OpenMP

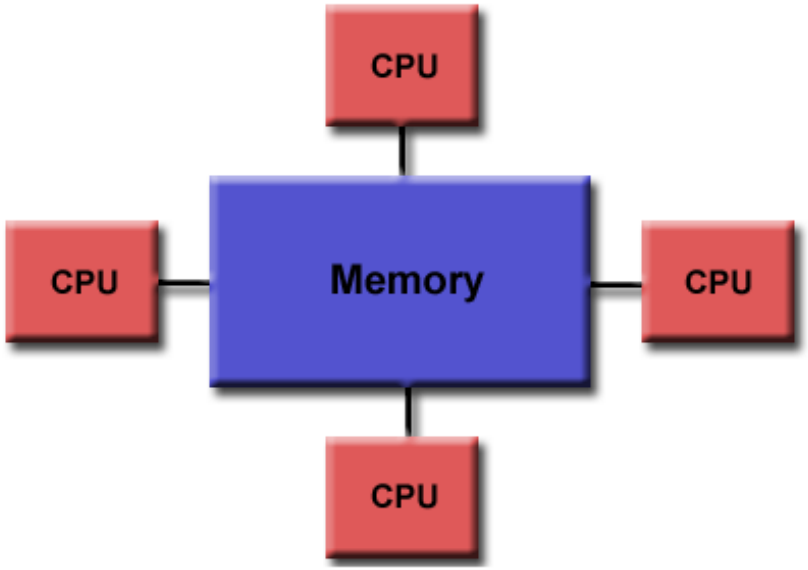
What is OpenMP?

- An Application Program Interface (API)

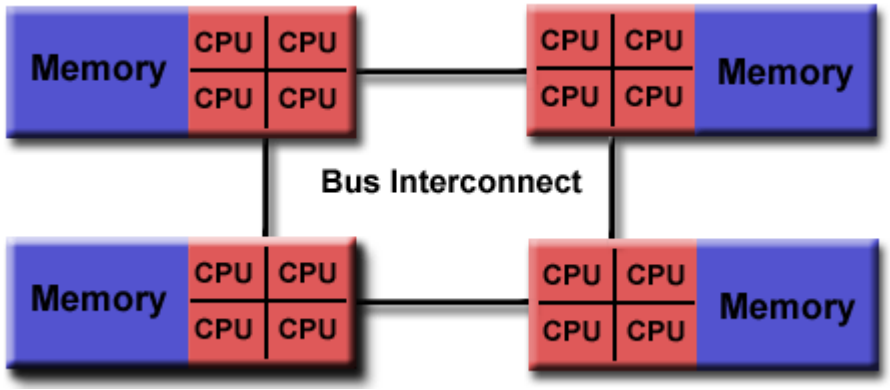
  used to explicitly direct multi-threaded, shared memory parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- An abbreviation for: Open Multi-Processing

Shared Memory Model

- Uniform memory access



- Non-uniform memory access



Parallelism

- Thread Based Parallelism:
  - OpenMP programs accomplish parallelism exclusively through the use of threads.
  - A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
  - Threads exist within the resources of a single process. Without the process, they cease to exist.
  - Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Parallelism

- Explicit Parallelism:
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives....
  - Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

OpenMP

- OpenMP has become an important standard for parallel programming
  - Makes the simple cases easy to program
  - Avoids thread-level programming for simple parallel code patterns
- We are looking at OpenMP because
  - It’s a nice example of a small domain-specific language
  - Shows how otherwise difficult problems can be solved by sharply reducing the space of problems

OpenMP

- Language extension for C/C++
- Uses #pragma feature
  - Pre-processor directive
  - Ignored if the compiler doesn't understand
- Using OpenMP

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

```
gcc –fopenmp program.c
```

- OpenMP support is now in gcc and clang

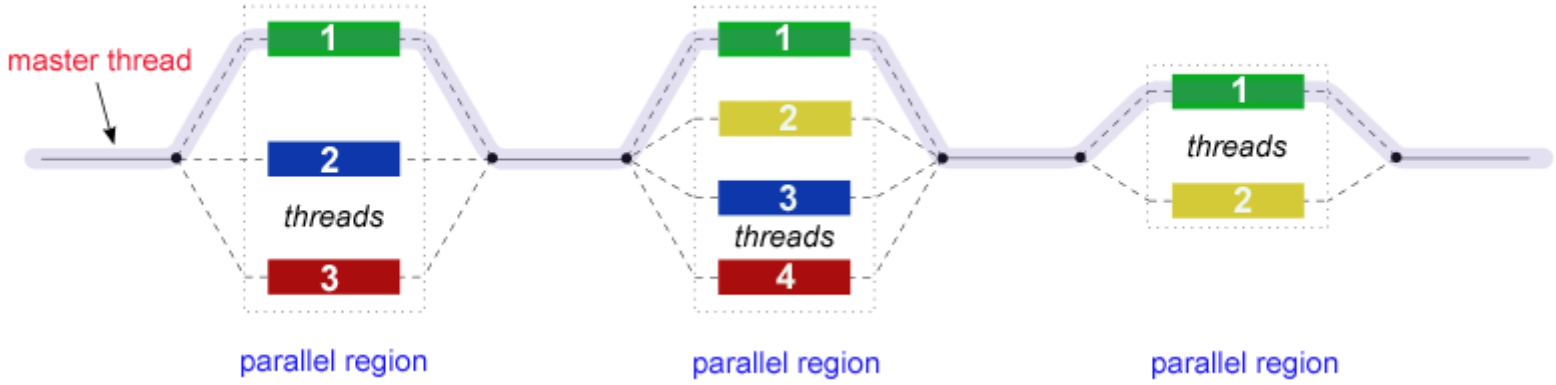
OpenMP

- OpenMP has grown over the years
  - It was originally designed for expressing thread parallelism on counted loops
    - Aimed at matrix computations
  - OpenMP 3 added “tasks”
    - Exploiting thread parallelism of
      - Non counted while loops
      - Recursive divide and conquer parallelism
  - OpenMP 4 adds SIMD and “accelerators”
    - Vector SIMD loops
    - Offload computation to GPUs

OpenMP

- We will look first at OpenMP thread programming
- Then we will add OpenMP 4
  - vector SIMD
  - And *possibly* a little GPU

Thread model



Threading Model

- OpenMP is all about threads
  - Or at least the core of OpenMP up to version 3
- There are several threads
  - Usually corresponding to number of available processors
  - Number of threads is set by the system the program is running on, not the programmer
  - Your program should work with any number of threads
- There is one master thread
  - Does most of the sequential work of the program
  - Other threads are activated for parallel sections

Threading Model

```
int x = 5;
#pragma omp parallel
 {
    x++;
 }
```

- The same thing is done by all threads
- All data is shared between all threads
- Value of x at end of loop depends on…
  - Number of threads
  - Which order they execute in
- This code is non-deterministic and will produce different results on different runs

Threading Model

- We rarely want all the threads to do exactly the same thing
- Usually want to divide up work between threads
- Three basic constructs for dividing work
  - **Parallel for**
  - **Parallel sections**
  - **Parallel task**

example code

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Hello!\n");
}
```

Parallel For

- Divides the iterations of a for loop between the threads

```
#pragma omp parallel for
    for (i = 0; i < n; i++ ) {
        a[i] = b[i] * c[i];
    }
```

- All variables shared
- Except loop control variable

Conditions for parallel for

- Several restrictions on for loops that can be threaded
- The loop variable must be of type integer.
- The loop condition must be of the form:

  i <, <=, > or >= loop_invariant_integer
- A loop invariant integer is an integer expression whose value doesn't change throughout the running of the loop
- The third part of the for loop must be either an integer addition or an integer subtraction of the loop variable by a loop invariant value
- If the comparison operator is < or <= the loop variable should be added to on every iteration, and the opposite for > and >=
- The loop must be a single entry and single exit loop, with no jumps from the inside out or from the outside in.

These restrictions seem quite arbitrary, but are actually very important practically for loop parallelisation.

Parallel for

- The iterations of the for loop are divided among the threads
- Implicit barrier at the end of the for loop
  - All threads must wait until all iterations of the for loop have completed

Parallel sections

- Parallel for divides the work of a for loop among threads
  - All threads do the same for loop, but different iterations
- Parallel sections allow different things to be done by different threads
  - Allow unrelated but independent tasks to be done in parallel.

Parallel sections

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        min = find_min(a);
    }
    #pragma omp section
    {
        max = find_max(a);
    }
}
```

Parallel sections

- Parallel sections can be used to express independent tasks that are difficult to express with parallel for
- Number of parallel sections is fixed in the code
  - Although the number of threads depends on the machine the program is running on

Parallel task

- Need constructs to deal with
  - Loops where number of iterations is not known
  - Recursive algorithms
- Tasks are parallel jobs that can be done in any order
  - They are added to a pool of tasks and executed when the system is ready

Parallel task

```
#pragma omp parallel
{
   #pragma omp single  // just one thread does this bit
   {
      #pragma omp task
      {
         printf(“Hello “);
      }
      #pragma omp task
      {
         printf(“world ”);
      }
   }
}
```

Parallel task

- Creates a pool of work to be done
- Pool is initially empty
- A task is added to the pool each time we enter a “task” pragma
- The threads remove work from the queue and execute the tasks
- The queue is disbanded when…
  - All enqueued work is complete
  - And…
    - End of parallel region or
    - Explicit #pragma omp taskwait

Parallel task

- Tasks are very flexible
  - Can be used for all sorts of problems that don't fit well into parallel for and parallel sections
  - Don't need to know how many tasks there will be at the time we enter the loop
- But there is an overhead of managing the pool
- Order of execution not guaranteed
  - Tasks are taken from pool in arbitrary order whenever a thread is free
- But it is possible for one task to create another
  - Allows a partial ordering of tasks
  - If task A creates task B, then we are guaranteed that task B starts after task A

Task example: Linked List

```
p = list_head;
#pragma omp parallel
{
   #pragma omp single
   {
   while ( p != NULL) {
      #pragma omp task firstprivate(p)
      {
         do_some_work(p);
      }
      p = p->next;
      }
   }
}
```

Mixing constructs

```
#pragma omp parallel {
    /* all threads do the same thing here */
    #pragma omp for
    for ( i = 0; i < n; i++ ) {
        /*loop iterations divided between threads*/
    }
    /* there is an implicit barrier here that makes all threads wait until all are finished */
    #pragma omp sections
    {
        #pragma omp section
        {
            /* executes in parallel with code from other section */
        }
        #pragma omp section
        {
            /* executes in parallel with code from other section */
        }
    }
    /* there is an implicit barrier here that makes all threads wait until all are finished */
    /* all threads do the same thing again */
}
```

Scope of data

- By default, all data is shared
- This is okay if the data is not updated
- A really big problem if multiple threads update the same data
- Two solutions
  - Provide mutual exclusion for shared data
  - Create private copies of data

Mutual exclusion

- Mutual exclusion means that only one thread can access something at a time
- E.g. x++;
  - If this is done by multiple threads there will be a race condition between different threads reading and writing x
  - Need to ensure that reading and writing of x cannot be interupted by other threads
- OpenMP provides two mechanisms for achieving this…
  - Atomic updates
  - Critical sections

Atomic updates

- An atomic update can update a variable in a single, unbreakable step

```
#pragma omp parallel
{
   #pragma omp atomic
   x++;
}
```

- In this code we are guaranteed that x will be increased by exactly the number of threads

Atomic updates

- Only certain operators can be used in atomic updates:
- x++, ++x, x--, --x
- x op= expr;
  - Where op is one of:
  - + * - / & ^ | << >>
- Otherwise the update cannot be atomic
  - Note that OpenMP is incredibly vague about what is an acceptable expression on the right-hand side (rhs) of the assignment
  - I only ever use constants for the rhs expression

Critical section

- A section of code that only one thread can be in at a time
- Although all threads execute same code, this bit of code can be executed by only one thread at a time

```
#pragma omp parallel
{
    #pragma omp critical
    {
        x++;
    }
}
```

- In this code we are guaranteed that x will be increased by exactly the number of threads

Named critical sections

- By default all critical sections clash with all others
  - In other words, it's not just this bit of code that can only have one thread executing it
  - There can only be one thread in any critical section in the program
- Can override this by giving different critical sections different names

```
#pragma omp parallel
{
    #pragma omp critical  (update_x)
    {
        x++;
    }
}
```

- There can be only one thread in the critical section called "update_x", but other threads can be in other critical sections

Critical sections

- Critical sections are much more flexible than atomic updates
- Everything you can do with atomic updates can be done with a critical section
- But atomic updates are…
  - Faster than critical sections
    - Or at least they might be faster, depending on how they are implemented by the compiler
  - Less error prone (in complicated situations)

Private variables

- By default all variables are shared
- But private variables can also be created
- Some variables are private by default
  - Variables declared within the parallel block
  - Local variables of function called from within the parallel block
  - The loop control variable in parallel for

Private variables

```
/* compute sum of array of ints */ --> live code!
int sum = 0;
#pragma omp parallel for
{
   for ( i = 0; i < n; i++ ) {
     #pragma omp critical
     sum += a[i];
   }
}
```

- Code works but is inefficient, because of contention between threads caused by the atomic update

Private variables

```
/* compute sum of array of ints */ --> live code!
int sum = 0;
#pragma omp parallel
{
    int local_sum = 0;
    #pragma omp for
    for ( i = 0; i < n; i++ ) {
        local_sum += a[i];
    }
    #pragma omp critical
    sum += local_sum;
}
```

- Does the same thing, but may be more efficient, because there is contention only in computing the final global sum

Private variables

```
/* compute sum of array of ints */
int sum = 0;
int local_sum;
#pragma omp parallel private(local_sum)
{
    local_sum = 0;
    #pragma omp for
    for ( i = 0; i < n; i++ ) {
        local_sum += a[i];
    }
    #pragma omp critical
    sum += local_sum;
}
```

- This time, each thread still has its own copy of local_sum, but another variable of the same name also exists outside the parallel region

Private variables

- Strange semantics with private variables
  - Declaring variable private creates new variable that is local to each thread
  - No connection between this local variable and the other variable outside
    - It's almost like creating a new local variable within a C/C++ block (that is a piece of code between a pair of matching curly brackets)
  - Local variable is given default value
    - Usually zero
  - Value of "outside" version of the private variable is undefined after parallel region (!)
    - So it's not exactly like declaring a new local variable within a C/C++ block

firstprivate

- We often want a private variable that starts with the value of the same variable outside the parallel region
- The firstprivate construct allows us to do this

```
/* compute sum of array of ints */
int sum = 0;
int local_sum = 0;
#pragma omp parallel firstprivate(local_sum)
    {
    /* local_sum in here is initialised with local sum value from outside */
    #pragma omp for
    for ( i = 0; i < n; i++ ) {
            local_sum += a[i];
    }
    #pragma omp critical
    sum += local_sum;
}
```

Private variables and tasks

```
p = list_head;
#pragma omp parallel
{
    #pragma omp single
    {
    while ( p != NULL) {
       #pragma omp task // this code is broken!!!
       {
          do_some_work(p);
       }
          p = p->next;
       }
    }
}
```

- Problem that the value is p may change between the time that the task is created and the time that the task starts to execute
- The task needs its own private copy of p

Private variables and tasks

```
p = list_head;
#pragma omp parallel
{
    #pragma omp single
    {
    while ( p != NULL) {
        #pragma omp task firstprivate(p) // this code works...
        {  //  ... I think
            do_some_work(p);
        }
        p = p->next;
    }
    }
}
```

- Firstprivate declaration creates a new copy of p within the task
- Has value of original p at the point where the task was created

Shared variables

- By default all variables in a parallel region are shared
- Can also explicitly declare them to be shared
- Can opt to force all variables to be declared shared or non-shared
- Use “default(none)” declaration to specify this

Shared variables

```
/* example of requiring all variables be
   declared shared or non-shared */
#pragma omp parallel default(none) \
   shared(n,x,y) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)
        x[i] += y[i];
}
```

Reductions

- A reduction involves combining a whole bunch of values into a single value
  - E.g. summing a sequence of numbers
- Reductions are very common operation
- Reductions are inherently parallel
- With enough parallel resources you can do a reduction in O(log n) time
  - Using a “reduction tree”

Reductions

```
/* compute sum of array of ints */
int sum = 0;
#pragma omp parallel for reduction(+:sum)
{
    for ( i = 0; i < n; i++ ){
          sum += a[i];
    }
}
```

- A private copy of the reduction variable is created for each thread
- OpenMP automatically combines the local copies together to create a final value at the end of the parallel section

Reductions

- Reductions can be done with several different operators
- + - * & | ^ && ||
- Using a reduction is simpler than dividing work between the threads and combining the result yourself
- Using a reduction is potentially more efficient

Scheduling parallel for loops

- Usually with parallel for loops, the amount of work in each iteration is roughly the same
  - Therefore iterations of the loop are divided approximately evenly between threads
- Sometimes the work in each iteration can vary significantly
  - Some iterations take much more time
  - => Some threads take much more time
  - Remaining threads are idle
- This is known as poor “load balancing”

Scheduling parallel for loops

- OpenMP provides three “scheduling” options
- static
  - Iterations are divided evenly between the threads (this is the default)
- dynamic
  - Iterations are put onto a work queue, and threads take iterations from the queue whenever they become idle. You can specify a “chunk size”, so that iterations are taken from the queue in chunks by the threads, rather than one at a time
- guided
  - Similar to dynamic, but initially the chunk size is large, and as the loop progresses the chunk size becomes smaller
  - allows finer grain load balancing toward the end

Scheduling parallel for loops

- E.g. testing numbers for primality
- Cost of testing can vary dramatically depending on which number we are testing
- Use dynamic scheduling, with chunks of 100 iterations taken from work queue at a time by any thread

```
#pragma omp parallel for schedule(dynamic, 100)
{
   for ( i = 2; i < n; i++ ) {
        is_prime[i] = test_primality(i);
}
```

Conditional parallelism

- OpenMP directives can be made conditional on runtime conditions

```
#define DEBUGGING 1
#pragma omp parallel for if (!DEBUGGING)
   for ( i = 0; i < n; i++ )
        a[i] = b[i] * c[i];
```

- This allows you to turn off the parallelism in the program for debugging
- Once you are sure the sequential version works, you can then try to fix the parallel version
- You can also use more complex conditions that are evaluated at runtime

Conditional parallelism

- There is a significant cost in executing OpenMP parallel constructs
- Conditional parallelism can be used to avoid this cost where the amount of work is small

```
#pragma omp parallel for if ( n > 128 )
for ( i = 0; i < n; i++ )
   a[i] = b[i] + c[i];
```

- Loop is executed in parallel if n > 128
- Otherwise the loop is executed sequentially

Cost of OpenMP constructs

- The following number have been measured on an old 4-way Intel 3.0 GHz machine
- Source: “Multi-core programming: increasing performance through software threading” – Akhter & Roberts, Intel Press, 2006.
- Intel compiler & runtime library
- Cost usually 0.5 -2.5 microseconds
- Assuming clock speed of around 3 GHz
  - One clock cycle is ~0.3 nanoseconds
  - More than factor 1000 difference in time

Cost of OpenMP constructs

Construct	microseconds
parallel	1.5
barrier	1.0
schedule (static)	1.0
schedule (guided)	6.0
schedule (dynamic)	50
ordered	0.5
single	1.0
reduction	2.5
atomic	0.5
critical	0.5
lock/unlock	0.5

Cost of OpenMP constructs

- Some of these costs can be reduced by eliminating unnecessary constructs
- In the following code we enter a parallel section twice

```
#pragma omp parallel for
for ( i = 0; i < n; i++ )
    a[i] = b[i] + c[i];
    #pragma omp parallel for
    for ( j = 0; j < m; j++ )
        x[j] = b[j] * c[j];
```

- Parallel threads must be woken up at start of each parallel region, and put to sleep at end of each.

Cost of OpenMP constructs

- Parallel overhead can be reduced slightly by having only one parallel region

```
#pragma omp parallel
{
        #pragma omp for
        for ( i = 0; i < n; i++ )
                a[i] = b[i] + c[i];
        #pragma omp for
        for ( j = 0; j < m; j++ )
                x[j] = b[j] * c[j];
}
```

- Parallel threads now have to be woken up and put to sleep once for this code

Cost of OpenMP constructs

- There is also an implicit barrier at the end of each for
- All threads must wait for the last thread to finish
- But in this case, there is no dependency between first and second loop
- The “nowait” clause eliminates this barries

```
#pragma omp parallel
{
   #pragma omp for nowait
   for ( i = 0; i < n; i++ )
      a[i] = b[i] + c[i];
   #pragma omp for
   for ( i = 0; i < m; i++ )
      x[j] = b[j] * c[j];
}
```

- By removing the implicit barrier, the code may be slightly faster

Caching and sharing

- Shared variables are shared among all threads
- Copies of these variables are likely to be stored in the level 1 cache of each processor core
- If you write to the same variable from different threads then the contents of the different L1 caches needs to be synchronized in some way
  - Cache coherency logic detects that the cache line is invalid because variable has been updated elsewhere
  - Need to load the cache line again
  - This is expensive
- Sometimes called thrashing or “ping ponging” cache lines
- Should avoid modifying shared variables a lot

Caching and sharing

- Multiple threads do not have to modify the same variable to cause these sorts of performance problems with invalidation of shared cache lines
- You just need to share the same cache line, not necessarily the same data
- So you need to be careful about multiple threads writing to variables that are nearby in memory.
  - E.g. different threads writing to nearby elements of arrays in memory

Vector SIMD

- OpenMP is a very popular language for writing parallel code and from time to time there is a new version with new features
- OpenMP 4.0 added new constructs to direct the compiler to vectorize a loop

```
#pragma omp simd
for (i = 0; i < N; i++ ) {
    a[i] = a[i] + s * b[i];
}
```

Vector SIMD

- The semantics of OpenMP SIMD are a little bit odd
  - The SIMD directive tells the compiler to vectorize the code
  - It is quite different to the vectorization hints that many compilers provide
  - E.g. #pragma ivdep // directive accepted by Intel compiler
    - Tells the compiler that there are no loop carried dependences that it needs to worry about that would make vectorization illegal
    - But it does not guarantee that the loop will actually be vectorized

- OpenMP #pragma omp SIMD is different
  - It requires the compiler to vectorize the loop
  - The vectorization may be unsafe and wrong
  - But the compiler will make every attempt to vectorize the loop regardless of whether it is a good idea

Vector SIMD

- Strangely, OpenMP is quite vague about exactly what it means to vectorize a loop
  - With OpenMP threads, the semantics are clear
  - the iterations of the for loop are divided between the threads
- But OpenMP is unclear about whether the loop iterations should be allocated to different lanes
  - Although that seems to be the idea of OpenMP SIMD directives

Vector SIMD

- Sometimes a loop may have an inter-iteration data dependency
- E.g.

```
for ( i = 1; i < n; i++ ) {
    a[i] = sqrt(a[i-1]) + b[i];
}
```

- This loop cannot easily be vectorized because of the dependence between the current a[i] and the a[i-1] from the previous iteration

Vector SIMD

- But sometimes you can vectorized despite an inter-iteration dependency
- E.g.

```
for ( i = 10; i < n; i++ ) {
  a[i] = sqrt(a[i-10]) + b[i];
}
```

- This loop can be vectorized because the dependence distance is 10 iterations
- But it is not safe to vectorize more than 10 iterations at a time

Vector SIMD

- In OpenMP we write this:

```
#pragma omp simd safelen(10)
for ( i = 10; i < n; i++ ) {
  a[i] = sqrt(a[i-10]) + b[i];
}
```

- The "*safelen*" keyword tells the compiler that it should not generate vector code that operates on more than the specified number of iterations

Vector SIMD

- It’s also common for loops to contain function calls

```
#pragma omp simd safelen(10)
for ( i = 10; i < n; i++ ) {
  a[i] = my_func(a[i-10]) + b[i];
}
```

- It’s not easy for the compiler to vectorize this code because the *function my_func* probably operates on just one value at a time

Vector SIMD

- OpenMP allows us to declare that the compiler should create a vector version of the function

```
#pragma omp declare simd
float my_func(float) {
  // body of function
}
```

- The compiler will generate a version of the code that takes an entire vector of floats as a parameter, and computes a whole vector of results

Multithreaded programming