OpenMP Course

Victor Eijkhout

2024 COE 379L / CSE 392

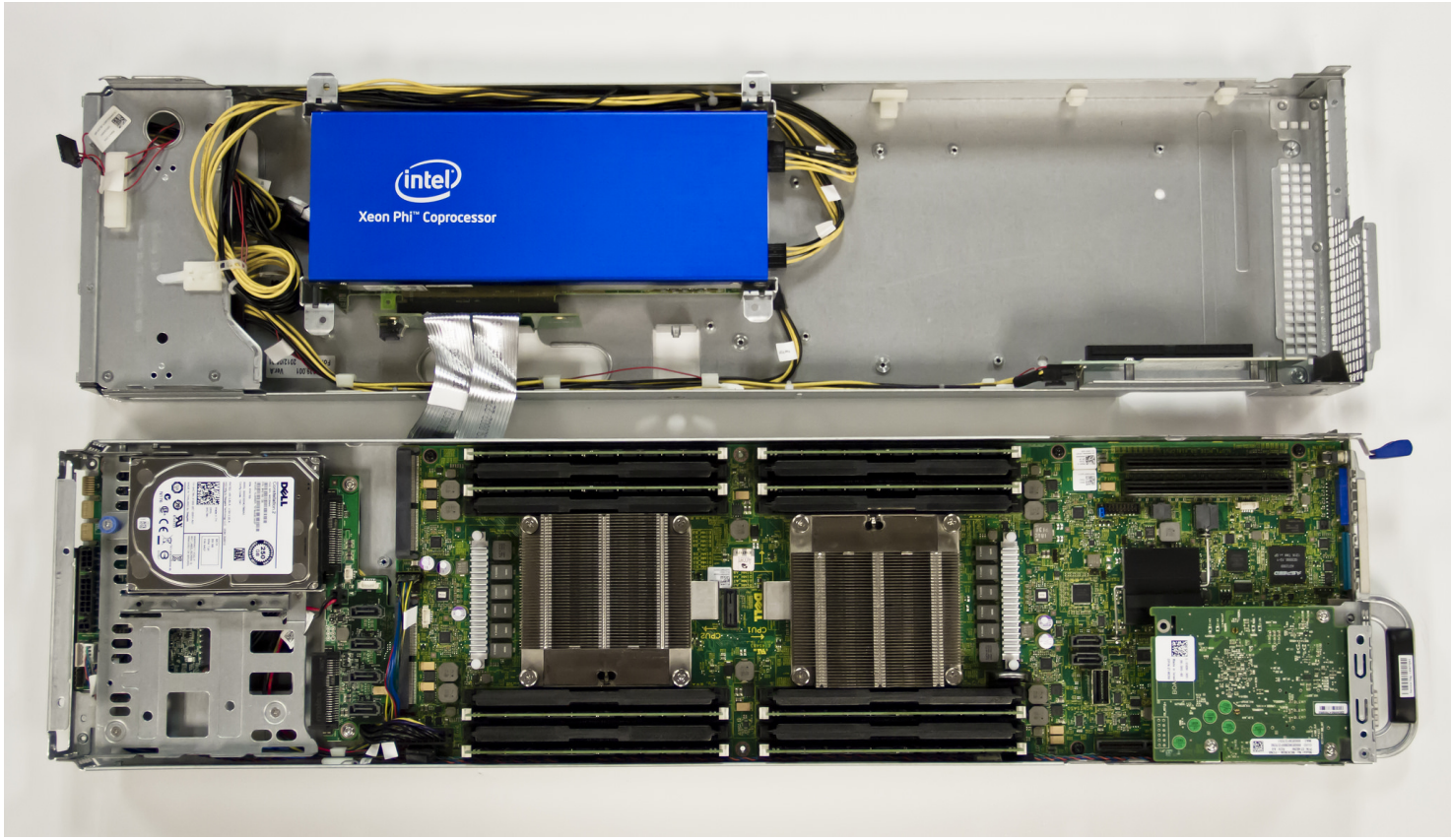
Justification

OpenMP is a flexible tool for incrementally parallelizing a shared memory-based code. This course introduces the main concepts through lecturing and exercises.

The Fork-Join model

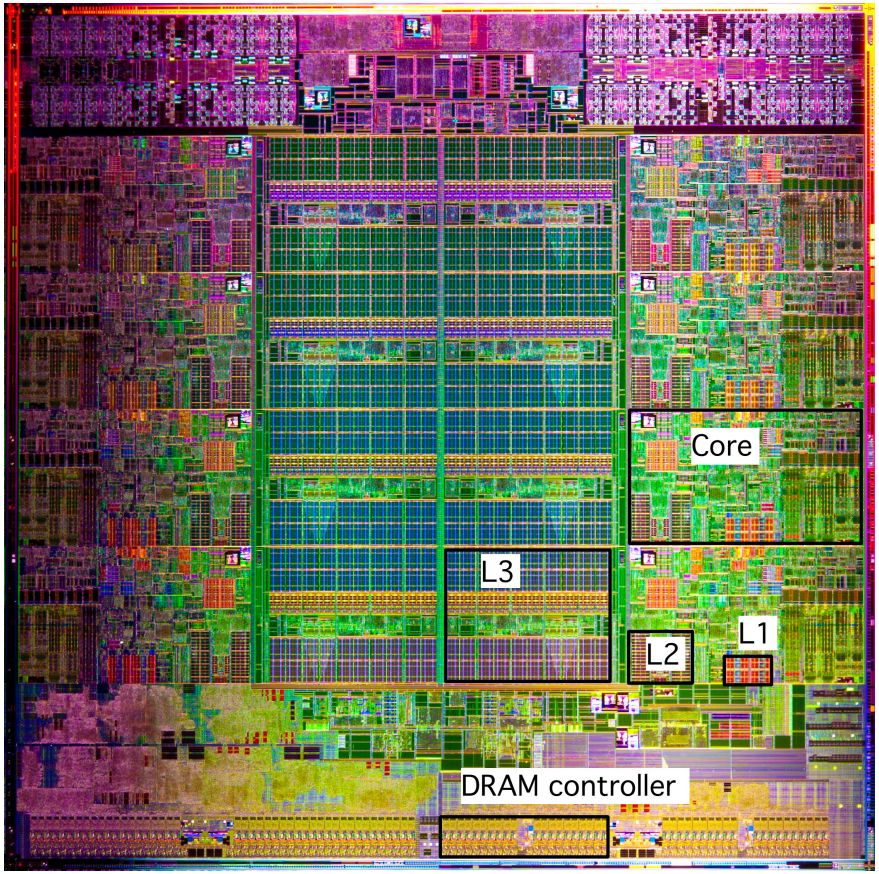
Computer architecture terminology

One cluster node:



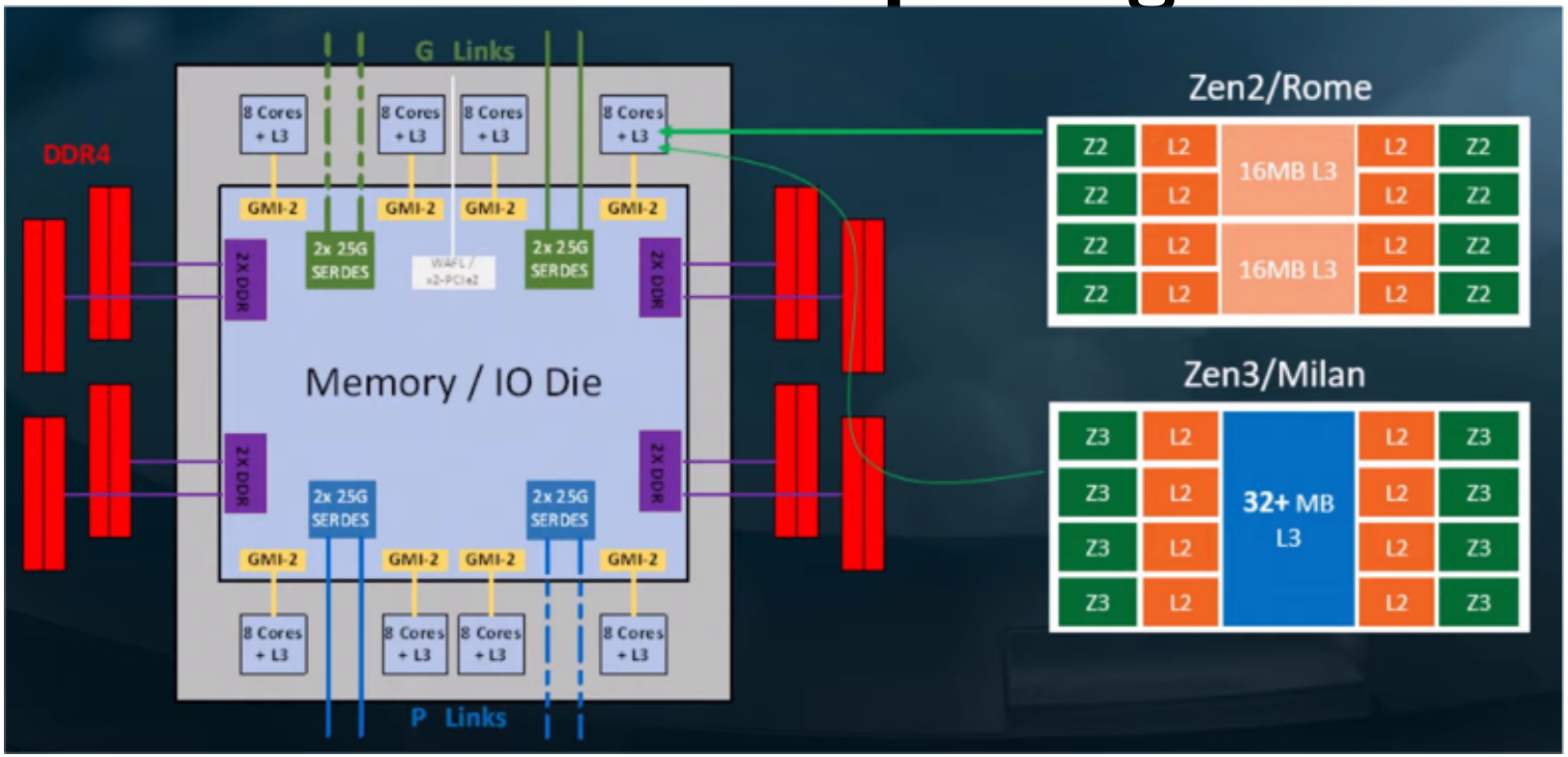
A node will have 1 or 2 or (sometimes) 4 ‘sockets’: processor chips.
There may be a co-processor attached.

Structure of a socket



- Eight cores per socket, 16 per node.
- Access to DRAM, both this socket and other
- Shared L3 cache, private L2 and L1

Modern chip design



- Shared memory
- Shared caches
- Local caches
- 'cc-Numa': Cache-coherent Non-Uniform Memory Access

OpenMP history

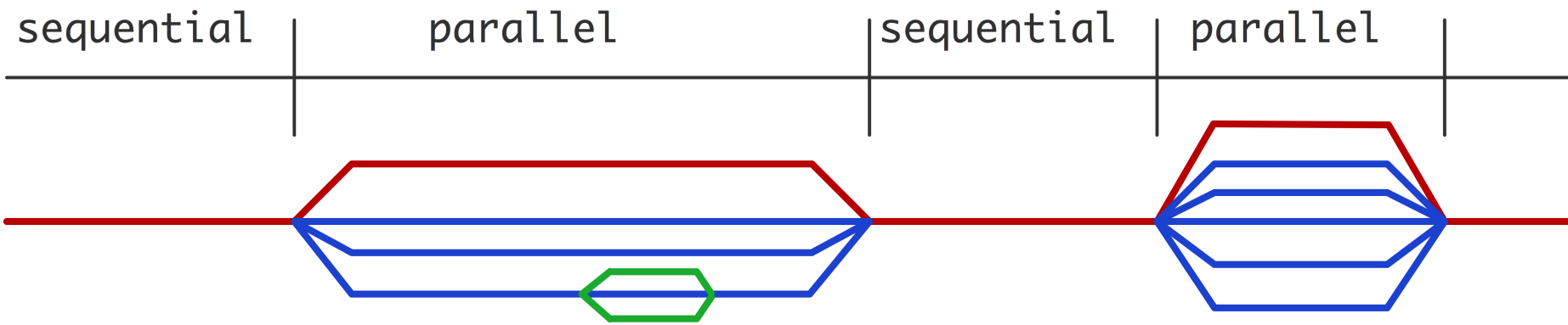
- First standard 1997, Fortran directives
- Current standard 5.2 as of 2021

History of shared memory computers:

- Intel Core processors: 2006;
  before that: 1980s systems from Alliant, Sequent, Denelcor;
  SGI Origin with up to 1024 processors
  supercomputers with multi-chip nodes (Intel Paragon, Cray T3D)

Threads

Process: stream of instructions
Thread: process can duplicate itself, same code, access to same data



The OS will place threads on different cores: parallel performance.
Note: threads are software. More threads than cores or fewer is allowed.
Ignore hyperthreads

To write an OpenMP program

```
1 #include "omp.h"
```

in C, and

```
1 use omp_lib
```

or

```
1 #include "omp_lib.h"
```

for Fortran.

To compile an OpenMP program

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler 23 and later
icc -o foo foo.c -fopenmp
# Intel compiler before 23
icc -o foo foo.c -qopenmp
```

Run as ordinary program.

CMake

```
1 find_package(OpenMP)
2 # depending on language:
3 target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_C )
4 target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_CXX )
5 target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_Fortran )
```

Let's try an example

How many cores do we have?

```
1 int omp_get_num_procs();
```

Compile and run

Query threads

```
1 int omp_get_num_threads();
```

Compile and run. Observe?

To run an OpenMP program

```
export OMP_NUM_THREADS=8
./my_omp_program
```

Quick experiment:

```
for t in 1 2 4 8 12 16; do
  OMP_NUM_THREADS=$t ./my_omp_program
done
```

Query threads'

Use a parallel region:

```
1 #pragma omp parallel
2 {
3   statements
4 }
```

or

```
1 !$omp parallel
2   statements
3 !$omp end parallel
```

Compile and run. Observe?

Query threads”

What happens if you write

```
1 int t;
2 #pragma omp parallel
3 {
4   t = omp_get_thread_num();
5   printf("%d\n",t);
6 }
```

```
1 {
2   int t; t = // ...
3   printf( ... );
4 }
```

```
1 int t;
2 #pragma omp parallel private(t)
```

Exercise 1 (`parallel`)

Write a 'hello world' program, where the print statement is in a parallel region. Compile and run.

Run your program with different values of the environment variable *`OMP_NUM_THREADS`*. If you know how many cores your machine has, can you set the value higher?

Exercise 2 (`parallel`)

Take the hello world program of exercise 1 and insert the above functions, before, in, and after the parallel region. What are your observations?

What happens if I press that button?

Who of you has tried setting the number of threads (much) larger than the number of cores? What happened?

Enclosed scopes

Data is (usually) shared in a parallel region:

Data is visible in enclosed scopes:

```
1 main() {
2   int x;
3 #pragma omp parallel
4   {
5      // you can use and set ‘x’ here
6   }
7   printf("x=%e\n",x); // value depends on what
8                       // happened in the parallel region
9 }
```

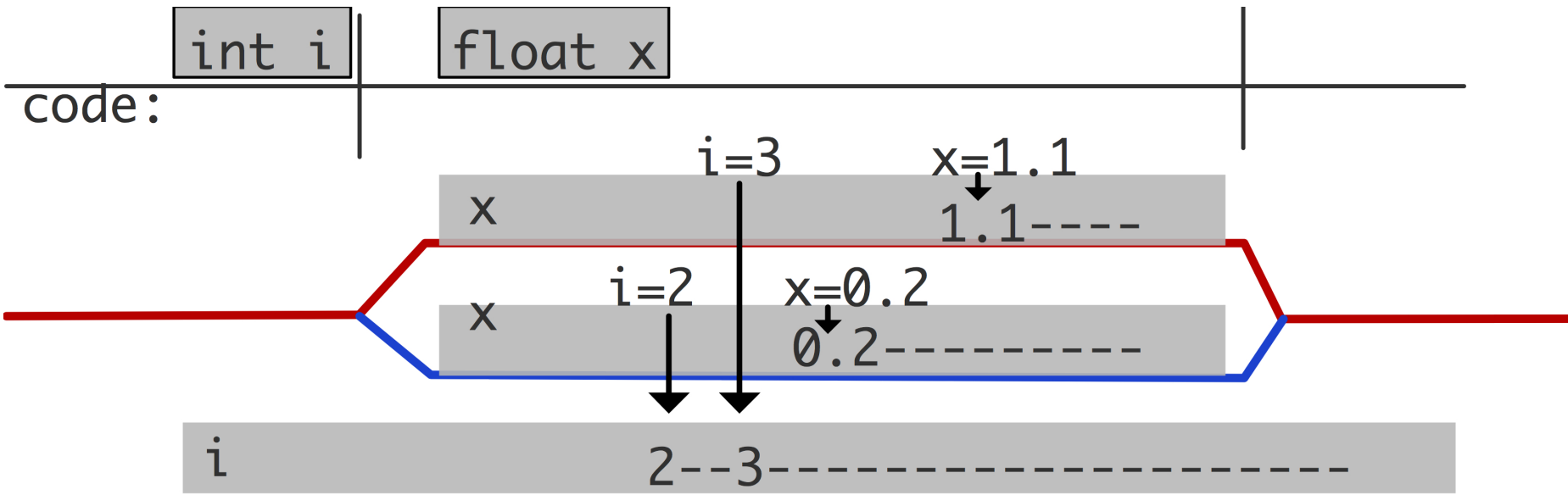
Note: this is usually the case, but strictly not the default.

Thread-local variables

```
1 {
2   int x;
3   #pragma omp parallel
4   {
5     double x;
6     // do something with x
7   }
8 }
```

- *A parallel region is a scope.*
- *Local variables are per thread.*

Locality of variables



Thread control

- Environment variable *OMP_NUM_THREADS*=4
- Clause *num_threads*(4)

```
1 #pragma omp parallel num_threads(4)
```

  usually not the best solution.

- Environment variable *OMP_THREAD_LIMIT*=56
  (useful for nested parallelism)

Limits on paralleism

```
1 #pragma omp parallel if(N>50)
```

Can you think of an application?
I mean, what's the harm of doing one small loop inefficiently?

Threads and threads

- Threads are software, cores are hardware.
- The OS can move threads between cores: not a good idea for performance.
- Set: `export OMP_PROC_BIND=true` and you'll be good in most of the cases.
- Look up 'affinity' in the OMP standard for all the details.

Nested parallelism

```
1 int main() {
2   ...
3 #pragma omp parallel
4   {
5   ...
6   func(...)
7   ...
8   }
9 } // end of main
```

```
1 void func(...) {
2 #pragma omp parallel
3   {
4   ...
5   }
6 }
```

Nesting control

By default, nested levels get only one thread.
Use:

```
1 void omp_set_max_active_levels(int);
2 int omp_get_max_active_levels(void);
```

or

```
OMP_MAX_ACTIVE_LEVELS=3
```

(Deprecated: boolean switch *OMP_NESTED*)

Loop parallelism

Loop parallelism

Much of parallelism in scientific computing is in loops:

- Vector updates and inner products
- Matrix-vector and matrix-matrix operations
- Finite Element meshes
- Multigrid

Work distribution

- Suppose loop iterations are independent:
- Distribute them over the threads:
- Use `omp_get_thread_num` to determine disjoint subsets.

```
1 #pragma omp parallel
2 {
3   int threadnum = omp_get_thread_num(),
4     numthreads = omp_get_num_threads();
5   int low = N*threadnum/numthreads,
6     high = N*(threadnum+1)/numthreads;
7   for (int i=low; i<high; i++)
8     // do something with i
9 }
```

Discuss.

Loops are easy

for directive before a loop:

```
1 #pragma omp parallel
2 {
3 #pragma omp for
4   for ( int i=0; i<N; i++)
5     ... something with i ...
6 }
```

- OpenMP does the partitioning of the iteration space
- blocks by default but other 'schedules' possible
- even dynamic load balancing.

Loops in Fortran

Fortran: optional matching end directive

```
1 !$omp parallel
2 !$omp do
3   do i=1,n
4     ... something with i ...
5   end do
6 !$omp end parallel
```

Workshare constructs

Here's the two-step parallelization in OpenMP:

- You use the `parallel` directive to create a team of threads;
- then you use a 'workshare' construct to distribute the work over the team.
- For loops that is the `for` (or `do`) construct.

Workshare construct for loops

Parallel and workshare together

```
1 #pragma omp parallel for
2   for (i=0; i<N; i++)
3     ... something with i ...
```

Range syntax

Parallel loops in C++ can use range-based syntax as of OpenMP-5.0:

```
1  // vecdata.cpp
2  vector<float> values(100);
3
4  #pragma omp parallel for
5  for ( auto& elt : values ) {
6    elt = 5.f;
7  }
8
9  float sum{0.f};
10 #pragma omp parallel for reduction(+:sum)
11 for ( auto elt : values ) {
12   sum += elt;
13 }
```

Tests show exactly the same speedup as the C code.

Ranges library

The C++20 `ranges` library is supported:

```
#pragma omp parallel for reduction(+:itcount)
for ( auto e : data
              | std::ranges::views::drop(1) )
        itcount += e;
#pragma omp parallel for reduction(+:itcount)
for ( auto e : data
              | std::ranges::views::transform
              ( []( auto e ) { return 2*e; } ) )
        itcount += e;
```

Index ranges

Use *iota_view* to obtain indices:

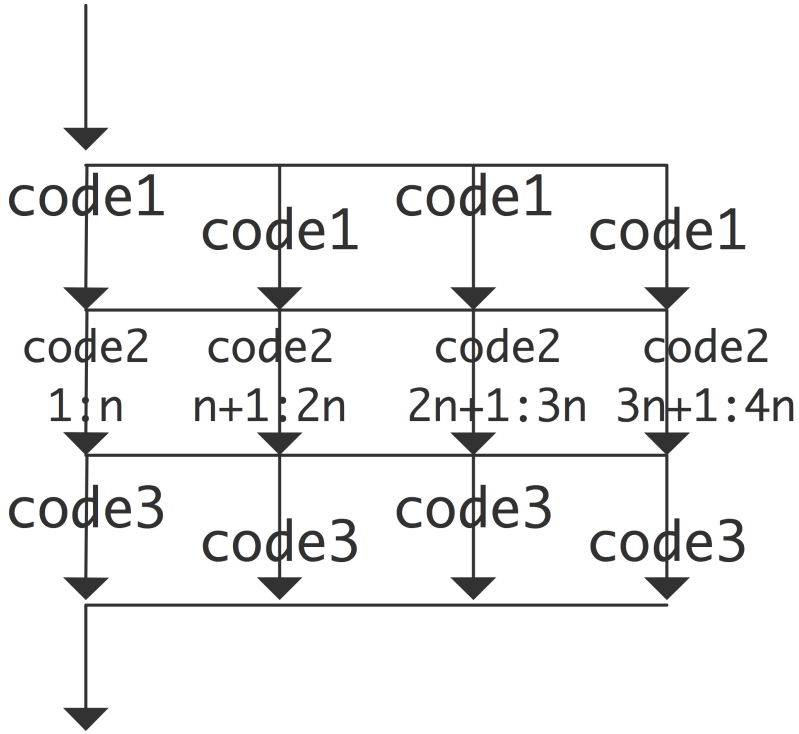
```
1 // iota.cpp
2 vector<long> data(N);
3 # pragma omp parallel for
4 for ( auto i : std::ranges::iota_view( 0UZ,data.size() ) )
5   data[i] = f(i);
```

Note that this uses C++23 suffix for *unsigned size_t*. For older versions:

```
1 iota_view( static_cast<size_t>(0),data.size() )
```

Stuff inside a parallel region

```
1 #pragma omp parallel
2 {
3   code1();
4   #pragma omp for
5   for (int i=1; i<=4*N; i++) {
6     code2();
7   }
8   code3();
9 }
```



Loops are static

- No *break* or so.
- *continue* or *cycle* allowed.
- Fixed upper bound;
- Simple loop increment;
- $\Rightarrow$ OpenMP needs to be able to calculate the number of iterations in advance.
- No *while* loops.

When is a loop parallel?

It's up to you!

This is parallelizable:

```
1   for (int i=low; i<hi; i++)
2      x[i] = // expression without x
```

Is this?

```
1 for (int i=low; i<hi; i++)
2    x[ f(i) ] = // expression
```

Histograms / binning.

Exercise 3

Consider the code

```
1 for (int i=0; i<n; i++)
2   x[i/2] += f(i)
```

Argue that this does not satisfy the above consition. Can you rewrite this loop to be parallelizable?

Exercise 4 (`vectorsum`)

Run the following snippet

```
1 // vectorsum.c
2 for ( int iloop=0; iloop<nloops; ++iloop ) {
3   for ( int i=0; i<vectorsize; ++i ) {
4     outvec[i] += invec[i]*loopcoeff[iloop];
5   }
6 }
```

1. Sequentially
2. On one thread
3. With more than one thread.

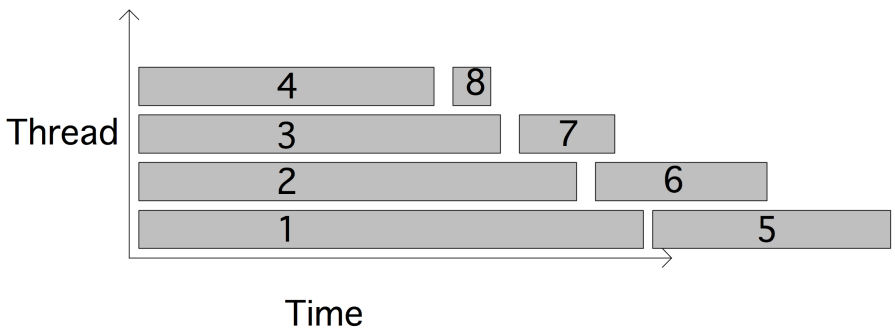
Do a scaling study. What all can you do to improve performance? Read back through previous slides.

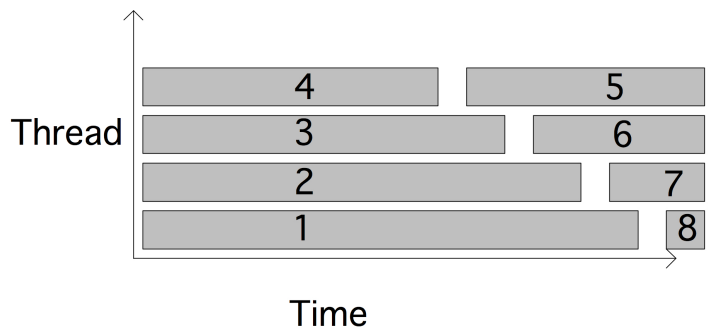
More about loops

Loop schedules

- Static scheduling of iterations.
  (default in practice though not formally)
  Very efficient. Good if all iterations take the same amount of time.
  `schedule(static)`
- Other possibility: dynamic.
  Runtime overhead; better if iterations do not take the same amount of time.
  `schedule(dynamic)`

Four threads, 8 tasks of decreasing size
dynamic schedule is better:

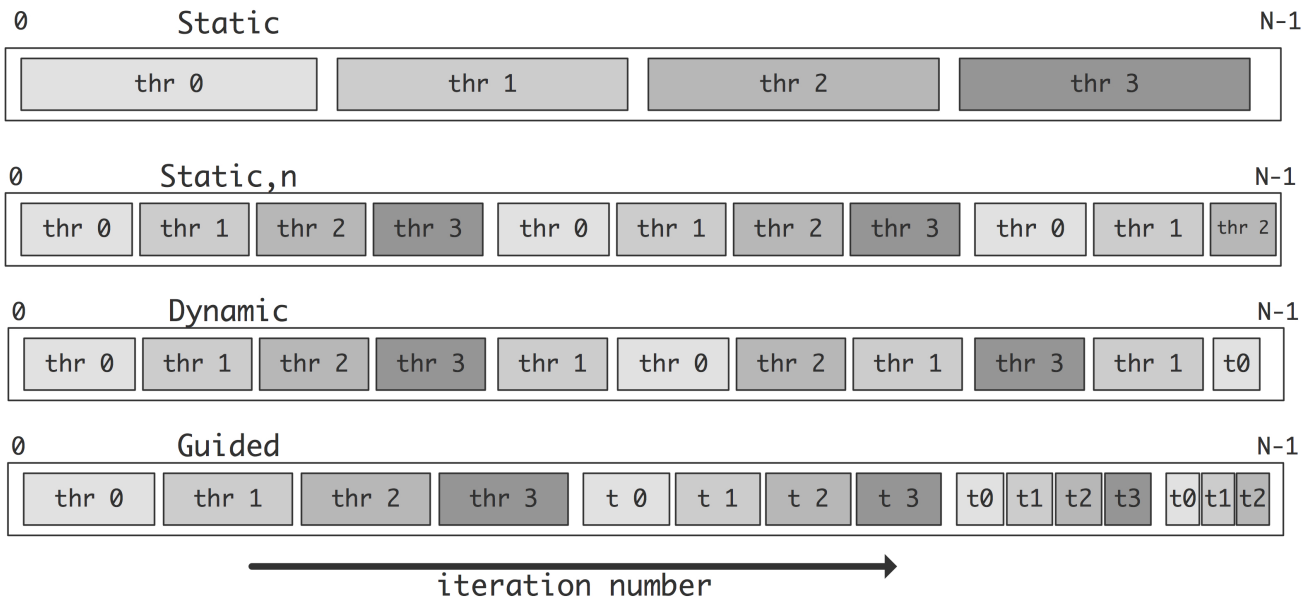




Chunk size

With N iterations and t threads:

- Static: each thread gets N/t iterations.
  explicit chunk size: `schedule(static,123)`
- Dynamic: each thread gets 1 iteration at a time
  explicit chunk size: `schedule(dynamic,45)`



Guided schedule

Use decreasing chunk size (with optional minimum chunk):
`schedule(guided,6)`

More flexible than dynamic, less overhead than dynamic.

Collapse

- Parallelize loop nest
- More iterations ⇒ better performance
- Inner loop needs to be directly nested
  bounds simple function of outer bounds and var

```
1  // triangle.c
2  #pragma omp parallel for collapse(2)
3    for ( int i=0; i<vectorsize; ++i ) {
4      for ( int j=i+1; j<vectorsize; ++j ) {
5        double
6          dx= particles[XCOORD(i)]-particles[XCOORD(j)],
7          dy= particles[YCOORD(i)]-particles[YCOORD(j)];
8        double r = sqrt( dx*dx + dy*dy ),
9          f = (particles[MASS(i)]*particles[MASS(j)])/(r*r);
10     }
11   }
```

Ordered loops

- Ordered iterations: normally OpenMP can execute iterations in any sequence. You can force ordering if you absolutely have to.
- Bad for performance and generally not needed.
- Use for I/O:

```
1 #pragma omp parallel for ordered schedule(dynamic)
2 for ( i=... ) {
3   x[i] = ...
4   #pragma omp ordered
5     printf(x[i])
6 }
```

Loop data

Private vs shared

Statements such as:

```
1 #pragma omp parallel private(x)
2 #pragma omp parallel shared(y)
3 #pragma omp parallel default(none) shared(x) private(y)
```

- Shared data: comes from outside the parallel region
- Private data: either private versions of outside, or local defined
- Default is shared. Dangerous!
- Parallel loop variables are always private.

Where is the bug?

```
1   int i,j;
2   double temp;
3 #pragma omp parallel for private(temp)
4    for(i=0;i<N;i++){
5     for (j=0;j<M;j++){
6        temp = b[i]*c[j];
7        a[i][j] = temp * temp + d[i];
8 } }
```

Is there a different way of handling *temp*?

More private/shared

```
1  float x=5.5;
2  #pragma omp parallel firstprivate(x)
3    // x is now private, and has value 5.5
4
5  float y;
6  #pragma omp parallel for lastprivate(y)
7  for ( /* looping */ ) {
8    // stuff
9    y = something;
10 }
11 // now y has the sequentially last value
```

Reduction

Exercise 5

Extend the program from exercise 2. Make a complete program based on these lines:

Code:

```
1 // reduct.c
2 int tsum=0;
3 #pragma omp parallel
4 {
5   tsum += // expression
6 }
7 printf("Sum is %d\n",tsum);
```

Output:

```
1 With 4 threads, sum s/b
      ↪6
2 Sum is 6
3 Sum is 5
4 Sum is 1
5 Sum is 4
6 Sum is 6
7 Sum is 5
8 Sum is 6
9 Sum is 5
10 Sum is 3
11 Sum is 4
```

Compile and run again. (In fact, run your program a number of times.)
Do you see something unexpected? Can you think of an explanation?

Shared memory problems

Race condition: simultaneous update of shared data:

process 1: `I=I+2`
process 2: `I=I+3`

Results can be indeterminate:

scenario 1.		scenario 2.		scenario 3.	
I = 0					
read I = 0	read I = 0	read I = 0	read I = 0	read I = 0	
compute I = 2	compute I = 3	compute I = 2	compute I = 3	compute I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			read I = 2
					compute I = 5
					write I = 5
I = 3		I = 2		I = 5	

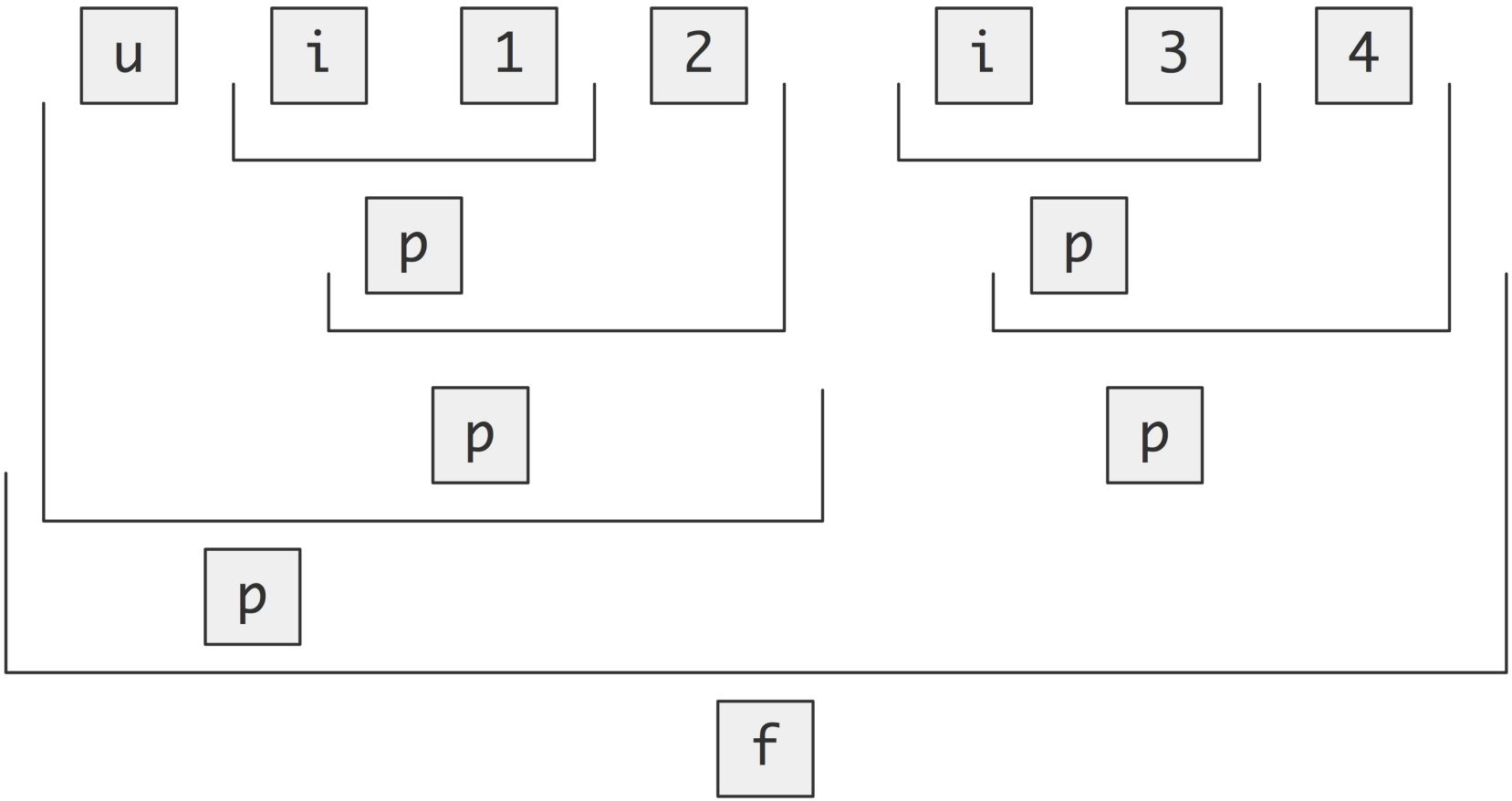
Reductions

- Inner product loop:

```
1 // reductsum.c
2 float s = 0;
3 #pragma omp parallel for reduction(+:s)
4 for (int idata=0; idata<ndata; idata++)
5   s += x[idata]*y[idata];
```

- Use the `reduction(+:s)` clause.
- All the usual operations are available; you can also make your own.

How OMP reducts



Exercise 6 (`pi`)

Compute π by *numerical integration*. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using *Riemann sums*.

- Let $f(x) = \sqrt{1-x^2}$ be the function that describes the quarter circle for $x = 0 \ldots 1$;
- Then we compute

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \qquad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Write a program for this, and parallelize it using OpenMP parallel for directives. Measure attained speedup.

1. Put a `parallel` directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or `parallel do`). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a `critical` directive in front of the update. (Yes and very much no.)
4. Remove the `critical` and add a clause `reduction(+:quarterpi)` to the `for` directive. Now it should be correct and efficient.

Exercise 7 (`piadapt`)

We continue with exercise 6. We add 'adaptive integration'where needed, the program refines the step size[1]. This means that the iterations no longer take a predictable amount of time.

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the *`reduction`* clause to fix this.
2. Your code should now see a decent speedup, but possible not for all cores. It is possible to get completely linear speedup by adjusting the schedule.

   Start by using *`schedule`*`(static,`*n*`)`. Experiment with values for *n*. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try *`schedule`*`(dynamic)`. This will actually give a fairly bad result. Why? Use *`schedule`*`(dynamic,$n$)` instead, and experiment with values for *n*.
4. Finally, use *`schedule`*`(guided)`, where OpenMP uses a heuristic. What results does that give?

[1] It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.

same exercise

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the `reduction` clause to fix this.
2. Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.
   Start by using `schedule(static,$n$)`. Experiment with values for *n*. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try `schedule(dynamic)`. This will actually give a fairly bad result. Why? Use `schedule(dynamic,$n$)` instead, and experiment with values for *n*.
4. Finally, use `schedule(guided)`, where OpenMP uses a heuristic. What results does that give?
5. `schedule(auto)` : leave it up to the system.
6. `schedule(runtime)` : leave it up to environment variables; good for experimenting.

Reduction on arrays, static

```
1 // reductarray.c
2 int data[nthreads];
3 #pragma omp parallel for schedule(static,1) \
4   reduction(+:data[:nthreads])
5 for (int it=0; it<nthreads; it++) {
6   for (int i=0; i<nthreads; i++)
7     data[i]++;
8 }
```

Reduction on arrays, dynamic

```
1 int *alloced = (int*)malloc( nthreads*sizeof(int) );
2 for (int i=0; i<nthreads; i++)
3   alloced[i] = 0;
4 #pragma omp parallel for schedule(static,1) \
5   reduction(+:alloced[:nthreads])
6 for (int it=0; it<nthreads; it++) {
7   for (int i=0; i<nthreads; i++)
8         alloced[i]++;
9 }
```

Reductions on C++ vectors

Use the *data* method to extract the array on which to reduce. However, this does not work:

```
1 vector<float> x;
2 #pragma omp parallel reduction(+:x.data())
```

because the reduction clause wants a variable, not an expression, for the array, so you need an extra bare pointer:

```
1 // reductarray.cpp
2 vector<int> data(nthreads,0);
3 int *datadata = data.data();
4 #pragma omp parallel for schedule(static,1) reduction(+:datadata[:nthreads])
```

Reduction on arrays, warning

Each thread gets a copy of the array on the stack
$\Rightarrow$ possible stack overflow

set *OMP_STACKSIZE*

also see `ulimit` on the Unix level.

User-defined reductions

```
1    #pragma omp declare reduction
2    ( identifier : typelist : combiner )
3    [initializer(initializer-expression)]
4
```

where:

`combiner` is an expression that updates the internal variable *omp_out* as function of itself and *omp_in*.

`initializer` sets *omp_priv* to the identity of the reduction;
Often: *initializer*(*omp_priv*=*omp_orig*) to use the initial value.

User-defined reductions, example

```
1 // reductpositive.c
2 int reduce_without_zero(int r,int n);
```

```
1 #pragma omp declare reduction \
2   (rwz:int:omp_out=reduce_without_zero(omp_out,omp_in)) \
3   initializer(omp_priv=-1)
4   m = -1;
5 #pragma omp parallel for reduction(rwz:m)
6   for (int idata=0; idata<ndata; idata++)
7     m = reduce_without_zero(m,data[idata]);
```

Exercise 8

Write a custom reduction that finds the coordinate (x, y) with maximal norm $\sqrt{x^2 + y^2}$. Test on an array of randomly generated coordinates. Decide on how to store this:

```
1 double coords[N][2];
2 double coords[2*N];
3 struct coord { double x,double y };
4 double *coords = (double*) malloc(2*N*sizeof(double));
```

In C++ consider writing a `Coordinate` class and overloading an operator on it.

sample output for prev exercise

```
1 Execute program reductcoord cores 4
2 xy[0] = 0.396465 0.840485
3 xy[1] = 0.353336 0.446583
4 xy[2] = 0.318693 0.886428
5 xy[3] = 0.015583 0.584090
6 xy[4] = 0.159369 0.383716
7 xy[5] = 0.691004 0.058859
8 xy[6] = 0.899854 0.163546
9 xy[7] = 0.159072 0.533065
10 xy[8] = 0.604144 0.582699
11 xy[9] = 0.269971 0.390478
12 Max coordinate: 0.318693 0.886428
```

Workshare constructs

What is worksharing again?

- The `omp parallel` creates a team of threads.
- Now you need to distributed work among them.
- Already seen: `for`, `do`
- Similar: `sections`
- Not obvious: `single`
- Fortran only: `workshare` works with array notation
- Story in itself: `task`

Sections

Independent separate calculations:

```
1   double fx = f(x), gx = g(x), hx = h(x);
2   ..... fx ... gx ... hx ....
```

```
1 // sections.c
2 #pragma omp parallel sections
3 {
4   #pragma omp section
5   fx = f(1.);
6   #pragma omp section
7   gx = g(1.);
8   #pragma omp section
9   hx = h(1.);
10 }
11 float s = fx+gx+hx;
```

Sections with reduction

Same code, but with reduction:

```
1  float s=0;
2  #pragma omp parallel sections reduction(+:s)
3  {
4    #pragma omp section
5    s += f(1.);
6    #pragma omp section
7    s += g(1.);
8    #pragma omp section
9    s += h(1.);
10 }
```

Single

```
1  int a;
2  #pragma omp parallel
3  {
4    #pragma omp for reduction(+:x)
5    for ( ... i ... )
6      x += // something
7    #pragma omp single
8      y = fx();
9    #pragma omp for
10   for ( ... i ... )
11     z[i] = ... i ... y ...
12 }
```

- Is executed by a single thread.
- Has implicit barrier, so the result is available to all threads after.

Masked/Master

- Masked/Master: single thread execution
- Not a workshare, so no barrier

```
1 int a;
2 #pragma omp parallel
3 {
4   // stuff
5   #pragma omp master
6     printf("...",a);
7   // stuff
8 }
```

(*masked* introduced in 5.1; *master* deprecated in 5.2)

Master wrongly used

```
1 int a;
2 #pragma omp parallel
3 {
4   #pragma omp master
5     a = f(); // some computation
6   // WRONG: ‘a’ may not be available
7   #pragma omp sections
8     // various different computations using ‘a’
9 }
```

Exercise 9

What is the difference between this approach and how the same computation would be parallelized in MPI?

Fortran: workshare

```
!! workshare1d.F90
  integer,parameter :: dim=100000,iterations=1000
  real(4),dimension(dim) :: A,B,C
            !$omp parallel workshare
            C = C + A*B
            !$omp end parallel workshare
            !!codesnippet fworkshare
         end do
      else if (version==2) then
         vnames(version) = "do loop"
         do it=1,iterations
            !$omp parallel do
            do i=1,dim
               C(i) = C(i) + A(i) * B(i)
            end do
            !$omp end parallel do
         end do
      else if (version==3) then
         vnames(version) = "simd loop"
         do it=1,iterations
            !$omp parallel do simd
            do i=1,dim
               C(i) = C(i) + A(i) * B(i)
            end do
            !$omp end parallel do simd
         end do
      end if
      t = omp_get_wtime()-t
      prin  '( a,x,a,x,f7.5)', trim(vnames(version)),"time:",t
      exact = 2*iterations
      if ( abs(c(1)-exact)>1.e-5) then
```

Thread data

Shared and private data

You have already seen some of the basics:

- Data declared outside a parallel region is shared.
- Data declared in the parallel region is private.
  (Fortran does not have this block scope mechanism)

```
1 int i;
2 #pragma omp parallel
3 { double i; .... }
```

- You can change all this with clauses:

```
1 int i;
2 #pragma omp parallel private(i)
```

Variables in loops

```
1 int i; double t;
2 #pragma omp parallel for
3   for (i=0; i<N; i++) {
4     t = sin(i*pi*h);
5     x[i] = t*t;
6   }
```

- The loop variable is automatically private.
- The temporary t is shared, but conceptually private to each iteration: needs to be declared private.
  (What happens if you don't?)

Copying to/from private data

- Private data is uninitialized

```
1 int i = 3;
2 #pragma omp parallel private(i)
3   printf("%d\n",i); // undefined!
```

- To import a value:

```
1 int i = 3;
2 #pragma omp parallel firstprivate(i)
3   printf("%d\n",i); // undefined!
```

- `lastprivate` to preserve value of last iteration.

Default behaviour

- `default(shared)` or `default(private)`
- useful for debugging: `default(none)`
  because you have to specify everything as shared/private

Persistent thread data

- Private data disappears after the parallel region. What if you want data to persist?
- Directive `threadprivate`

```
1 double seed;
2 #pragma omp threadprivate(seed)
```

- Standard application: random number generation.
- Tricky: has to be global or static.

Arrays

- Statically allocated arrays can be made private.
- Dynamically allocated ones can not: the pointer becomes private.

Synchronization

Need for synchronization

- The loop and sections directives do not specify an ordering, sometimes you want to force an ordering.
- Barriers: global synchronization.
- Critical sections: only one process can execute a statement this prevents race conditions.
- Locks: protect data items from being accessed.

Barriers

- Every workshare construct has an implicit barrier:

```
#pragma omp parallel
{
  #pragma omp for
    for ( .. i .. )
      x[i] = ...
  #pragma omp for
    for ( .. i .. )
      y[i] = .. x[i] .. x[i+1] .. x[i-1] ...
}
```

  First loop is completely finished before second.

- Explicit barrier:

```
#pragma omp parallel
{
  x = f();
#pragma omp barrier
  .... x ...
}
```

Nowait

```
1 #pragma omp parallel
2 {
3   #pragma omp for nowait
4   for ( int i=0; i<N; ++i )
5     x[i] = // something
6   #pragma omp for
7   for ( int i=0; i<N; ++i )
8     y[i] = ... x[i] ...
9 }
```

Needed:

- In the same parallel region (why?)
- Same number of iterations
- Same schedule

Critical sections

- Critical section: One update at a time.

```
#pragma omp parallel
{
  double x = f();
#pragma omp critical
  global_update(x);
}
```

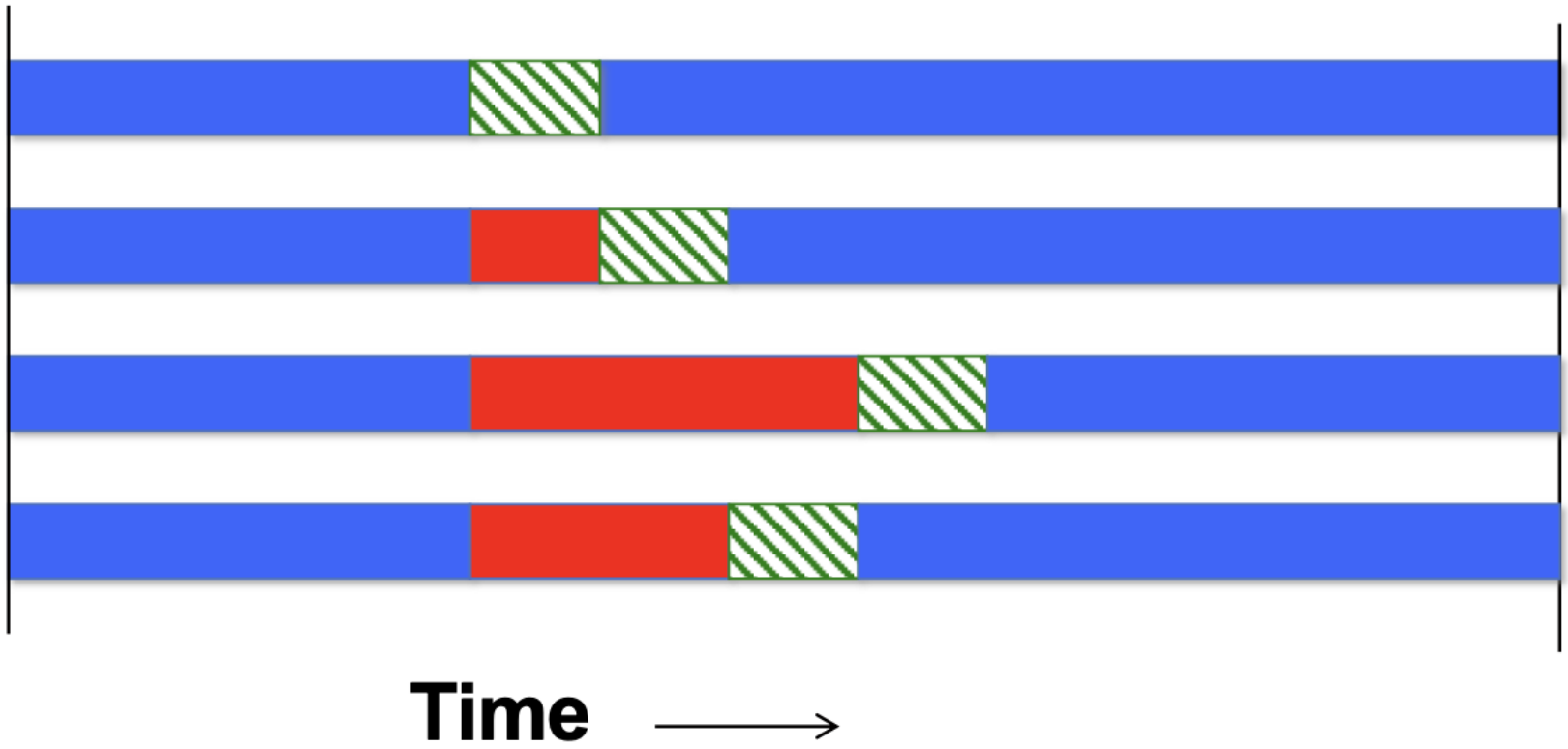
- `atomic` : special case for simple operations, possible hardware support

```
#pragma omp atomic
  t += x;
```

Warning

- Critical sections are not cheap! The operating system takes thousands of cycles to coordinate the threads.
- Use only if minor amount of work.
- Do not use if a reduction suffices.
- Name your critical sections.
- Explore locks if there may not be a data conflict.

Critical sections and idle time



In effect, the execution becomes sequential!

Do not misuse critical sections

```
1 #pragma omp parallel for
2 for ( /* i in whatever */ ) {
3   #pragma omp critical
4   s += /* expression with i */
5 }
```

- Use reduction if appropriate
- Only use if the enough other work in the iteration

Locks

- Critical sections are coarse:
  they dictate exclusive acces to a *statement*
- Suppose you update a big table
  updates to non-conflicting locations should be allowed
- Locks protect a single data item.

Lock create and destoy

- Variable to hold lock
- Create/destroy actual lock

```
1 // lock.c
2 omp_lock_t the_lock;
3 omp_init_lock( &the_lock );
4 omp_destroy_lock( &the_lock );
```

Use of lock

- Same thread sets/unsets lock
- While lock is set, other threads can not pass

```
1 #pragma omp parallel
2   {
3     omp_set_lock( &the_lock );
4     sum += omp_get_thread_num();
5     omp_unset_lock( &the_lock );
6   }
```

Exercise: histogram

Basic idea:

```
1 for ( i /* lots of cases */ )
2   bin[ property(i) ]++;
```

Without a specific application, we let the bin number be random:

MISSING SNIPPET histobasic in codesnippetsdir=../../snippets

Histogram 2

- Make this omp parallel
- Observe that the result is not correct.
- Experiment with number of threads and bins.

Histogram 3

- Solve with histogram
- Experiment with number of threads and bins.

Histogram 3

- Solve with locks
  (How many locks are needed?)
- Experiment with number of threads and bins.

Tasks

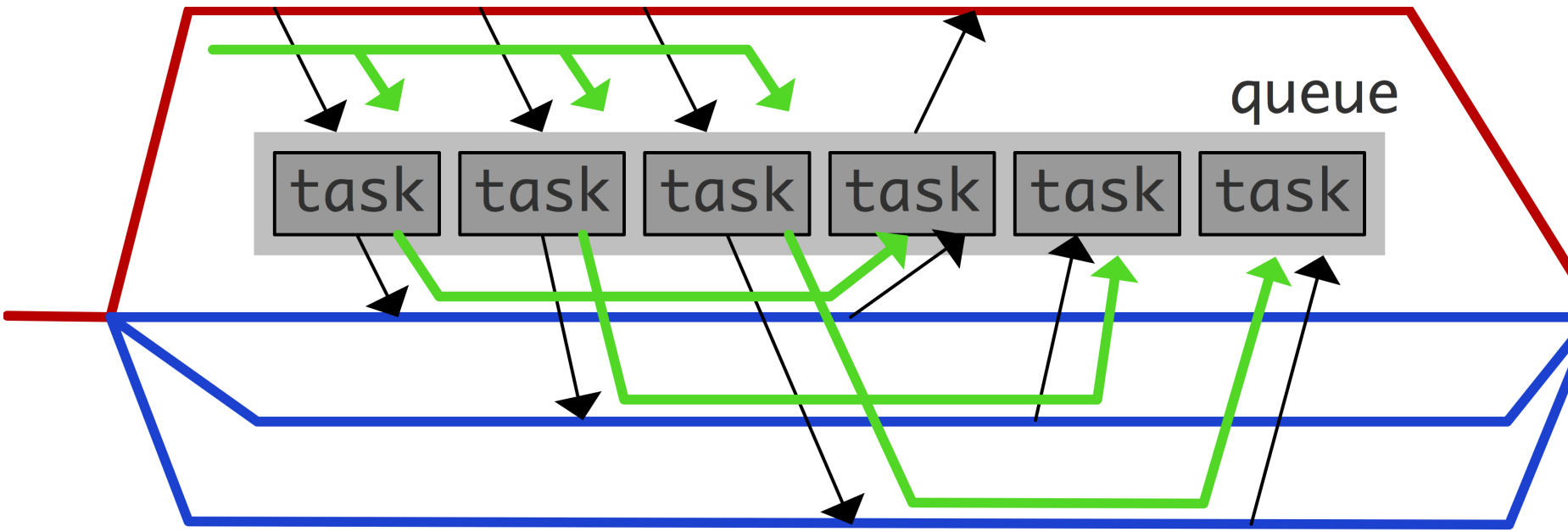
Tasks

- Loops generate 'implicit tasks'
  need static number of iterations
- Also: 'explicit tasks'
  can deal with dynamic work:
- Example linked lists or trees
- Tasks are very flexible:
  you create work, it goes on a queue, gets executed later

```
1 p = head_of_list();
2 while (!end_of_list(p)) {
3 #pragma omp task
4   process( p );
5   p = next_element(p);
6 }
```

Threads, tasks, queues

- There is one queue (per team), not visible to the programmer.
- One thread starts generating tasks.
- Tasks can recursively generate tasks.
- You never know who executes what.



Generation idiom

```
1 #pragma omp parallel
2 #pragma omp single
3 {
4   ...
5   #pragma omp task
6   { ... }
7   #pragma omp taskwait
8   ...
9 }
```

(Complicated logic of scheduling points; hardly ever matters)

Recursive generation

```
1  void f() {
2    do_something();
3    #pragma omp task
4      do_task1();
5    #pragma omp task
6      do_task2();
7    #pragma omp taskwait
8    do_something_more();
9  }
10 ...
11 #pragma omp parallel
12 #pragma omp single
13 #pragma omp task
14   f();
```

- Any thread can create tasks
- including threads that are executing tasks

Exercise 10 (`taskfactor`)

Use tasks to find the smallest factor of a large number (using $2999 \cdot 3001$ as test case): generate a task for each trial factor.

- Turn the factor finding block into a task.
- Run your program a number of times:

  ```
  for i in ‘seq 1 1000‘ ; do ./taskfactor ; done | grep -v 299
  ```

  Does it find the wrong factor? Why? Try to fix this.
- Once a factor has been found, you should stop generating tasks. Let tasks that should not have been generated, meaning that they test a candidate larger than the factor found, print out a message.

Task data space

- Environment captured as *firstprivate* unless explicitly otherwise
- Makes sense: execution is deferred

```
1 #pragma omp parallel
2 #pragma omp single
3 for ( int i /* ... */ )
4   #pragma omp task
5   f(i);
```

- Shared data has to be captured explicitly

Task data: tricky exception

```
1 #pragma omp parallel shared(i)
2 #pragma omp single
3 for ( i /* ... */ ) // note: no 'int'
4   #pragma omp task firstprivate(i)
5   f(i);
```

Deferred and undeferred tasks

- Tasks are unusally 'deferred': they are executed at some undetermined point in the future.
- Tasks can also be undeferred: executed here and now.
- Prime example: with *if* clause

```
1 #pragma omp task if (level>5)
```

Task synchronization

Mechanisms for task synchronization:

- `taskwait`: wait for all previous tasks (not nested)
- `taskgroup`: wait for all tasks, including nested
- `depend`: synchronize on data items.

Task wait

Insider parallel region:

```
1 #pragma omp task
2 ...
3 #pragma omp task
4 ...
5 #pragma omp taskwait
```

- Wait on tasks spawned directly
- Not 'descendant' tasks

Task group

```
1 #pragma omp taskgroup
2 {
3   #pragma omp task
4   foo(0);
5   #pragma omp task
6   for ( int i=1; i<N; ++i )
7     #pragma omp task
8     foo(i)
9 }
```

- Wait for tasks on this level
- . . . and tasks created by tasks on this level

Example: tree traversal

```
1 int process( node n ) {
2   if (n.is_leaf)
3     return n.value;
4   for ( c : n.children) {
5     #pragma omp task
6     process(c);
7 #pragma omp taskwait
8   return sum_of_children();
9 }
```

Exercise: Fibonacci

Fix the errors in this program.

```
1  long fib(int n) {
2    if (n<2) return n;
3    else { long f1,f2;
4      #pragma omp task
5      f1 = fib(n-1);
6      #pragma omp task
7      f2 = fib(n-2);
8      return f1+f2;
9    }
10 }
11
12 #pragma omp parallel
13 #pragma omp single
14   printf("Fib(50)=%ld",fib(50));
```

Task cancelling

```
1 !$omp cancel construct [if (expr)]
```

where construct is `parallel`, `sections`, `do` or `taskgroup`.

More cancelling

Need to set `OMP_CANCELLATION` explicitly.

Set more cancellation points:

```
1 #pragma omp cancellation point <construct>
```

Task dependencies

```
1 #pragma omp task
2   x = f()
3 #pragma omp task
4   y = g(x)
```

```
1 #pragma omp task depend(out:x)
2   x = f()
3 #pragma omp task depend(in:x)
4   y = g(x)
```

Sequence tasks by indicating

- The output of one task
- the needed input of another
- (only between sibling tasks)

Exercise: dependencies

```
1 for i in [1:N]:
2     x[0,i] = some_function_of(i)
3     x[i,0] = some_function_of(i)
4
5 for i in [1:N]:
6     for j in [1:N]:
7         x[i,j] = x[i-1,j]+x[i,j-1]
```

- Use tasks
- Is there a different way to do this?

Fibonacci memo'ization

```
1  long fibvalues[100];
2  void fibonacci(n) {
3    if (n>=2) {
4      #pragma omp task \
5         depend( in:fibvalues[n-2],in:fibvalues[n-1] ) \
6         depend( out:fibvalues[n] )
7      fibvalues[n] = fibvalues[n-2]+fibvalues[n-1];
8  };
9
10 #pragma omp parallel
11 #pragma omp single
12   for (i<50)
13     fibonacci(i);
```

Taskloop

```
1 #pragma omp taskloop grainsize(5)
2 for ( int i=0; i<N; ++i )
3   f(i)
```

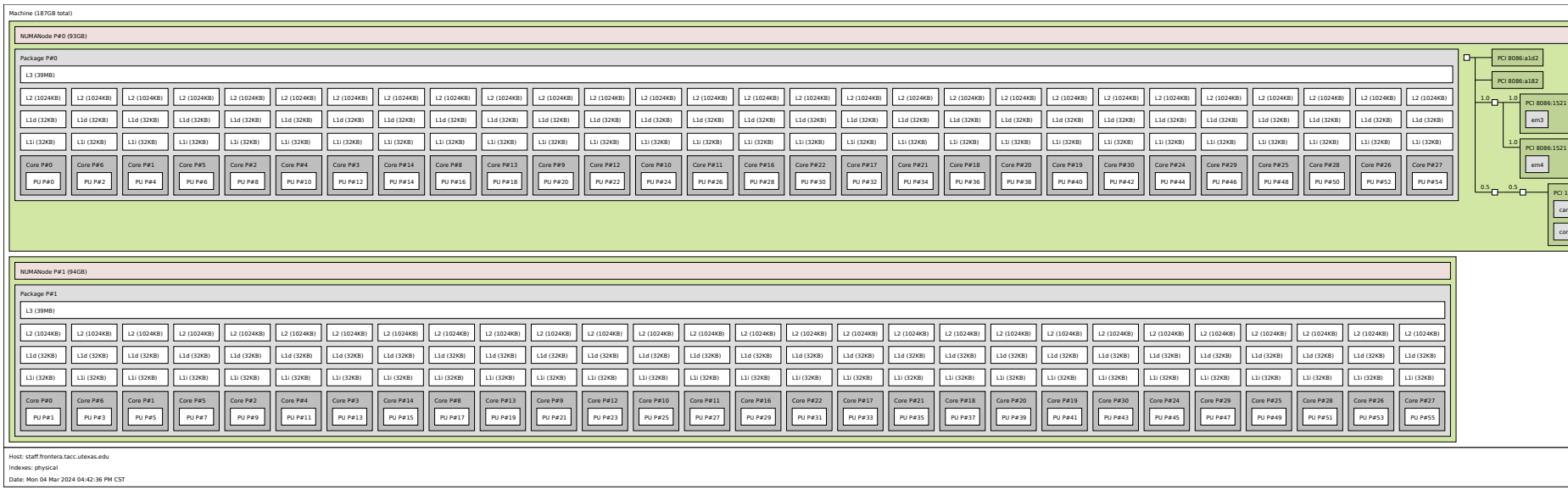
- Turn iterations into tasks
- Implicit taskgroup, unless *nogroup* specified

Affinity and other memory issues

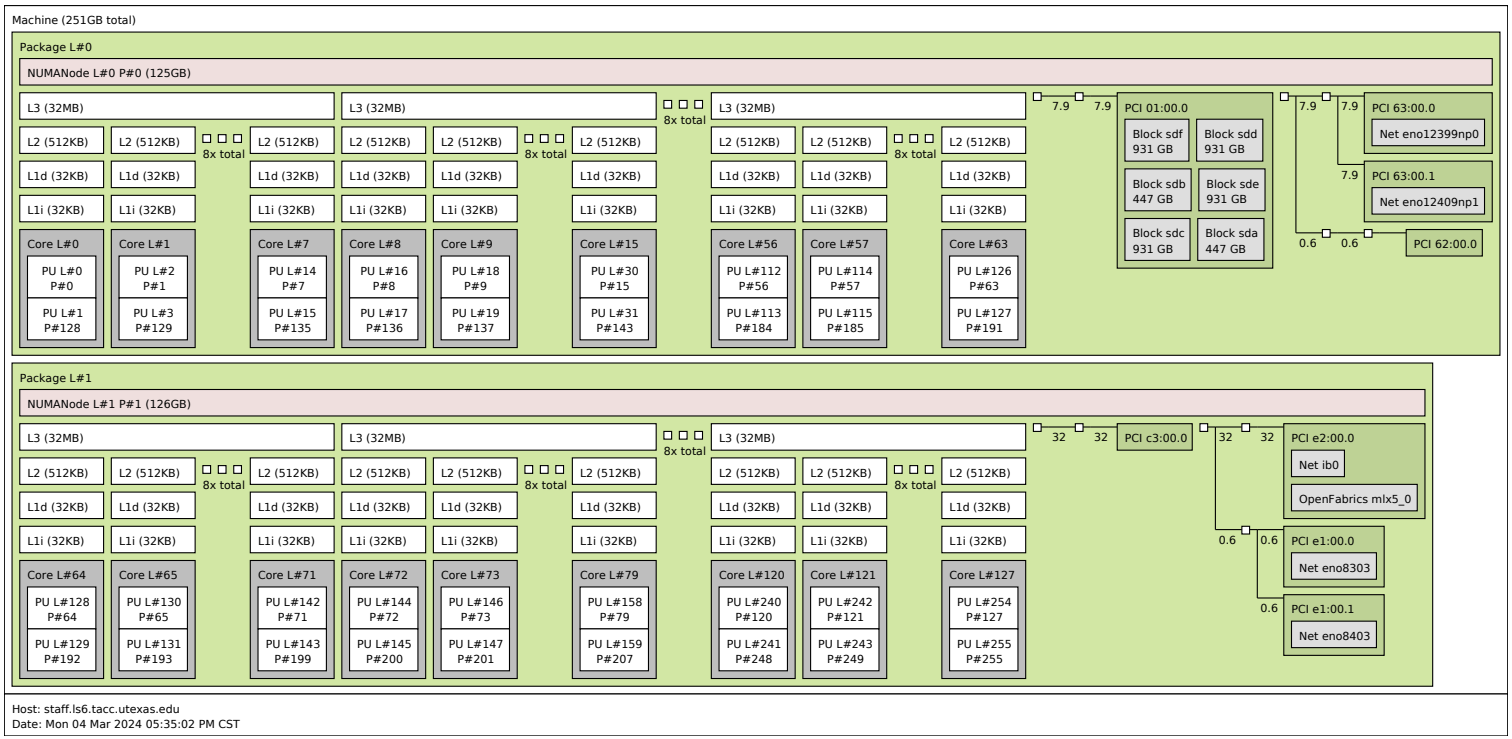
Random issues

- If you have two threads and two cores, how do you place the threads?
- The OS can 'migrate' threads. Is that good?

Frontera


Machine (187GB total)
NUMANode P#0 (93GB)
Package P#0
L3 (39MB)
NUMANode P#1 (94GB)
Package P#1
L3 (39MB)
PCI 8086:a1d2
PCI 8086:a182
PCI 8086:1521
em3
PCI 8086:1521
em4
Host: staff.frontera.tacc.utexas.edu
Indexes: physical
Date: Mon 04 Mar 2024 04:42:36 PM CST

Lonestar6

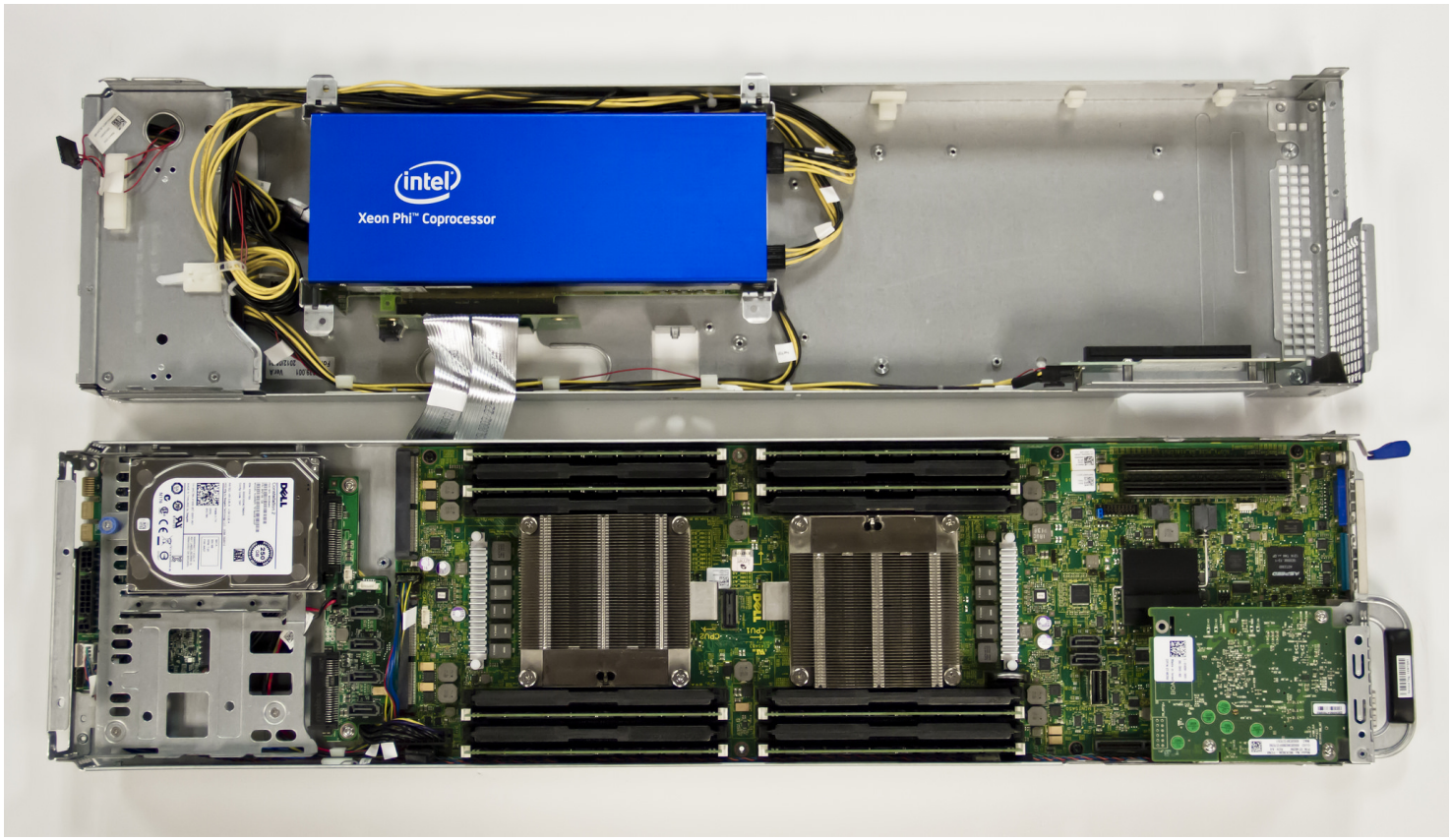

Machine (251GB total)
Package L#0
NUMANode L#0 P#0 (125GB)
L3 (32MB)
L2 (512KB)
L1d (32KB)
L1i (32KB)
8x total
Core L#0
PU L#0 P#0
PU L#1 P#128
Core L#1
PU L#2 P#1
PU L#3 P#129
Core L#7
PU L#14 P#7
PU L#15 P#135
Core L#8
PU L#16 P#8
PU L#17 P#136
Core L#9
PU L#18 P#9
PU L#19 P#137
Core L#15
PU L#30 P#15
PU L#31 P#143
Core L#56
PU L#112 P#56
PU L#113 P#184
Core L#57
PU L#114 P#57
PU L#115 P#185
Core L#63
PU L#126 P#63
PU L#127 P#191
7.9
PCI 01:00.0
Block sdf 931 GB
Block sdd 931 GB
Block sdb 447 GB
Block sde 931 GB
Block sdc 931 GB
Block sda 447 GB
PCI 63:00.0
Net eno12399np0
PCI 63:00.1
Net eno12409np1
PCI 62:00.0
0.6
Package L#1
NUMANode L#1 P#1 (126GB)
Core L#64
PU L#128 P#64
PU L#129 P#192
Core L#65
PU L#130 P#65
PU L#131 P#193
Core L#71
PU L#142 P#71
PU L#143 P#199
Core L#72
PU L#144 P#72
PU L#145 P#200
Core L#73
PU L#146 P#73
PU L#147 P#201
Core L#79
PU L#158 P#79
PU L#159 P#207
Core L#120
PU L#240 P#120
PU L#241 P#248
Core L#121
PU L#242 P#121
PU L#243 P#249
Core L#127
PU L#254 P#127
PU L#255 P#255
32
PCI c3:00.0
PCI e2:00.0
Net ib0
OpenFabrics mlx5_0
PCI e1:00.0
Net eno8303
PCI e1:00.1
Net eno8403
Host: staff.ls6.tacc.utexas.edu
Date: Mon 04 Mar 2024 05:35:02 PM CST

Affinity

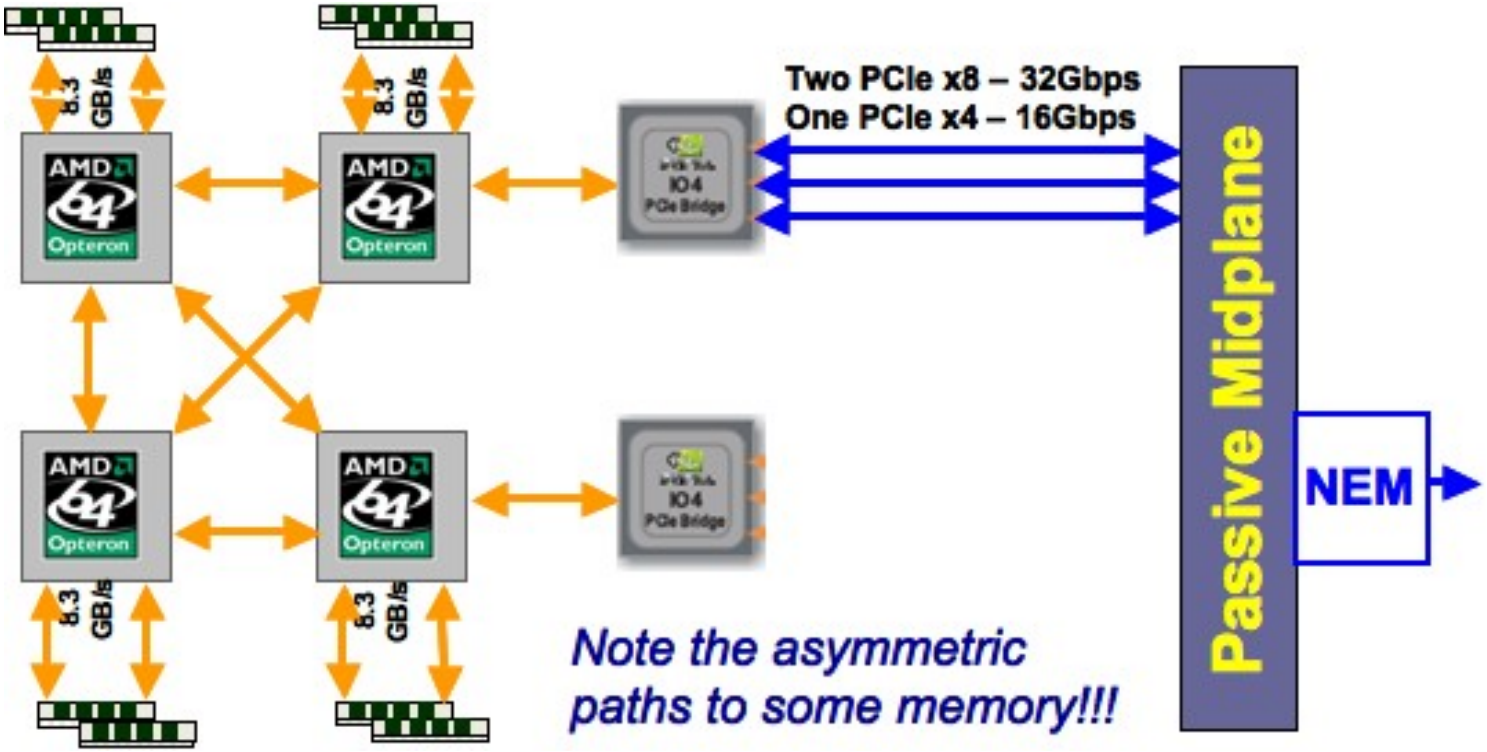
- *OMP_PROC_BIND*: either *true/false* or *close/spread*
- *OMP_PLACES*: *sockets/cores/threads* or explicit list.

Affinity utility

NUMA



NUMA


8.3 GB/s
AMD 64 Opteron
IO4 PCIe Bridge
Two PCIe x8 – 32Gbps
One PCIe x4 – 16Gbps
Passive Midplane
NEM
Note the asymmetric paths to some memory!!!

Virtual memory

- Memory is organized in ‘pages’, typically 4K, or 1M for ‘large’ pages.
- User code uses ‘logical addresses’ into these pages
- Black magic translates logical addresses to physical.

First-touch

```
1 double *x = (double*) malloc( lots );
2 // no memory has been reserved by malloc
3 for ( lots )
4   // now the pages get created
5   x[i] = something
```

- Pages get 'instantiated' (placed in memory) when they are first touched.
- In multi-socket designs, pages are instantiated in the memory of the touching thread.

First touch

```
1 double *x = (double*) malloc( lots );
2 for ( lots )
3   x[i] = 0.;
4 #pragma omp parallel for
5 for ( lots )
6   x[i] = f(i);
```

- first loop instantiates pages on the socket of the primary thread
- second loop: half the threads need to pull data from the other socket.
- Solution?

Memory model

Sequential consistency

Multi-threaded code:

- Instructions from threads are not ordered
- Instruction in each thread are ordered
- $\Rightarrow$ total execution is 'as if' some interleaving of the threads
- not a unique interleaving
  $\Rightarrow$ result is not unique

Dekker's algorithm

```
1     int a=0,b=0,r1,r2;
2 #pragma omp parallel sections shared(a, b, r1, r2)
3     {
4 #pragma omp section
5       {
6         a = 1;
7         r1 = b;
8       }
9 #pragma omp section
10      {
11        b = 1;
12        r2 = a;
13      }
14    }
```

Complicated interaction between threads!

Execution 1

a=0	b=0
a=1 r1=b=0	b=1 r2=a=1
r1+r2=1	

- Thread 1 completely before 2
- By symmetry, sum= 1 the other way too

Execution 2

a=0	b=0
a=1 r1=b=1	 b=1 r2=a=1
r1+r2=2	

- Both threads first statement
- ... before both threads second statement
- $\Rightarrow$ sum= 2 regardless micro-timing.

Compiler optimizations

- Compiler doesn't know about threads
- Compiler is allowed to exchange independent statements

a=0 r1=b=0 a=1	b=0 r2=a=0 b=1
r1+r2=0	

Not sequentially consistent!

Weak memory model

- Threads may keep a local copy of values
- $\Rightarrow$ a write by one thread may not be visible by another.
- Fix: explicitly 'flush' data to memory.

Flush

```
1      int a=0,b=0,r1,r2;
2  #pragma omp parallel sections shared(a, b, r1, r2)
3      {
4  #pragma omp section
5        {
6          a = 1;
7  #pragma omp flush (a,b)
8          r1 = b;
9          tasks++;
10       }
11 #pragma omp section
12       {
13         b = 1;
14 #pragma omp flush (a,b)
15         r2 = a;
16         tasks++;
17       }
18     }
```

Remaining topics

Accelerators

OpenMP 4 has mechanisms for offloading.

SIMD

Processors have 4 or 8-wide SIMD.
convert OpenMP loop to SIMD vector instructions.