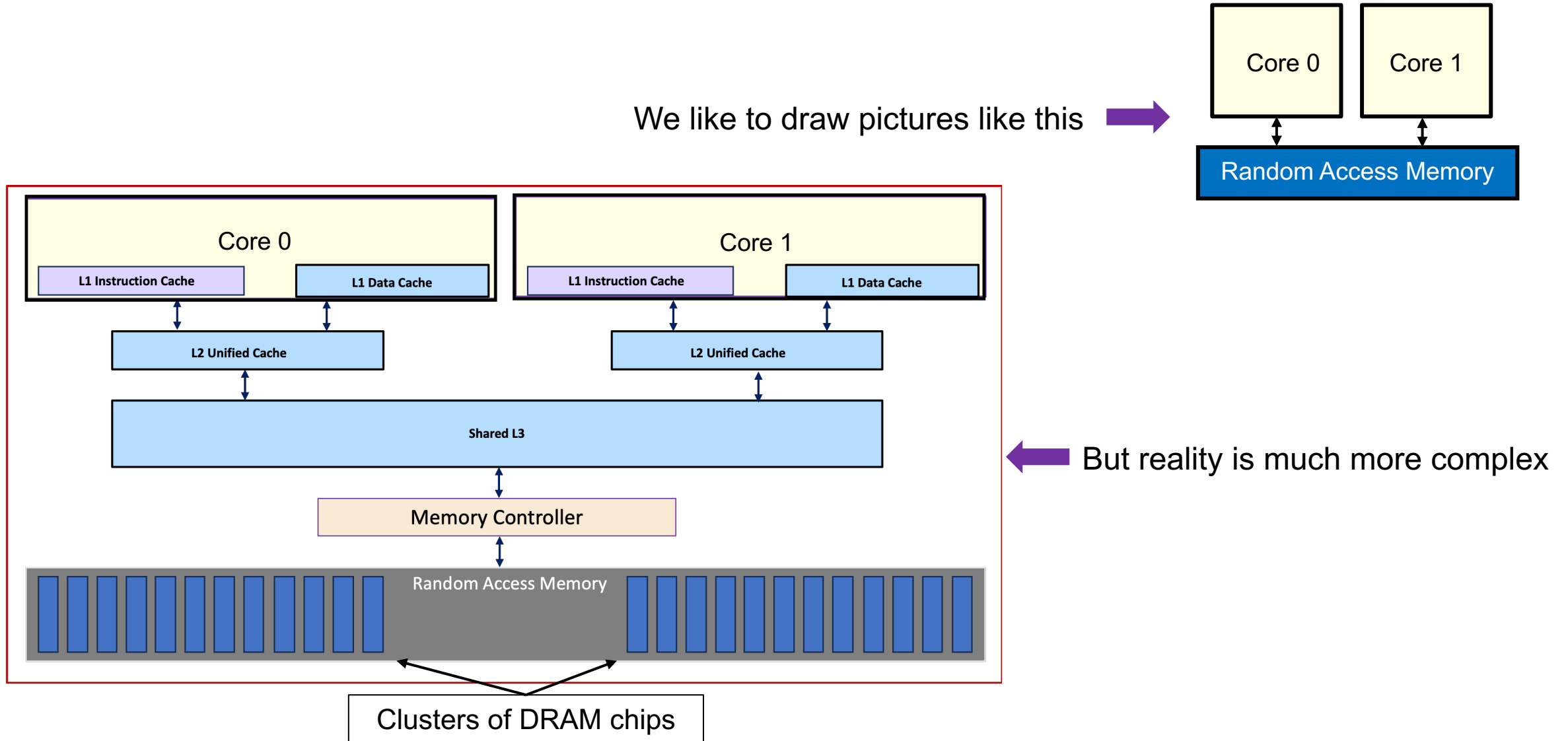
- A Very Brief OpenMP Review
- Irregular Parallelism and Tasks
- ➔ • OpenMP and Real Hardware: Dealing with NUMA
- Synchronization and the OpenMP Memory Model
- GPU Programming with OpenMP
- Recap

Slides on NUMA systems makes heavy use of content from Yun (Helen) He of NERSC, Christian Terboven (RWTH/AACHEN), and Michael Klemm (OpenMP)

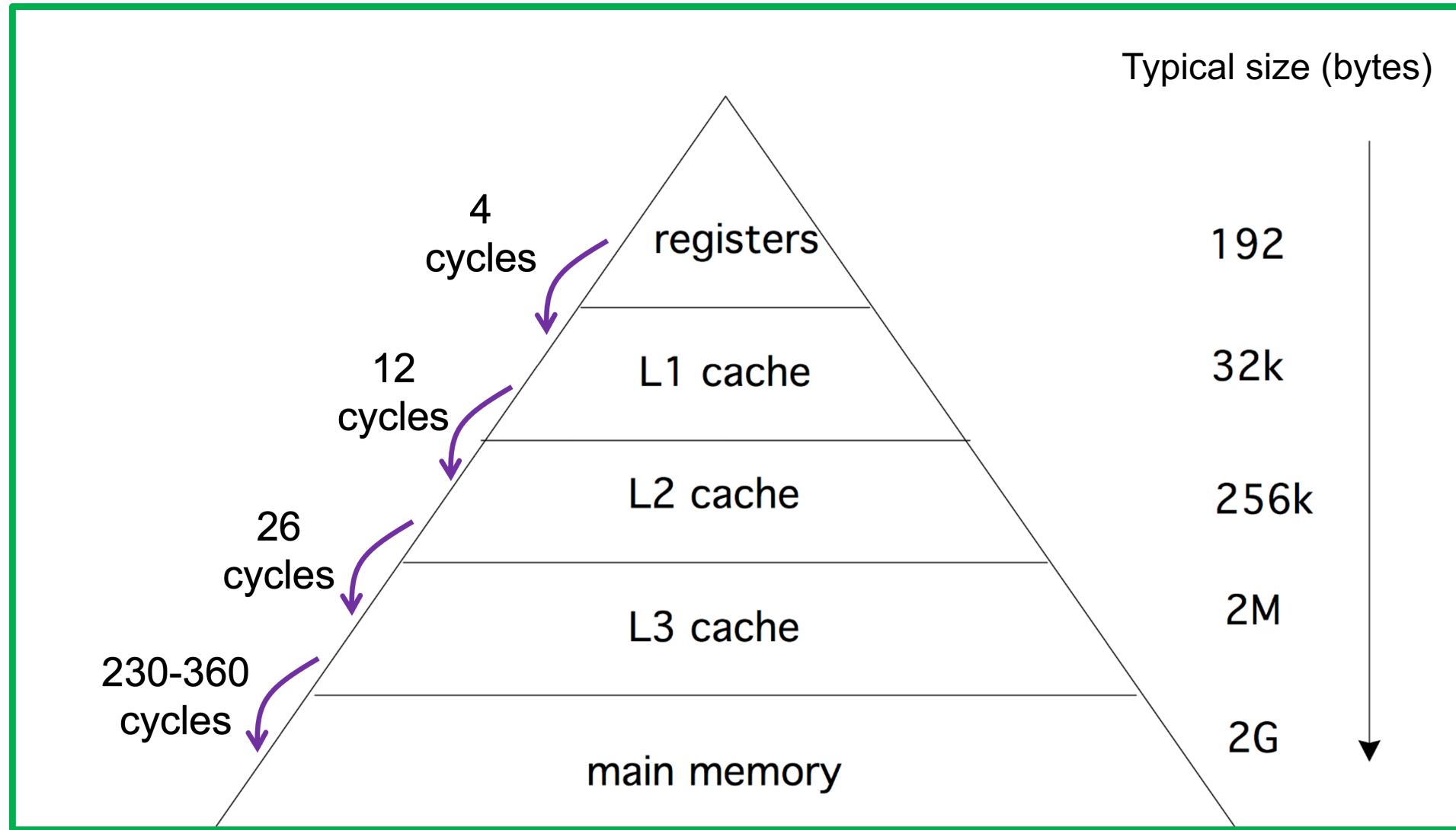
OpenMP Basic Definitions: Basic Solution Stack



SMP is a lie ... The memory is Non-Uniform

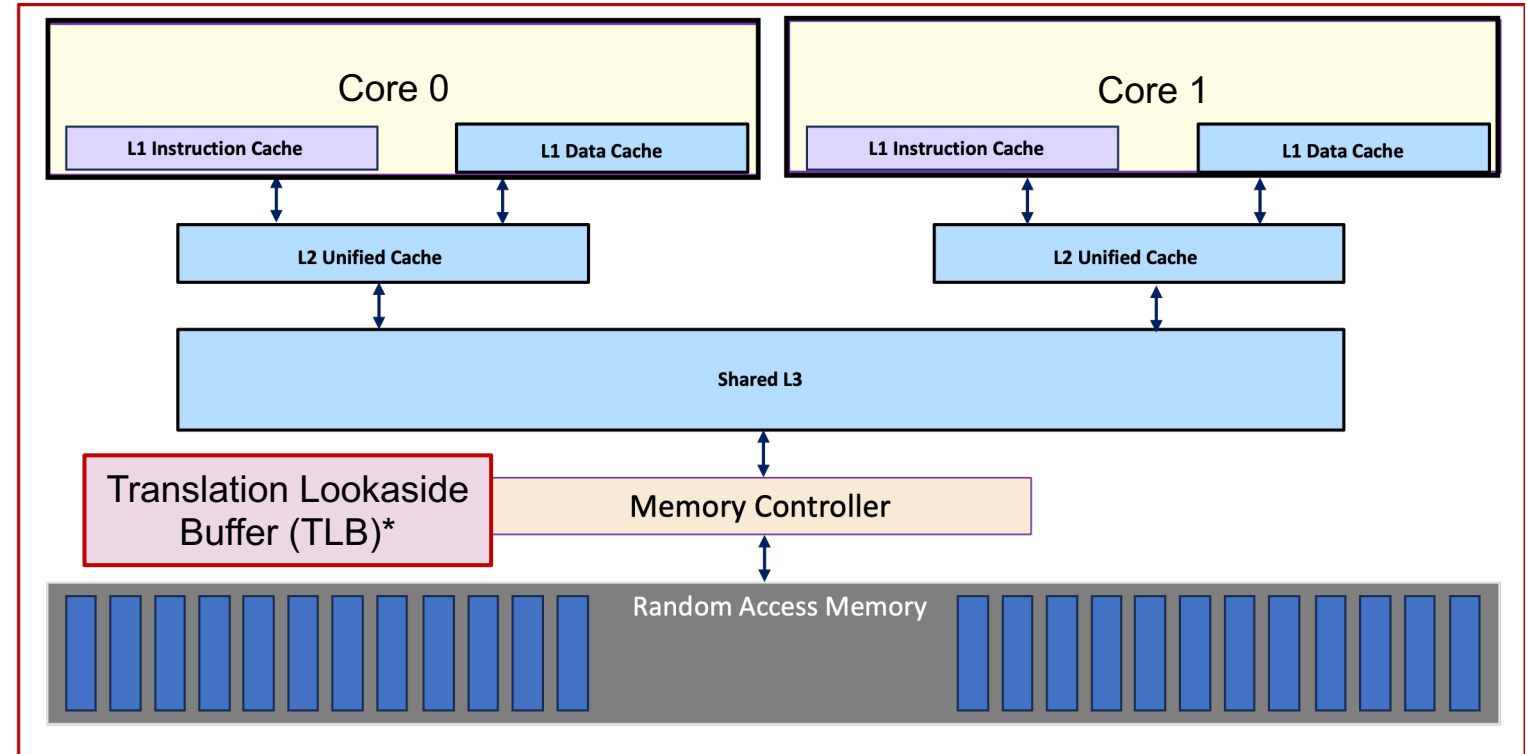


Latencies across the Memory Hierarchy



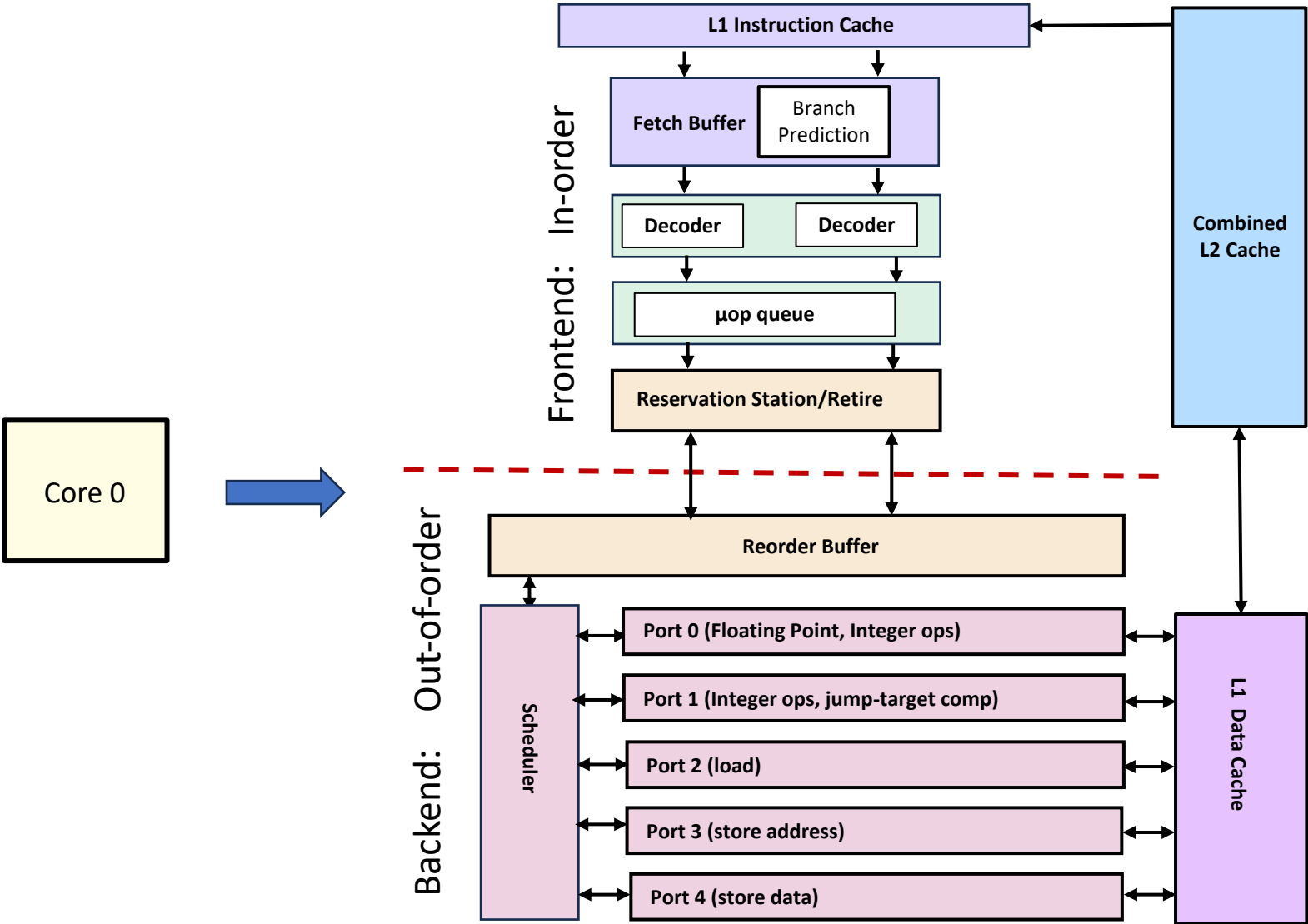
Memory Hierarchies are even more complex: the TLB

- 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

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”

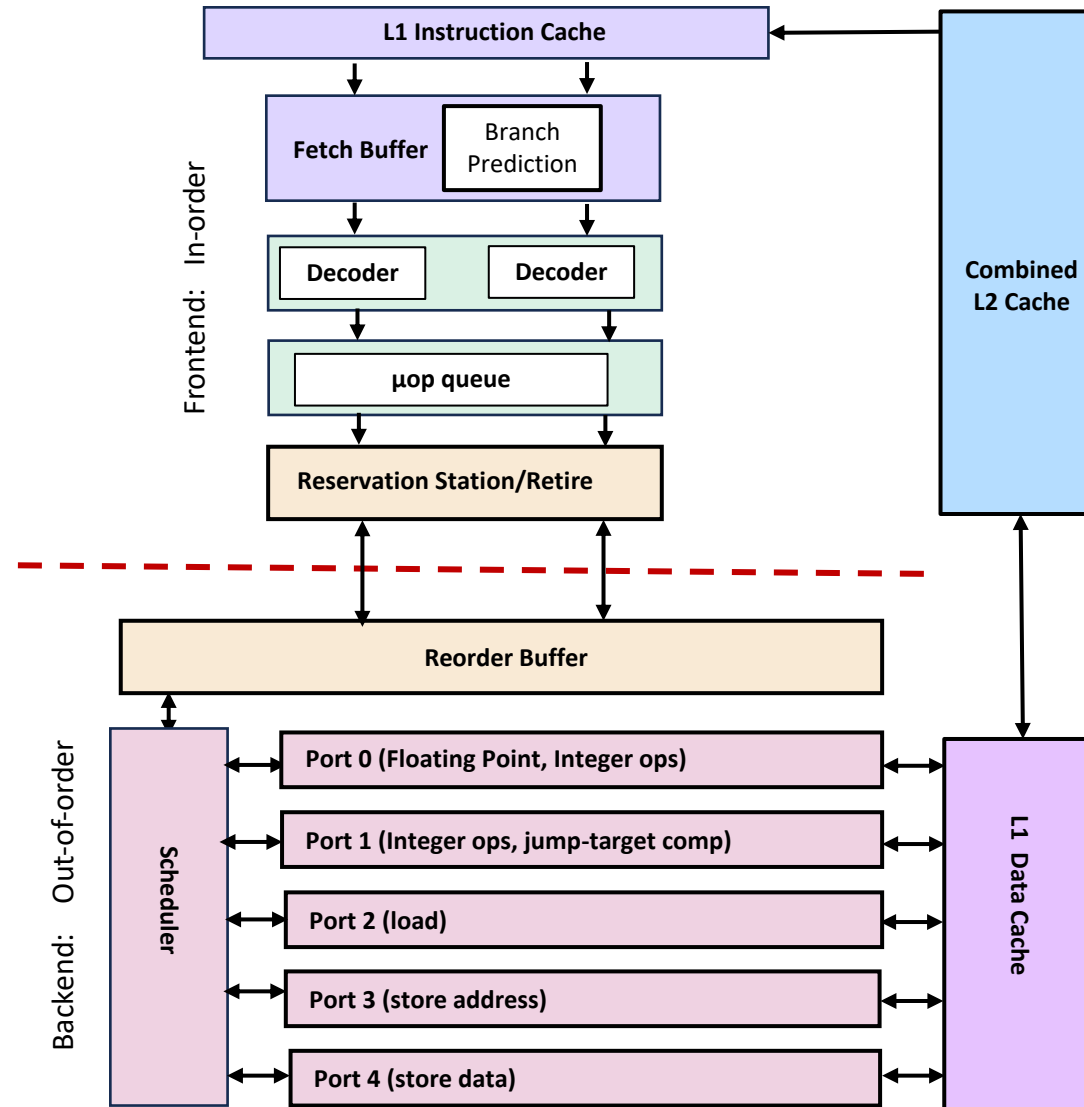
Backend executes decoded instructions.

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.

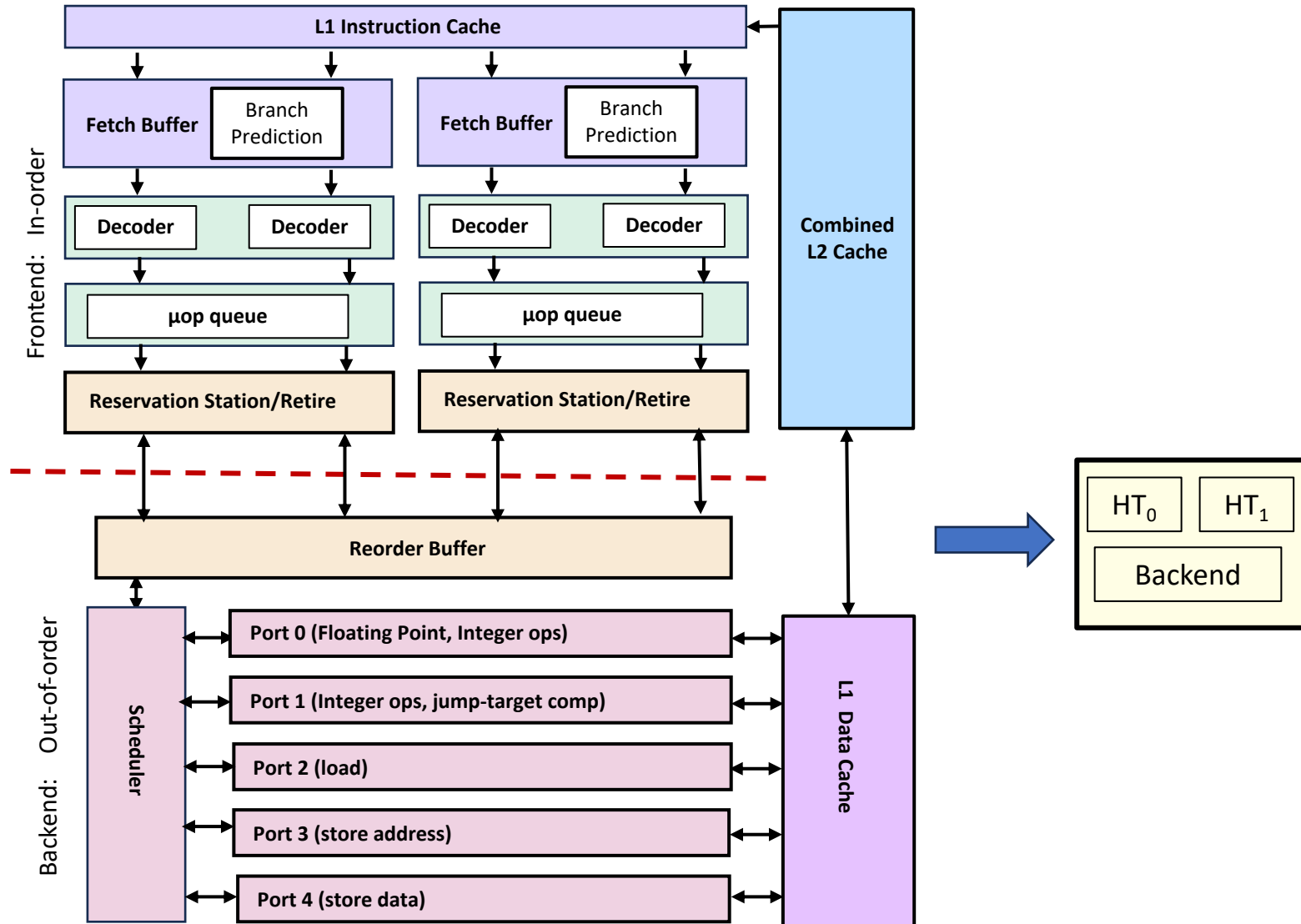


And its even worse

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

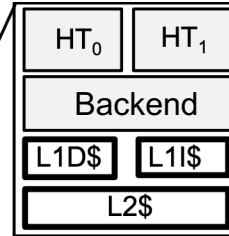
And then we put all this into a real two-socket node

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

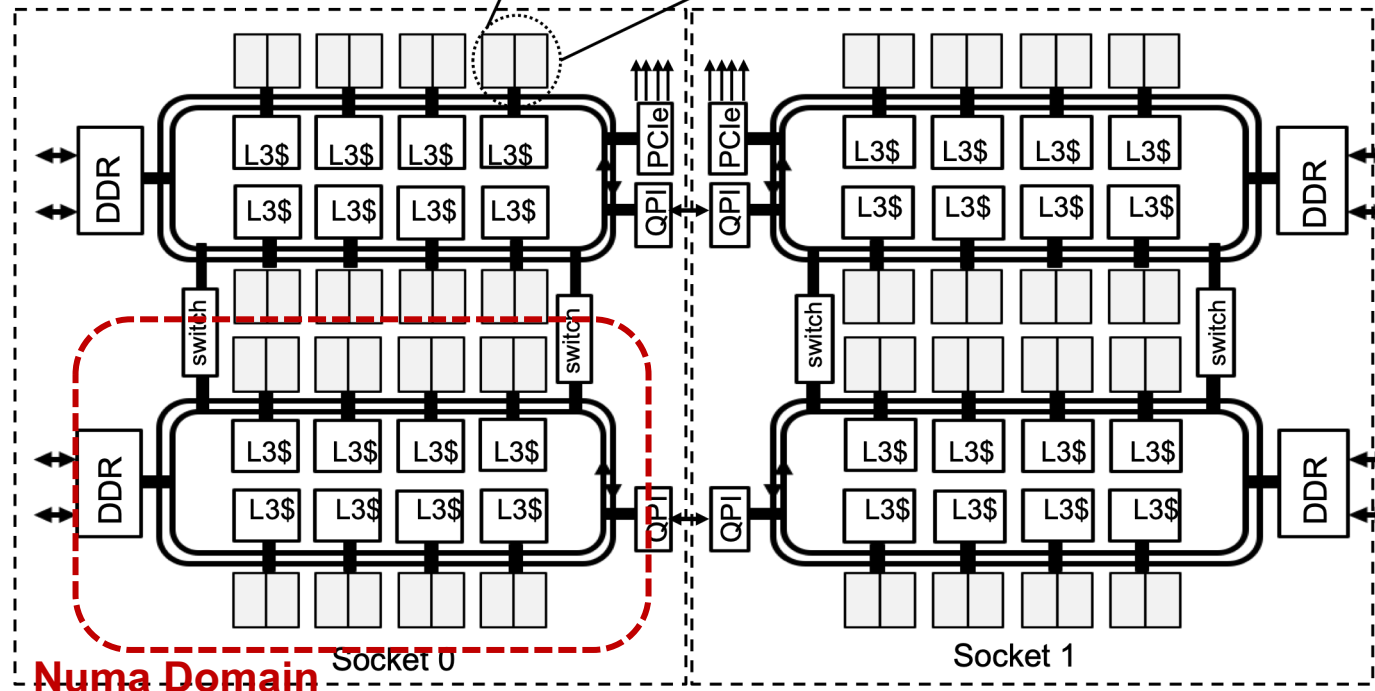
4 blocks of 8 core units connected by an on-chip-network with a DDR memory controller.

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.

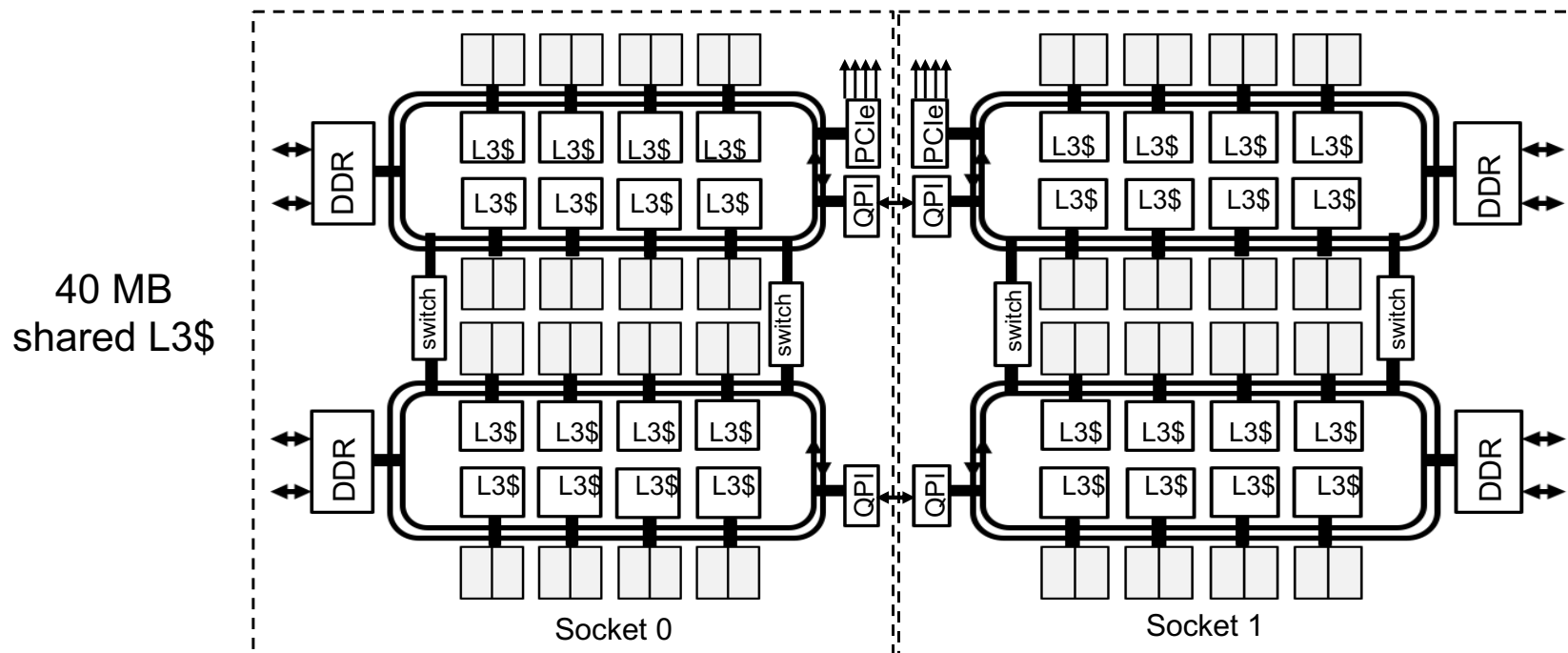
QPI: Quick Path Interconnect.
A coherent interconnect
between CPUs. Makes it
easy to build multi-CPU
nodes.

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)

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)

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

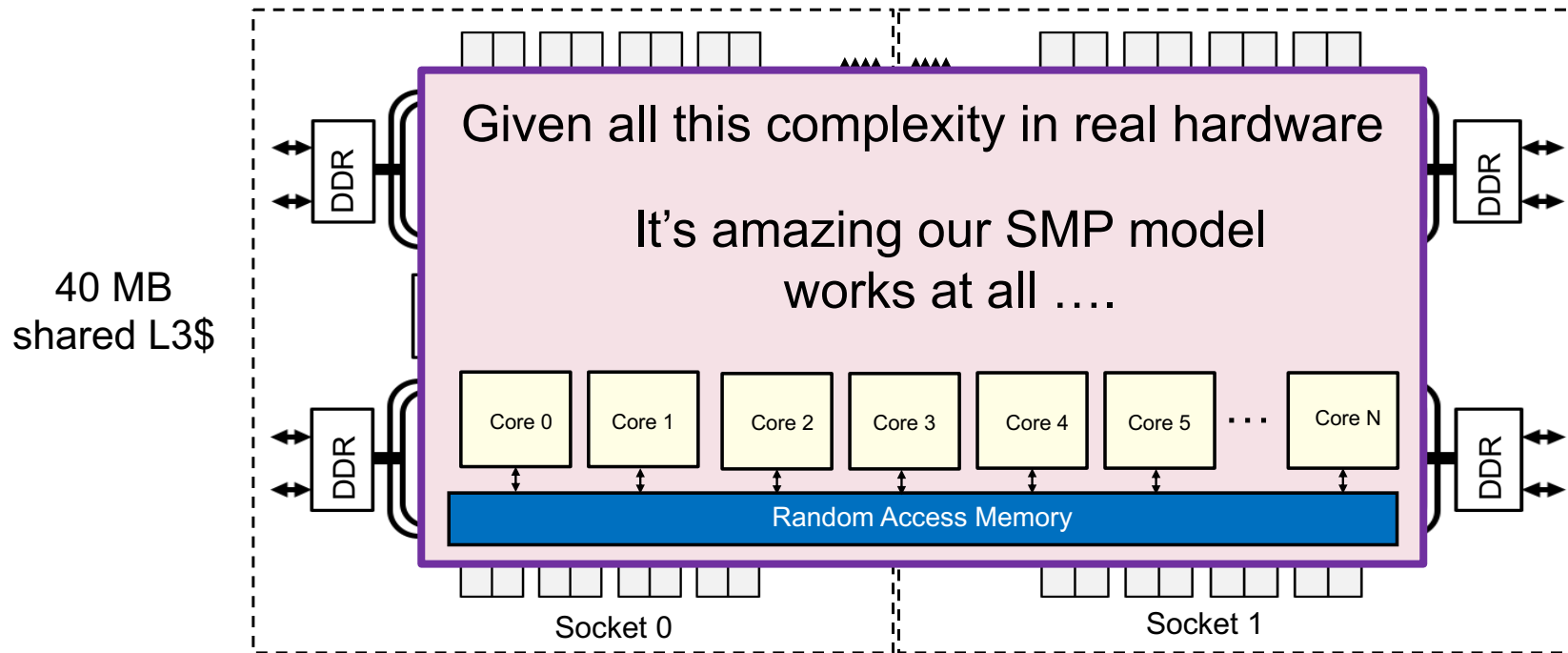


As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

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)

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

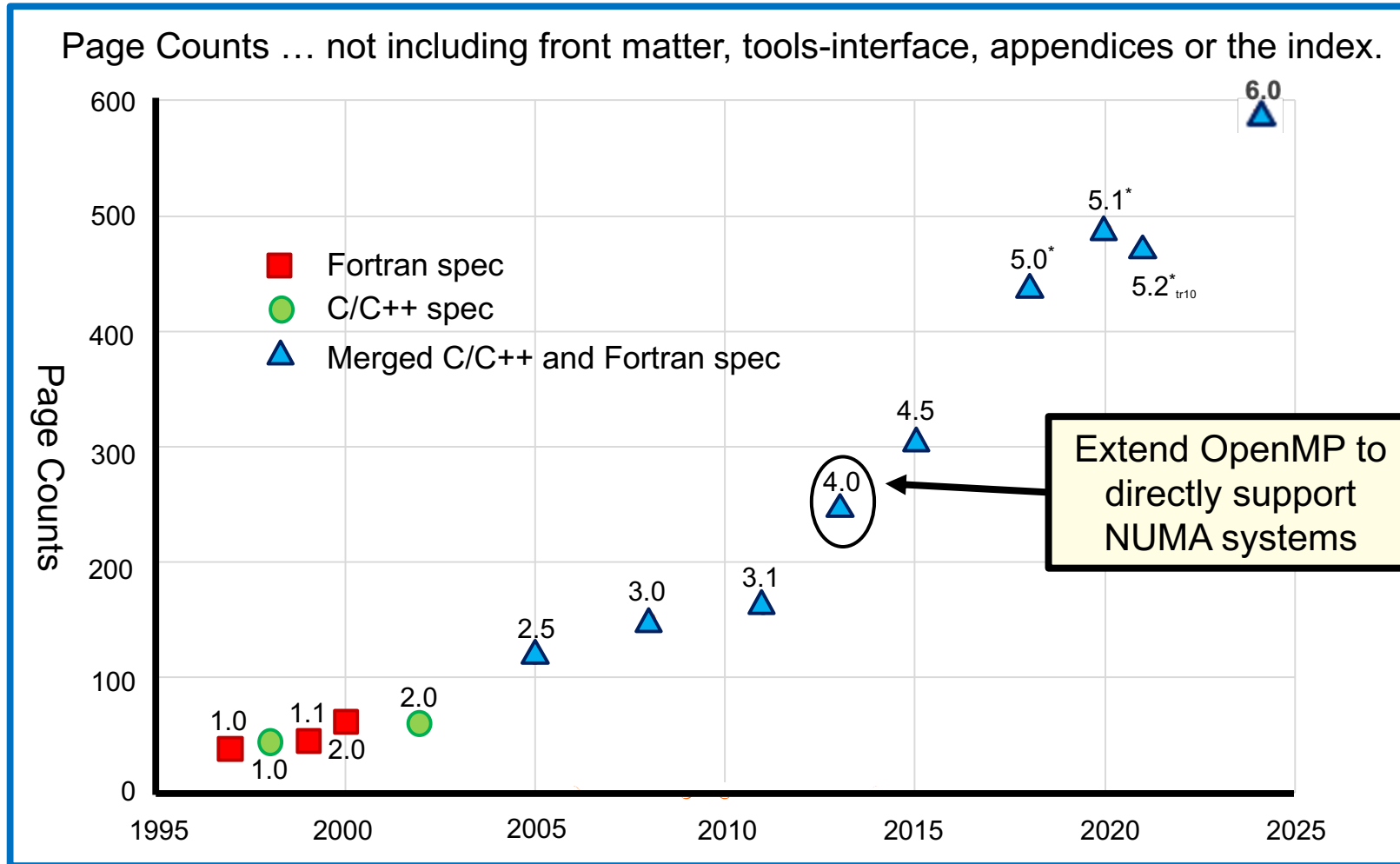


As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

Optimizing code for NUMA systems

The Growth of Complexity in OpenMP

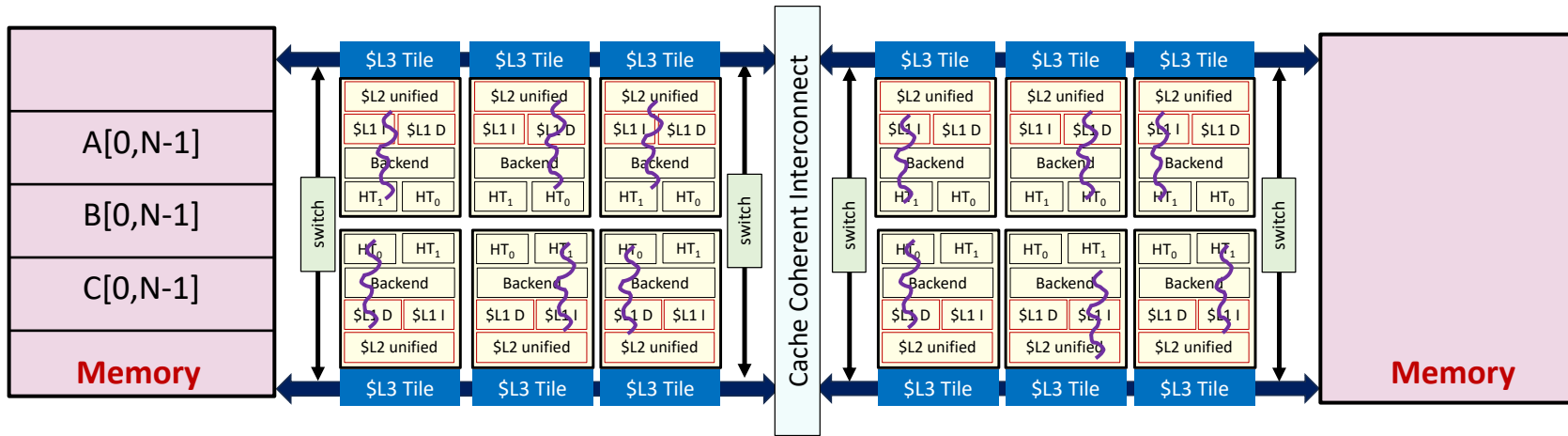
Our goal in 1997 ... A simple interface for application programmers



Consider the Stream Triad Benchmark

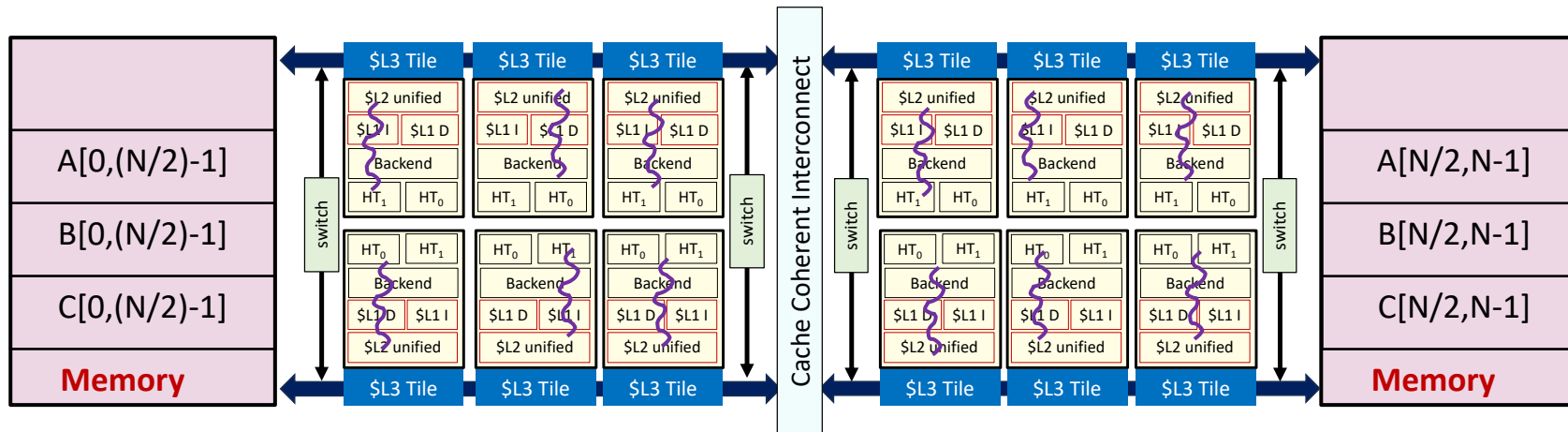
- Stream triad memory-bandwidth benchmark* running on a two-socket Intel® Xeon™ X5675 with 12 threads on 12 cores

```
// Stream Triad benchmark
#pragma omp parallel for
for (int i=0; i<N; i++)
    a[i] = b[i] + scalar * c[i];
```



3 arrays in one NUMA domain

triad
18.2 GB/s



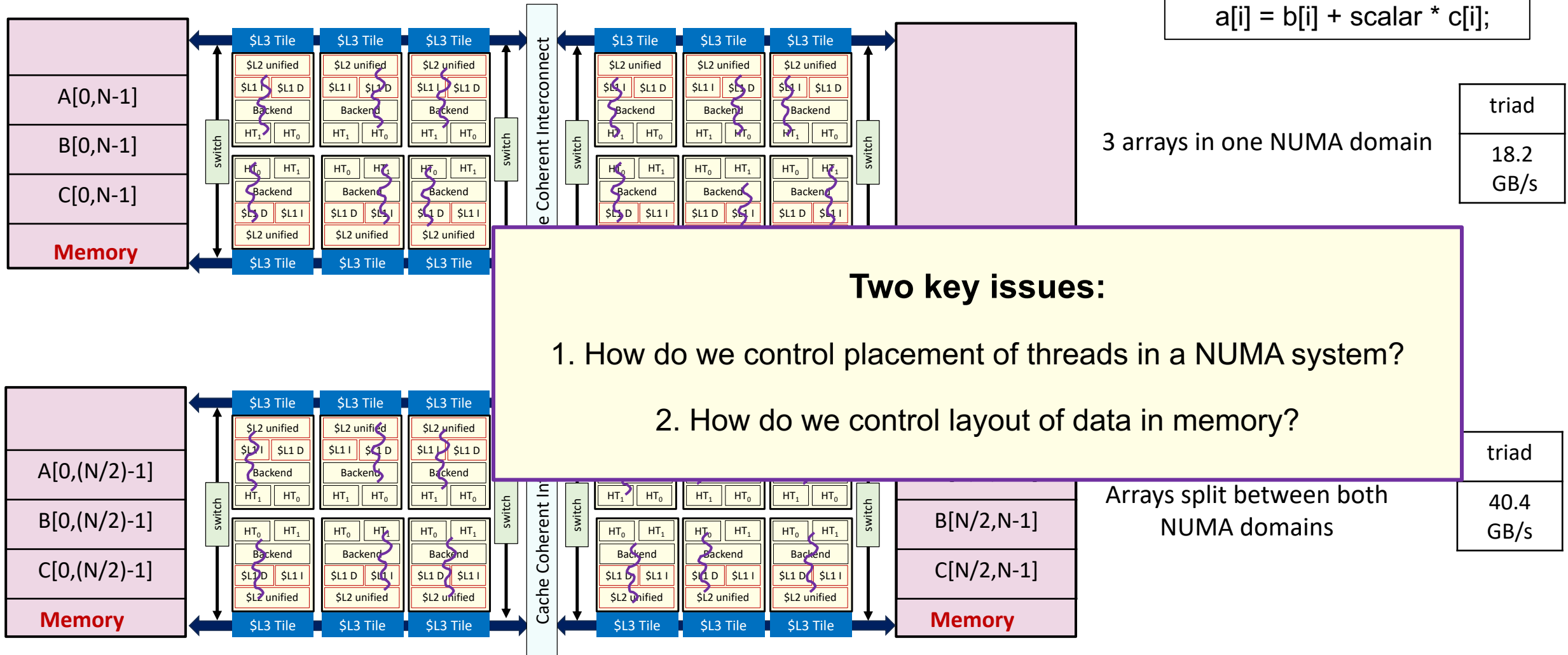
Arrays split between both NUMA domains

triad
40.4 GB/s

Consider the Stream Triad Benchmark

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

```
// Stream Triad benchmark
#pragma omp parallel for
for (int i=0; i<N; i++)
    a[i] = b[i] + scalar * c[i];
```



triad
18.2 GB/s

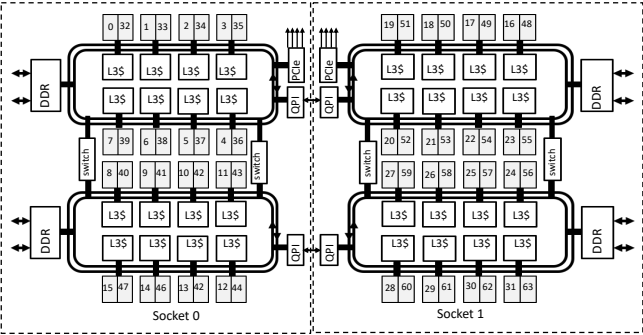
triad
40.4 GB/s

Controlling thread layout on a NUMA system:

- 1. How does the OS view components of processors?**
- 2. Where can threads go (places) and how do you line up threads to places (affinity)?**

Discover the OS view of virtual cores

- 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)

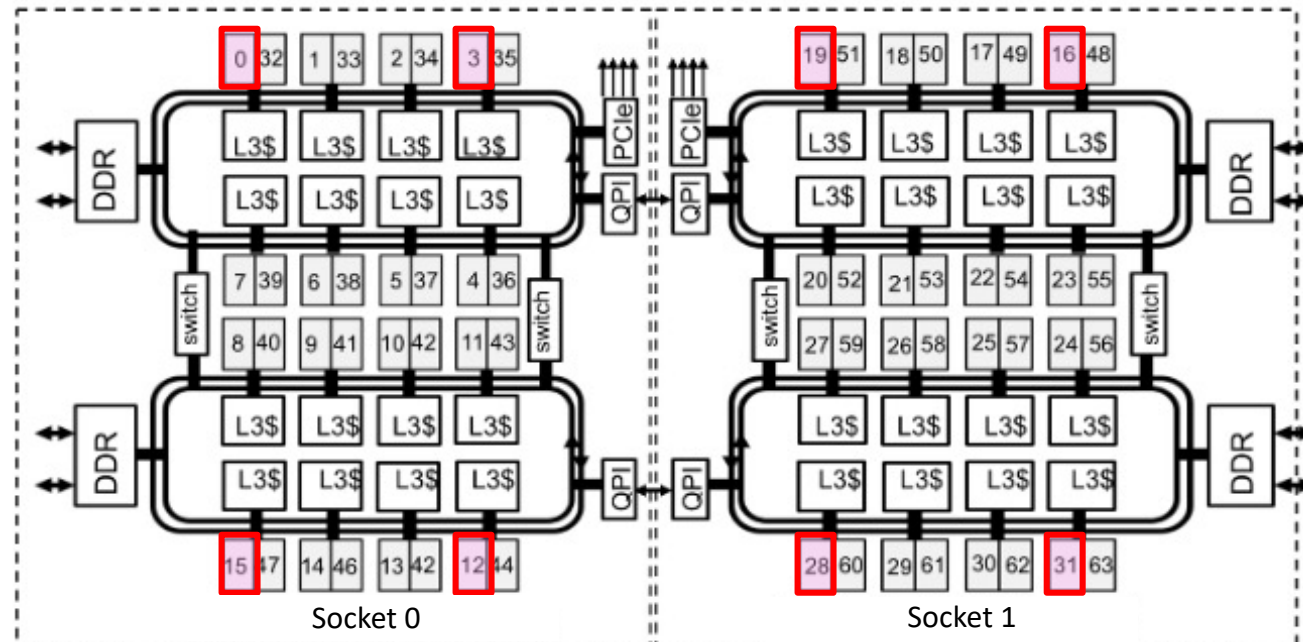
Using OMP_PLACES to select where to map threads

- Use a tool to learn the logical core IDs (processor units or PUs) as understood by the OS.
- OpenMP defines an environment variable (OMP_PLACES) to define where threads can be scheduled to execute.

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

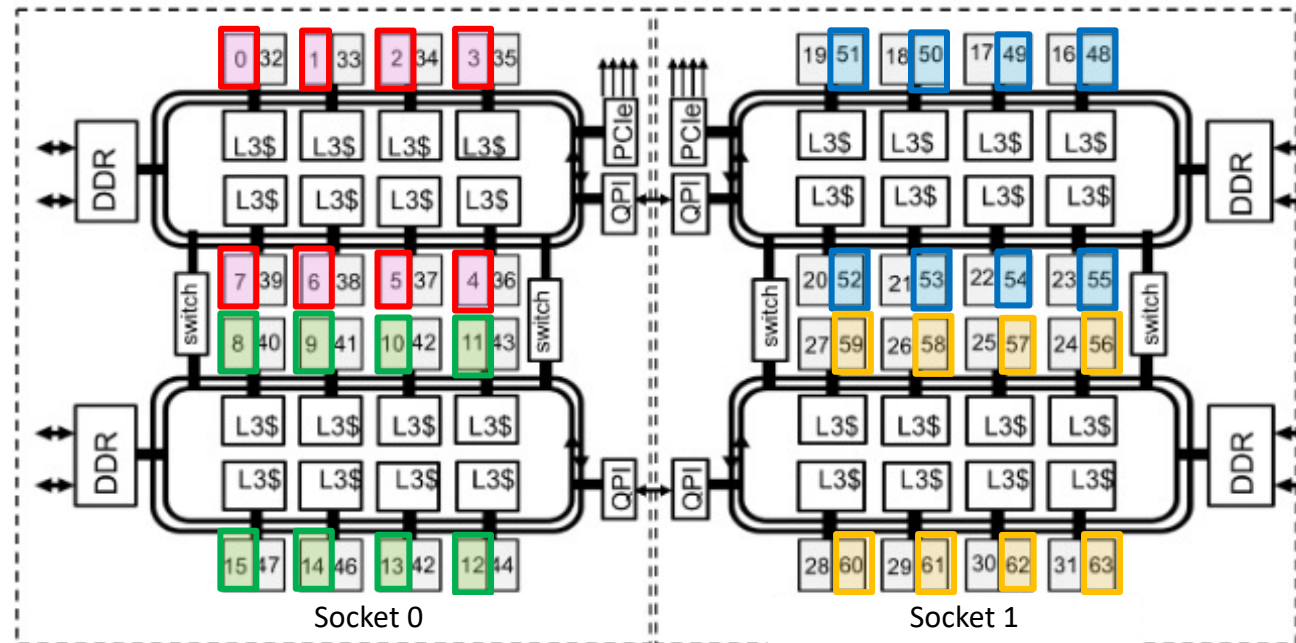
```
> export NUM_THREADS= 6
```

You can't control where the first thread goes, but the others will start at that first thread and wrap around the set of places



Using OMP_PLACES to select where to map threads

- Use a tool to learn the logical core IDs (processor units or PUs) as understood by the OS.
 - OpenMP defines an environment variable (OMP_PLACES) to define where threads can be scheduled to execute.
 - We can organize places into partitions to capture the structure of the NUMA system.
 - In the following, we have one places partition per NUMA domain*
- > export OMP_PLACES="{0:7:1},{8:15:1},{48:55:1},{56:63:1}"

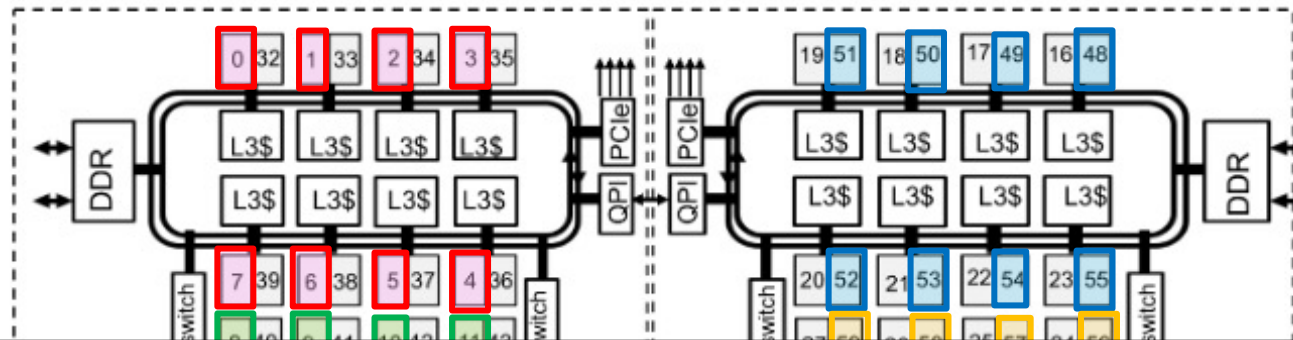


* Note that {0:7:1}. is equivalent to {0, 1, 2, 3, 4, 5, 6, 7}.

The details of what constitutes a NUMA domain is OS dependent ... it depends on how the OS maps memory pages onto DRAM and how caches are managed.

Using OMP_PLACES to select where to map threads

- Use a tool to learn the logical core IDs (processor units or PUs) as understood by the OS.
 - OpenMP defines an environment variable (OMP_PLACES) to define where threads can be scheduled to execute.
 - We can organize places into partitions to capture the structure of the NUMA system.
 - In the following, we have one places partition per NUMA domain*
- > export OMP_PLACES="{0:7:1},{8:15:1},{48:55:1},{56:63:1}"



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

* Note that {0:7:1}. is equivalent to {0, 1, 2, 3, 4, 5, 6, 7}.

The details of what constitutes a NUMA domain is OS dependent ... it depends on how the OS maps memory pages onto DRAM and how caches are managed.

Hardware Abstraction and OMP_PLACES

- OMP_PLACES can be an abstract name instead of 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)

- Examples:
 - `export OMP_PLACES=threads`
 - `export OMP_PLACES=cores`

Mapping threads to places: Thread affinity with OMP_PROC_BIND

- 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**

Example:

```
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

Mapping threads to places: Thread affinity with `proc_bind()` clause

- The `proc_bind()` clause 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**

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

Note: You **cannot** set places as a clause on a directive since the OS defines places when it launches the process to run a program

Examples: OMP_PROC_BIND

Assume 4 cores, 2 hardware threads per core, and 4 OpenMP threads

Set 4 place partitions: OMP_PLACES={0,1},{2,3},{4,5},{6,7}

- **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				

We have no control of where the initial thread is placed, but for these examples, we assume it is placed on HT₀ of Core 0.

With close, threads placed in consecutive locations

- **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	

With spread, threads placed in first place of each partition

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

OMP_PROC_BIND Choices for STREAM Benchmark

OMP_NUM_THREADS=32

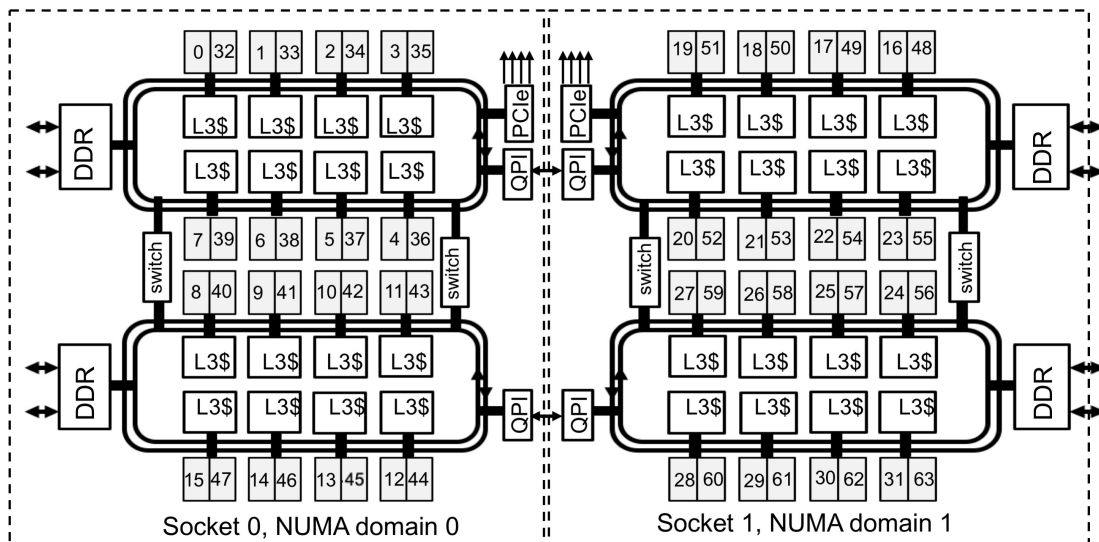
OMP_PLACES=threads

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

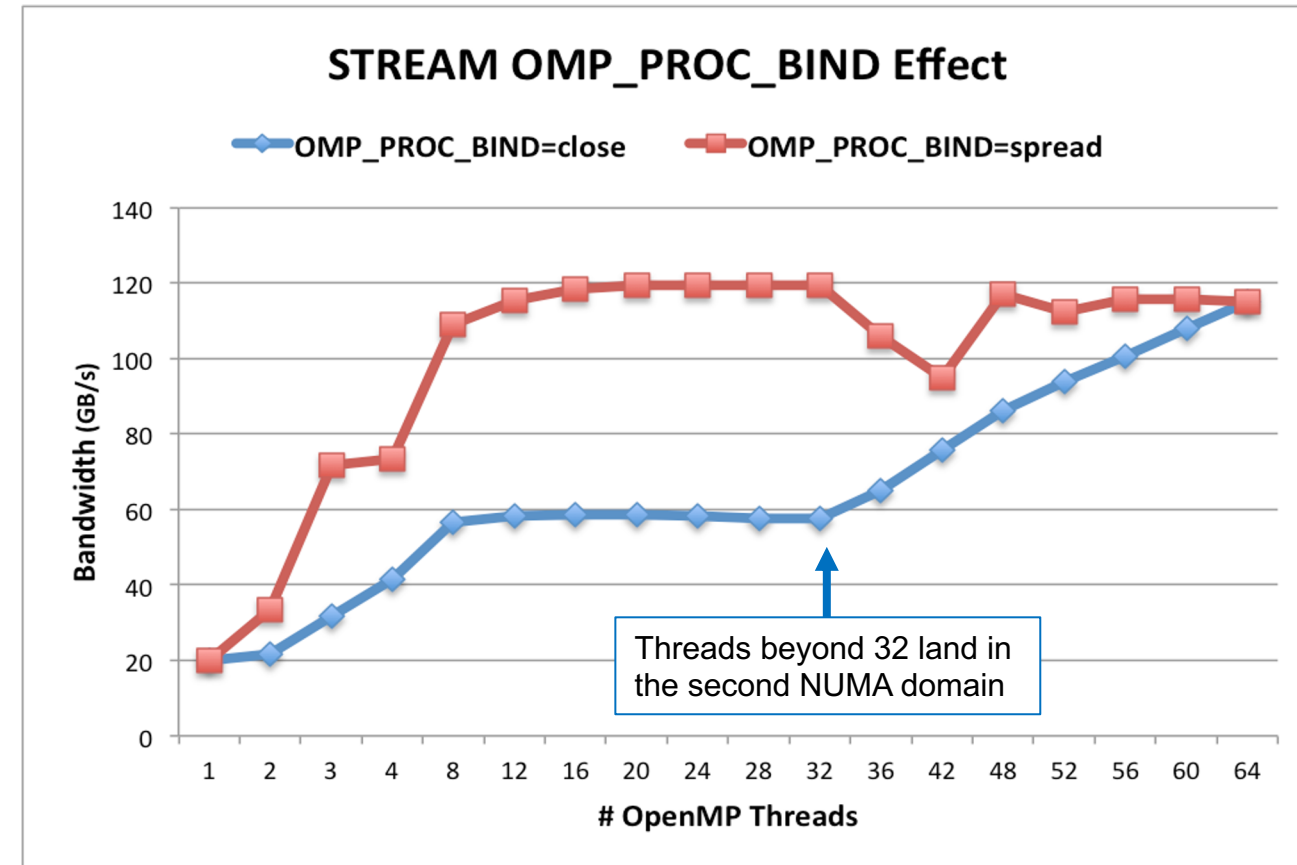
Threads 0 to 31 bind to cores 0,1,2,... to 31. Both sockets and memory are used to maximize memory bandwidth.



Blue: OMP_PROC_BIND=close

Red: OMP_PROC_BIND=spread

In both cases, assume the arrays are spread out across all memory domains (a topic we'll address soon)



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

Aligning memory to threads ... First touch

Memory Affinity: Exploiting “First Touch” page mapping policy

Step 1.1 Initialization by primary thread only

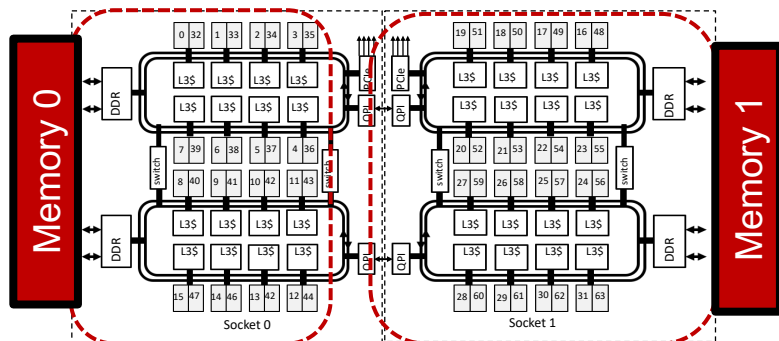
```
for (j=0; j<VectorSize; j++) {  
    a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}  
}
```

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];
```



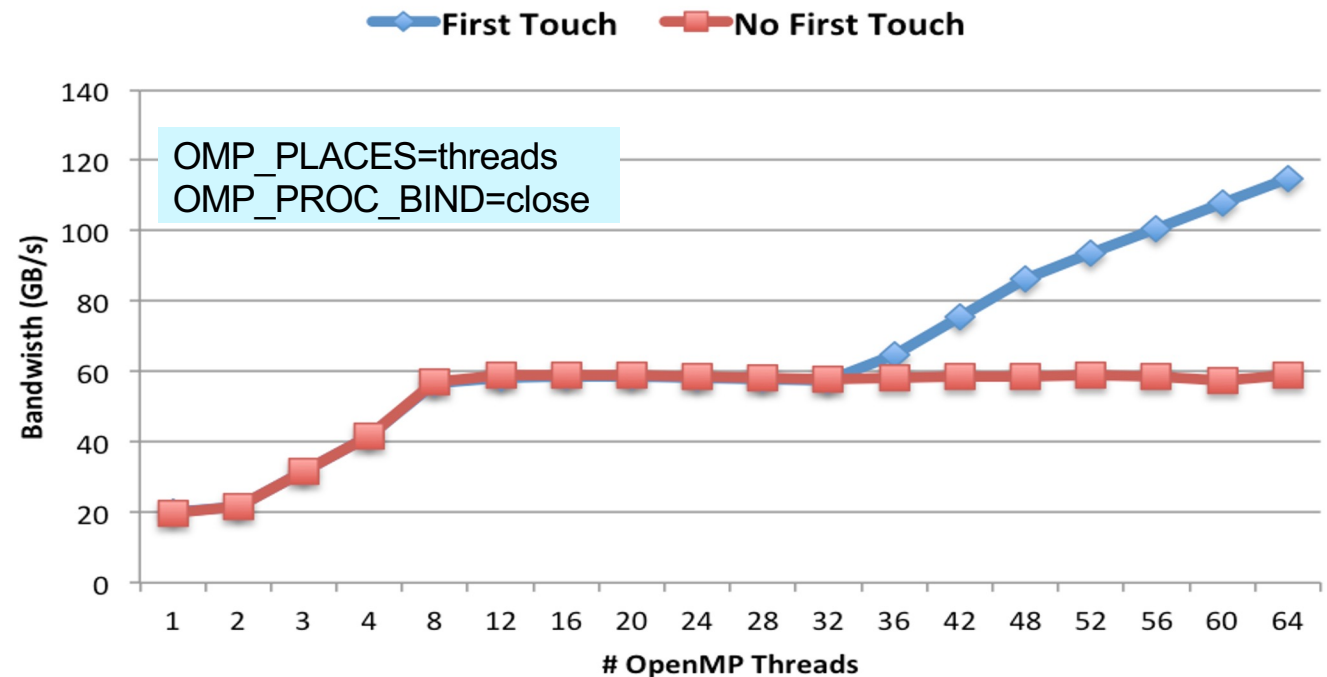
A dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

- 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: memory is local to the thread that initializes it.

Red: step 1.1 + step 2. Arrays come from Memory 0 only

Blue: step 1.2 + step 2. Arrays come from both Memory domains

STREAM NUMA Effect



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

Example: working with the First Touch Policy

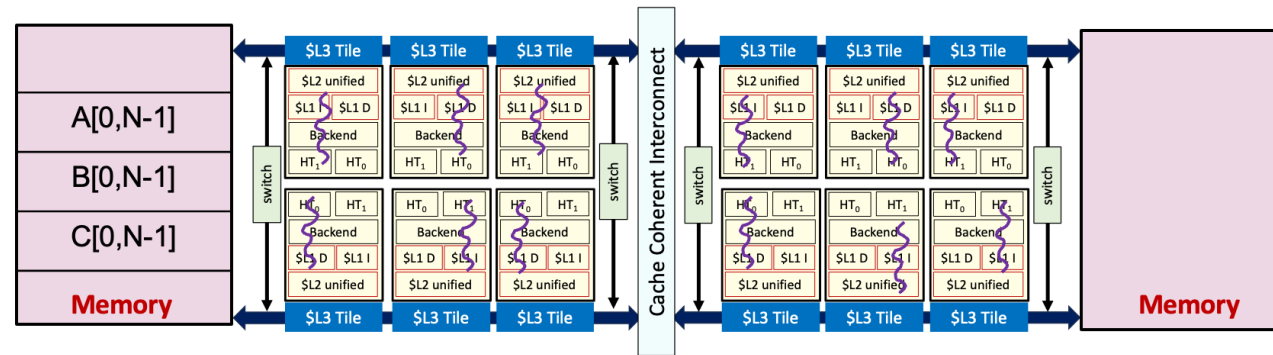
Remember this slide?

Consider the Stream Triad Benchmark

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

```
// Stream Triad benchmark
#pragma omp parallel for
for (int i=0; i<N; i++)
    a[i] = b[i] + scalar * c[i];
```

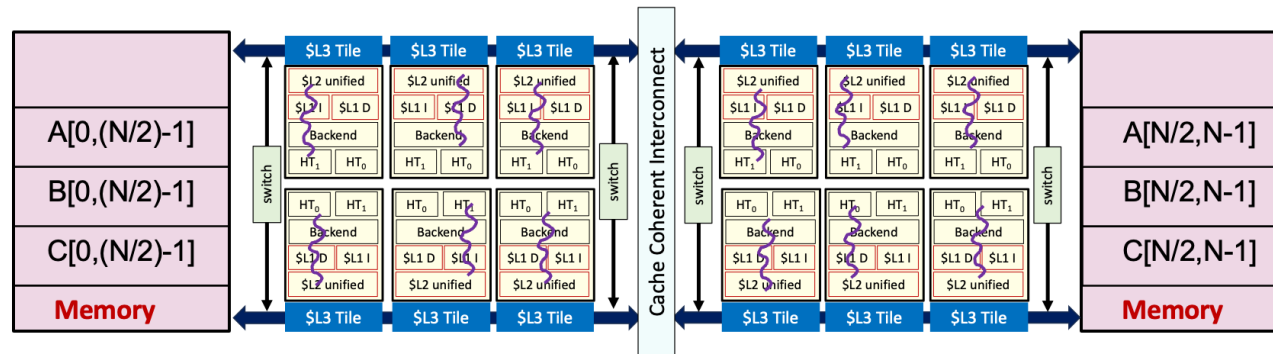
Arrays A, B, and C
initialized on
primary thread



3 arrays in one NUMA domain

triad
18.2 GB/s

Arrays A, B, and C
initialized with the
same “parallel for
schedule” as used
for the computation



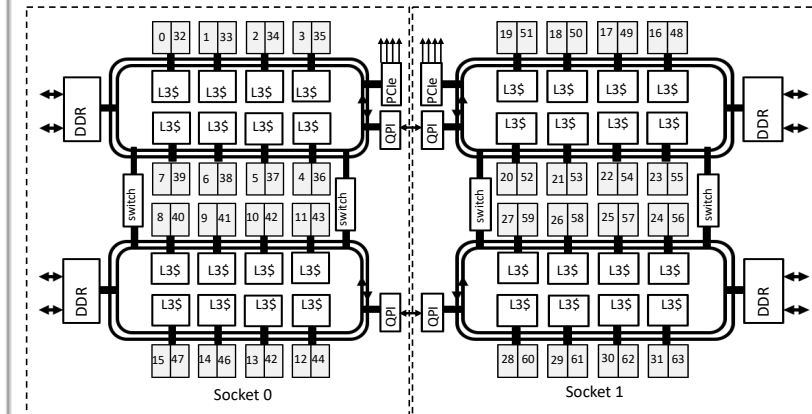
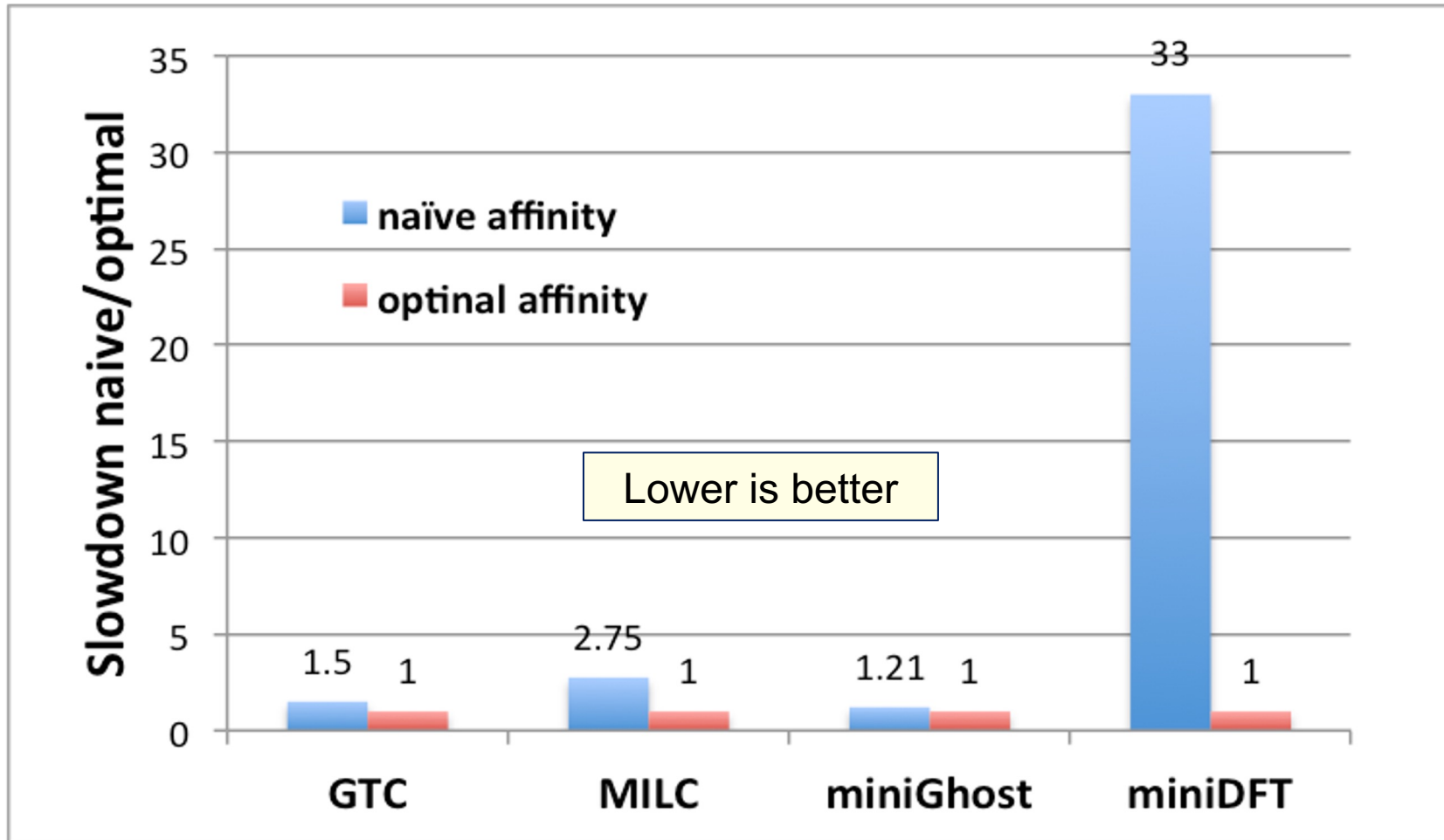
Arrays split between both
NUMA domains

triad
40.4 GB/s

**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



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

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 far 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

Exercise: Matrix Transpose

- Consider the program transpose.c
 - Generates a square matrix, transposes the matrix and tests the result.
- This is all memory movement. Optimize the transpose operation:

```
export OMP_PROC_BIND=spread
#pragma omp parallel proc_bind(spread)
export OMP_PLACES="{0,21, 42, 64}"
export OMP_PLACES="{0:7:1},{8:15:1},{48:55:1},{56:63:1}"
export OMP_PLACES=threads

#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();    int omp_get_thread_num();  int omp_set_num_threads();
double omp_get_wtime();
```

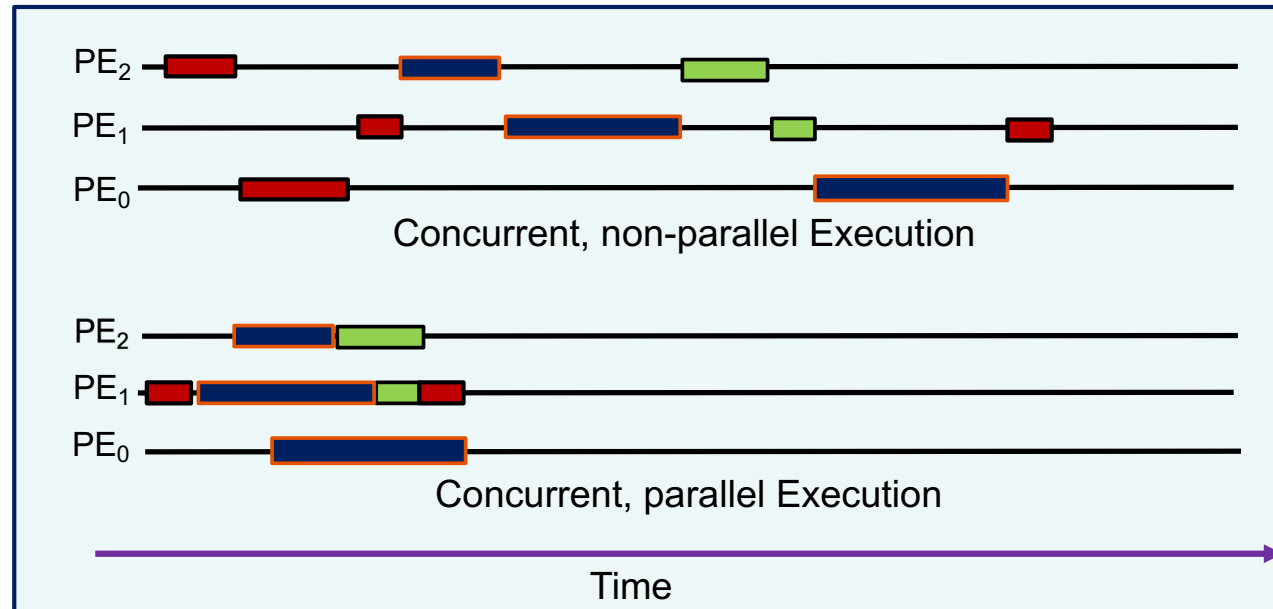
Annotations:

- Box 1: true, false, close, spread, primary (points to `spread` in `OMP_PROC_BIND` and `proc_bind`)
- Box 2: threads, cores, sockets (points to `threads` in `OMP_PLACES`)

- A Very Brief OpenMP Review
- Irregular Parallelism and Tasks
- OpenMP and Real Hardware: Dealing with NUMA
- ➡ • Synchronization and the OpenMP Memory Model
- GPU Programming with OpenMP
- Recap

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 an ordering constraint in the otherwise unordered execution of threads

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 in most Introduction to OpenMP courses

- Other, more advanced, synchronization operations:

- locks
 - atomic
 - flush

Covered in this section

Synchronization: Lock Routines

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

A lock implies a memory fence ... hence a compiler cannot move memory operations around a lock.

Note: a thread always accesses the most recent copy of the lock (which when we discuss memory models, means you don't need to use a flush on the lock variable).

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

Synchronization: Simple Locks Example

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

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

```
omp_lock_t odd_lck, even_lck;
```

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

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

One lock per case ... even and odd

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

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

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

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

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

```
        even_count++;
```

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

```
    }
```

```
    else{
```

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

```
        odd_count++;
```

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

```
    }
```

```
}
```

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

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

```
}
```

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

Free-up storage when done.

Synchronization: Atomic

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

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

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

Synchronization: Atomic

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

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

Atomic only protects the
read/update of X

The OpenMP 3.1 Atomics (1 of 2)

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

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

- Atomic can protect loads

pragma omp atomic read

v = x;

- Atomic can protect stores

pragma omp atomic write

x = expr;

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

pragma omp atomic update

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

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

This is the
original OpenMP
atomic

The OpenMP 3.1 Atomics (2 of 2)

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

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

- Where the statement is one of the following forms:

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

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

`{v = x; x binop = expr;}`

`{x binop = expr; v = x;}`

`{v=x; x=x binop expr;}`

`{X = x binop expr; v = x;}`

`{v = x; x++;}`

`{v=x; ++x;}`

`{++x; v=x;}`

`{x++; v = x;}`

`{v = x; x--;}`

`{v= x; --x;}`

`{--x; v = x;}`

`{x--; v = x;}`

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

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

```
#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;
}
```

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

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.

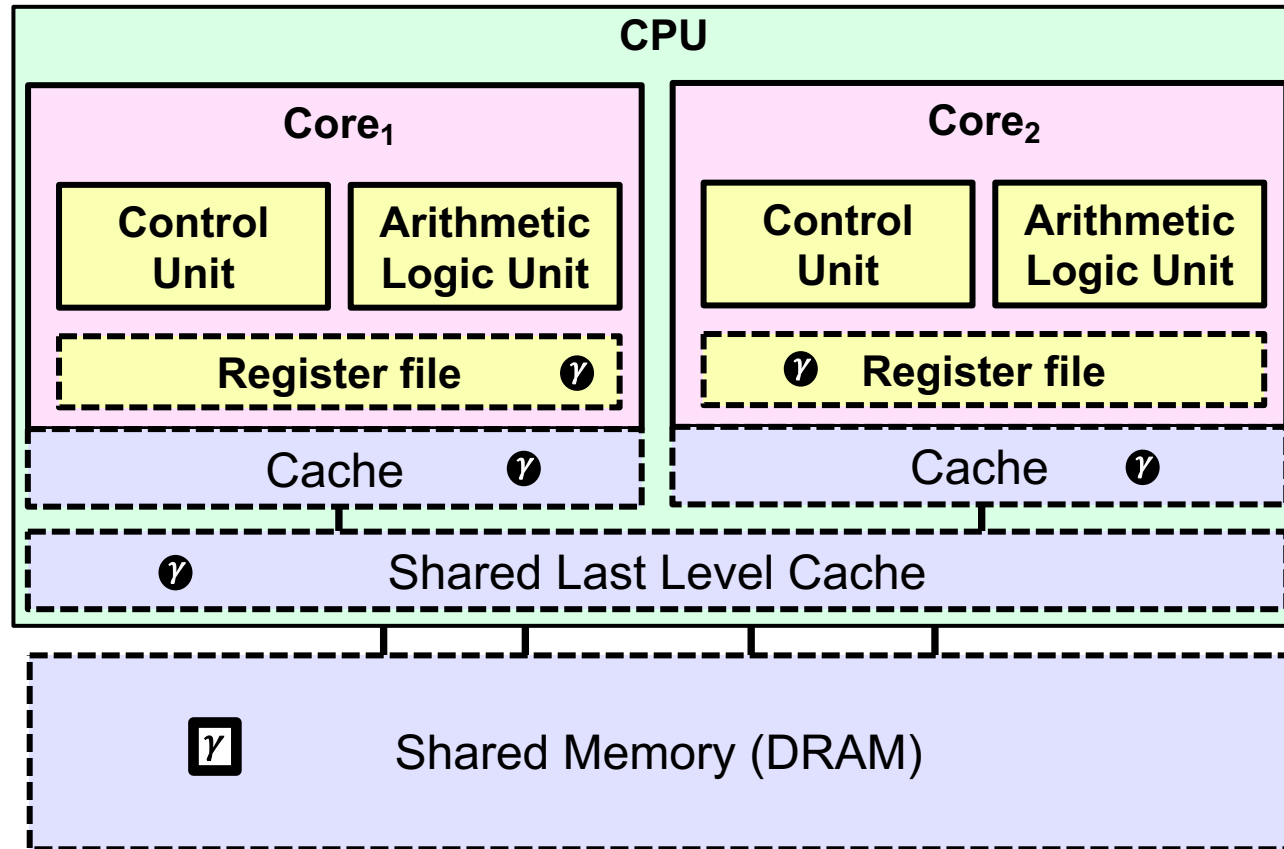
In the jargon of concurrent
programming, this is called
a “spin lock”

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 ...

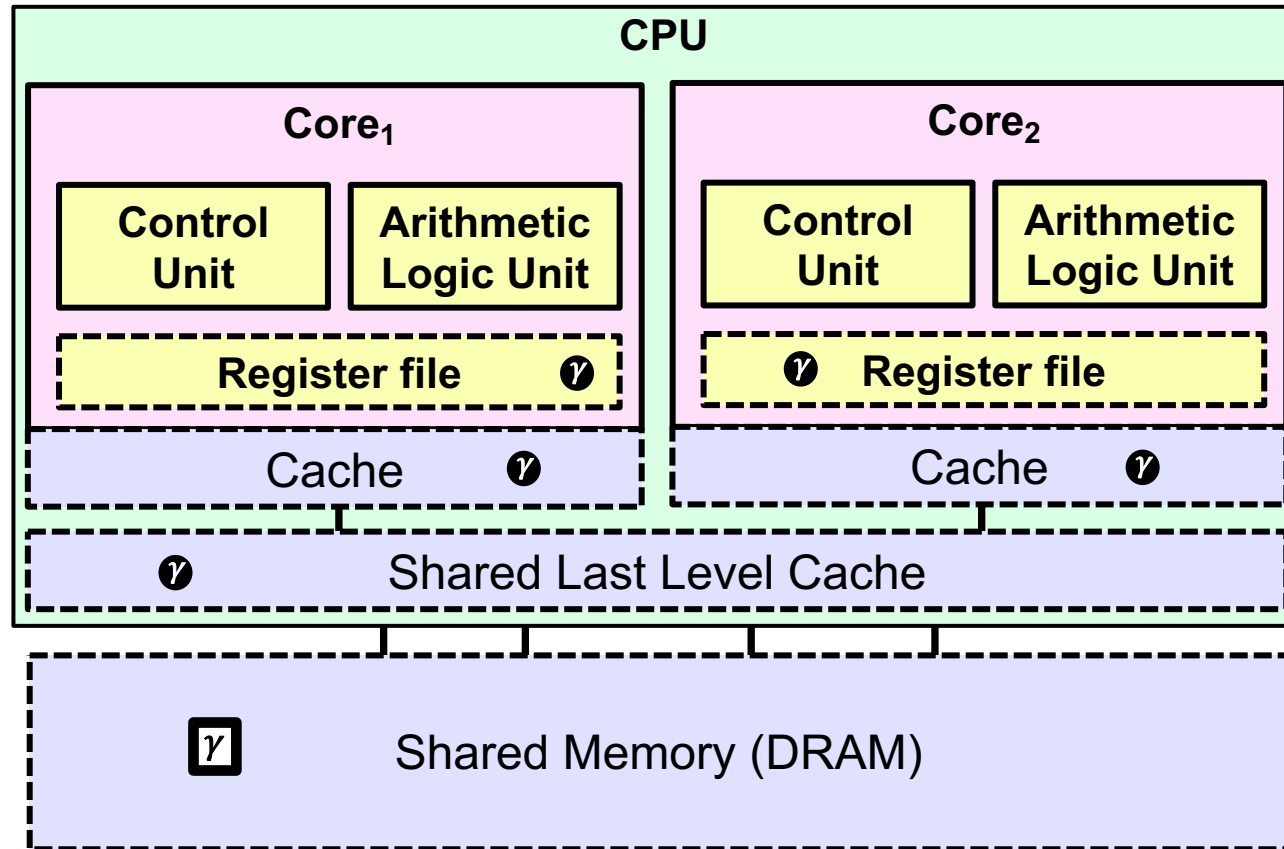
- A shared address space is a region of memory visible to the team of threads ... multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different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

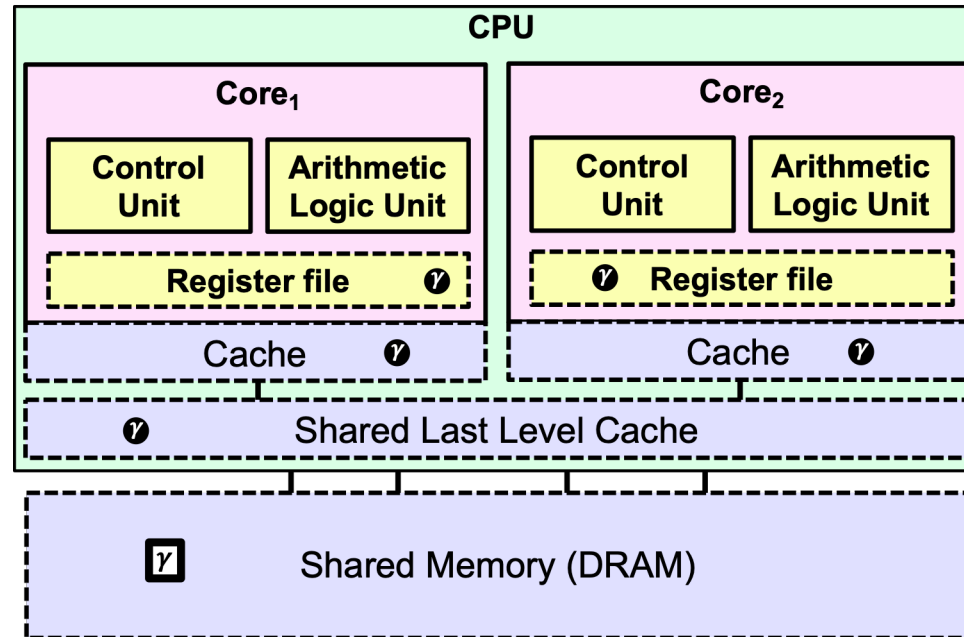
- Which value of γ is the one a thread should see at any point in a computation?

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;
}
```

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

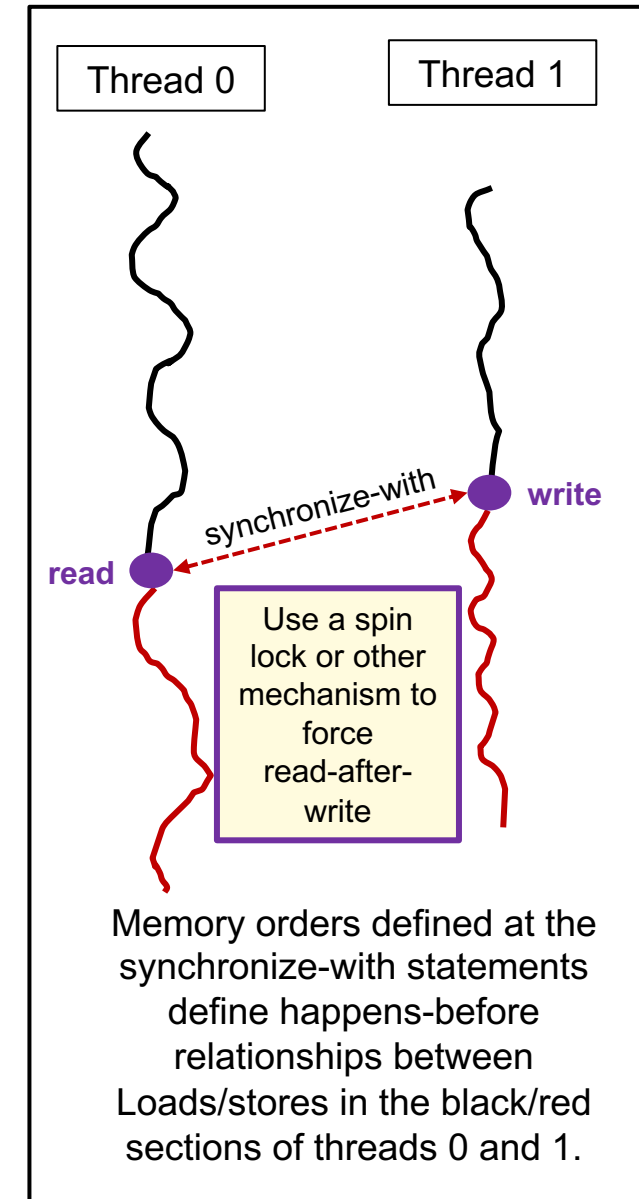
We need to enforce ordering constraints between the concurrent threads ... we need to consider the memory model and put the right synchronization constructs in place.

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.



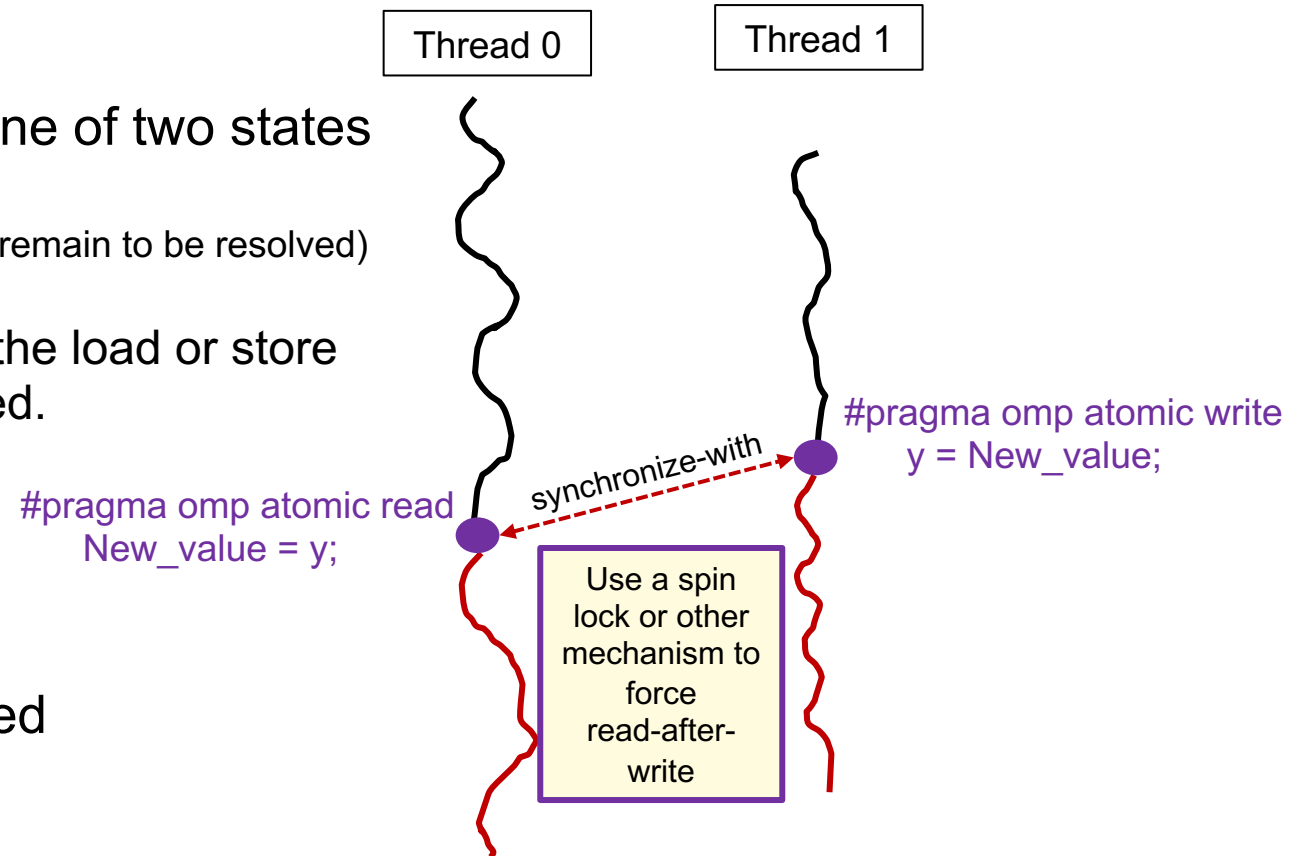
Atomic Operations and Synchronized-with

- 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).

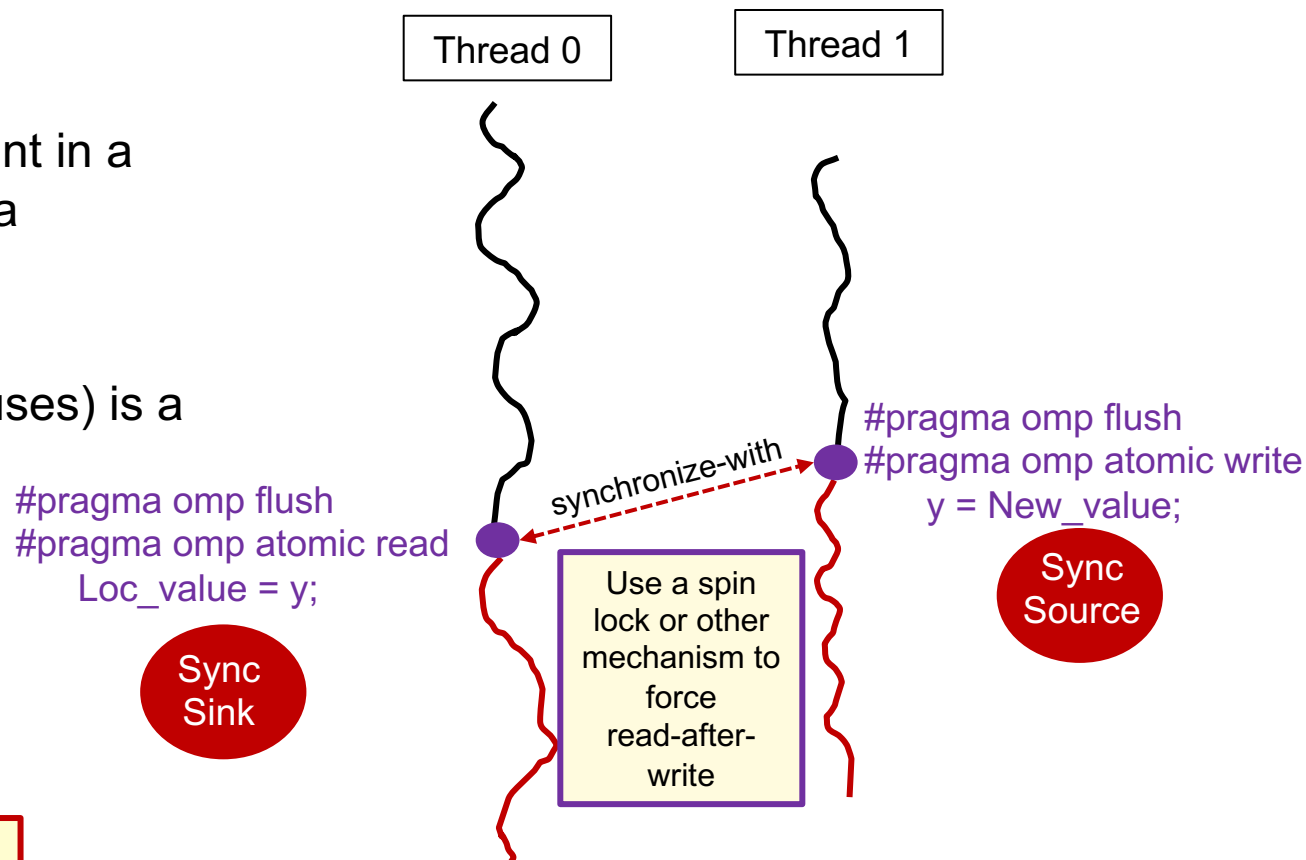


Memory orders

- 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

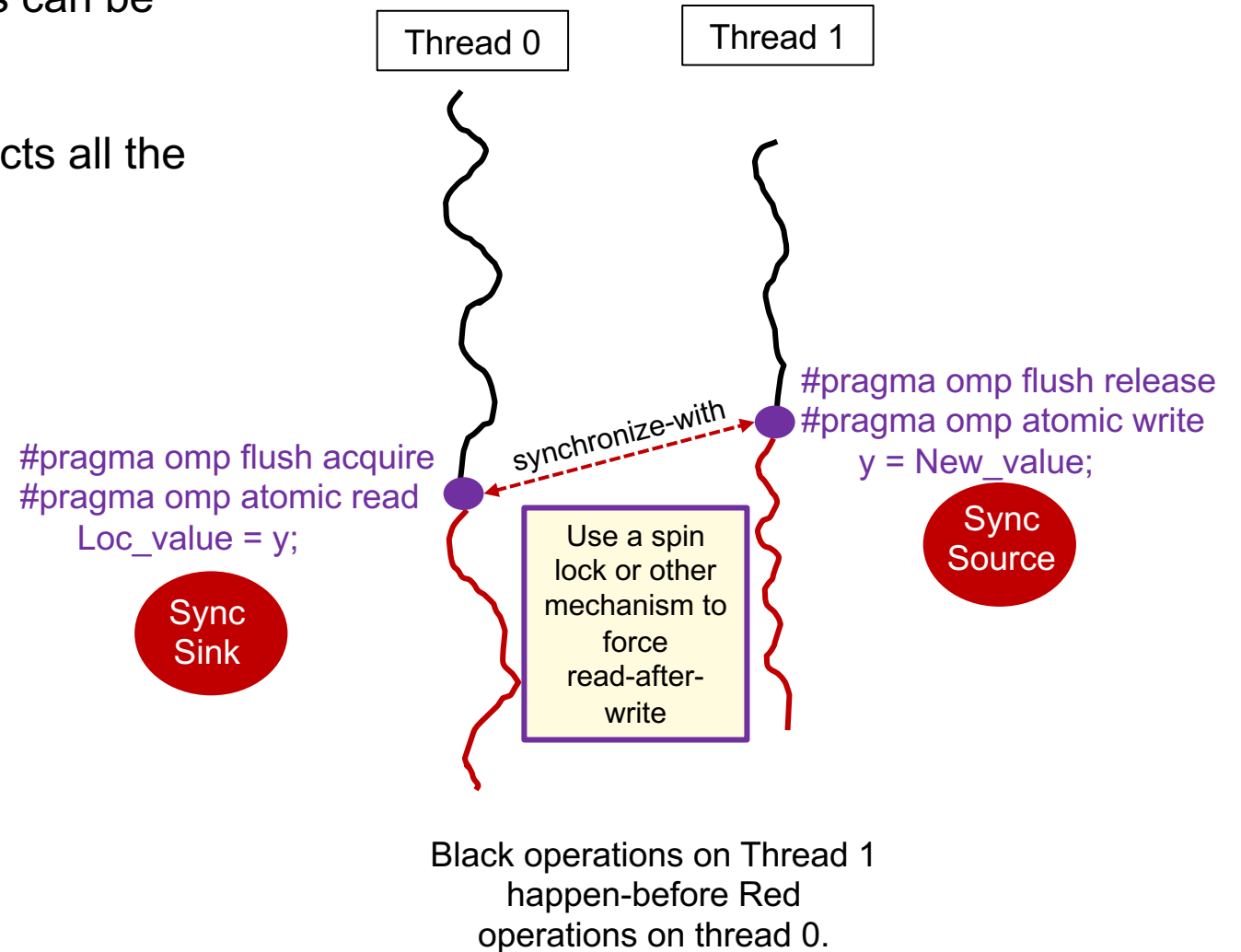
*all memory
operations stay
below the line*

- 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.



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.

`#pragma omp atomic read acquire`
`Loc_value = y;`

`#pragma omp atomic write release`
`y = New_value;`

Sync
Sink

synchronize-with

Use a spin
lock or other
mechanism to
force
read-after-
write

Sync
Source

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;
}
```

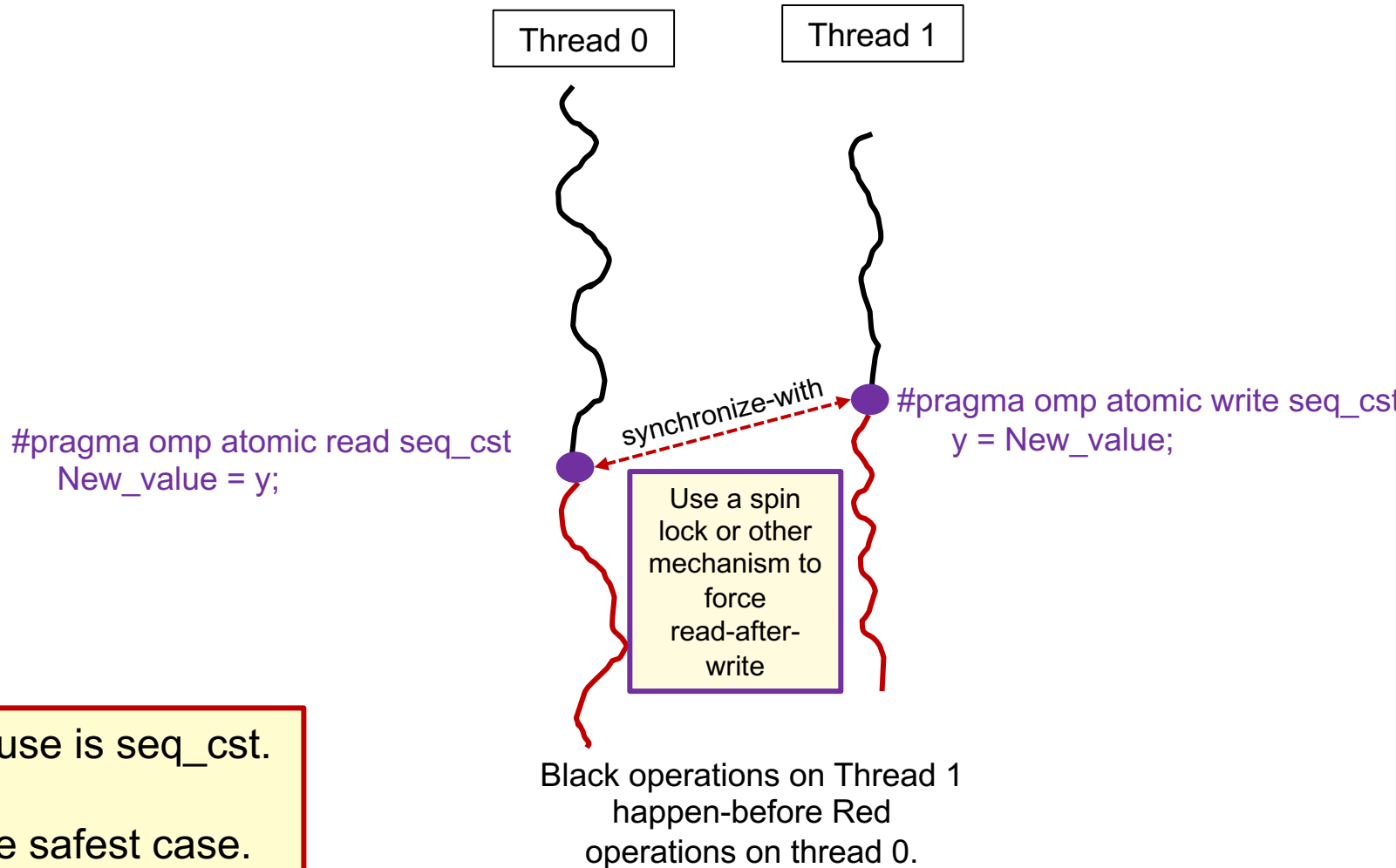
Other Memory orders in OpenMP

- 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.

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 multiple ways**. Try to minimize the time to generate the histogram.
- Time ONLY the assignment to the histogram. Can you beat the sequential time?

```
#define      num_trials    1000000 // number of x values
#define      num_buckets   50      // number of buckets in hitogram
static long xlow          = 0.0;   // low end of x range
static long xhi           = 100.0; // High end of x range

int main (){
    double x[num_trials]; // array used to assign counters in the histogram
    long   hist[num_buckets]; // the histogram
    double bucket_width; // the width of each bucket in the histogram
    double time;

    seed(xlow, xhi); // seed the random number generator over range of x
    bucket_width = (xhi-xlow)/(double)num_buckets;

    // fill the array. << code not shown >>

    // initialize the histogram << code not shown >>

    // Assign x values to the right histogram bucket
    time = omp_get_wtime();
    for(int i=0;i<num_trials;i++){

        long ival = (long) (x[i] - xlow)/bucket_width;

        hist[ival]++;

    }

    time = omp_get_wtime() - time;

    // compute statistics and output results << code not shown >>
    return 0;
}
```

Only focus
on this part of
the program

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]);
}
```

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

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

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

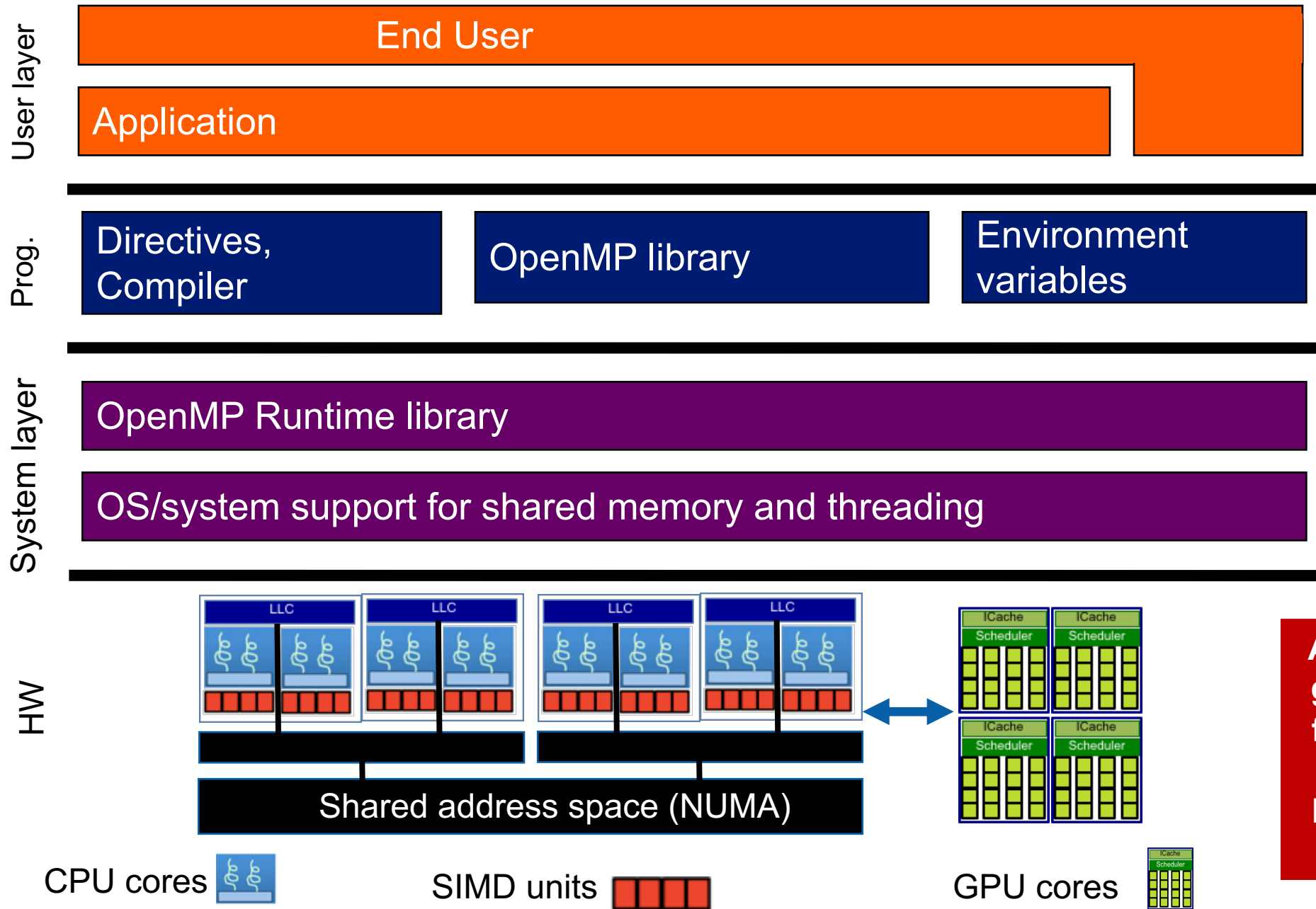
1000000 random values in X sorted into 50 bins. Four threads on a dual core Apple laptop (Macbook air ... 2.2 Ghz Intel Core i7 with 8 GB memory) and the gcc version 9.1. Times are for the above loop only (we do not time set-up for locks, destruction of locks or anything else)

- A Very Brief OpenMP Review
- Irregular Parallelism and Tasks
- OpenMP and Real Hardware: Dealing with NUMA
- Synchronization and the OpenMP Memory Model

➡ • GPU Programming with OpenMP

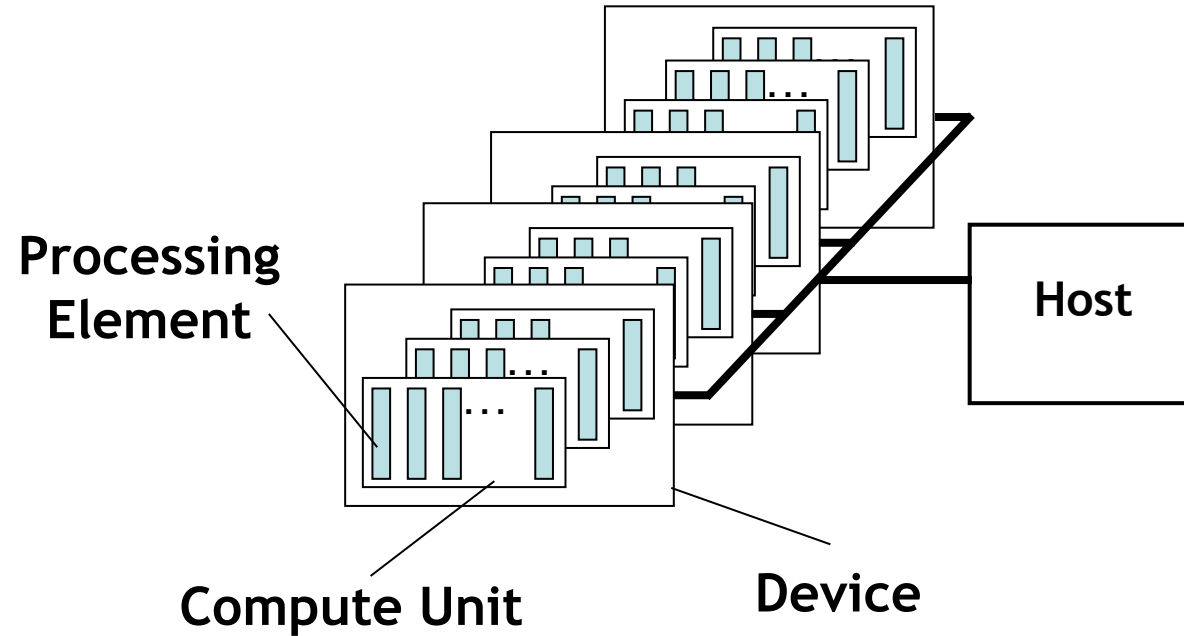
- Recap

OpenMP Basic Definitions: Basic Solution Stack



At last we get to the full set of on-node hardware options

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

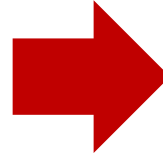
Traditional Loop based vector addition (vadd)

```
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];
}

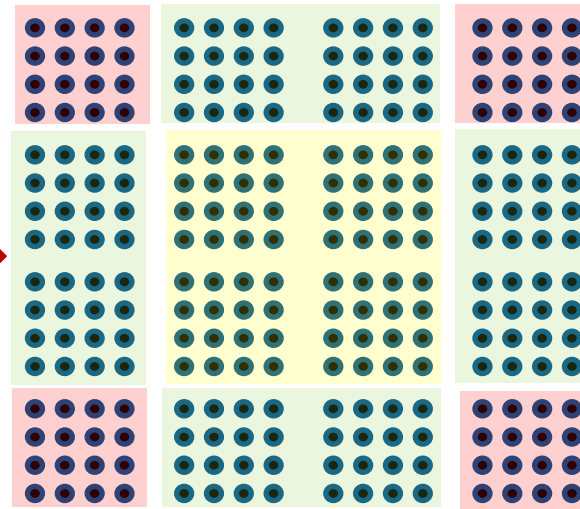
int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a, sizeof(float) * N);
    // ... allocate other arrays (b and c)
    // and fill with data

    // 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.



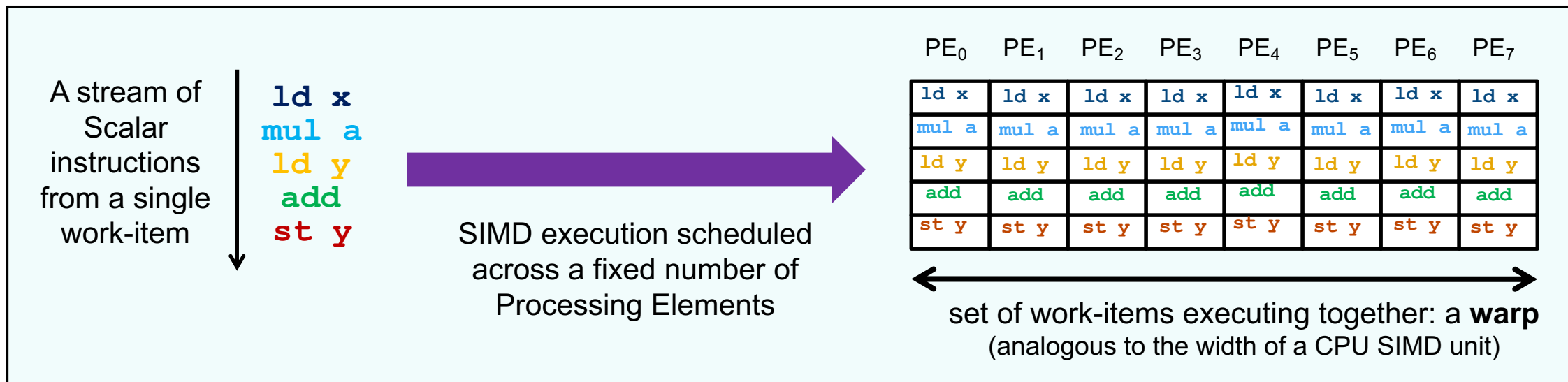
3. Map data structures onto the same index space

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

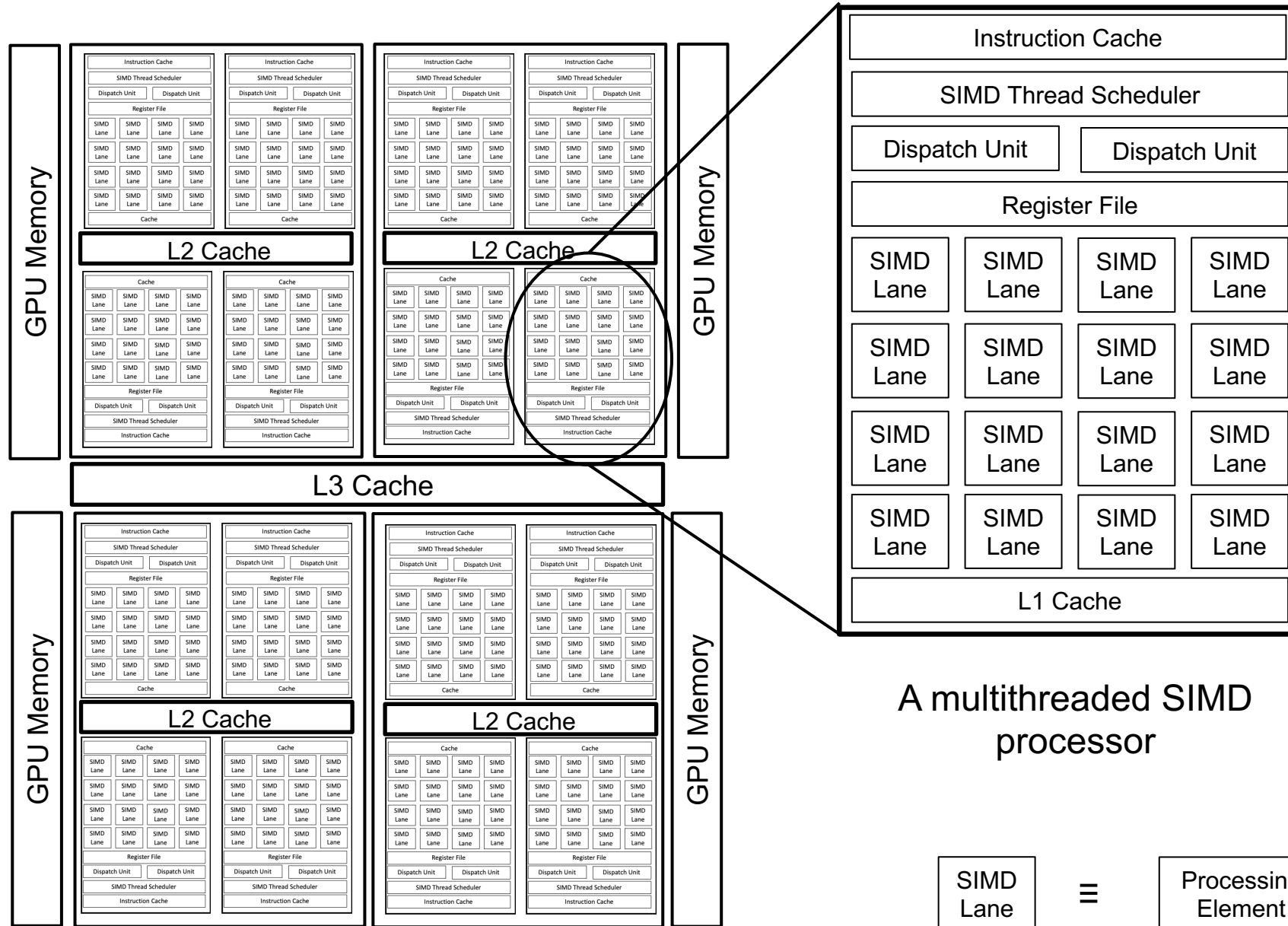


GPU nomenclature is really messed up. (sorry about that ... we tried to unify around OpenCL but failed).

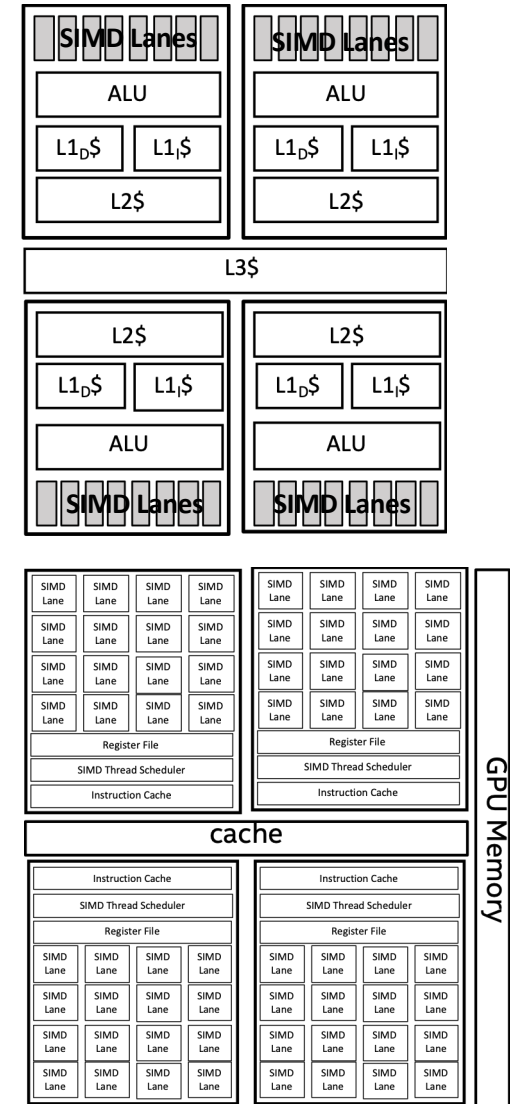
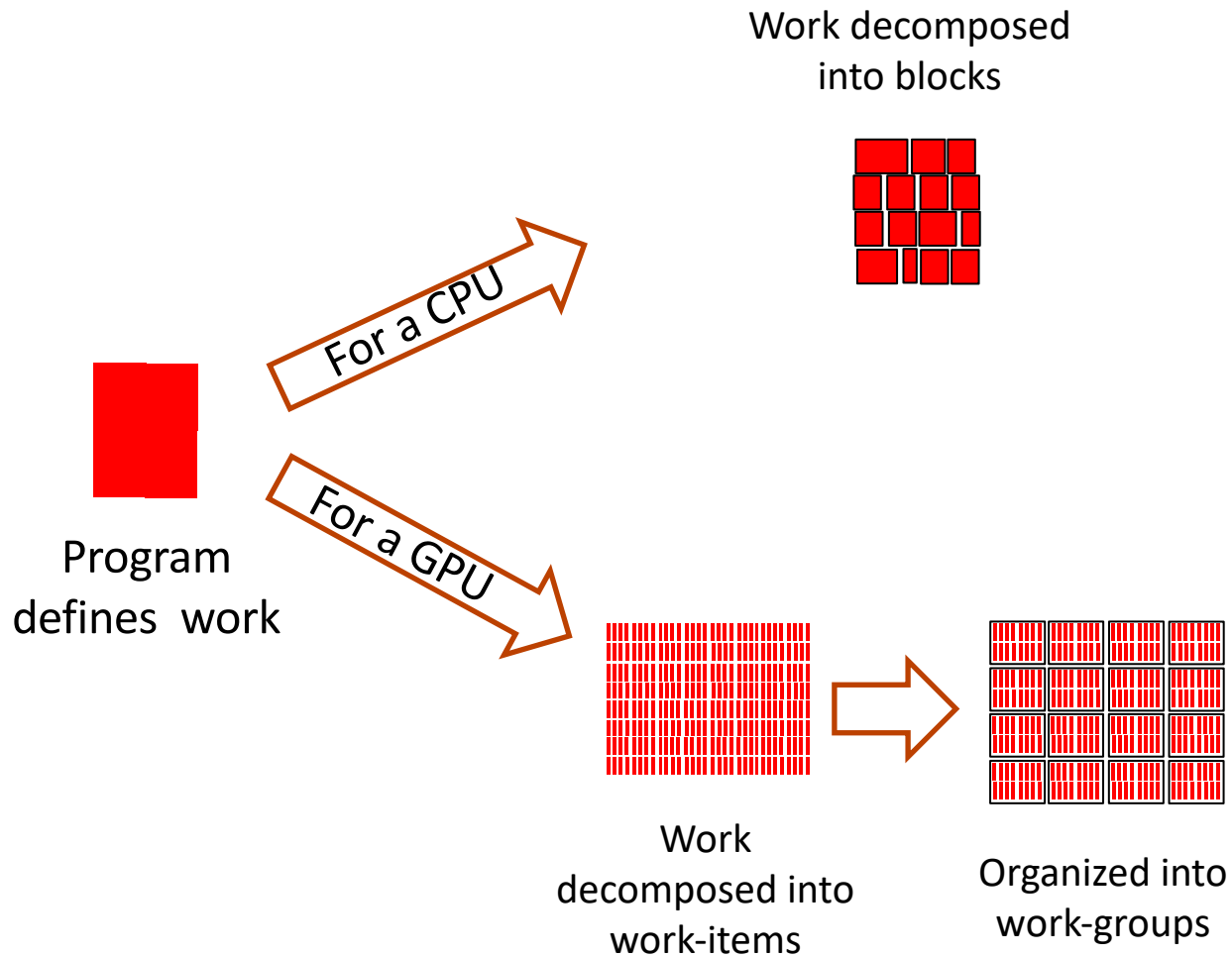
Instruction stream at finest grain	Work-item, CUDA Thread
Blocks for scheduling work-items	work-group, thread block
Execution width for work-items	Subgroup, warp
Finest grained processing element (PE) in a GPU	SIMD Lane, Processing Element, CUDA Core
Block of PEs driven by a single Instruction sequencer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

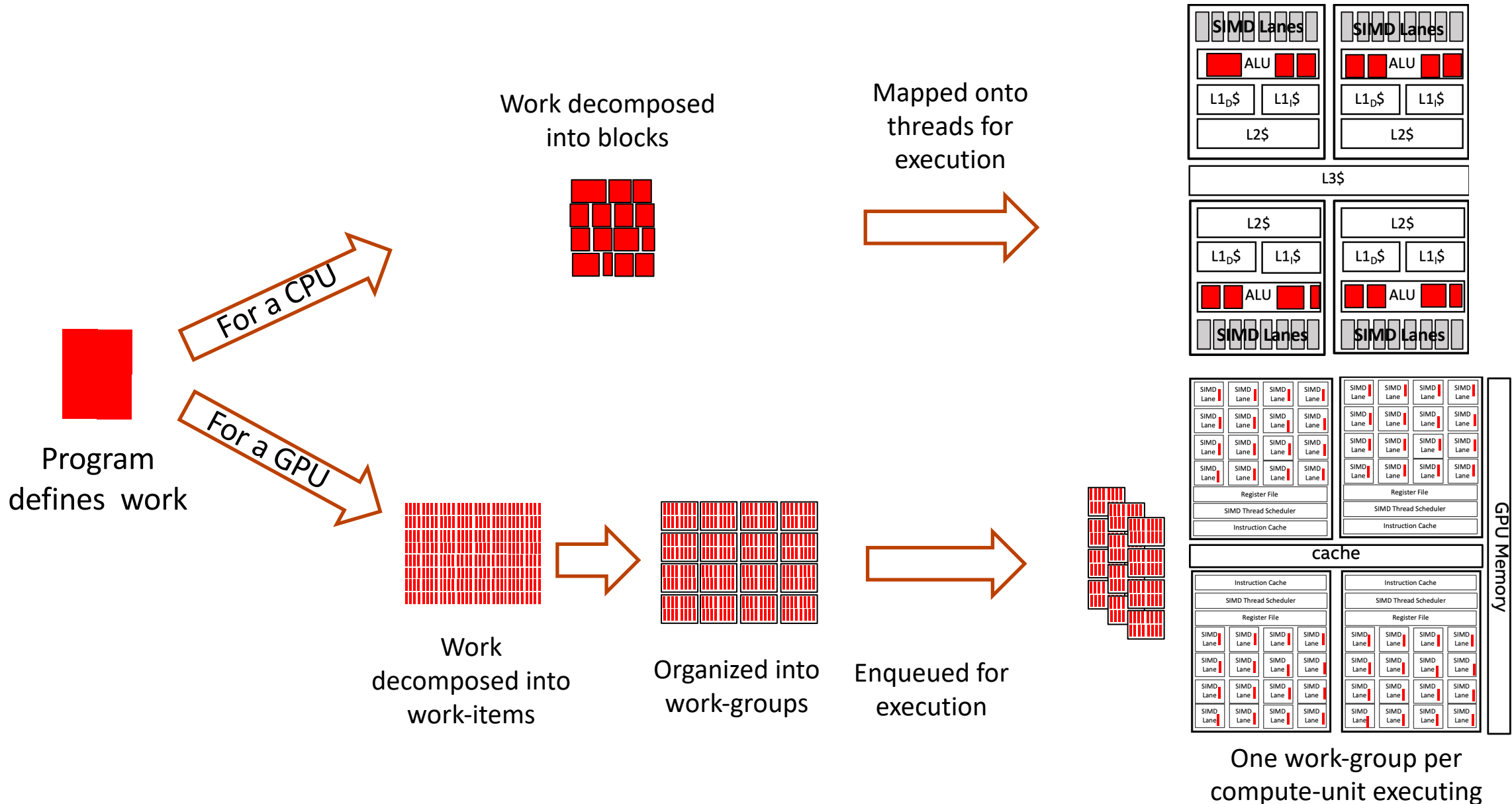


Executing a program on CPUs and GPUs



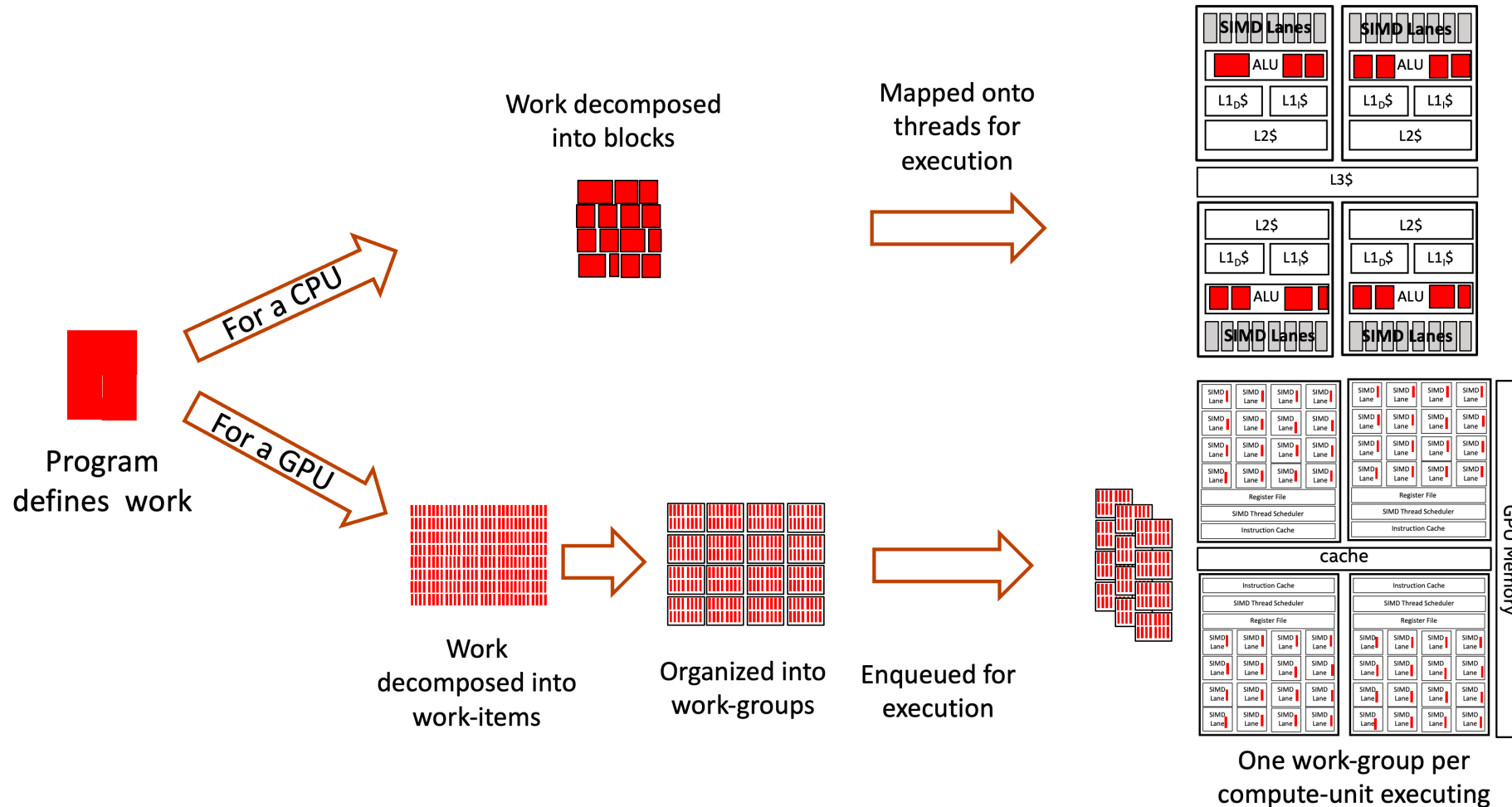
One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

Executing a program on CPUs and GPUs



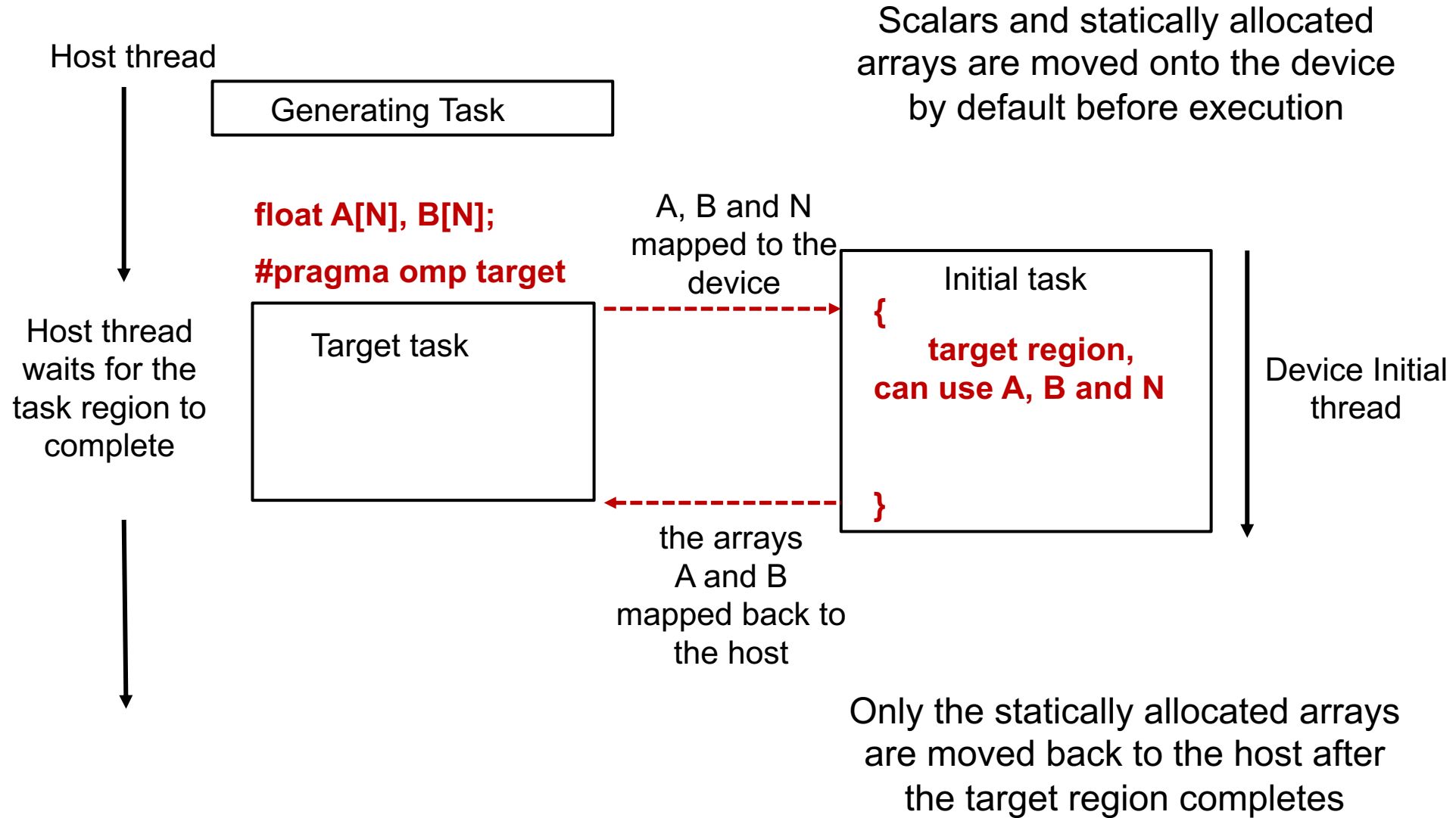
For a CPU, the threads are all active and able to make forward progress.

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



Default Data Sharing: example

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

1. Variables created in host memory.

```
#pragma omp target
```

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

```
{
```

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

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

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

4. Execution on the device.

```
    }
```

```
} // end of target region
```

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;

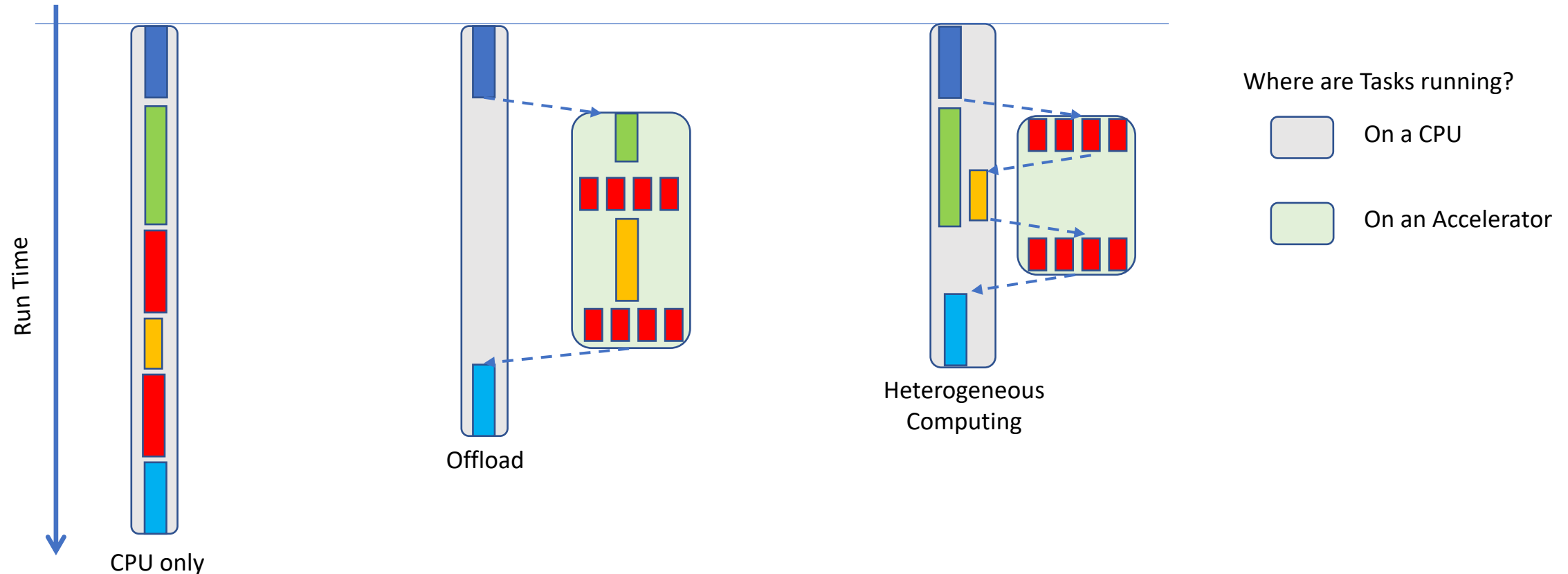
    // fill the arrays
    #pragma omp parallel for
    for (int i=0; i<N; i++){
        a[i] = (float)i;
        b[i] = 2.0*(float)i;
        c[i] = 0.0;
        res[i] = i + 2*i;
    }

    // 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:

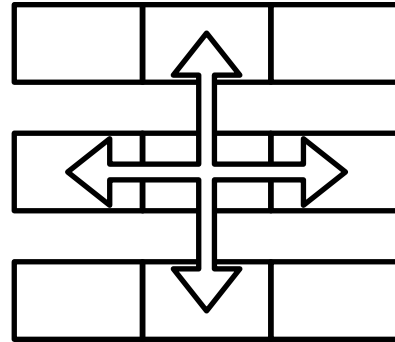
$$\frac{\partial u}{\partial t} \approx \frac{u(t + 1, x, y) - u(t, x, y)}{dt}$$

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

- 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.

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) {
```

Loop over time steps

```
    for (int i = 0; i < n; ++i) {
```

```
        for (int j = 0; j < n; ++j) {
```

Loop over NxN spatial domain

```
            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);
```

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

```
        }
```

```
    }
```

```
    // Pointer swap to get ready for next step
```

```
    tmp = u;
```

```
    u = u_tmp;
```

```
    u_tmp = tmp;
```

```
}
```

```
}
```

Heat diffusion problem: 5-point stencil code

```

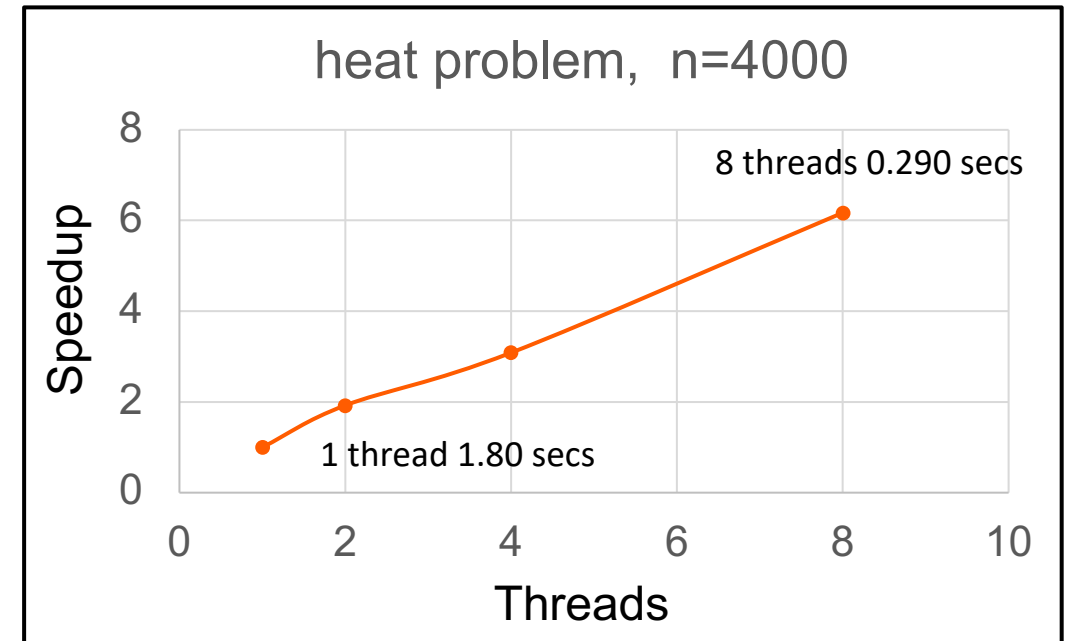
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

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

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

    #pragma omp 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

```

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

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** $(2*N^2)*\text{sizeof}(\text{TYPE})$ 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** $(2*N^2)*\text{sizeof}(\text{TYPE})$ 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.

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)
#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:n*n])

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])
}
```

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			

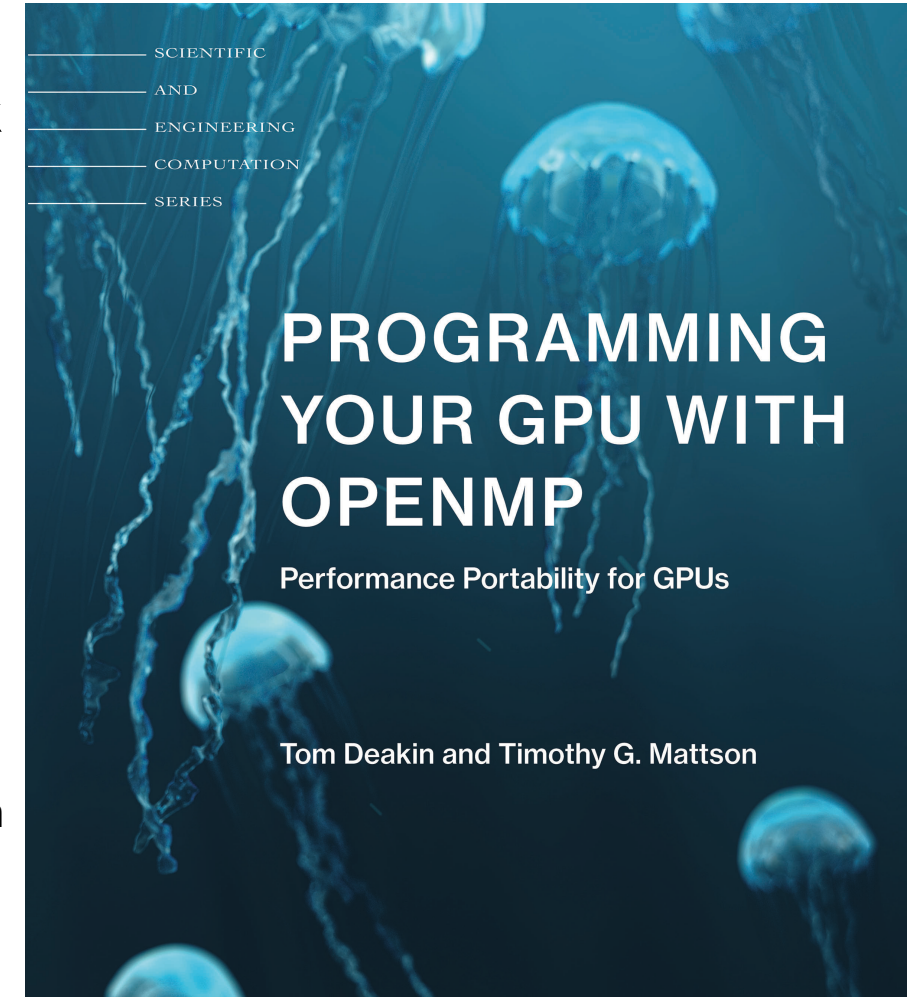
All times in seconds

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

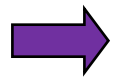
GPU programming with OpenMP

- 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 programming 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

- A Very Brief OpenMP Review
- Irregular Parallelism and Tasks
- OpenMP and Real Hardware: Dealing with NUMA
- Synchronization and the OpenMP Memory Model
- GPU Programming with OpenMP

 • Recap

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

Resources

- www.openmp.org has a wealth of helpful resources

OpenMP
The OpenMP API specification for parallel programming

Home Specifications Community Resources News & Events About

Specifications

Home > Specifications

OpenMP 5.2 Specification

- OpenMP API 5.2 Specification – Nov 2021
 - Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.2 Reference Guide (English) (Japanese)
- OpenMP API 5.2 Supplementary Source Code
- OpenMP API 5.2 Examples – April 2022
 - Softcover Book on Amazon
- OpenMP API 5.2 Stack Overflow

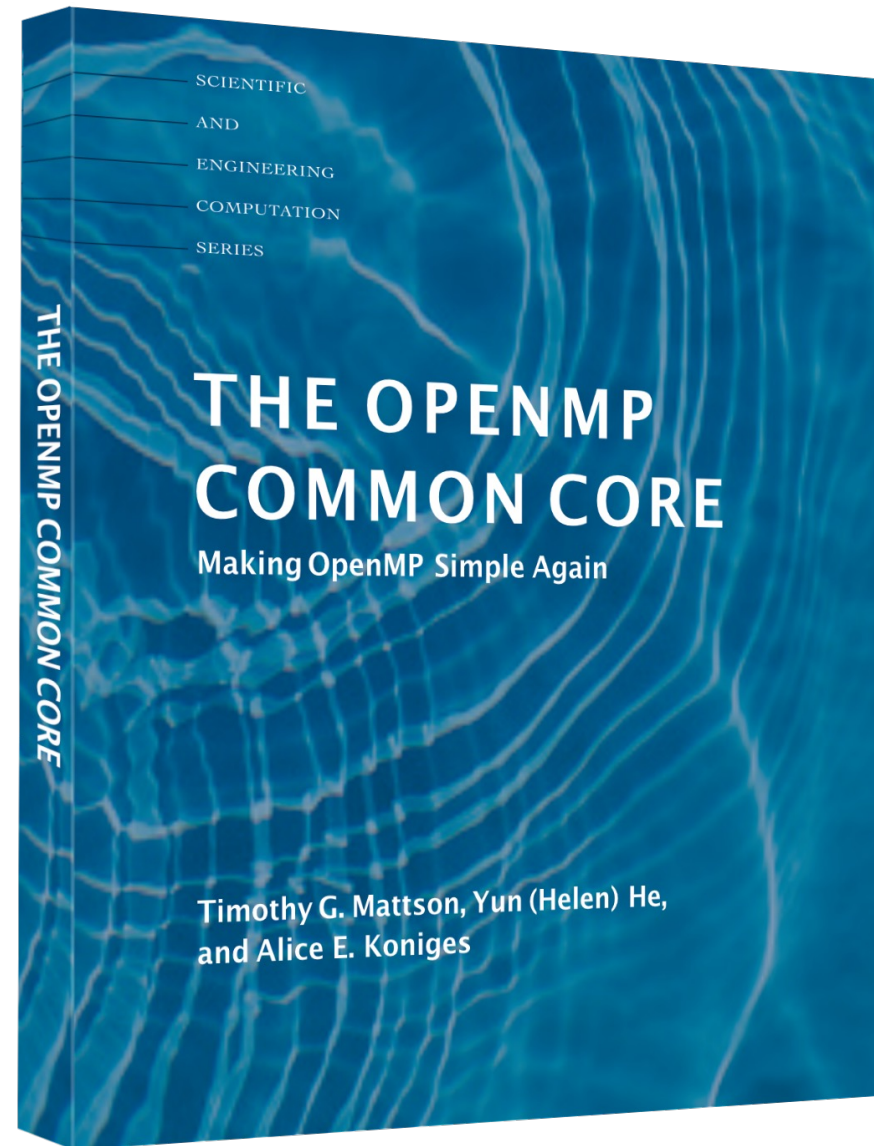
OpenMP 5.1 Specification

- OpenMP API 5.1 Specification – Nov 2020
 - HTML Version
 - Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.1 Reference Guide
- OpenMP API 5.1 Supplementary Source Code
- OpenMP API 5.1 Examples – August 2021
- OpenMP API 5.1 Stack Overflow

Including a comprehensive collection of examples of code using the OpenMP constructs

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use
- It's geared towards people learning OpenMP, but as one commentator put it ... **everyone at any skill level should read the memory model chapters.**
- Available from MIT Press



Books about OpenMP

A great book that covers
OpenMP features beyond
OpenMP 2.5

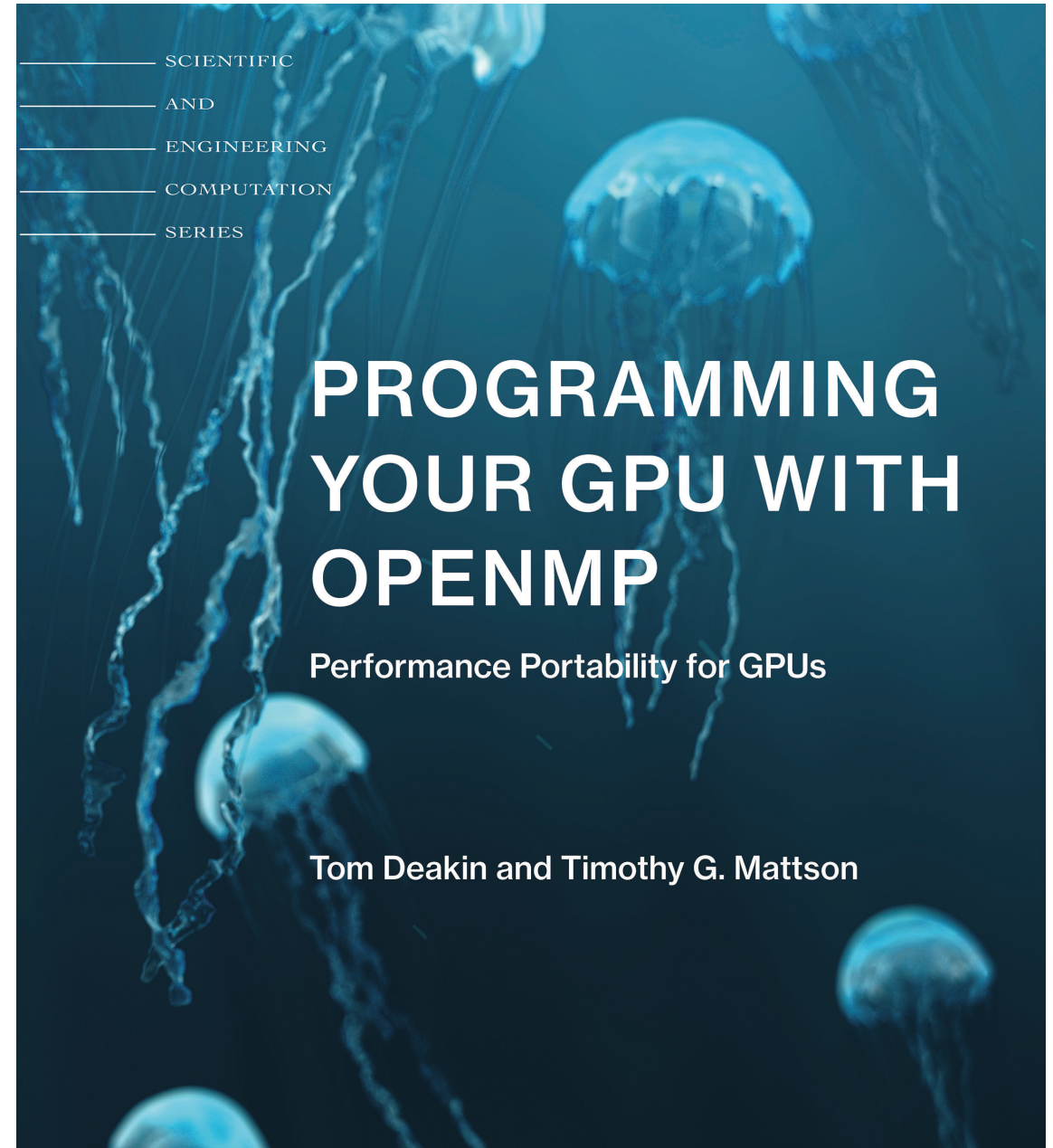


Books about OpenMP

The latest book on OpenMP ...

Came out in early November 2023.

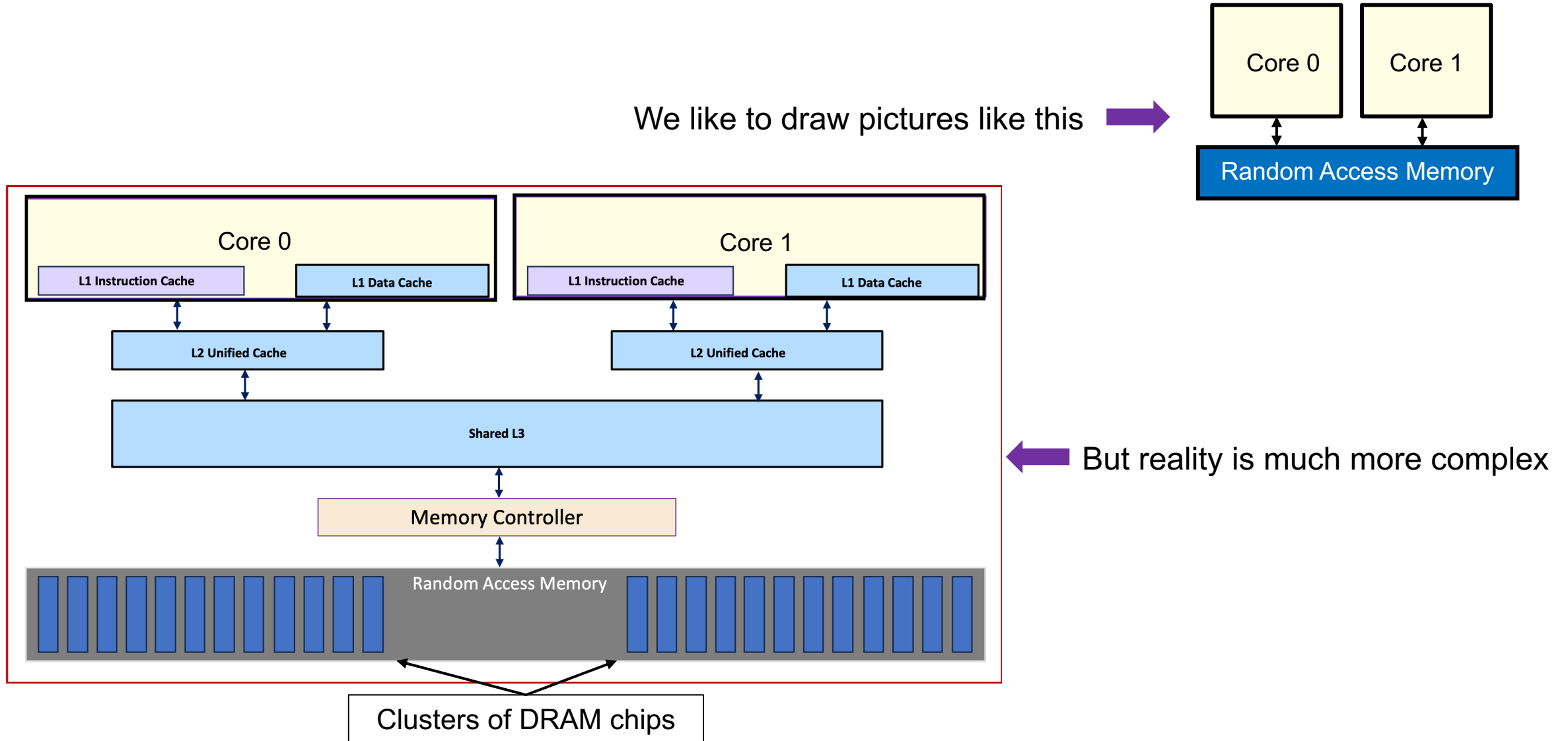
A book about how to use OpenMP to
program a GPU.



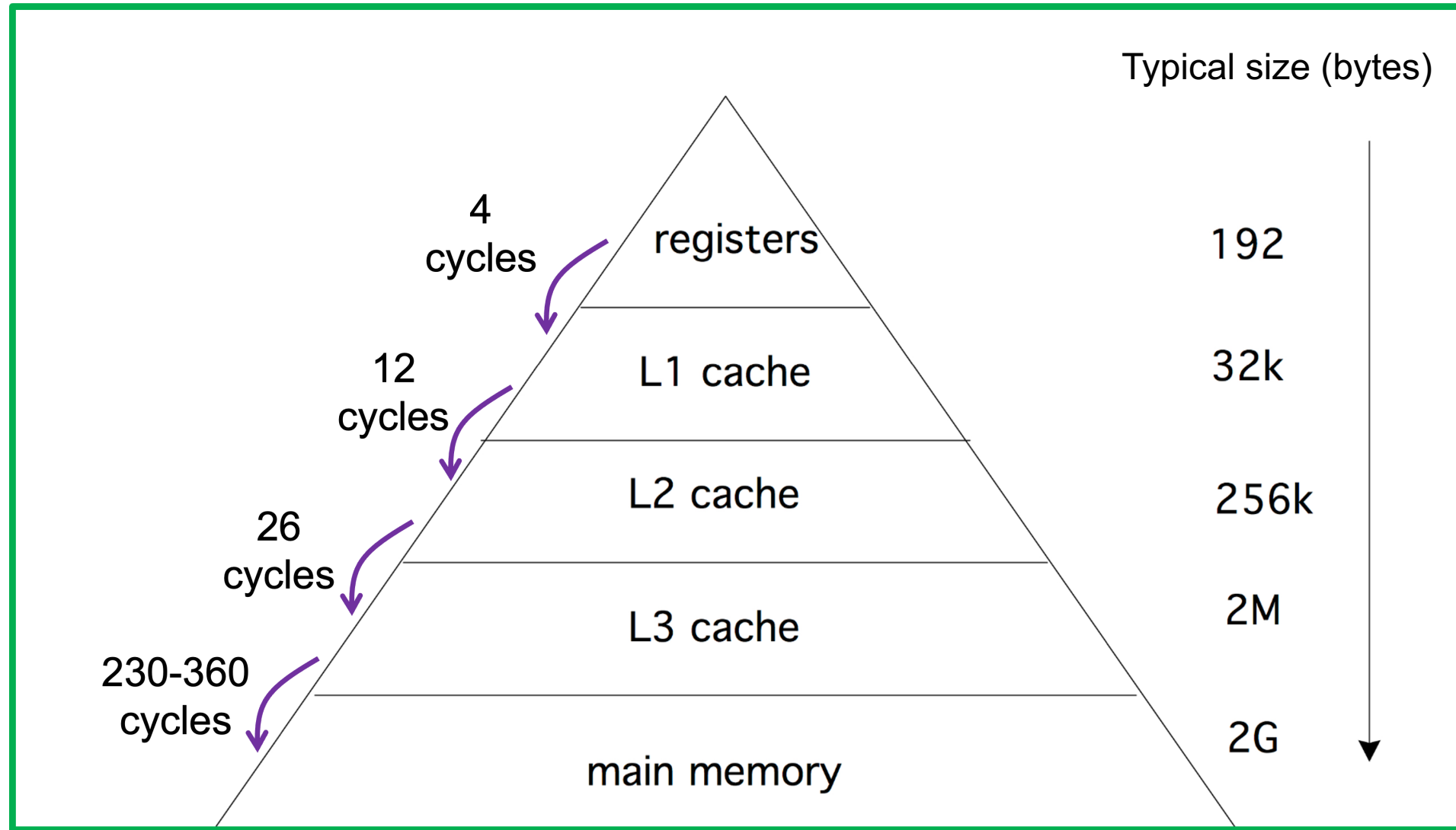
Extra Content

- 
- Memory performance, the TLB, and the tile construct
 - HPCorpus and OpenMP usage
 - Introduction to the SIMD constructs
 - NUMA: additional content

SMP is a lie ... The memory is Non-Uniform

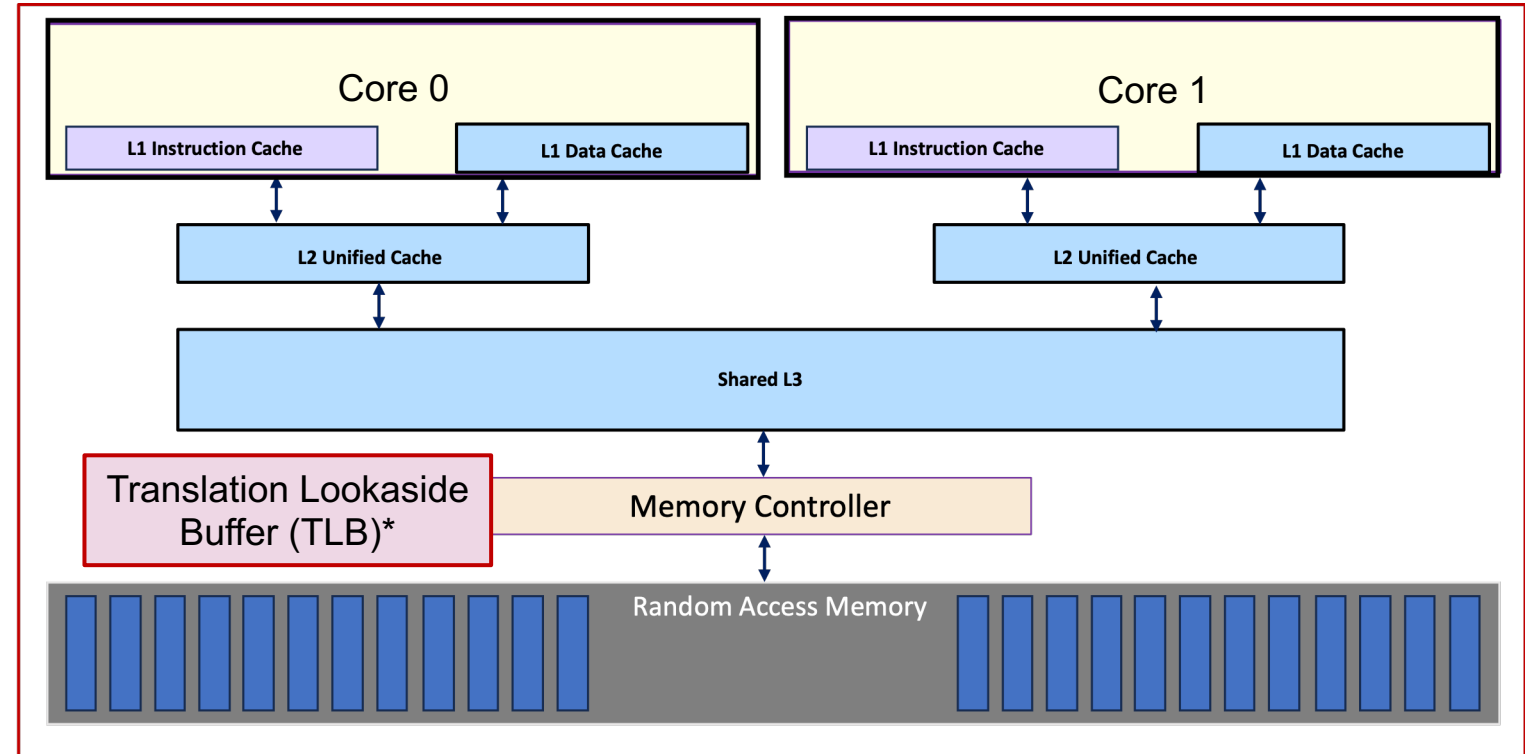


Latencies across the Memory Hierarchy



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.

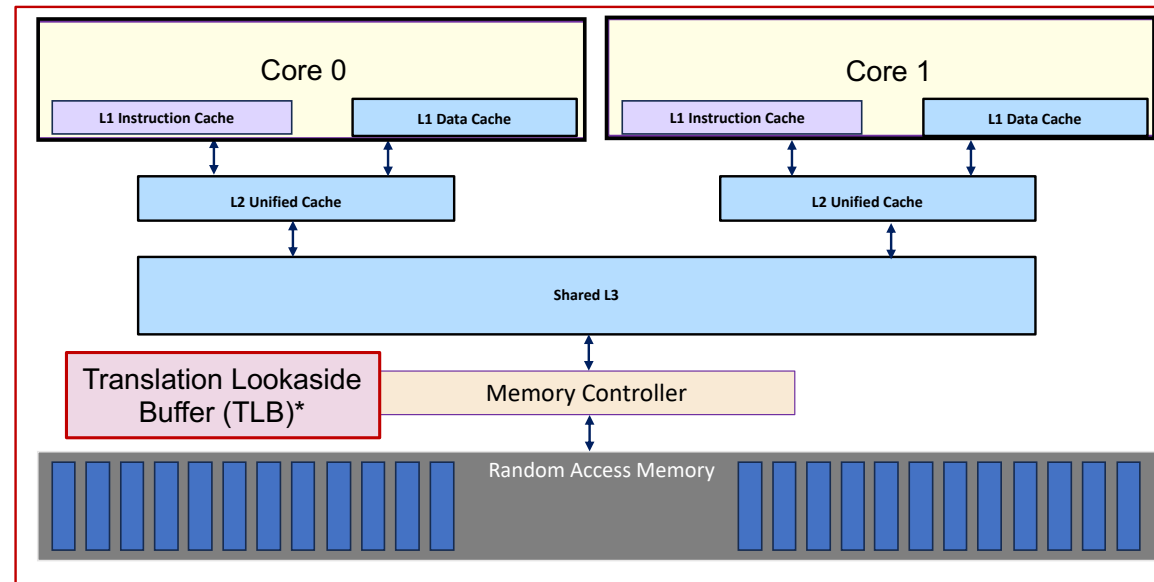
$$A \in \mathbb{R}^{N \times N}$$

$$B \in \mathbb{R}^{N \times N}$$

```
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

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

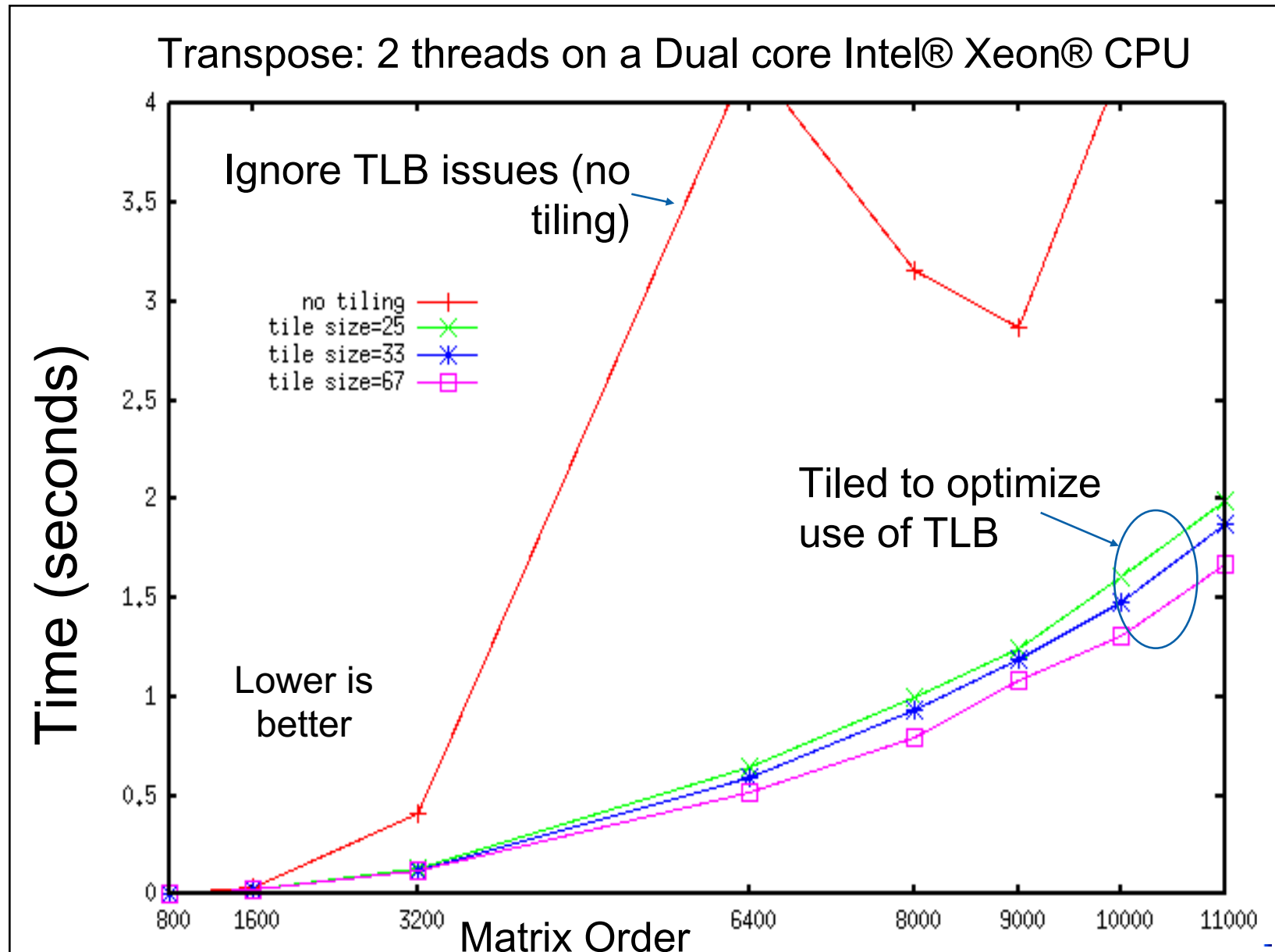
- Solution ... break the loops into blocks so we reduce the number of page faults

```
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?



Optimizing Matrix Transpose for the TLB

- OpenMP version 5.1 added a construct to do this sort of tiling ... i.e., the code in these two boxes are equivalent

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



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

1st loop (i)
tile size

2nd loop (j)
tile size

OpenMP allows for partial tiles, so $(N \% \text{tile_size})$ can be non-zero
The sizes clause is required

Note: tile_size must be known at compile time, so use a literal or a #define statement

Optimizing Matrix Transpose for the TLB

- The parallel for applies to the outer most loop ... i.e., the code in these two boxes are equivalent

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



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

OpenMP allows for partial tiles, so $(N \% \text{tile_size})$ can be non-zero

Note: tile_size must be known at compile time, so use a literal or a #define statement

Extra Content

- Memory performance, the TLB, and the tile construct



- HPCorpus and OpenMP usage
- Introduction to the SIMD constructs
- NUMA: additional content

What do people use from OpenMP

- The fact is ... parallel programming language designers have no idea which features from their languages are actually used.
 - Design is largely driven by anecdote ... which is undependable and based on “who you know”.
- Wouldn't it be nice if we had systematic studies of parallel programs to measure exactly which features are used?
 - Unfortunately, collecting the data to support such a study would be difficult/expensive.

Enter Stage Left ...

- Gal Oren at Technion and Yuval Pinter at Ben-Gurion University (and their students).
- Research on AI used in Machine programming ... in particular AI to help generate OpenMP programs.
- Central hypothesis: Its better to have a small Large-Language-Model trained on data that matches the problem domain ... Scope is all you need to get more value from your large language model
- To explore this hypothesis, they collected data restricted to HPC code to be used to train HPC specific large language models. They call this data set **HPCorpus**.

-
- Tal Kadosh, Niranjana Hasabnis, Vy A. Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, Timothy Mattson, and Gal Oren . "**Scope is all you need: Transforming LLMs for HPC Code.**" *arXiv preprint arXiv:2308.09440* (2023). **IXPUG'23**
 - Tal Kadosh, Nadav Schneider, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. "**Advising OpenMP Parallelization via a Graph-Based Approach with Transformers.**" *arXiv preprint arXiv:2305.11999* (2023). **IWOMP'23**
 - Gal Oren. "**Unlocking the Potential of Large Language Models for High Performance Computing Code.**" *AI Assisted Software Development for HPC* (2023). <https://www.youtube.com/watch?v=iqk-rd2Txdw> , **AI4DEV'23**,
 - Nadav Schneider, Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. "**MPI-RICAL: Data-Driven MPI Distributed Parallelism Assistance with Transformers.**" *AI Assisted Software Development for HPC* (2023). **AI4DEV'23**
 - Re'em Harel, Yuval Pinter, and Gal Oren. "**Learning to parallelize in a shared-memory environment with transformers.**" *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (2023). **PPoPP'23**

HPCorpus data collection

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



```
1 WITH selected_repos as (  
2   SELECT f.id, f.repo_name as repo_name, f.ref as  
   ↪ ref, f.path as path  
3   FROM `bigquery-public-data.github_repos.files` as  
   ↪ f  
4   JOIN `bigquery-public-data.github_repos.licenses`  
   ↪ as l on l.repo_name = f.repo_name  
5   ),  
6   deduped_files as (  
7     SELECT f.id, MIN(f.repo_name) as repo_name,  
   ↪ MIN(f.ref) as ref, MIN(f.path) as path  
8     FROM selected_repos as f  
9     GROUP BY f.id  
10  )  
11  SELECT  
12    f.repo_name, f.ref, f.path, c.copies, c.content,  
13    FROM deduped_files as f  
14    JOIN `bigquery-public-data.github_repos.contents`  
   ↪ as c on f.id = c.id  
15  WHERE  
16    NOT c.binary  
17    AND (f.path like '%.c' OR f.path like '%.cpp' OR  
   ↪ f.path like '%.f' OR f.path like '%.f90' OR  
   ↪ f.path like '%.f95')
```

Query used with Google BigQuery to collect HPCorpus data

HPCorpus data collection



Scan all C, C++ and Fortran codes
from github with “last updated” dates
between 201

The BIG idea ...

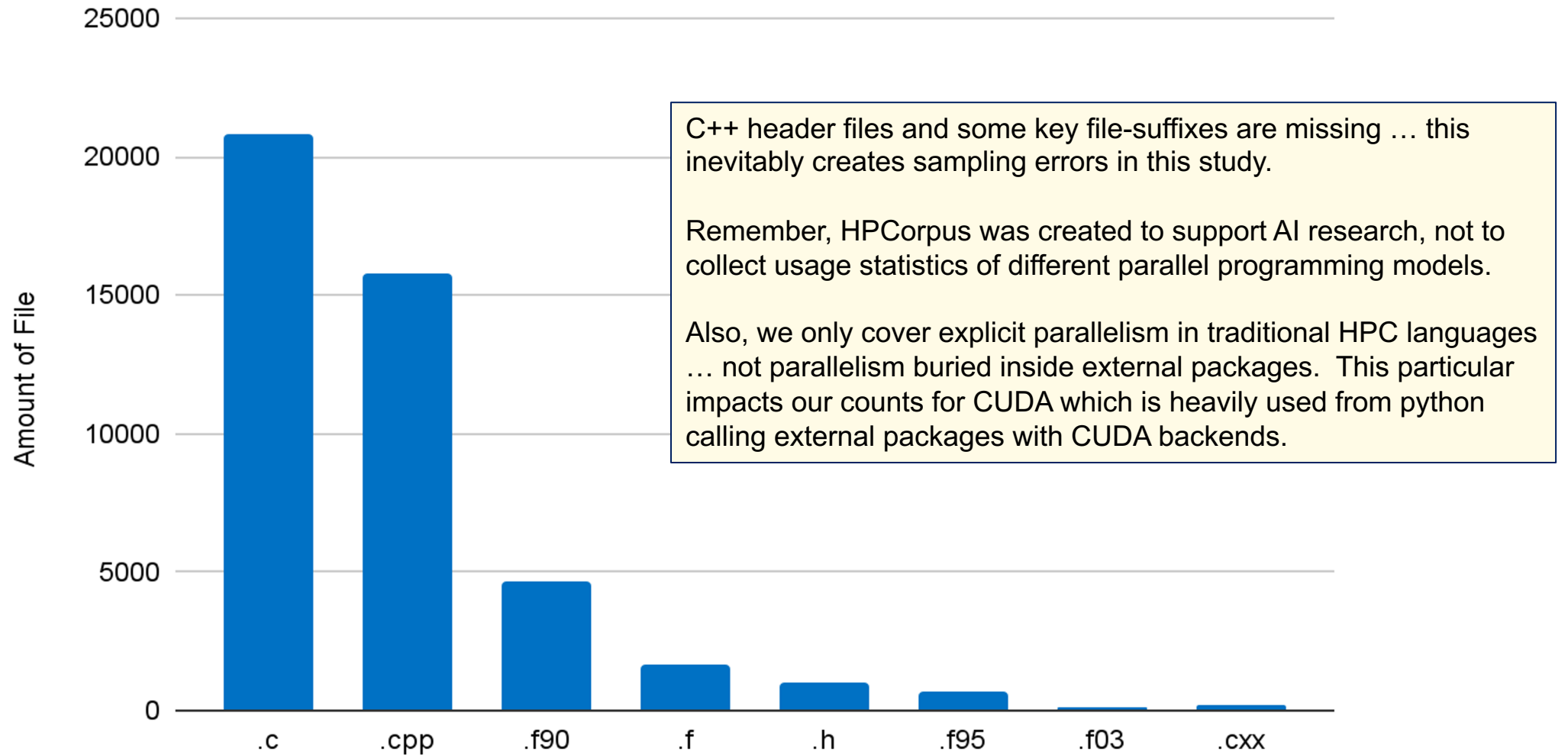
HPCorpus was collected to train AI models for machine
programming, but it can also be used to generate
quantitative data on Programming model usage in HPC

	Repos			
C	144,522			
C++	150,481	26.16	4,735,196	68,233,984
Fortran	3,683	0.68	138,552	359,272

```
1  WITH selected_repos as (  
2  SELECT f.id, f.repo_name as repo_name, f.ref as  
   ↪ ref, f.path as path  
3  FROM `bigquery-public-data.github_repos.files` as  
   ↪ f  
   ↪ JOIN `bigquery-public-data.github_repos.licenses`  
   ↪ ON f.repo_name = f.repo_name  
   ↪ JOIN `bigquery-public-data.github_repos.contents`  
   ↪ ON f.id = c.id  
   ↪ WHERE  
   ↪ NOT c.binary  
   ↪ AND (f.path like '%.c' OR f.path like '%.cpp' OR  
   ↪ f.path like '%.f' OR f.path like '%.f90' OR  
   ↪ f.path like '%.f95')  
14  
15  
16  
17
```

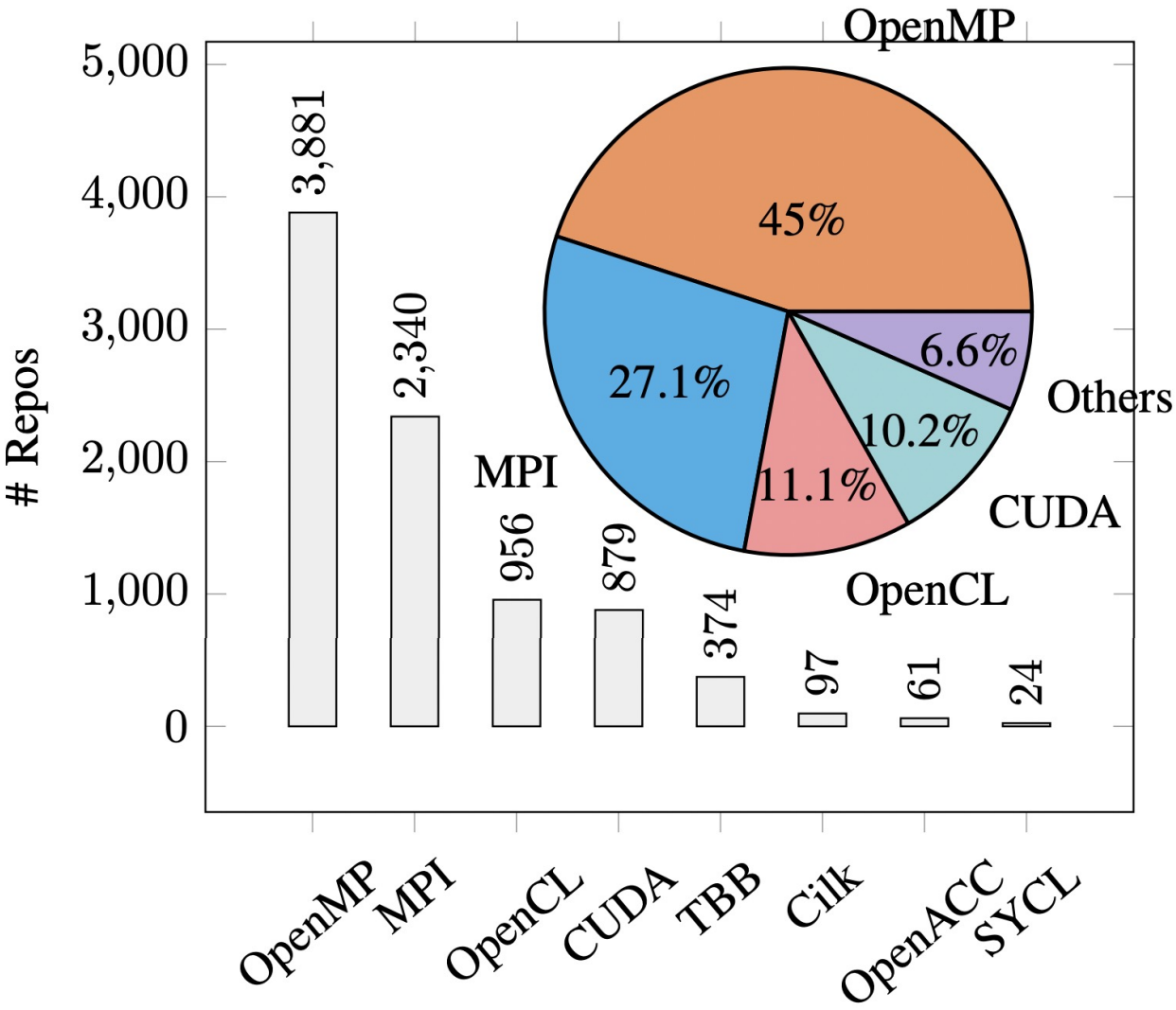
Query used with Google BigQuery to collect HPCorpus data

Language/file-type breakdown for files in HPCorpus



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

Programming models usage 2013 to 2023



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

Aggregate numbers over all repositories from 2013 to 2023

What are people actually using from OpenMP

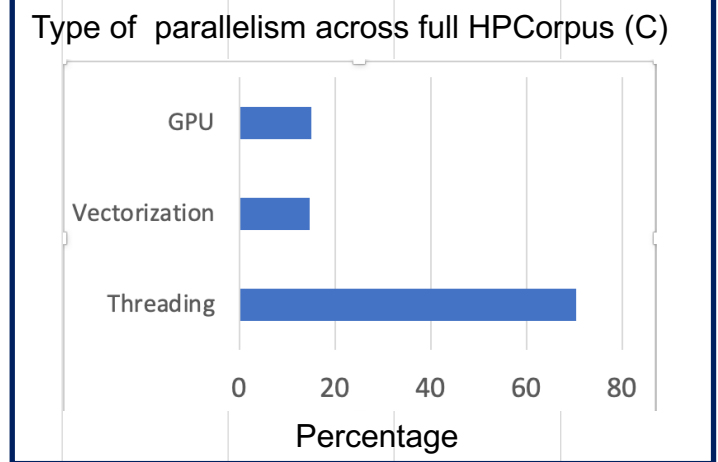
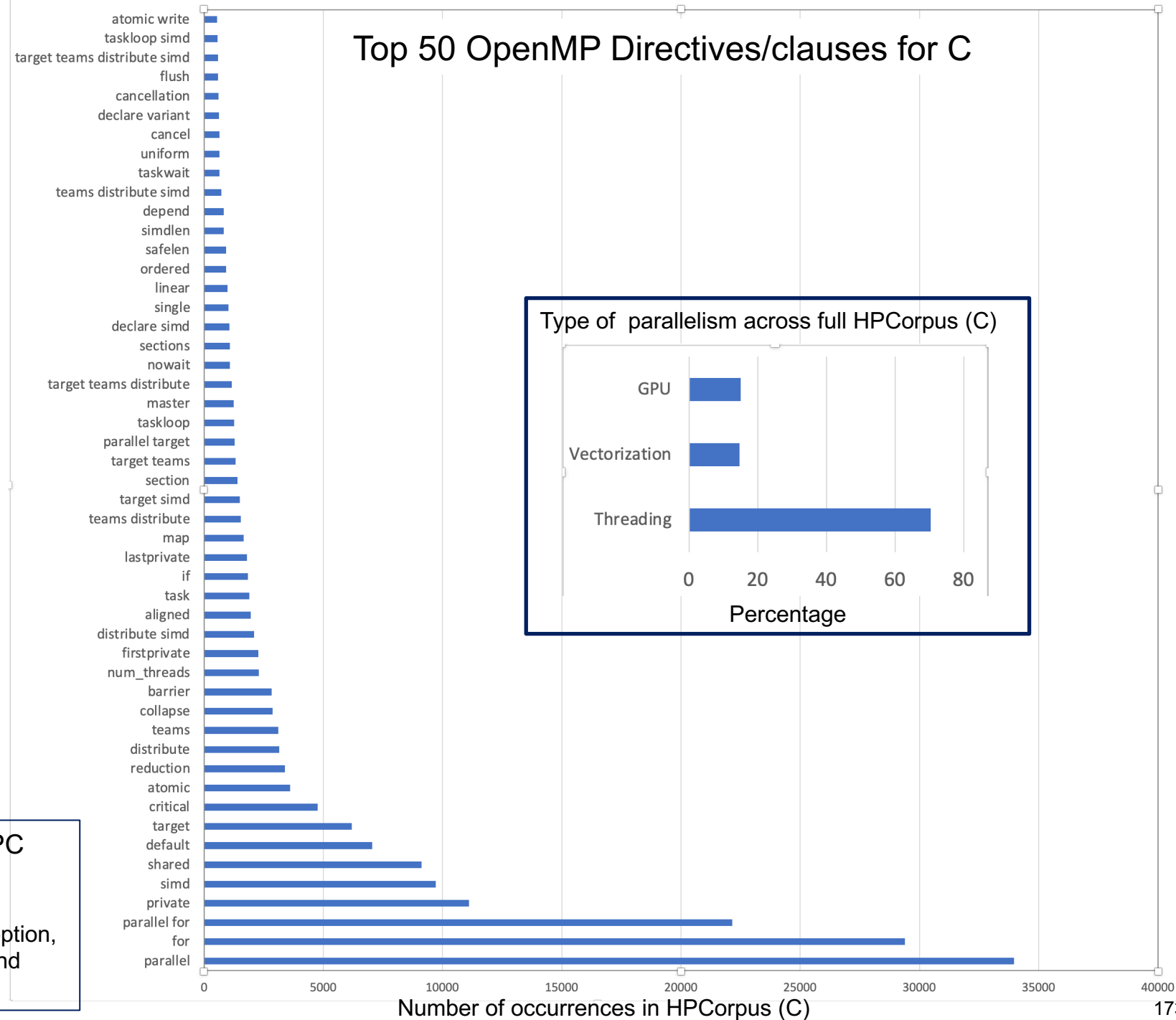
With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

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.

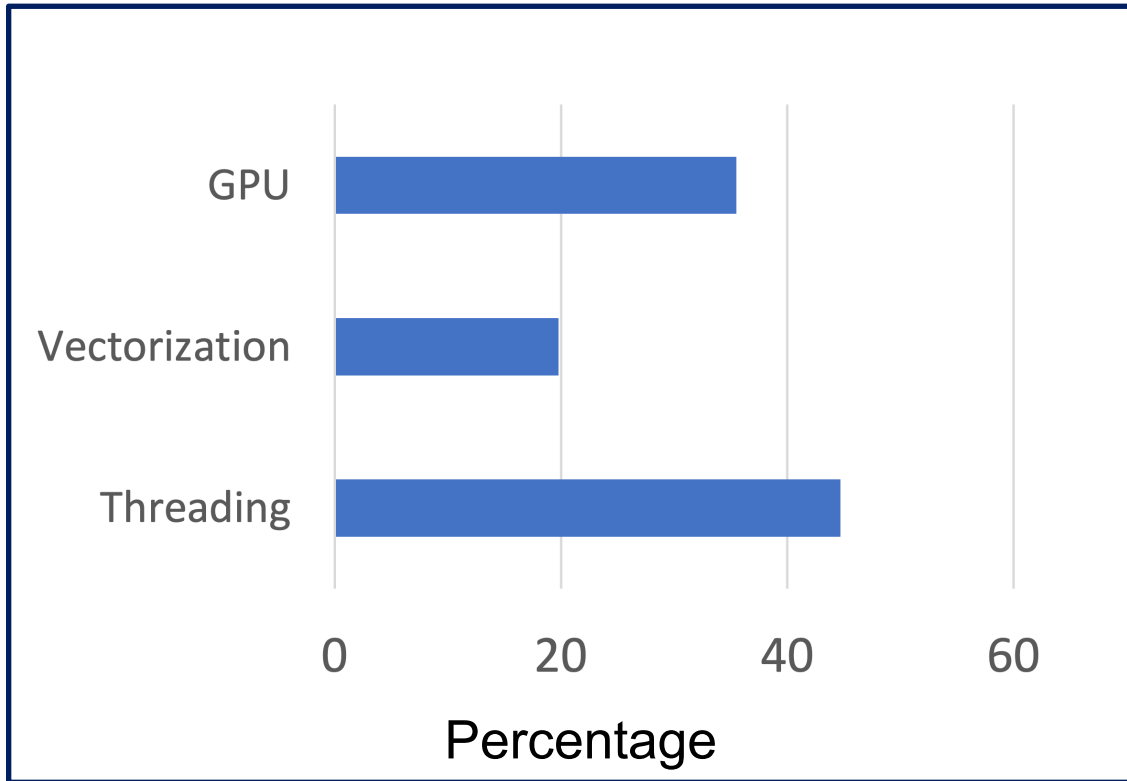
Quantifying OpenMP: Statistical Insights into Usage and Adoption, Tal Kadosh, Niranjana Hasabnis, Tim Mattson, Yuval Pinter, and Gal Oren, IEEE HPEC 2023

Top 50 OpenMP Directives/clauses for C

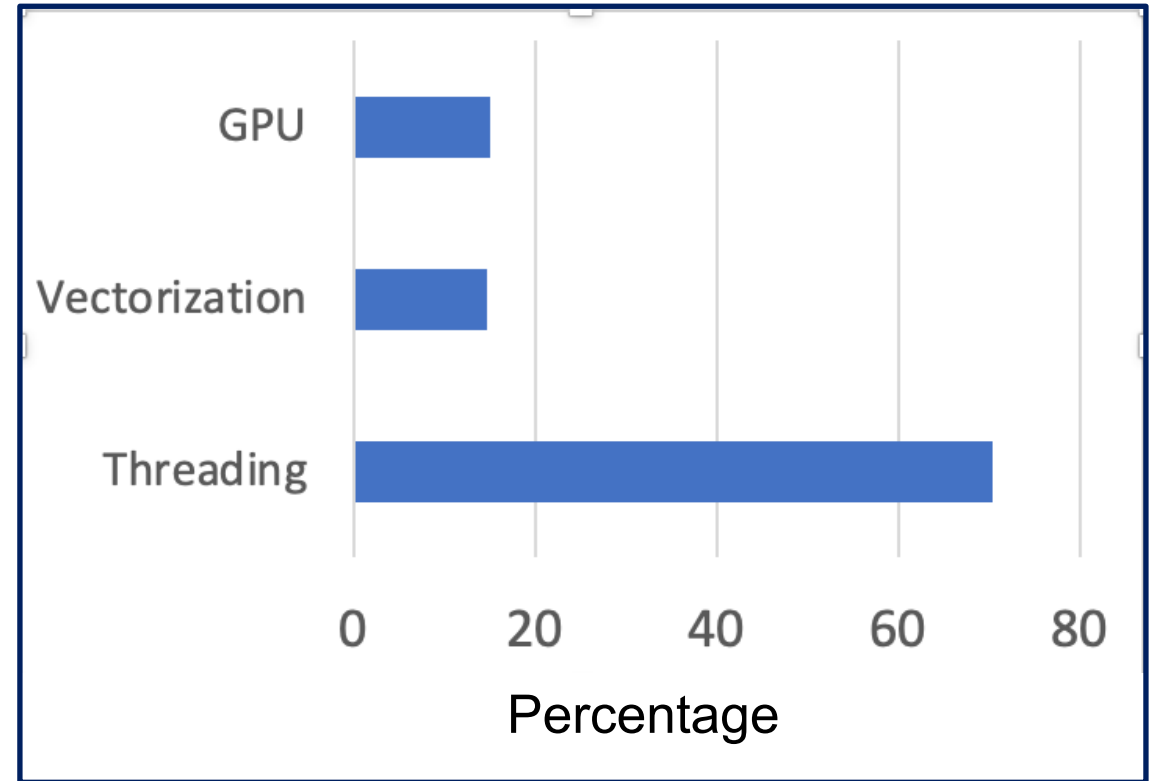


C++ vs C OpenMP programs in HPCorpus

Type of parallelism across full HPCorpus (C++)



Type of parallelism across full HPCorpus (C)



C++ programmers are MUCH more aggressive about the newer types of parallelism in OpenMP

What are people actually using from OpenMP

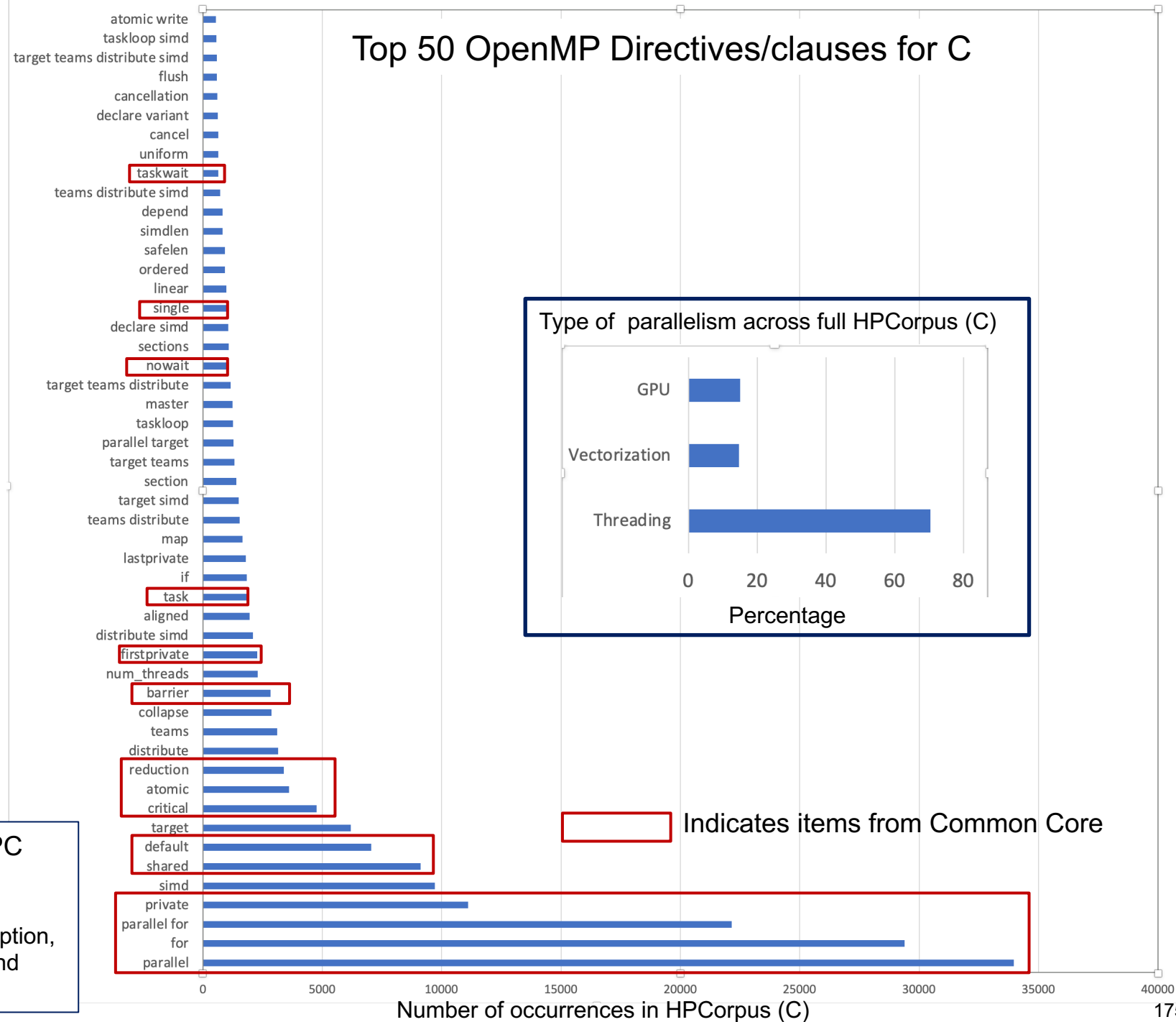
With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

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

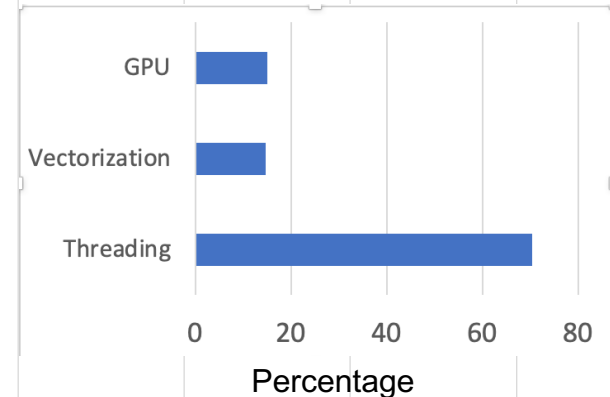
HPCorpus ... a data set created by scraping “all” HPC codes from github written in C, C++ and Fortran.

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

Top 50 OpenMP Directives/clauses for C



Type of parallelism across full HPCorpus (C)



What are people actually using from OpenMP

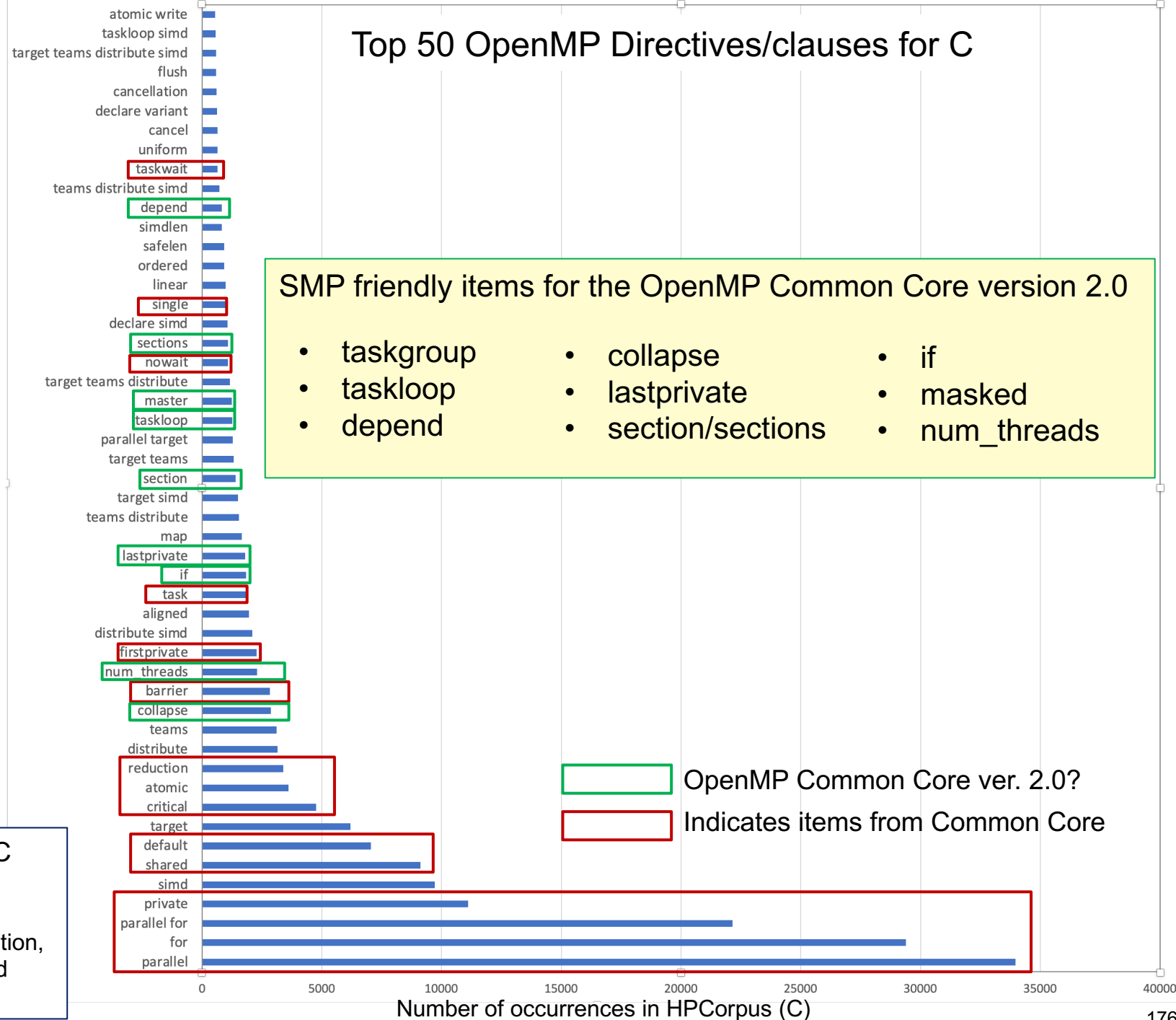
With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

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, Niranjana Hasabnis, Tim Mattson, Yuval Pinter, and Gal Oren, IEEE HPEC 2023

Top 50 OpenMP Directives/clauses for C

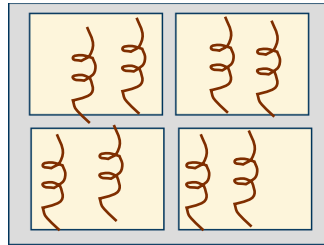


Extra Content

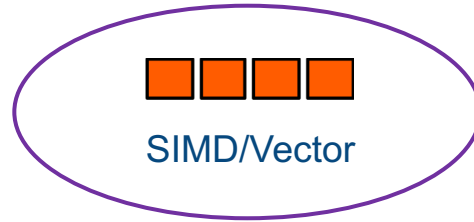
- Memory performance, the TLB, and the tile construct
- HPCorpus and OpenMP usage
-  • Introduction to the SIMD constructs
- NUMA: additional content

For hardware ... parallelism is the path to performance

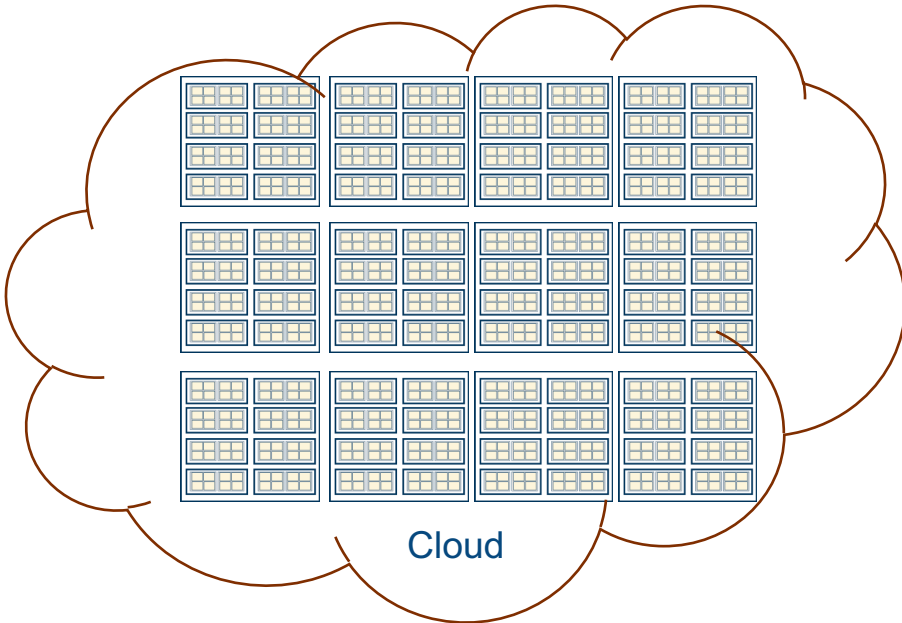
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



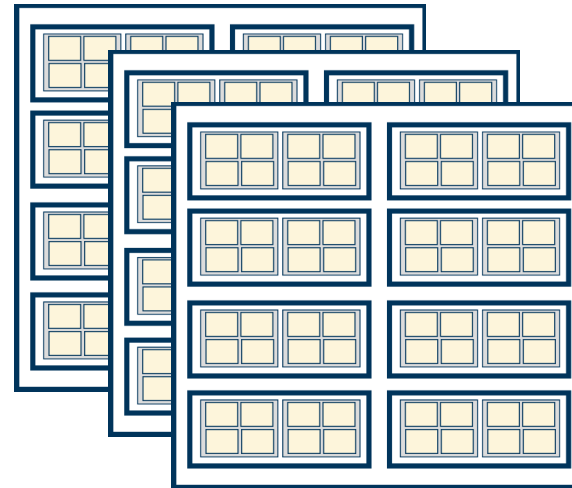
CPU



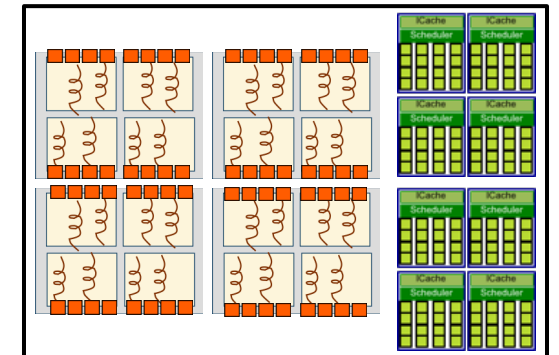
GPU



Cloud

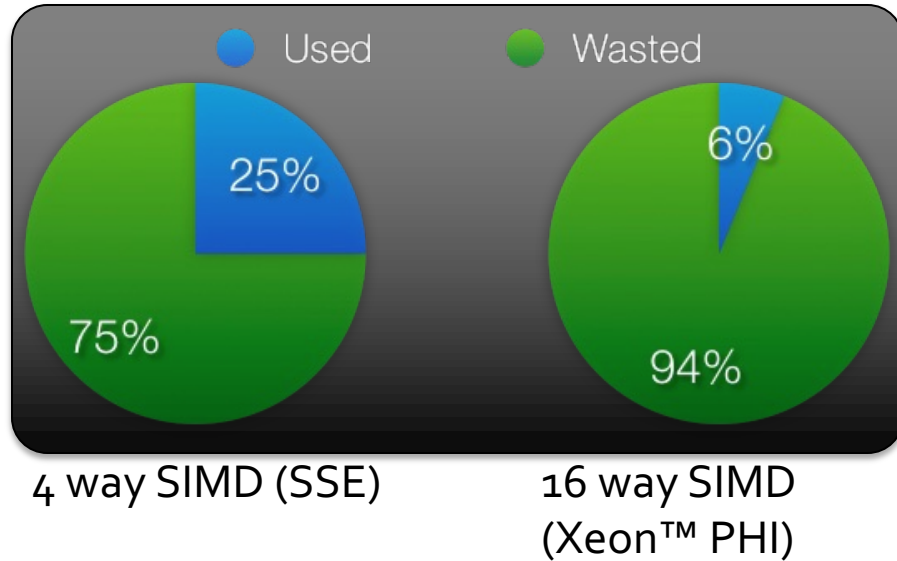


Cluster

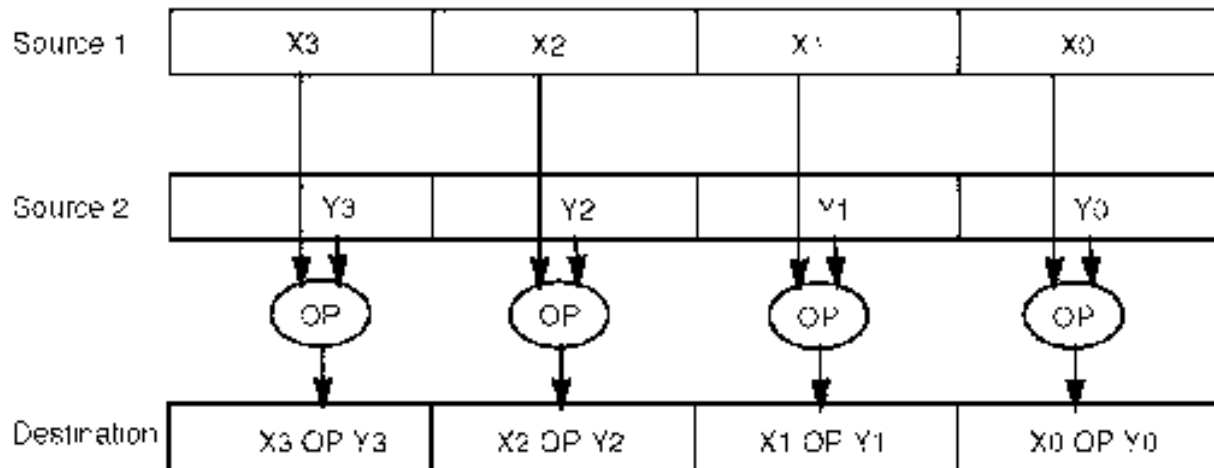


Heterogeneous node

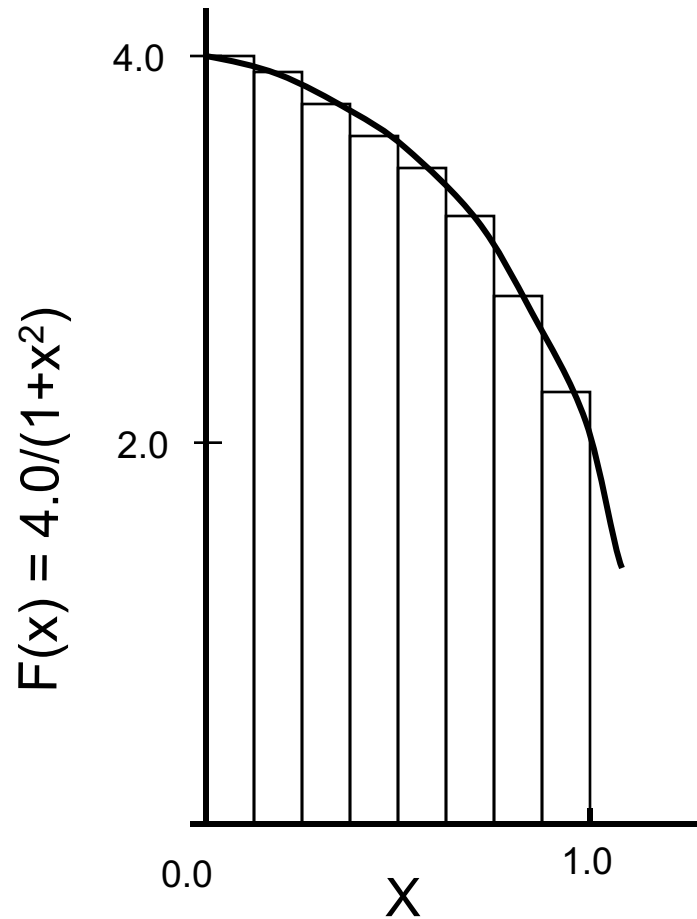
Vector (SIMD) Programming



- Architects love vector units, since they permit space- and energy-efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
 - Let the compiler do the job
 - Assist the compiler with language level constructs for explicit vectorization.
 - Use intrinsics ... an assembly level approach.



Example Problem: Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Serial PI program

Literals as double (no-vec), 0.012 secs

Literals as Float (no-vec), 0.0042 secs

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

    step = 1.0/(float) num_steps;

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

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Explicit vectorization of our Pi Program: Step 1 ... Unroll the loop

```
float pi_unroll(int num_steps)
{
    float step, x0, x1, x2, x3, pi, sum = 0.0;

    step = 1.0f/(float) num_steps;

    for (int i=1;i<= num_steps; i=i+4){    //unroll by 4, assume num_steps%4 = 0
        x0 = (i-0.5f)*step;
        x1 = (i+0.5f)*step;
        x2 = (i+1.5f)*step;
        x3 = (i+2.5f)*step;
        sum += 4.0f*(1.0f/(1.0f+x0*x0) + 1.0f/(1.0f+x1*x1) + 1.0f/(1.0f+x2*x2) + 1.0f/(1.0f+x3*x3));
    }

    pi = step * sum;
    return pi;
}
```

- We need one iteration to fit in the vector unit
- What is the width of your vector unit?
- We'll use SSE which is 128 bits wide.
- A float in C is 32 bits wide ... 4 floats fits in 128 bits

So unroll our loop by four

Explicit vectorization of our Pi Program: Step 2 ... Add SSE intrinsics

```
#include <immintrin.h>
float pi_sse(int num_steps)
{
    float scalar_one = 1.0, scalar_zero = 0.0, ival, scalar_four = 4.0, step, pi, vsum[4];
    step = 1.0/(float) num_steps;

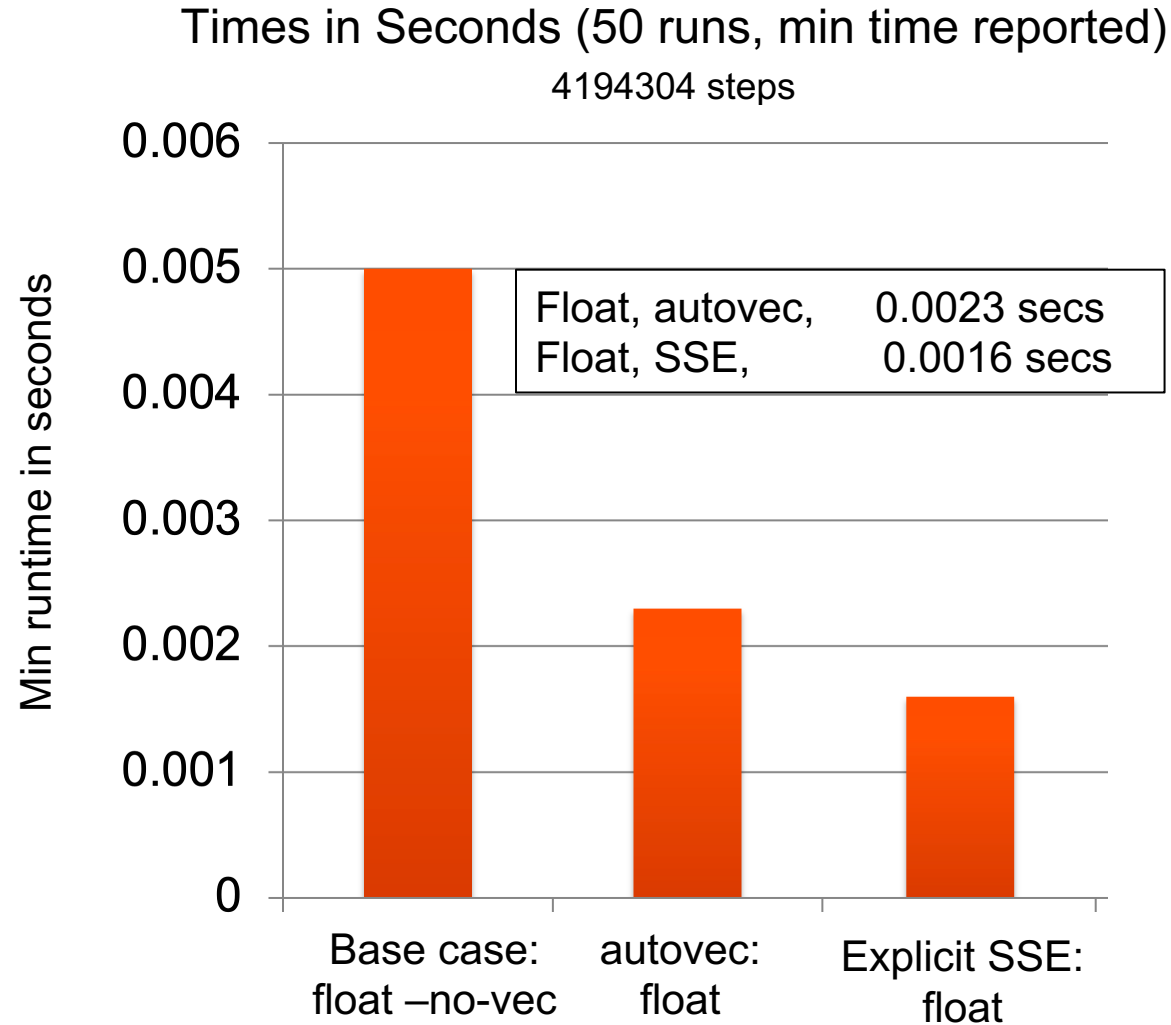
    __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
    __m128 one     = _mm_load1_ps(&scalar_one);
    __m128 four    = _mm_load1_ps(&scalar_four);
    __m128 vstep   = _mm_load1_ps(&step);
    __m128 sum     = _mm_load1_ps(&scalar_zero);
    __m128 xvec;   __m128 denom; __m128 eye;

    for (int i=0;i< num_steps; i=i+4){           // unroll loop 4 times
        ival      = (float)i;                     // and assume num_steps%4 = 0
        eye       = _mm_load1_ps(&ival);
        xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
        denom     = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
        sum       = _mm_add_ps(_mm_div_ps(four,denom),sum);
    }
    _mm_store_ps(&vsum[0],sum);
    pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
    return pi;
}
```

The vast majority of programmers never write explicitly vectorized code.

It is important to understand explicit vectorization so you appreciate what the compiler does to vectorize code for you

PI program Results:



- Intel Core i7, 2.2 Ghz, 8 GB 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

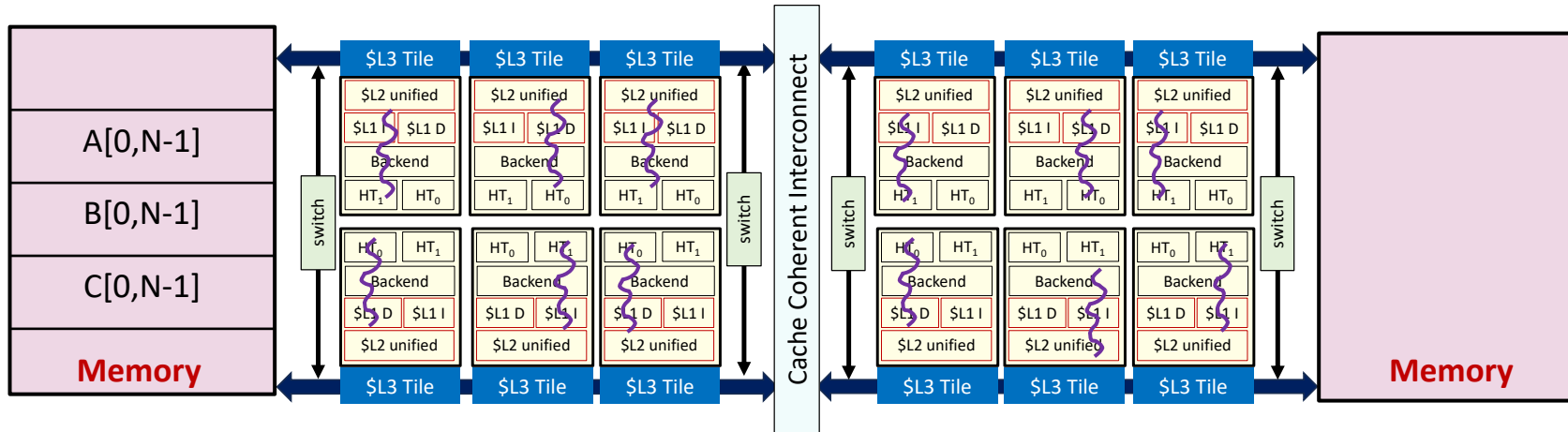
Extra Content

- Memory performance, the TLB, and the tile construct
- HPCorpus and OpenMP usage
- Introduction to the SIMD constructs

-  • NUMA: additional content

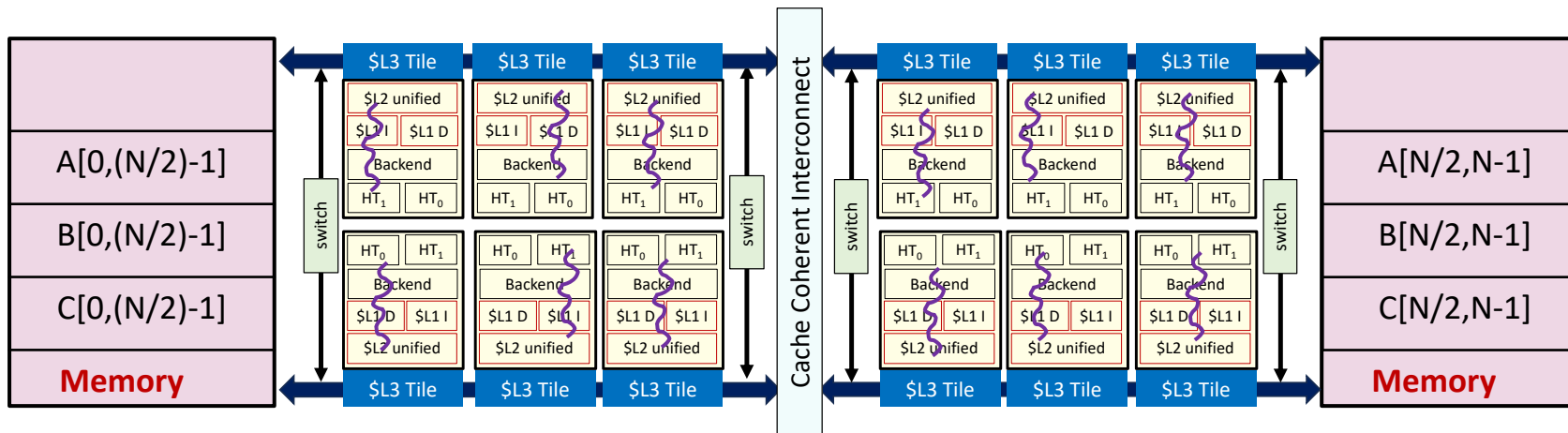
Example: Stream Benchmark (all four cases)

- 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

Nested parallelism

Process and Thread Affinity in Nested OpenMP

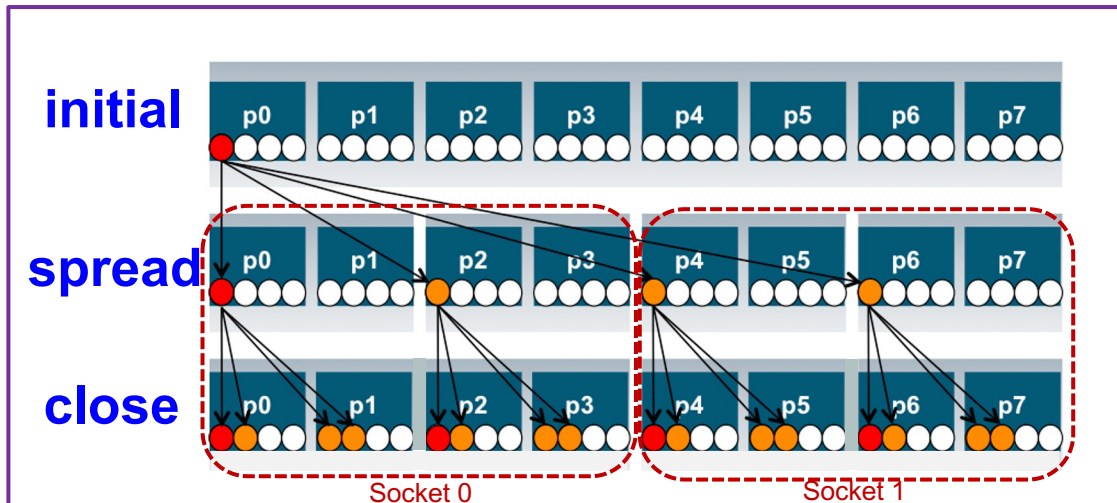
Consider a
program with
nested parallel
regions

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

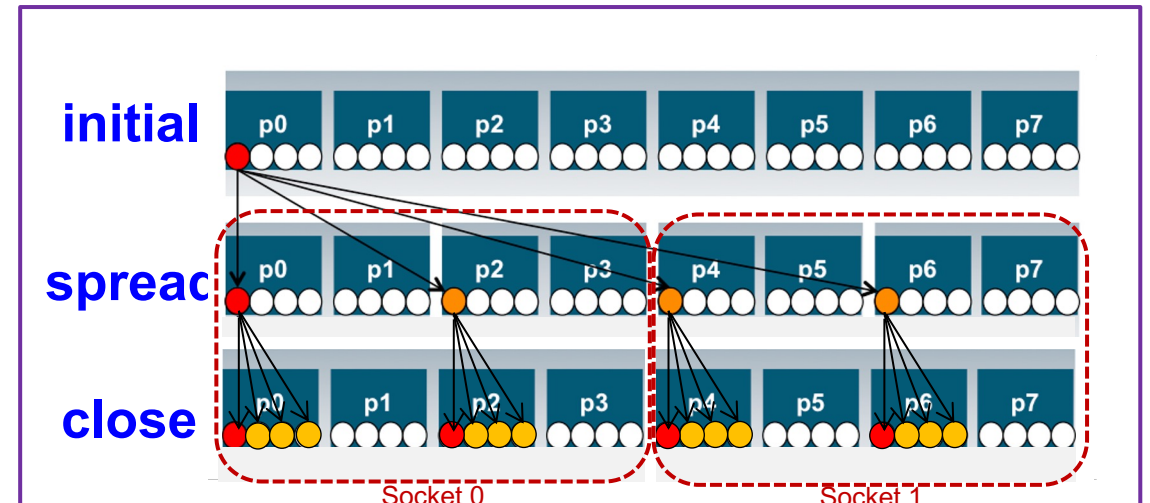
Running on a system with 2 sockets, 4 cores per socket, 4 hardware-threads per core

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

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



Cyclic distribution between “close” cores



Distribution across four hardware threads

**Controlling exactly which
cores/hardware-threads are used in a
computation**

Benchmarking: Sometimes you need to Control Everything!

- Goal: To compare different programming systems applied to the same problem:
 - We must control everything we can to make sure any observed differences are due to the different programming systems.
- We need to know exactly which cores we are using and how thread IDs map onto cores ... so we can understand data detailed memory movement and make sure it's the same between the different test cases.

Step 1: Know Your System

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

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

SMT enabled ... two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

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

Step 1: Know Your System

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

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

SMT enabled ... two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

The numbering of these “cores” (in 2 sockets).

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

Step 2: Setup a Runscript (so you can reproduce the computations later)

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

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

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

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

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

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

Step 3: Run the script and save results

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

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

Normally, this is going too far, but for benchmarking, this could be important.

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

**Additional content about less
commonly used OpenMP features for
NUMA systems**

OpenMP task-to-data Affinity (in OpenMP 5.0)

- Affinity hints can be provided for OpenMP tasks, resulting data to be closer to tasks
- Useful for multi-socket systems

```
void task_affinity() {  
    double* B;  
    #pragma omp task shared(B) affinity(A[0:N])  
    B = init_B_and_important_computation(A);  
  
    #pragma omp task firstprivate(B) affinity(B[0:N])  
    important_computation_too(B);  
  
    #pragma omp taskwait  
}
```

Memory Allocators (in OpenMP 5.0)

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

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

Using Memory Allocators

```
void allocator_example(omp_allocator_t *my_allocator) {  
    int a[M], b[N];  
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)  
    #pragma omp allocate(b) // use default OMP_ALLOCATOR  
  
    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);  
  
    #pragma omp parallel private(a) allocate(omp_low_lat_mem_alloc:a)  
    {  
        some_parallel_code();  
    }  
    omp_free(p);  
}
```

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

Affinity Verification Methods

- 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
```