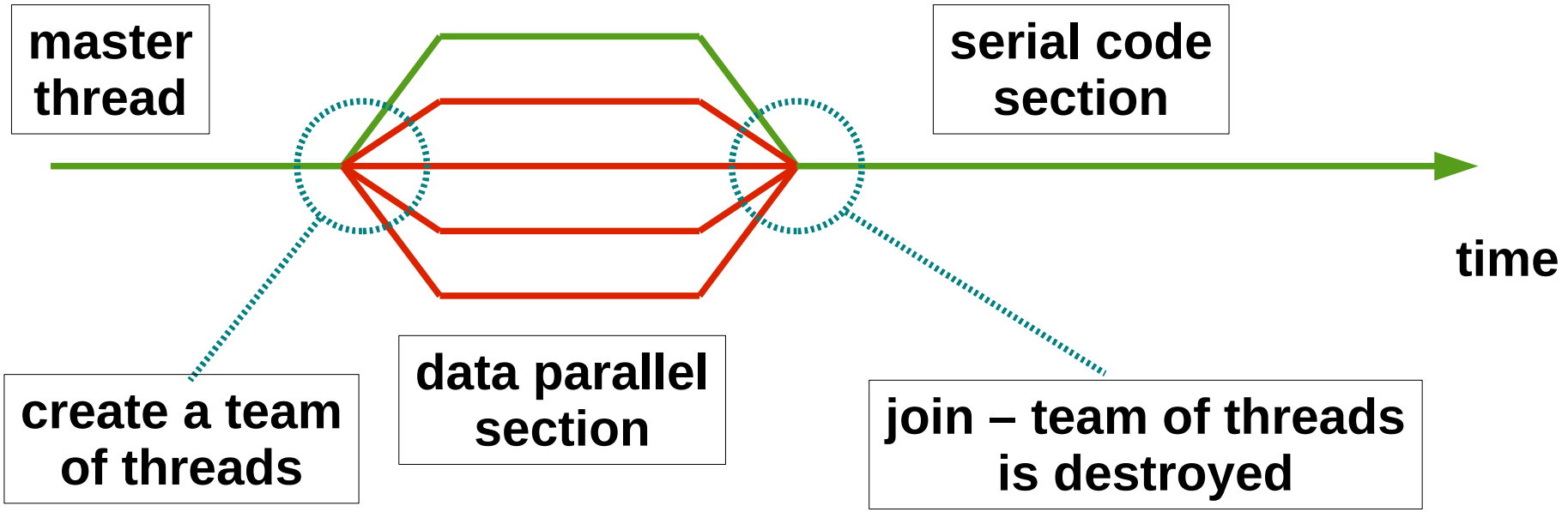
OpenMP1

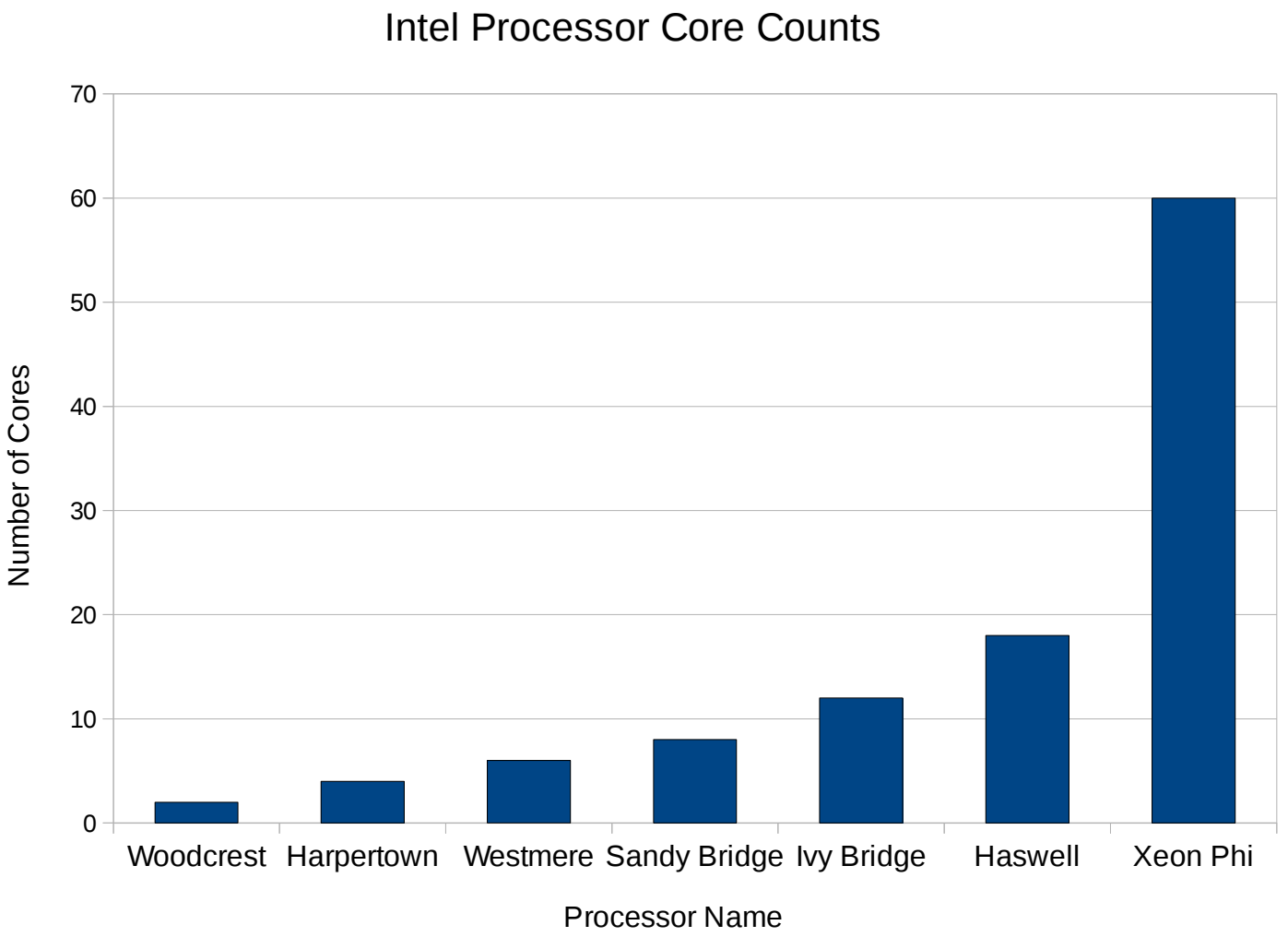
Sunway-TaihuLight: >10M cores, ~40K nodes, 93PFlops
Take a look at http://www.top500.org - lots of interest

OpenMP™



University of BRISTOL

But Why are Threads Important?



Threads are not Processes

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler.
A thread is a light-weight process.

The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process.

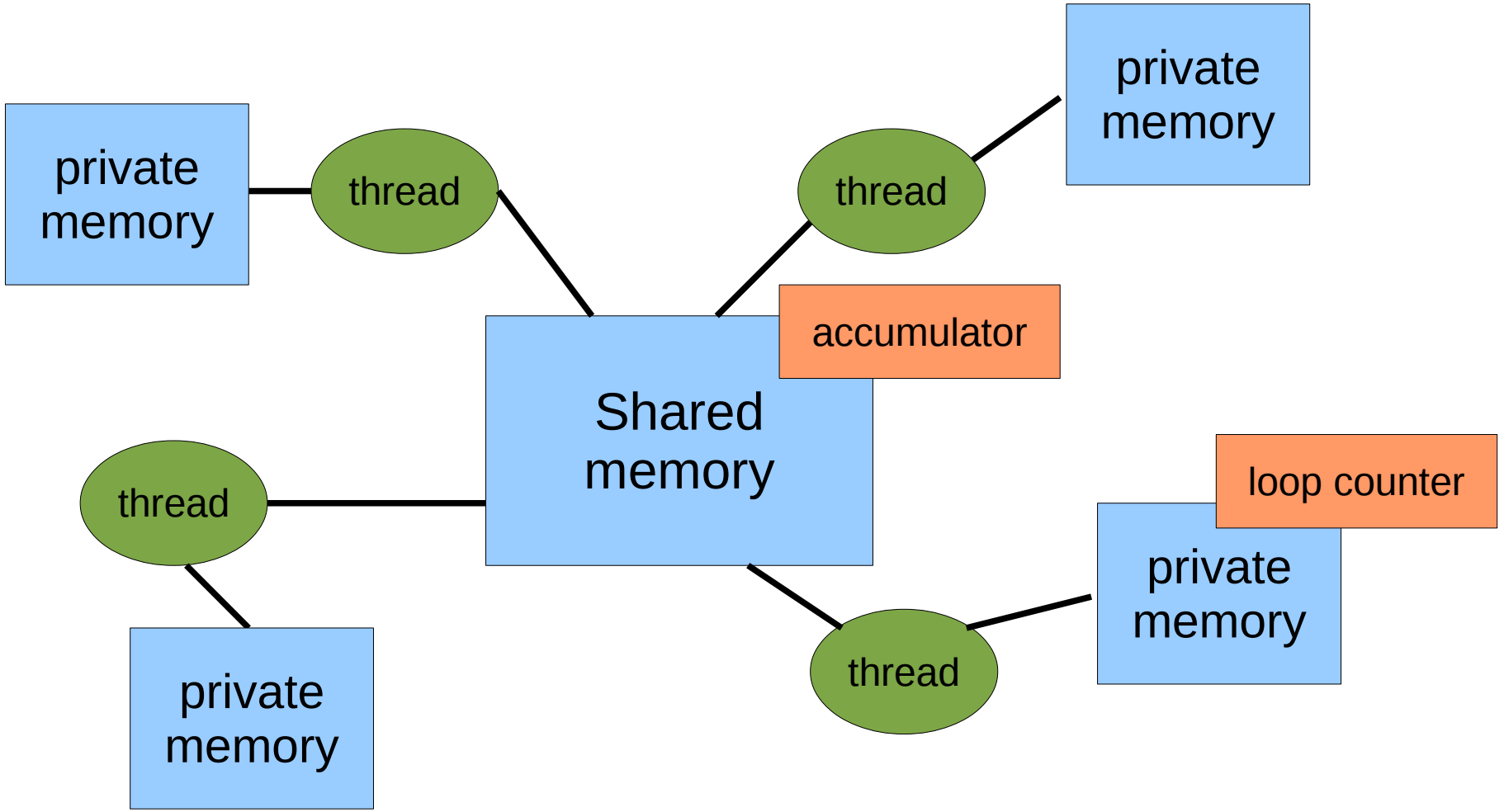
Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).

Thank you, Wikipedia.

..and that's why we can do shared memory parallel programming

Threads can use both shared & private memory



But.. Suddenly Things are more Complex: Thread Safety

A piece of code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads

What are the dangers with these two code snippets?

```
static int total_count
```

```
func(.., double *data_store,..)
```

The Old Way: Posix Threads

```
#include <pthread.h>
#define NUM_THREADS 3  /* in addition to the Master thread.. */

void* run(void *threadID) {
  printf("Hello, world from pthread %ld\n", (long)threadID);
  pthread_exit(NULL);
}

int main(int argc, char** argv)
{ ...
  pthread_t threads[NUM_THREADS];  /* to hold opaque thread IDs */

  for (ii=0; ii<NUM_THREADS; ii++) {
    retval = pthread_create(&threads[ii], NULL, run, (void *)ii);
  }
  printf("Hello, world from the MASTER thread\n");
  pthread_exit(NULL);
}
```

OK, but a bit tedious & error prone

OpenMP Instead: Serial Code with #pragma compiler directives

```
#include <omp.h>

int main (int arc, char *argv[]) {
  int tid;

  /* Fork a team of threads
     giving them their own copies of variables */
#pragma omp parallel private(tid)
  {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello, world from thread = %d\n", tid);
  }  /* All threads join master thread and disband */
}
```

All threads execute parallel block: {..}

OpenMP and Order of Execution

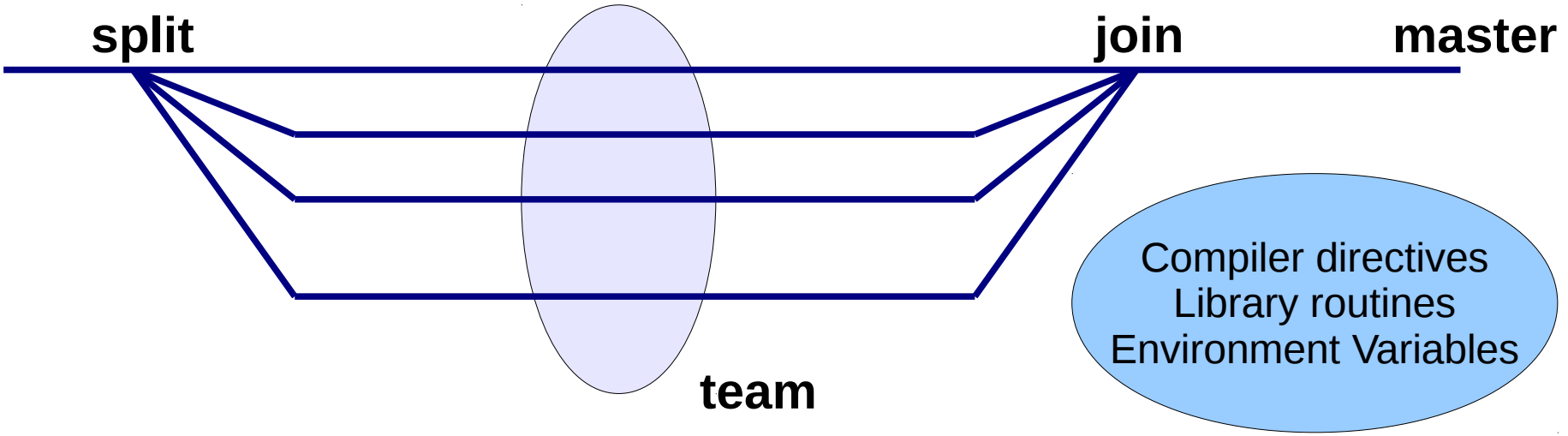
```
Hello, world from thread = 1
Hello, world from thread = 0
Hello, world from thread = 2
Hello, world from thread = 3
Number of threads = 4
```

Note:

- The order in which threads execute is not determined
- The print statement has been made thread-safe (this won't be the case in, e.g. MPI)

OpenMP Key Features

- Pragmatic design criteria:
  - Sequential equivalence (compile w/ or wo/ -fopenmp)
  - Incremental parallelism
  - Much use of loop-splitting programming model

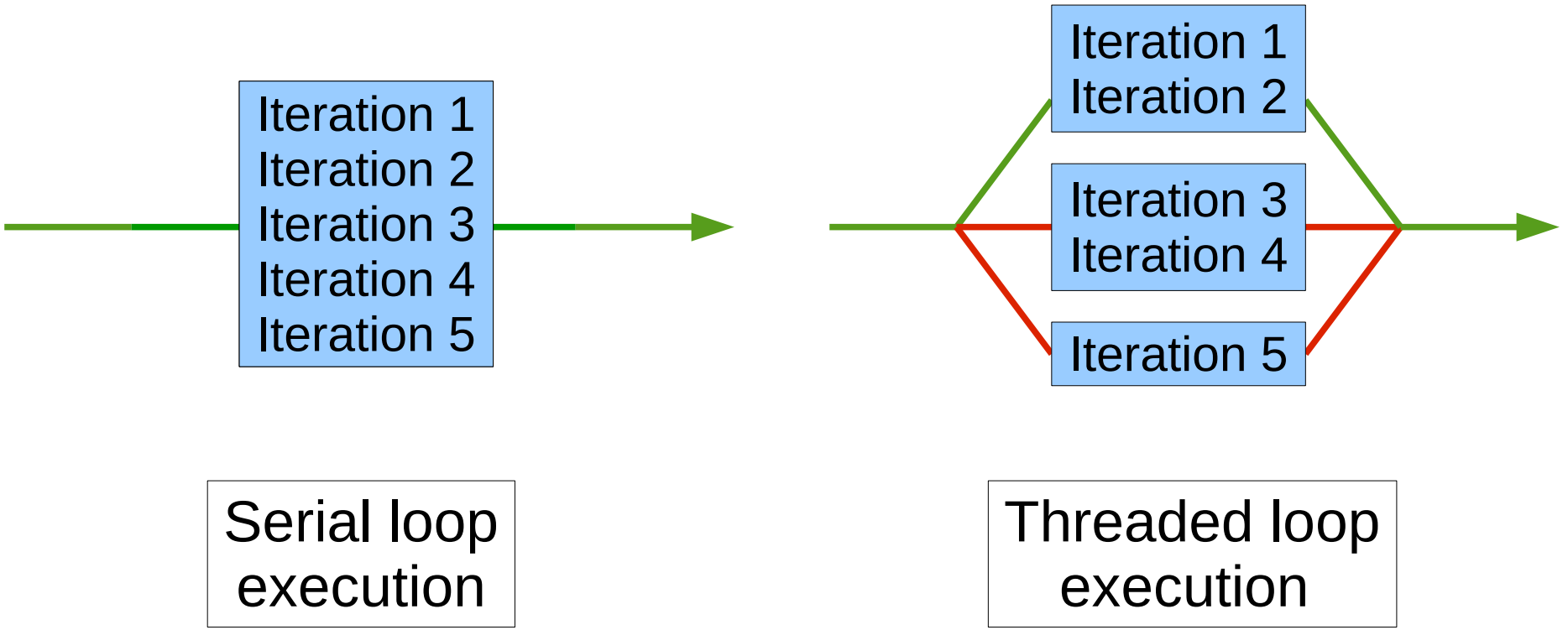


OpenMP Key Features

- Compiler directives: **`#pragma`**
- Thread team creation: **`parallel`**
  - Implicit join in C (at end of structured block)
- Data attributes: **`shared, private,..`**
- Team operations: **`reduction`**
- Work sharing:
  - Loops: **`for`**
  - Less structured: **`section, task`**

OpenMP is available
For several languages:
C, Fortran, C++,..

Worksharing For Loop



Serial loop execution

Threaded loop execution

OpenMP Key Features (1/2)

- Syncronisation
  - **Implicit** at end of worksharing loops.
  - Mutex (one processor at a time): **`critical`**
  - Wait for all threads: **`barrier`**
  - One thread executes: **`single, master`**
- You can make calls to runtime library
  - How many threads in team?
    - **`omp_get_num_threads()`**
  - Which thread am I?
    - **`omp_get_thread_num()`**

OpenMP Key Features (2/2)

- Control no. of threads via an environment var:
  - **`OMP_NUM_THREADS`**
- Or in the program:
  - **`omp_set_num_threads(N)`**
- Can compile with/without activation flag
  - gcc: **`-fopenmp`**

Parallel For Loop

- Parallel region must have been started:

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

- Or roll both directives into one line:

```
#pragma omp parallel for
```

- Can have additional clauses, for scope of variables or loop scheduling instructions:

```
#pragma omp parallel for private(...)
#pragma omp parallel for schedule(..)
#pragma omp parallel for nowait
```

Further Reading

- Take a look at Tim Mattson's OpenMP tutorials on youtube:
  - http://tinyurl.com/leajtxn
  - Tim is a Senior Research Scientist at Intel. You can find out more at his personal webpage: http://timmattson.com/
- http://openmp.org/wp/resources/#Tutorials
- https://mitpress.mit.edu/books/using-openmp

Practical

- Download the tarball and notes from the course website.
- There are 5 examples to work through:
  - Hello World
  - False Sharing
  - Scheduling
  - Matrix Multiply & Heat Equation.
  - Sections (for writing task parallel code)

Summary

An Introduction to OpenMP:

- Threads vs. processes and shared memory.
- Key features, e.g. #pragma compiler directives.
- Key concepts, e.g. worksharing for loops.
- Practical: Go and try the example programs on BlueCrystal.

Appendices

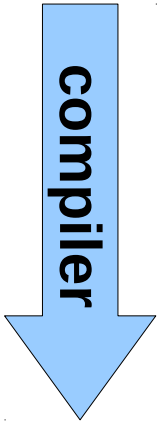
OpenMP 'under the hood'

- i.e. how it uses pthreads

“Under the Hood”

Taken from a blog by Michael Klemm at Intel: http://tinyurl.com/33smp6z

```
#include <stdio.h>
int main(int argc, char** argv) {
#pragma omp parallel num_threads(NUM_THREADS)
    printf("Hello World\n");
    return 0;
}
```



thunk

```
void main_omp_func_0() {
    printf("Hello World\n");
}
int main(int argc, char **argv) {
    _omp_start_parallel_region(main_omp_func_0);
    main_omp_func_0();
    _omp_end_parallel_region();
    return 0;
}
```

cf our early pthreads example

“Under the Hood”: Data Scoping

```
#include <stdio.h>
int main(int argc, char** argv) {
    int sh1 = 42;
    int sh2 = 43;
    int pr  = 44;
    int fp  = 45;
    int rd  = 0;
#pragma omp parallel shared(sh1,sh2) private(pr)
            firstprivate(fp) reduction(+:rd)
    {
        printf(“sh1=%d sh2=%d pr=%d fp=%d\n”,sh1,sh2,pr,fp);
        rd = 1;
    }
```

```
void main_omp_func_0(void *sh, int fp) {
    int pr;
    int rd = 0;
    prinft(“sh1=%d sh2=%d pr=%d fp=%d\n”,
           ((int*) sh)[0], ((int*) sh)[1], pr, fp);
    rd = 1;
    omp_reduce_add_int(address_of(sh[2]), rd);
}
```