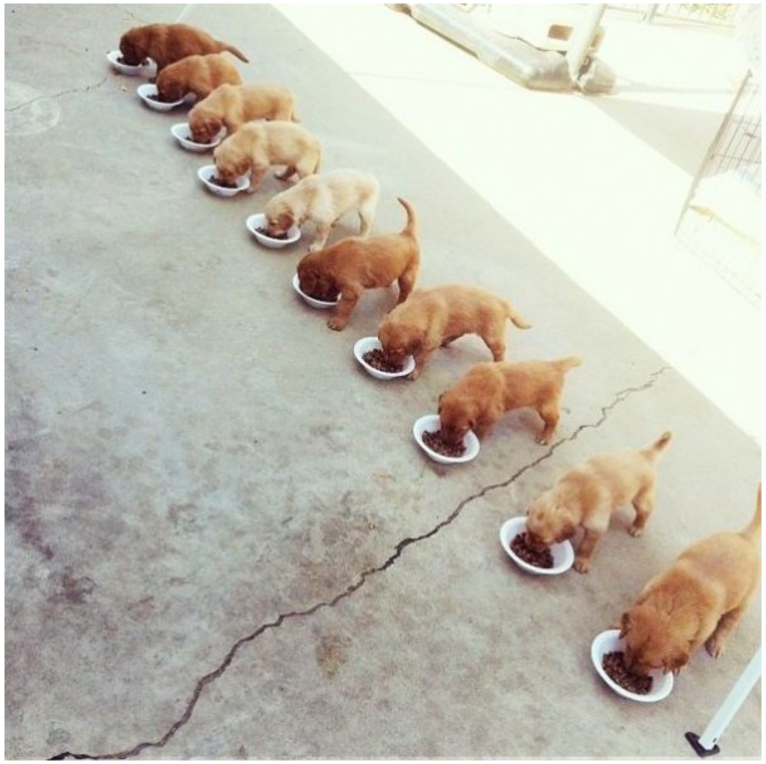
OpenMP 3

Multi-threaded Programming



theory

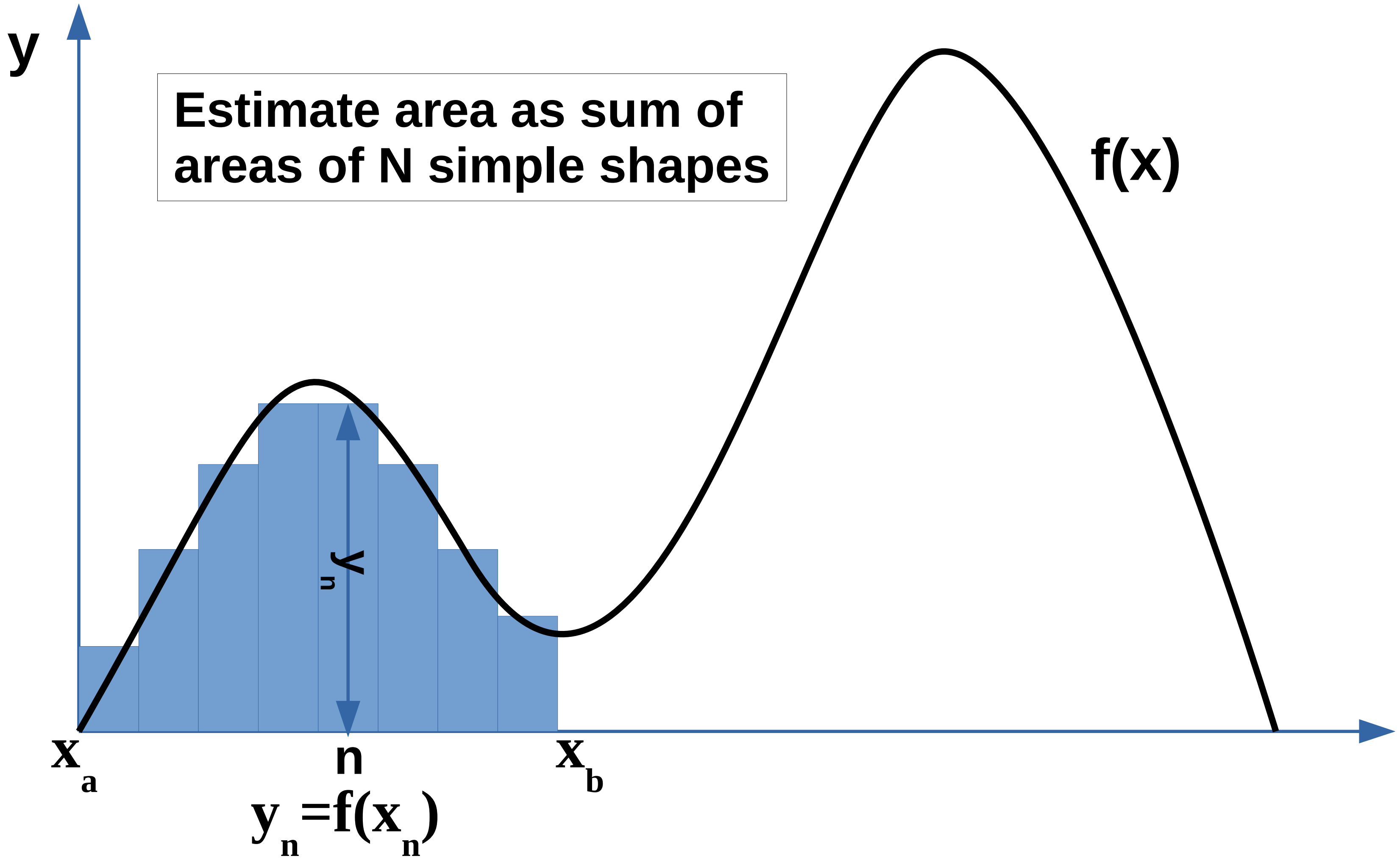
practice

University of BRISTOL

Overview

- Continuing with our numerical integration example and reviewing performance.
- We now encounter ‘false sharing’ as a performance gotcha.
- Also consider how many times you create and destroy teams of threads, because both operations carry an overhead.
- Let’s be clear about what we can hope to gain by going parallel.

Example: numerical integration



(e.g. estimate pi by numerically integrating $4/1+x^2$)

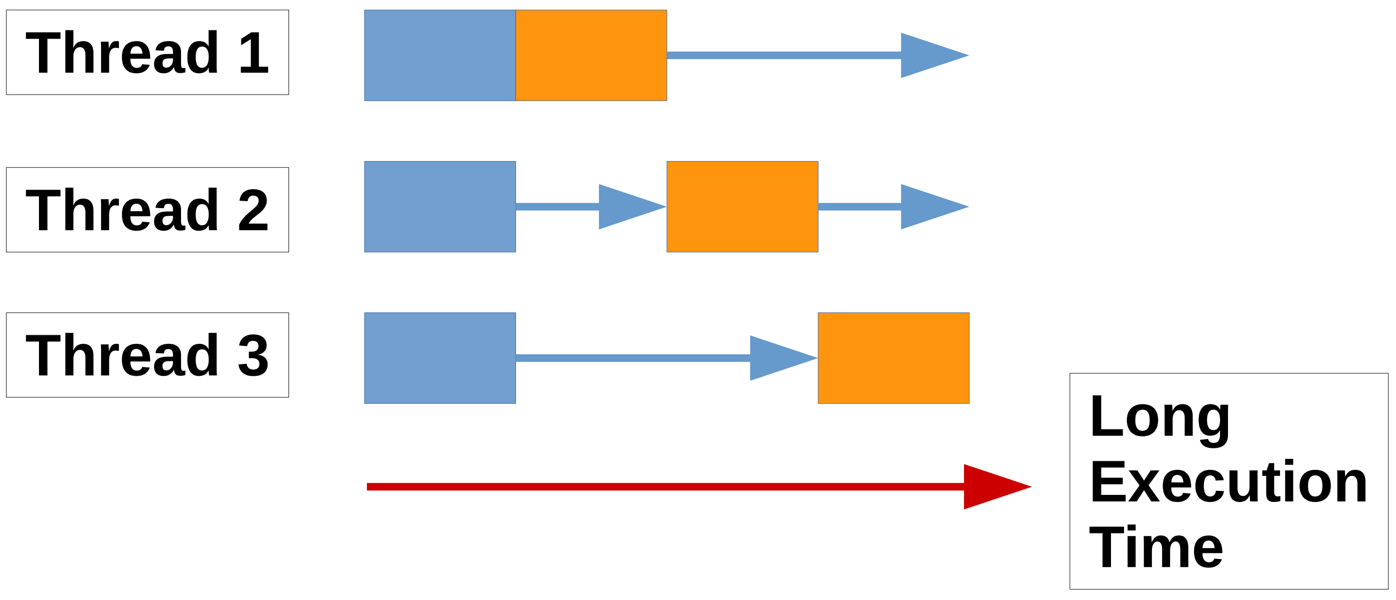
Shared Accumulator

```
#pragma omp parallel for shared(sum) private(x)
  for (ii=0; ii<num_steps; ii++) {
    x = (ii+0.5)*step;
#pragma omp critical
    sum += 4.0/(1.0+x*x); /* all threads access! */
  }

  /* master thread only */
  pi = step * sum;

  printf("pi is:\t\t%.16f\n",pi);
  printf("error is:\t%.16f\n",fabs(pi – PI25DT));
```

Syntactically fine, but load imbalance
Gives us poor performance

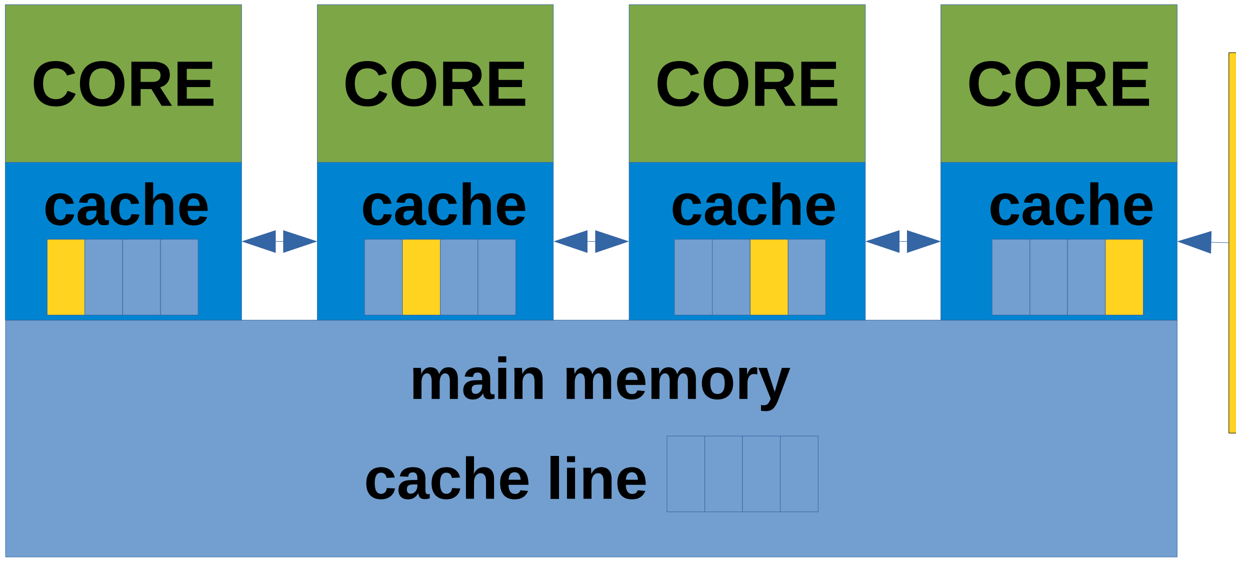
University of BRISTOL

Shared Accumulator



→	Idle time in critical region
🟦	Work done in parallel for
🟧	Work done in critical region

Performance: Cache Coherency & False Sharing



- Caches must be kept synchronised.
- Done in h/w, BUT still Takes time.

'**False sharing**' is an example of cache thrashing in a concurrent context. It occurs when two threads access different words that are on the same cache line.

e.g. A small array that will fit in cache, one cell used per processor.

x86: Cache lines are 64 bytes

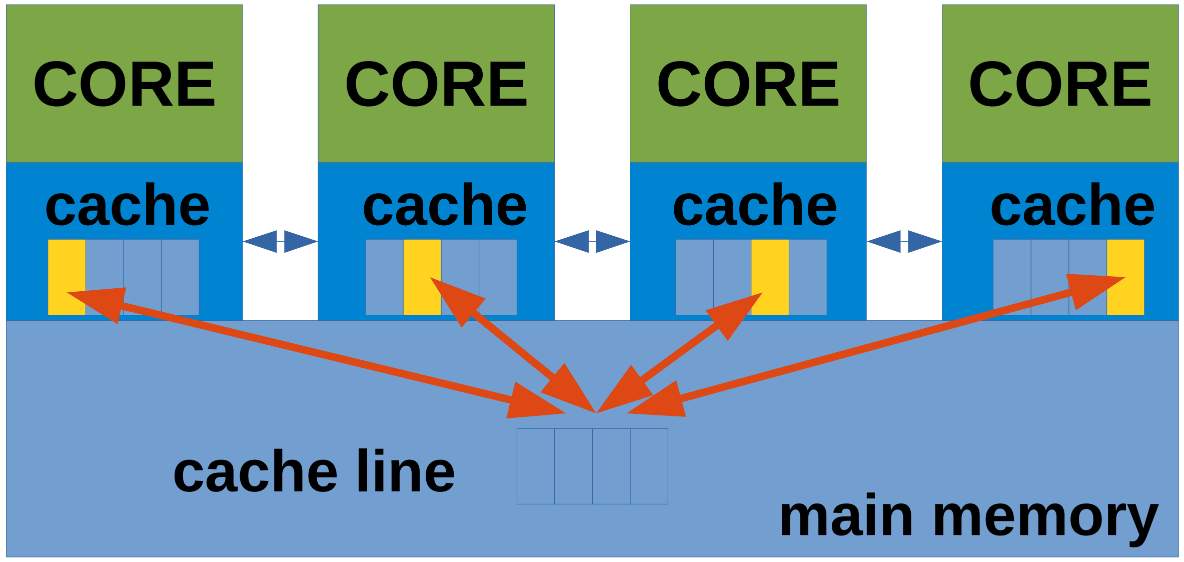
Explore: `/sys/devices/system/cpu/cpu0/cache`

Array of Accumulators

```
double sum[NUM_THREADS];      /* array to store partial sums */

#pragma omp parallel shared(sum,pi) private(x,thread_id)
  {
    thread_id = omp_get_thread_num();
    sum[thread_id] = 0.0;
#pragma omp for
    for (ii=0; ii<num_steps; ii++) {
      x = (ii+0.5)*step;
      sum[thread_id] += 4.0/(1.0+x*x);
      /* .. note that whole array will likely be in cache */
    }
  }

  /* back to just master thread */
  /* total the partial sums */
  for (ii=0; ii<NUM_THREADS; ii++) {
    pi += sum[ii];
  }
  pi *= step;
```

Nice try, but..

Array of Accumulators



Constantly 'thrashing cache'

Private Accumulators

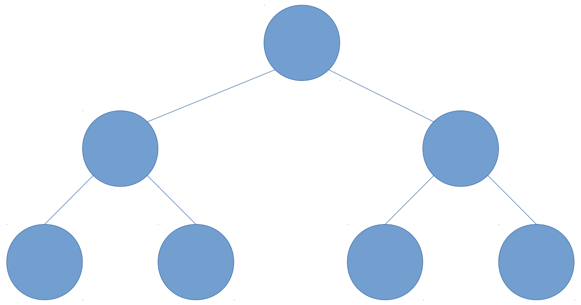
```
#pragma omp parallel shared(pi) private(x) \
                    firstprivate(sum)
  {
#pragma omp for
    for (ii=0; ii<num_steps; ii++) {
      x = (ii+0.5)*step;
      /* partial sum is now private in each thread
       * note the use of firstprivate to ensure an
       * initial value for the accumulator */
      sum += 4.0/(1.0+x*x);
    }
    /* a critical section is a sequence of mutual
     * exclusion (mutex) locks which ensures access
     * for only one thread at a time */
#pragma omp critical
    pi += sum;                    /* but total is shared */
  }
```

Much better, but still not best..

Reduction

```
#pragma omp parallel for reduction(+:sum) private(x)
    for (ii=0;ii<num_steps; ii++){
      x = (ii+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
      /* ..note that sum is now in a reduction */
    }
```

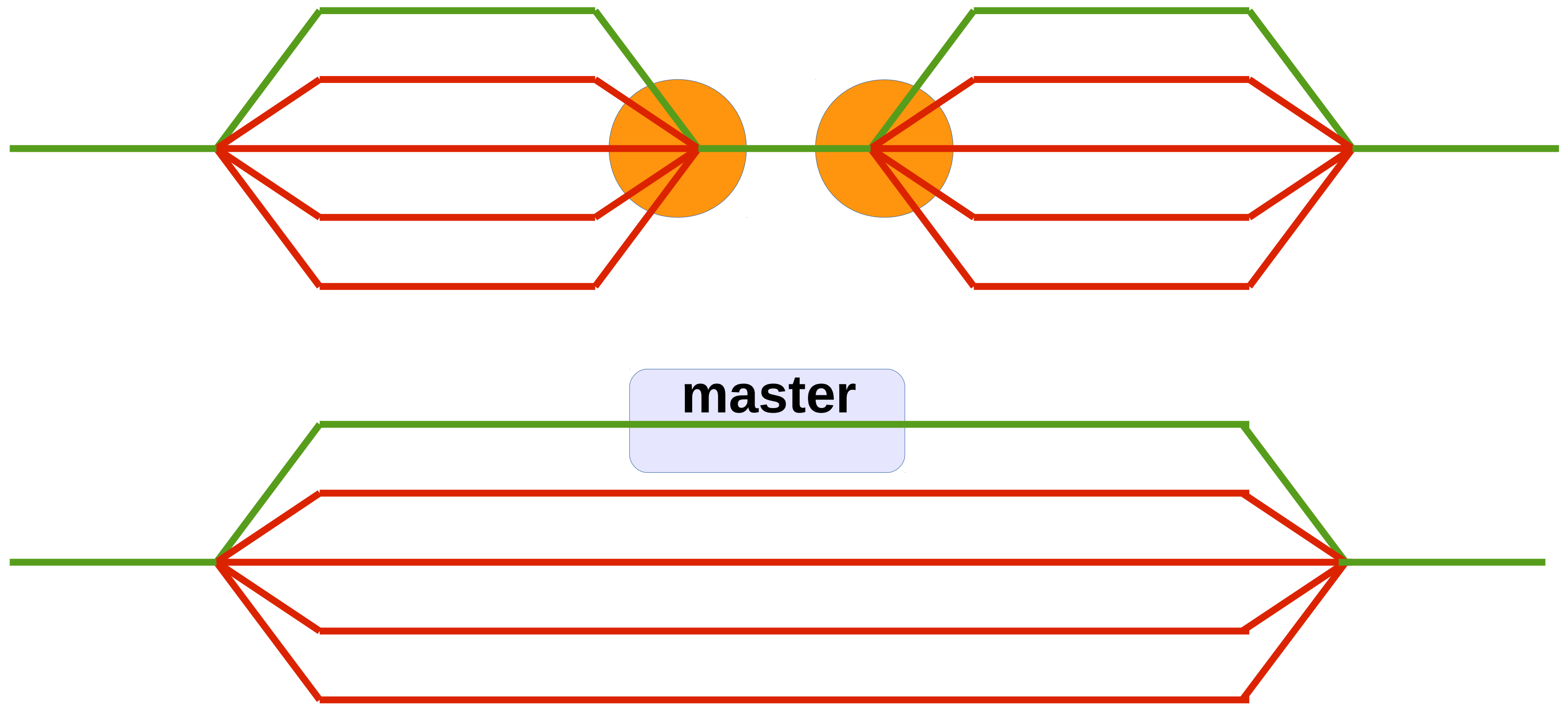
- **Much neater! (and potential better load balance)**
- **Several reduction operators available: +,-,*,logical & bitwise ops**
  - **Fortran also allows 'min' and 'max'**



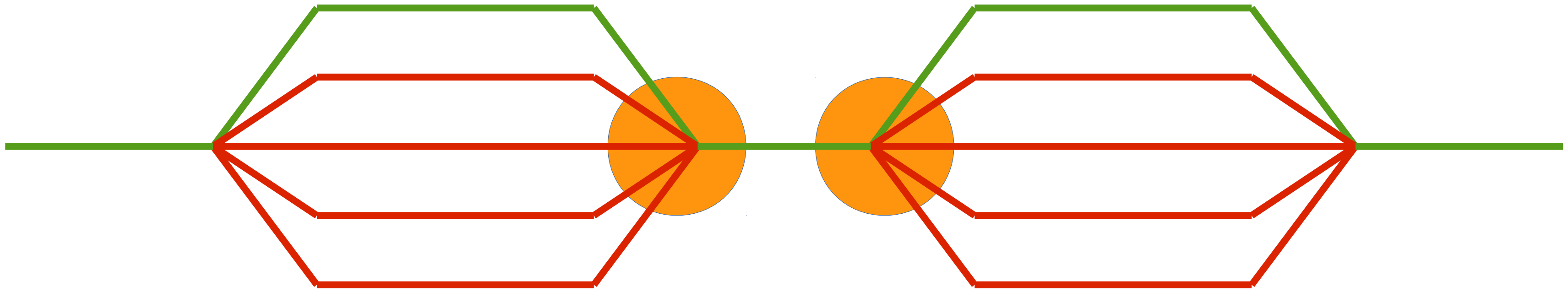
Better load balance through tree structure

University of BRISTOL

Performance: Cost of Team Creation & Destruction

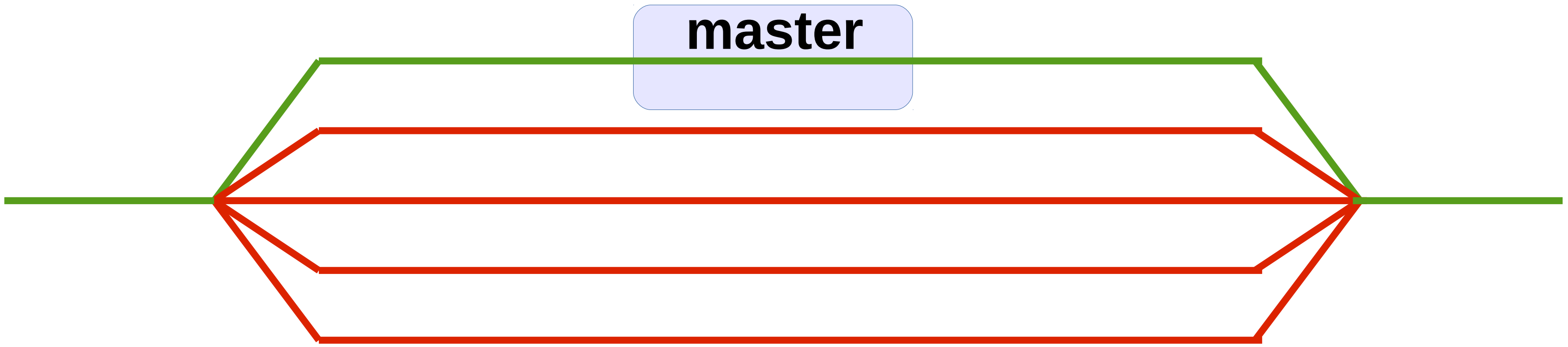


Creating & Destroying Teams



```
#pragma omp parallel for
…
#pragma omp parallel for
…
```

Creating & Destroying Teams



```
#pragma omp parallel
#pragma omp for
...
#pragma omp for
...
```

Threading Overheads

On my (old) (4-core, single-socket, i3) desktop..

```
  for(j=0; j<REPEAT; j++) {
#pragma omp parallel for
    for (i=0; i<N; i++) {
      a[i] = b[i];
    }
  }
```

```
real 0m3.526s
user0m13.566s
sys  0m0.024s
```

```
#pragma omp parallel private(j)
  for(j=0; j<REPEAT; j++) {
#pragma omp for
    for (i=0; i<N; i++) {
      a[i] = b[i];
    }
  }
```

```
real 0m1.090s
user0m3.944s
sys  0m0.004s
```

Here we see the cost of repeatedly creating and destroying teams of threads.

Threading Overheads

```
#pragma omp parallel for shared(sum)
private(x,x2)
  for (ii=0; ii<num_steps; ii++) {
    x = (ii+0.5)*step;
    x2 = 4.0/(1.0+x*x);
#pragma omp critical
    sum += x2; /* all threads access! */
  }
```

Here we see that the critical region accounts for the majority of the run time.

```
                                  ON SUMMARY (mean):
---------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
             msec    total msec                   usec/call
---------------------------------------------------------------------------------
100.0          48     4:56.696          1          1  296696151 .TAU application
100.0       0.0423     4:56.647          1          1  296647246 parallel begin/end [OpenMP]
100.0       0.0535     4:56.647          1          1  296647204 parallelfor (parallel begin/end) [OpenMP location:
file:/panfs/panasas01/isys/ggdagw/hpc-course/openmp/examples/example2/parallel_shared_pi.c <26, 32>]
100.0       0.0357     4:56.647          1          1  296647150 for enter/exit [OpenMP]
100.0      1:37.962     4:56.647          1    2.5E+07  296647115 parallelfor (loop body) [OpenMP location:
file:/panfs/panasas01/isys/ggdagw/hpc-course/openmp/examples/example2/parallel_shared_pi.c <26, 32>]
 64.9       35,857     3:12.667    2.5E+07    2.5E+07          8 critical enter/exit [THROTTLED]
 52.9      1:58.603     2:36.809    2.5E+07    2.5E+07          6 critical (critical enter/exit) [OpenMP location:
file:/panfs/panasas01/isys/ggdagw/hpc-course/openmp/examples/example2/parallel_shared_pi.c <30, 31>]
[THROTTLED]
```

Potential Benefits: **Amdahl's Law**

$$speedup = \frac{T_1}{T_p}$$

Best you can ever get

$$speedup_{max} = \frac{1}{1-P}$$

If you've coded it perfectly

$$speedup_{ideal} = \frac{1}{\frac{P}{N} + S}$$

	Ideal *parallel* speed-up		
N	P=0.5	P=0.9	P=0.99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90

- **T_1** is the execution time on a single processor
- **T_P** is the execution time on a parallel computer
- **S** is the serial fraction of the program
- **P** is the parallelisable fraction of the program
- **N** is the number of processors available

'strong' scaling

Amdahl, Gene (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". AFIPS Conference Proceedings (30): 483-485

University of BRISTOL

Potential Benefits: **Gustafson's Law**

'weak' scaling

$$speedup = \frac{T_1}{T_p} = \frac{T_a(x) + N \times T_b(x)}{T_a(x) + T_b(x)} \to N$$

• **x**	a measure of problem size
• **N**	processors
• **$T_a(x)$**	fraction of time spent executing the serial part of the program
• **$T_b(x)$**	fraction of time spent executing the parallel part

*As **x** increases*:

- **T_a** → 0
- **T_b** → 1

It is assumed that T_a is the minor fraction. Its relative importance will diminish <u>with an increase in x</u> and ultimately, speed-up will approach N.

“Reevaluating Amdahl's Law”, John L. Gustafson, Communications of the ACM 31(5), 1988. pp. 532-533.

Summary

- Use OpenMP wisely:
  - Watch out for load imbalance and false sharing.
  - Use barriers (only) when you need them.
  - Don't create and destroy more thread teams than you need to.
  - Explore the use of 'schedule' if you suspect asymmetries and imbalance.
  - Be savvy about what extra performance more threads can bring you.