Tutorial on OpenMP programming
Victor Eijkhout
SSiASC 2016

Victor Eijkhout

Justification

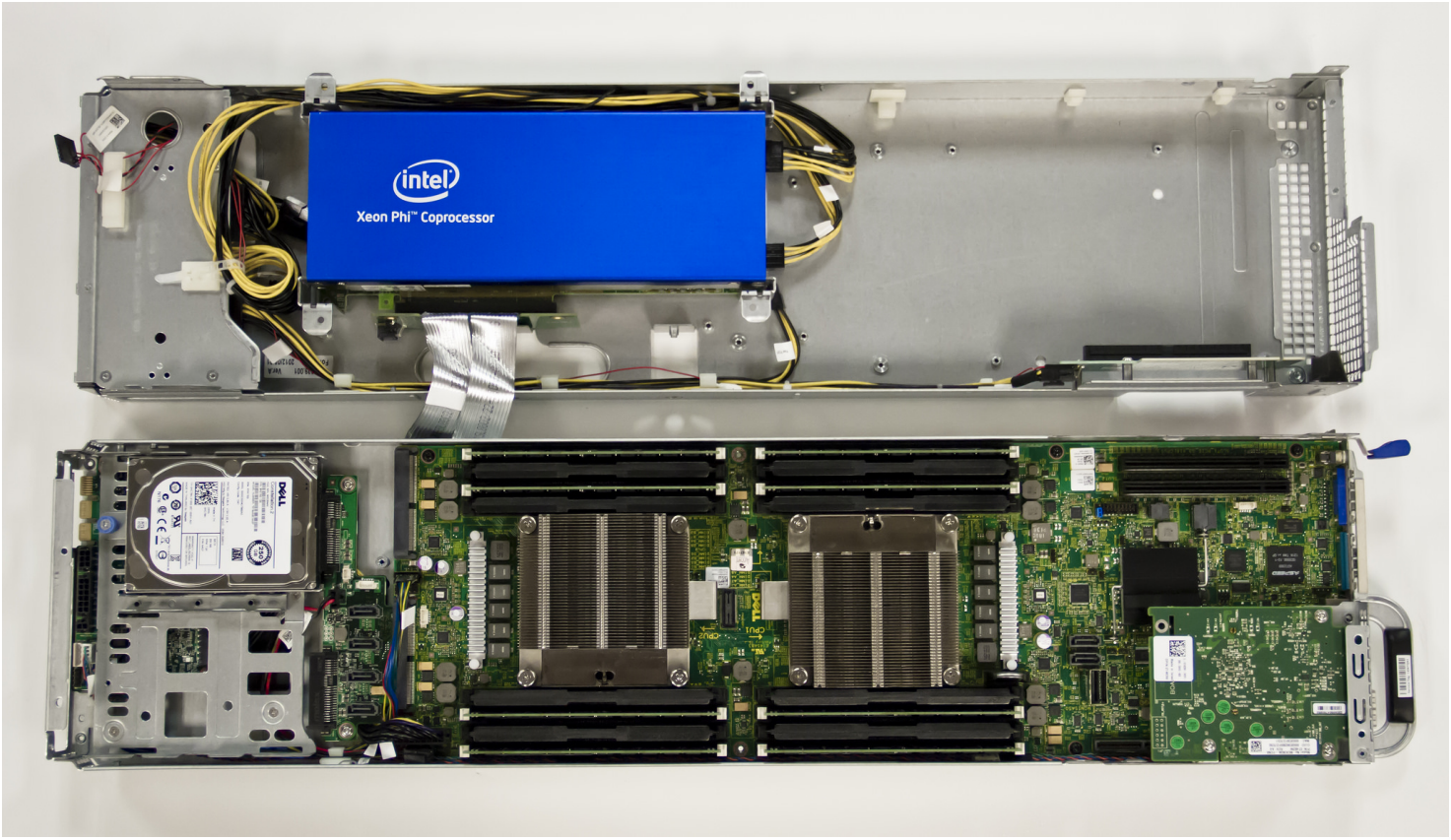
OpenMP is a flexible tool for incrementally parallelizing a shared memory-bas
code. This course introduces the main concepts through lecturing and exercis

Part I

The Fork-Join model

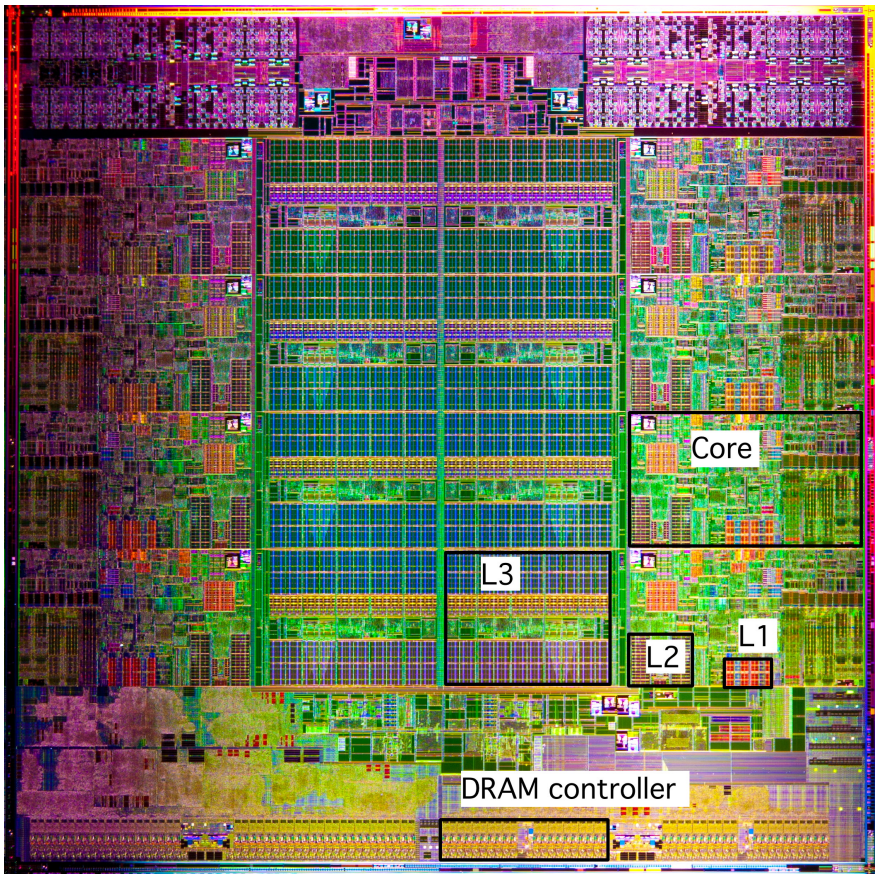
Computer architecture terminology

One cluster node:



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

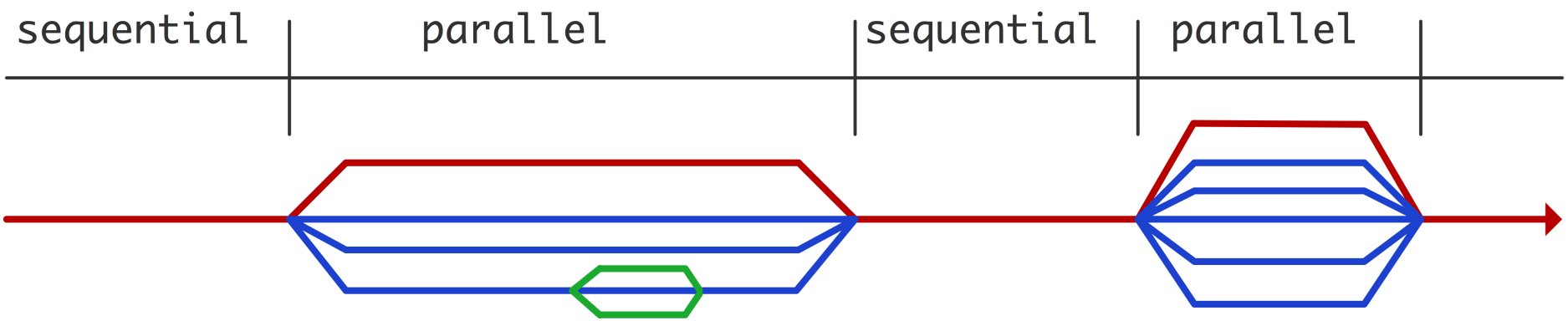
Structure of a socket



Eight cores per socket, making 16 per node.
They all access the same data.

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.

To write an OpenMP program

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

in C, and

```
use omp_lib
```

or

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

for Fortran.

To compile an OpenMP program

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -openmp
```

To run an OpenMP program

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

Stampede has 16 cores; more than 16 threads does not make sense.

Quick experiment:

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

Exercise 1 (`parallel`)

Take the hello world program of exercise **??** and insert the above functions, b
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?

Threads and threads

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

Exercise 2

Extend the program from exercise 1. Make a complete program based on the lines:

Code:

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

Output:

```
With 4 threads, sum s
Sum is 6
Sum is 5
Sum is 1
Sum is 4
Sum is 6
Sum is 5
Sum is 6
Sum is 5
Sum is 3
Sum is 4
```

Compile and run again. (In fact, run your program a number of times.) Do y 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	

Part II

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.
- How would you do this specifically?

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 tea
- For loops that is the `for` (or `do`) construct.

Workshare construct for loops

C: directive followed by statement or block:

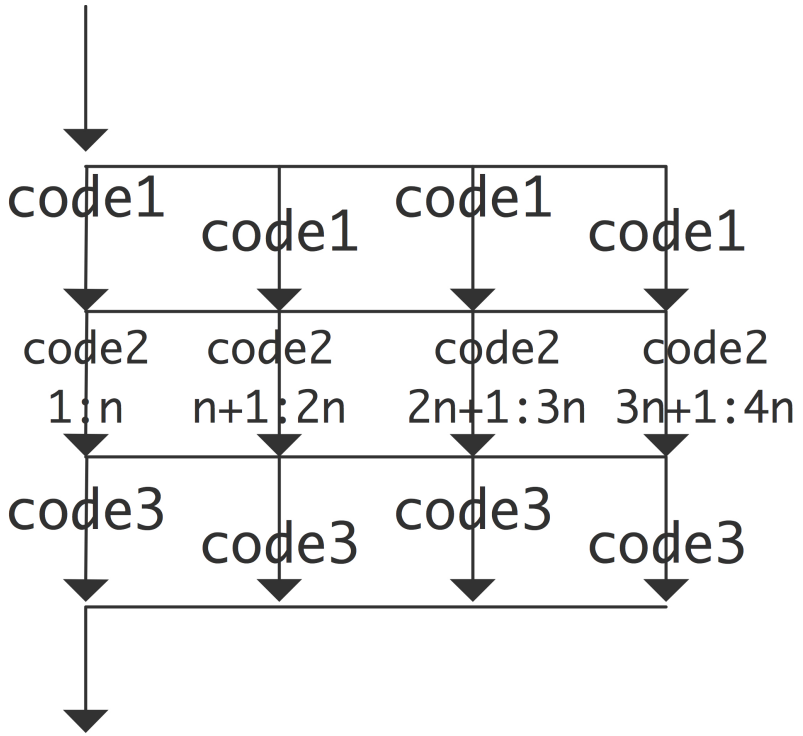
```
#pragma omp parallel
{
#pragma omp for
  for (i=0; i<N; i++)
    ... something with i ...
}
```

Fortran: matching end directive

```
!$omp parallel
!$omp do
  do i=1,n
    ... something with i ...
  end do
!$omp end do
!$omp end parallel
```

Stuff inside a parallel region

```
#pragma omp parallel
{
  code1();
#pragma omp for
  for (i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```



Exercise 3 (`pi`)

Compute π by *numerical integration*. We use the fact that π is the area of the unit circle, a 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.

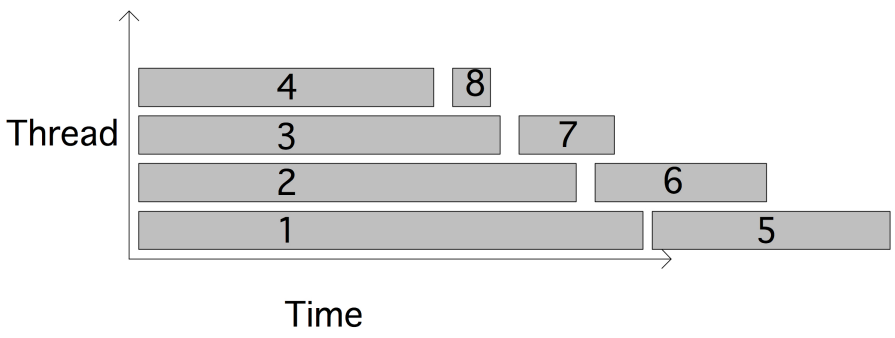
1. Put a `parallel` directive around your loop. Does it still compute the right result? Do time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or `parallel do`). Now is the result correct? execution speed up? (The answers should now 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` directi Now it should be correct and efficient.

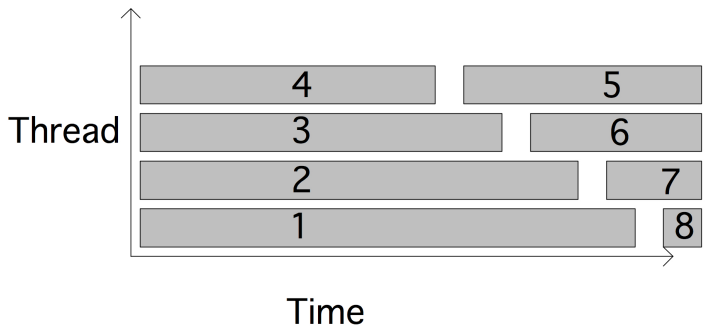
Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread a sequential code?

Loop schedules

- Default: static scheduling of iterations.
  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 t
  `schedule(dynamic)`

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

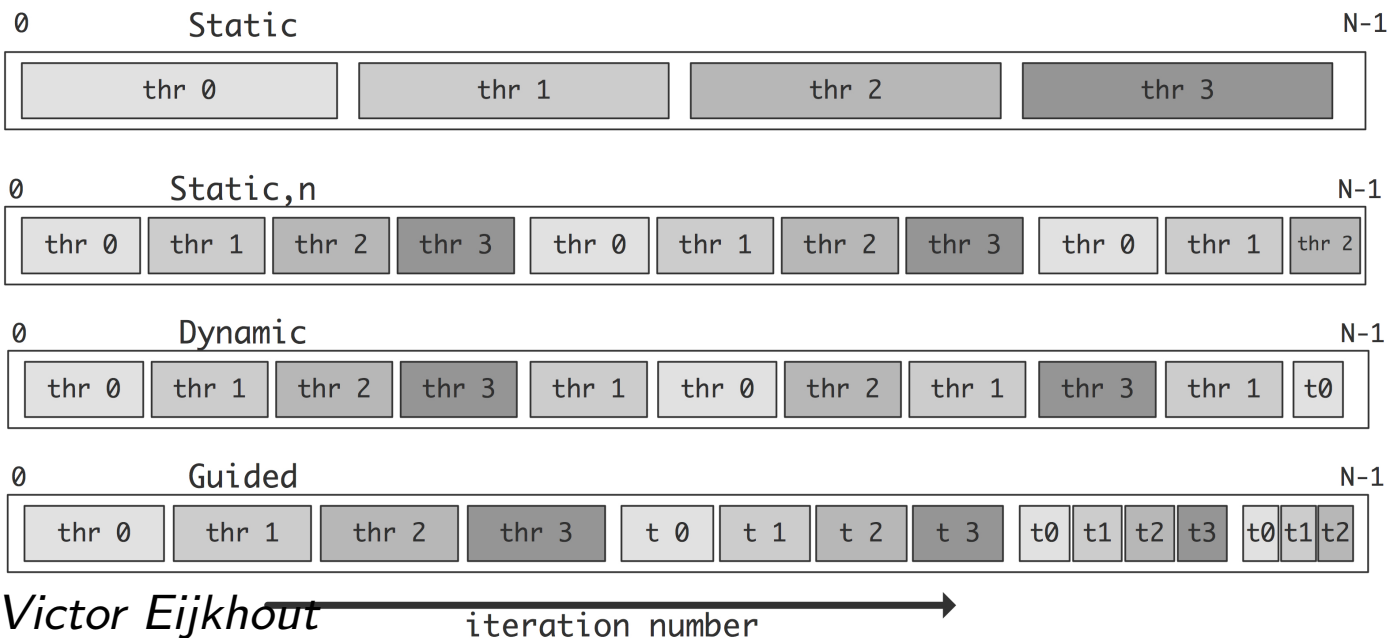




Chunk size

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)`
- Help from OpenMP:
  guided schedule uses decreasing chunk size (with optional minimum chunk):
  `schedule(guided,6)`



Reductions

- Inner product loop:

```
s = 0.;
for (i=0; i<N; i++)
  s += x[i]*y[i];
```

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

Exercise 4 (piadapt)

We continue with exercise 3. We add 'adaptive integration': where needed, the program ref
the step size[1]. This means that the iterations no longer take a predictable amount of time.

```
1  for (int i=0; i<nsteps; i++) {
2      double
3        x = i*h,x2 = (i+1)*h,
4        y = sqrt(1-x*x),
5        y2 = sqrt(1-x2*x2),
6        slope = (y-y2)/h;
7      if (slope>15) slope = 15;
8      int
9      samples = 1+(int)slope, is;
10     for (int is=0; is<samples;
       ↪is++) {
11         double
12         hs = h/samples,
13         xs = x+ is*hs,
14         ys = sqrt(1-xs*xs);
15         quarterpi += hs*ys;
16         nsamples++;
17     }
18 }
19   pi = 4*quarterpi;
```

1. Use the *omp* `parallel for` construct to parallelize the loop. As in the previous lab, y
   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 possib
   get completely linear speedup by adjusting the schedule.
   Start by using *schedule*(*static*,*n*). Experiment with values for *n*. When can you ge
   better speedup? Explain this.

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 `reductio` clause to fix this.
2. Your code should now see a decent speedup, using up to 8 cores. Howev is possible to get completely linear speedup. For this you need to adjust schedule.
   Start by using `schedule(static,$n$)`. Experiment with values for *n*. can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try `schedule(dynamic)`. This wi 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.

More loop topics

- Multiple loops can be collapsed: `collapse(2)`. Improves performance.
- Ordered iterations: normally OpenMP can execute iterations in any sequ
  You can force ordering if you absolutely have to. Bad for performance!
- There is a barrier at the end of a `for`: use `nowait` to let threads continu

Part III

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, but compiler suppo
  seems mediocre)
- Story in itself: `task`

Sections

Independent separate calculations:

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

```
#pragma omp sections
{
  #pragma omp section
  fx = f(x)
  #pragma omp section
  gx = g(x)
  #pragma omp section
  hx = h(x)
}
```

Adding them together:

```
  s = f(x)+g(x)+h(x);
```

Use reduction.

Single

```
int a;
#pragma omp parallel
{
  #pragma omp single
    a = f(); // some computation
  #pragma omp sections
    // various different computations using ‘a’
}
```

- Is executed by a single thread.
- Has implicit barrier, so the result is available to everyone after.
- `master` is similar, does not have barrier.

Exercise 5

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

Part IV

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)

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

- You can change all this with clauses:

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

Variables in loops

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

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

Copying to/from private data

- Private data is uninitialized

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

- To import a value:

```
int i = 3;
#pragma omp parallel firstprivate(i)
  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`

```
double seed;
#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.

Part V

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 ...
}
```

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 o 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.

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.

Part VI

Tasks

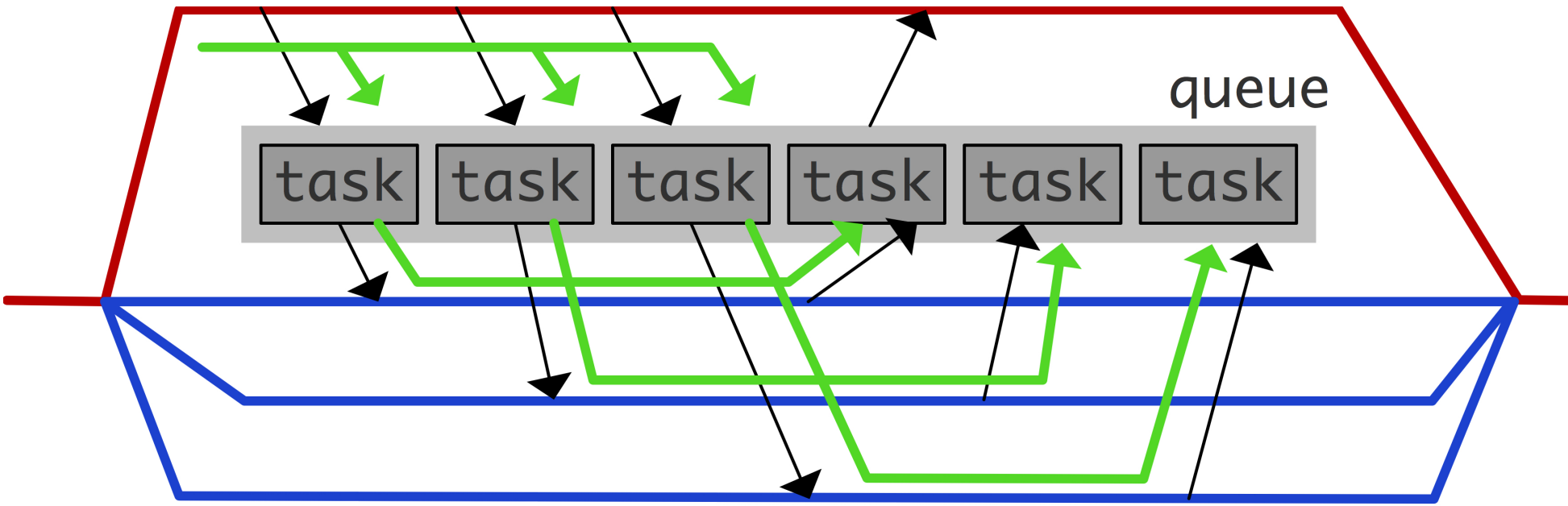
More flexibility

- You have seen loops and sections.
- How about linked lists or trees?
- Tasks are very flexible:
  you create work, it goes on a queue, gets executed later

```
p = head_of_list();
while (!end_of_list(p)) {
#pragma omp task
  process( p );
  p = next_element(p);
}
```

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.



Exercise 6 (taskfactor)

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

```
1     int factor=0;
2   #pragma omp parallel
3   #pragma omp single
4     for (int f=2; f<4000; f++) {
5       { // see if 'f' is a factor
6         if (N%f==0) { // found factor!
7           factor = f;
8         }
9       }
10      if (factor>0)
11        break;
12    }
13    if (factor>0)
14      printf("Found a factor: %d\n",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 2
```

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.

Example: tree traversal

```
int process( node n ) {
  if (n.is_leaf)
    return n.value;
  for ( c : n.children) {
#pragma omp task
    process(c);
#pragma omp taskwait
  return sum_of_children();
}
```

Example: Fibonacci

```
long fib(int n) {
  if (n<2) return n;
  else { long f1,f2;
#pragma omp task
  f1 = fib(n-1);
#pragma omp task
  f2 = fib(n-2);
#pragma omp taskwait
  return f1+f2;
}

#pragma omp parallel
#pragma omp single
  printf("Fib(50)=%ld",fib(50));
```

(what is conceptually wrong with this example?)

Fibonacci once more

```
long fibs[100];
void fib(n) {
  if (n>=2) {
    #pragma omp task \
       depend( in:fibs[n-2],in:fibs[n-1] ) \
       depend( out:fibs[n] )
    fibs[n] = fibs[n-2]+fibs[n-1];
};

#pragma omp parallel
#pragma omp single
  for (i<50)
    fib(i);
```

Part VII

Remaining topics

Affinity

How do you place threads on cores?

- Two socket design NUMA
- Intel KNL has quadrants and hardware multi-threading
- `OMP_PROC_BIND` and `OMP_PLACES`

Accelerators

OpenMP 4 has mechanisms for offloading.

SIMD

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