



Multithreading in C/C++

ICS432
**Concurrent and High-Performance
Programming**

Henri Casanova (henric@hawaii.edu)

Pthreads

- Almost every modern language gives you mechanisms for multi-threading
- C/C++ is no exception
- Low-level mechanisms are implemented in the Pthreads library (POSIX standard)
 - It is a bit cumbersome, but you can do pretty much everything with it
- We are not going to do any Pthreads in this course, but let's just look at a small example to give you a sense of what it looks like....

Pthread Hello World (Thread code)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

// Data structure to pass arguments to threads
struct argstruct {
    int *array;
    int size;
};

// Thread's "main" function
void *do_work(void *arg) {
    int *A = ((struct argstruct *)arg)->array;
    int n = ((struct argstruct *)arg)->size;
    long sum = 0;
    for (int i=0; i < n; i++) sum += A[i];
    return (void*)sum; // hideous
}
```

In Pthreads, we can only pass a `void*` to a thread. So one typically creates a data structure with the arguments we want to pass to a thread, and then pass a pointer to it!

Pthread Hello World (Thread code)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

// Data structure to pass arguments to threads
struct argstruct {
    int *array;
    int size;
};

// Thread's "main" function
void *do_work(void *arg) {
    int *A = ((struct argstruct *)arg)->array;
    int n = ((struct argstruct *)arg)->size;
    long sum = 0;
    for (int i=0; i < n; i++) sum += A[i];
    return (void*)sum; // hideous
}
```

Therefore, the first thing thread does is "unpack" its arguments

Pthread Hello World (Thread code)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

// Data structure to pass arguments to threads
struct argstruct {
    int *array;
    int size;
};

// Thread's "main" function
void *do_work(void *arg) {
    int *A = ((struct argstruct *)arg)->array;
    int n = ((struct argstruct *)arg)->size;
    long sum = 0;
    for (int i=0; i < n; i++) sum += A[i];
    return (void*)sum; // hideous
}
```

A Pthread must return a void *. In this case we (horribly) cast a long to a void * to return an integer from the thread

Pthread Hello World (Full Program)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

// Data structure to pass arguments to threads
struct argstruct {
    int *array;
    int size;
};

// Thread's "main" function
void *do_work(void *arg) {
    int *A = ((struct argstruct *) (arg))->array;
    int n = ((struct argstruct *) (arg))->size;
    long sum = 0;
    for (int i=0; i < n; i++) sum += A[i];
    return (void*)sum; // hideous
}
```

```
int main(int argc, char **argv) {
    // Allocate and initialize an array
    int *A = (int *)malloc( 100 * sizeof(int) );
    for (int i = 0; i < 100 ; i++) A[i] = rand() % 100;
    // Set up thread argument structure
    struct argstruct *arg = (struct argstruct *)
        calloc(1, sizeof(struct argstruct));
    arg->array = A; arg->size = 100;
    // Launch a thread
    pthread_t thread;
    pthread_create(&thread, NULL, do_work, arg);
    int return_value;
    // Wait for the thread
    pthread_join(thread, (void*)&return_value);
    printf("sum = %d\n", return_value);
}
```

Beyond Pthreads

- Pthreads are cumbersome, especially for novice programmers
 - Creating tons of data structures and casting gets old really quickly
- There have been many attempts at making it easier so that the developer does not have to write Pthread code by hand
- There are libraries, such as Boost (for C++), that provide easier APIs than Pthreads, and use Pthreads underneath
- There are also language extensions (that require a compliant compiler that generates Pthread code for you)
- A common C language extension is CILK
 - Research project at MIT that went commercial
 - Acquired/supported by Intel (CILKPlus in forks of GCC/Clang) until it became obsolete ~2018
- Let's look at one CILK hello world program just so that you have a sense of what it looked like...

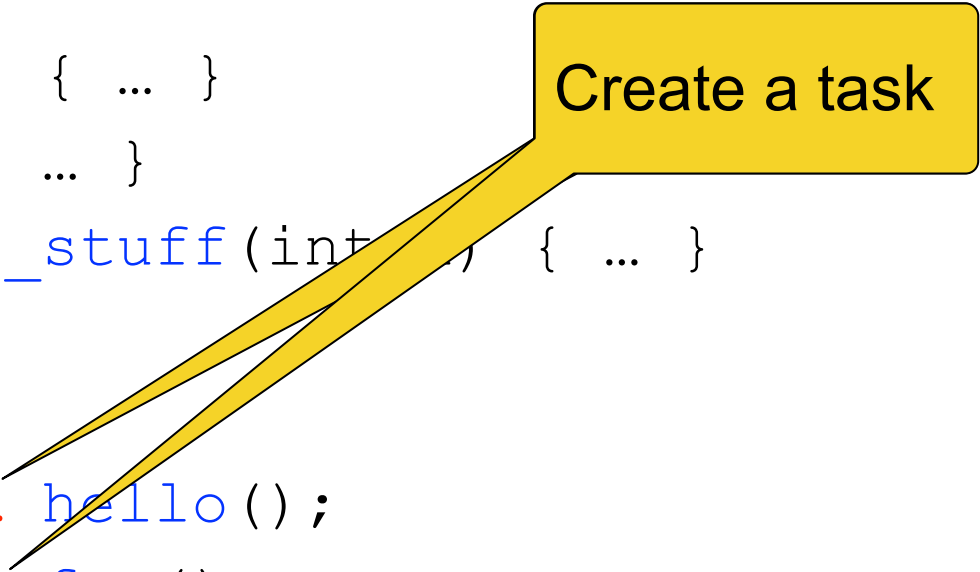
CILK in 10 seconds

```
void hello() { ... }
void foo() { ... }
void compute_stuff(int x) { ... }

int main() {
    cilk_spawn hello();
    cilk_spawn foo();
    cilk_sync;
    cilk_for (int i = 0; i <= 10000; i++) {
        compute_stuff(i);
    }
}
```


CILK in 10 seconds


```
void hello() { ... }  
void foo() { ... }  
void compute_stuff(int i) { ... }  
  
int main() {  
    cilk_spawn hello();  
    cilk_spawn foo();  
    cilk_sync;  
    cilk_for (int i = 0; i <= 10000; i++) {  
        compute_stuff(i);  
    }  
}
```



Create a task

CILK in 10 seconds


```
void hello() { ... }  
void foo() { ... }  
void compute_stuff(int x) { ... }  
  
int main() {  
    cilk_spawn hello();  
    cilk_spawn foo();  
    cilk_sync;  
    cilk_for (int i = 0; i <= 10000; i++) {  
        compute_stuff(i);  
    }  
}
```



Wait for (join with) all tasks

CILK in 10 seconds

```
void hello() { ... }  
void foo() { ... }  
void compute_stuff(int x) { ... }  
  
int main() {  
    cilk_spawn hello();  
    cilk_spawn foo();  
    cilk_sync;  
    cilk_for (int i = 0; i <= 10000; i++) {  
        compute_stuff(i);  
    }  
}
```



Parallel for loop!

C++ and `std::thread`

- Since C++ 11, threads and all synchronization operations are available in the standard library!
- It basically does everything we have learned how to do in Java
- It looks much, much nicer than using PThreads directly
- Again, we're not going to do this in this course, but let's just look at a small example...

C++ 11 Thread Example

```
#include <thread>
#include <iostream>
std::condition_variable cond;
std::mutex m_mutex;
```

```
int main(int argc, char **argv) {
    std::thread alarm_clock(alarm_clock_function, 2);
    std::thread sleeper(sleeper_function);
    sleeper.detach();
    alarm_clock.join();
}
```

```
void sleeper_function() {
    while (true) {
        std::unique_lock<std::mutex> mlock(m_mutex);
        cond.wait(mlock);
        std::cout << "I am awake!" << std::endl;
    }
}
```

```
void alarm_clock_function(int num_iterations) {
    for (int i=0; i < num_iterations; i++) {
        int duration = 1 + rand() % 4;
        std::this_thread::sleep_for(std::chrono::seconds(duration));
        std::cout << "Alarm clock is going off" << std::endl;
        cond.notify_one();
    }
}
```

What now?

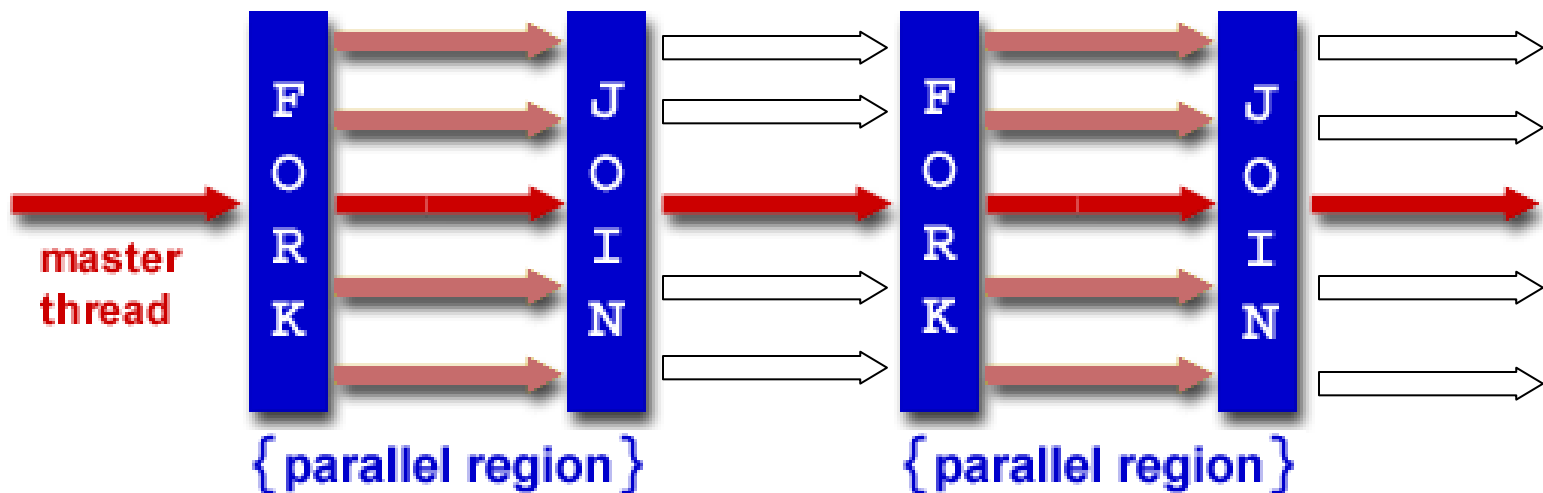
- We can apply what we've learned in Java to many different programming languages directly
- I could now go into “this is how it's done in C#”, etc.
- But they are all the same, with syntactic differences
 - Besides JavaScript, which is weaker
 - Besides Rust, which is trying to do things differently
 - Besides Python, which is weird
- We'll about about those “special” cases later in the semester
- But for now, let's look at something quite different for C/C++, and quite popular: OpenMP

OpenMP

- A well-established and popular way to do multi-threading in C/C++ is OpenMP
 - A library with some simple functions
 - The definition of a few C pragmas
 - pragmas are a way to “extend” C
 - provide an easy way to give hints/information to a compiler
 - This “compiler directive” way of programming goes way back...
 - A compiler that supports OpenMP
 - e.g., gcc since version 4.2 circa 2007 (compile with the -fopenmp flag)
- It provides a somewhat rigid but still extremely useful programming model
- The goal is: make it easy for developers

Fork-Join Programming Model

- Program begins with a **Master thread**
- **Fork:** **Teams of threads** created at times during execution
- **Join:** Threads in the team synchronize (**barrier**) and only the master thread continues execution



OpenMP and #pragma

- One needs to specify blocks of code that are executed in parallel
- A *parallel region*:

#pragma omp parallel [clauses]

- Defines a region of the code that will be executed in parallel
- The “clauses” specify many useful things, and we’ll see a few of them
- All threads in the region execute the same code by default

OpenMP Example #1

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("Start\n");

    /* Start Parallel Code */
    #pragma omp parallel
    { // note the `{`
        printf("Hello World\n");
    } // note the `}'

    /* Resume Serial Code */
    printf("Done\n");
}
```

- OpenMP is finicky about braces (new lines needed!)
- What we now need is to specify how many threads should run in the parallel region...

OpenMP Example #1

```
#include <stdio.h>
#include <omp.h>
int main() {
    printf("Start\n");
    omp_set_num_threads(4);

    /* Start Parallel Code */
#pragma omp parallel
{
    printf("Hello World\n");
}

    /* Resume Serial Code */
    printf("Done\n");
}
```

- Let's now have each thread print their identity...

OpenMP Example #1

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("Start\n");
    omp_set_num_threads(4);

#pragma omp parallel
{
    printf("Hello World (%d/%d)\n",
          omp_get_thread_num(),
          omp_get_num_threads());
}
    printf("Done\n");
}
```

- Let's compile and run this program on my laptop...
- Note that my laptop is a Mac, so I need to run gcc (not clang, which sadly doesn't support OpenMP out of the box, and "looks like" gcc!)

Data: Private or Shared?

- When using OpenMP threads, we must decide which variables they share and which variables they don't share
- **Shared** variable: all threads “see” the same variable in RAM (the master's)
- **Private** variable: each thread has its own copy of the variable
 - Done for you by the compiler!
- Let's see this on a tiny program where each thread computes a partial sum of an array...

OpenMP Example #2

```
#define N 2000
int main(){
    omp_set_num_threads(4);
    int A[4*N];
    for (int i=0; i < 4*N; i++) A[i] = i;

    int sum, i, start, end;
#pragma omp parallel shared(A) private(sum, i, start, end)
    {
        sum = 0;
        start = N * omp_get_thread_num();
        end = N * (1 + omp_get_thread_num());
        for (i=start; i < end; i++) sum += A[i];
        printf("Thread #%d: sum=%d\n", omp_get_thread_num(),
            sum);
    }
}
```

OpenMP Example #2

```
#define N 2000
int main() {
    omp_set_num_threads(4);
    int A[4*N];
    for (int i=0; i < 4*N; i++) A[i] = i;

    int sum, i, start, end;
    #pragma omp parallel shared(A) private(sum, i, start, end)
    {
        sum = 0;
        start = N * omp_get_thread_num();
        end = N * (1 + omp_get_thread_num());
        for (i=start; i < end; i++) sum += A[i];
        printf("Thread #%d: sum=%d\n",
            omp_get_thread_num(), sum);
    }
}
```

Declaring A as private would copy the array, which is unnecessary and harmful to performance

Declaring any of these as shared would cause bugs

Data: Private or Shared?

- It's very common to introduce performance or correctness bugs by mis-labelling variables in terms of shared and private
- OpenMP allows you to define the default behavior:

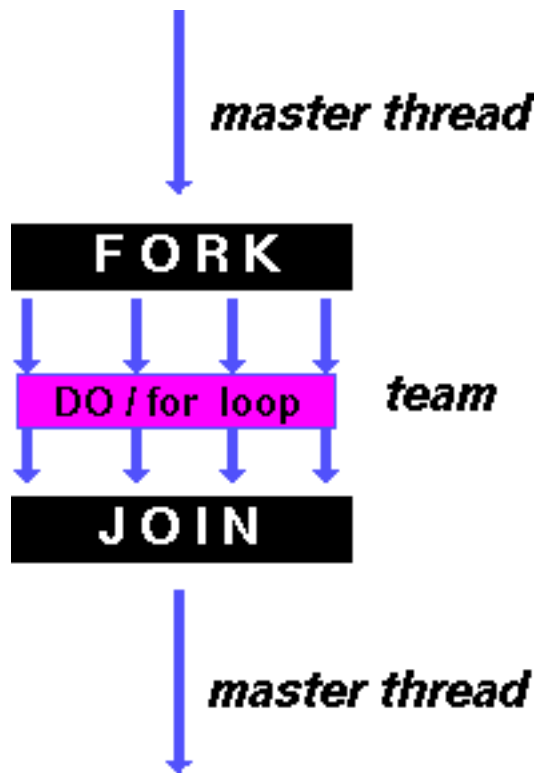
```
#pragma omp parallel shared(A) default(private)
```

- There are other useful variations
 - **firstprivate**: initialization from the master's copy
 - **lastprivate**: the master gets the last value updated by the last thread to do an update
 - etc.
- (Look at public on-line resources for all details)

Work Sharing directives

- We have seen the concept of a **parallel region** to run multiple threads
- Work Sharing directives make it possible to have threads “share work” within a parallel region:
 - For Loop
 - Sections
 - Single

For Loops



- Share iterations of the loop across threads
- To implement **Data-parallelism**
- Program correctness must NOT depend on which thread executes which iteration
 - No ordering!

OpenMP Example #3

```
#include <omp.h>
#define N 10
main () {
    int i, chunk;
    float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a, b, c) private(i)
    {
#pragma omp for
        for (i=0; i < N; i++) {
            printf("Thread %d, i=%d", omp_get_thread_num(), i);
            c[i] = a[i] + b[i];
        }
    }
}
```

OpenMP Example #3

```
#include <omp.h>
#define N 10
main () {
    int i, chunk;
    float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a, b, c) private(i)
    {
        #pragma omp for
        for (i=0; i < N; i++) {
            printf("Thread %d, i=%d", omp_get_thread_num(), i);
            c[i] = a[i] + b[i];
        }
    }
}
```

Needed for
performance

Required for
correctness

For Loop and “nowait”

- With “nowait”, threads do not synchronize at the end of the loop
 - i.e., threads may exit the “#pragma omp for” at different times

```
#pragma omp parallel shared(a, b, c) private(i) {  
    #pragma omp for nowait  
    for (i=0; i < N; i++) {  
        // do some work  
    }  
    // Threads may get here at different times  
}
```

Loop Parallelizations

- Not all loops can be safely parallelized
 - Not specific to OpenMP, but OpenMP makes loop parallelization so easy that one may forget that “just add a `#pragma`” doesn’t always work
- If there are **inter-iteration dependencies**, we can have race conditions
- Simple example:

```
for (i=1; i < N; i++) {  
    a[i] += a[i-1] * a[i-1]  
}
```

 - We have a “data dependency”: iteration i needs data produced by previous iteration $i-1$

Three Kinds of Dependencies

- Consider a for loop: `for (i=0; i < N; i++)`
- Data dependency:
 - Example: `a[i] += a[i-1] * a[i-1];`
 - Iteration i needs data produced by a previous iteration
- Anti-dependency:
 - Example: `a[i] = a[i+1] * 3;`
 - Iteration i must use data before it is updated by the next iteration
- Output dependency
 - Example: `a[i] = 2*a[i+1]; a[i+2] = 3 * a[i]`
 - Different iterations write to the same addresses

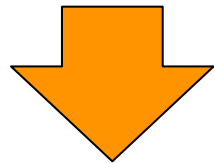
Dealing with Dependencies

- Due to dependencies, some loops are inherently sequential and simply cannot be parallelized
- Some have race conditions that we could fix with locks (we will see how OpenMP does locks in a few slides)
 - But in this case, the loop likely becomes sequential
 - In fact, slower than sequential due to locking overhead

Dealing with Dependencies

- Sometimes, one can rewrite the loop to remove dependencies
- Example:

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

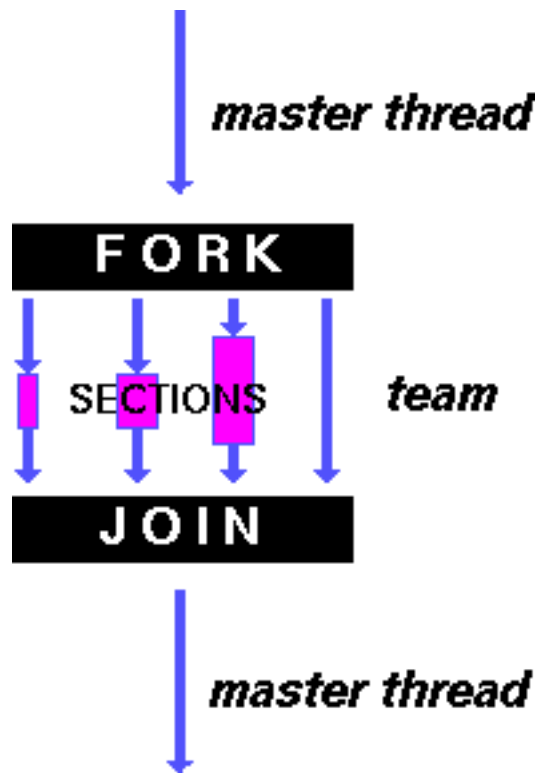


```
for (i=1; i < N; i++) {  
    a[i] = a[0] + i;  
}
```

Loops with Dependencies

- The bottom-line is that dependencies in loops make the job of the parallelizing developer/compiler difficult
- **Tons** of research has gone into this, with compilers being able to automatically parallelize some loops
- But many of the parallelization algorithms are very expensive (i.e., high complexity)
- Often, a developer will have to develop ingenuity to “expose loop parallelism”
- We saw a trivial example of this in the previous slide
- We’ll see another example at some point...

Sections



- Breaks work into **separate** sections
- Each section is executed by a thread
- To implement **Task-parallelism**
 - do different things on different data
- If more threads than sections, then some threads remain idle
- If fewer threads than sections, then some sections are serialized

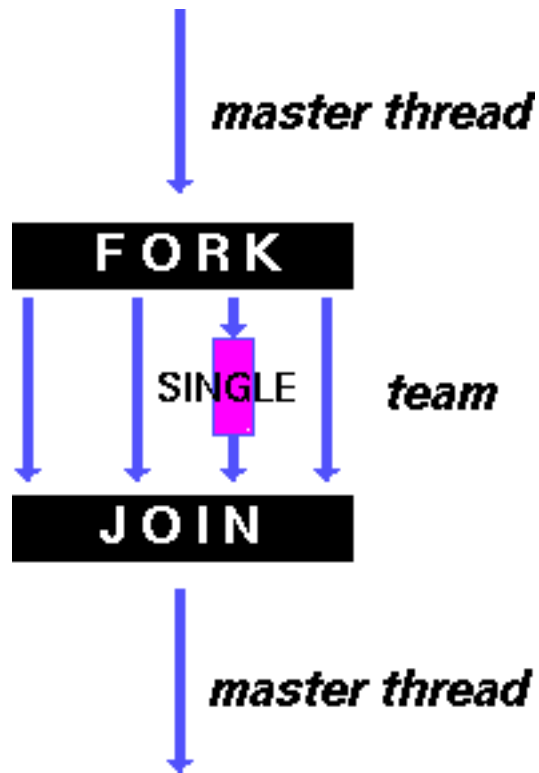
OpenMP Example #4

```
#include <omp.h>
#define N 10
main () {
    int i;
    float a[N], sum = 0.0, prod = 1.0;
    omp_set_num_threads(2);
    for (i=0; i < N; i++) a[i] = (i + 1) * 1.0;

    #pragma omp parallel shared(a) private(i)
    {
        #pragma omp sections
        {
            #pragma omp section // compute sum
            {
                for (i=0; i < N; i++)
                    sum += a[i];
            }

            #pragma omp section // compute product
            {
                for (i=0; i < N; i++)
                    prod *= a[i];
            }
        } /* end of sections */
    } /* end of parallel region */
}
```

Single



- Serializes a section of code within a parallel region
- Sometimes more convenient than terminating a parallel region and starting it later
 - Especially because variables are already shared/private, etc.
- Typically used to serialize a small section of the code that's not thread safe
 - e.g., I/O

Combined Directives

- It is cumbersome to create a parallel region and *then* create a parallel for loop, or sections, just to terminate the parallel region
- Therefore OpenMP provides a way to do both at the same time
 - `#pragma omp parallel for`
 - `#pragma omp parallel sections`
- Example:

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

Dealing with Race Conditions

- OpenMP doesn't magically fix race conditions:

```
int x = 0;
#pragma omp parallel sections shared(x)
{
    #pragma omp section
    { x++; }
    #pragma omp section
    { x--; }
}
```

- The above code has a race condition, just like our first race condition example in Java in this course

Synchronization Directives

- Directives for creating critical sections
 - **#pragma omp master**
 - Creates a region that only the master executes (therefore it's by definition a critical section)
 - **#pragma omp critical**
 - Creates a critical section
 - **#pragma omp atomic**
 - Create a “mini” critical section

- Let's see a critical section....

OpenMP Example #5

```
#include <stdio.h>
#include <omp.h>
#define N 10

int main () {
    int i;
    float a[N], sum=0.0;
    omp_set_num_threads(10);
    for (i=0; i < N; i++) a[i] = (i + 1) * 1.0;

    #pragma omp parallel shared(a, sum) private(i)
    {
        #pragma omp for
        for (i=0; i < N; i++) {
            #pragma omp critical
            {
                sum += a[i];
            }
        }
    }
    printf("sum = %f\n", sum);
}
```

This simply inserts Pthread lock/unlock calls in the code generated by the compiler

OpenMP Example #5

```
#include <stdio.h>
#include <omp.h>
#define N 10

int main () {
    int i;
    float a[N], sum=0.0;
    omp_set_num_threads(10);
    for (i=0; i < N; i++) a[i] = (i + 1) * 1.0;

    #pragma omp parallel shared(a, sum) private(i)
    {
        #pragma omp for
        for (i=0; i < N; i++) {
            #pragma omp critical
            {
                sum += a[i];
            }
        }
    }
    printf("sum = %f\n", sum);
}
```

This simply inserts Pthread lock/unlock calls in the code generated by the compiler

This is a terrible “mostly sequential” program, let’s rewrite it live using partial sums, and compare performance! (code on Web site)

OpenMP Atomic Directive

```
#pragma omp atomic
    i++;
```

- Only allowed for some expressions
 - `x = expr` (no mutual exclusion on expr evaluation)
 - `x++`
 - `++x`
 - `x--`
 - `--x`
- It's about atomic access to a memory location
- Some OpenMP implementations will just replace `atomic` by `critical` and create a basic blocks with lock/unlock around it
- But some may take advantage of **fast hardware instructions** that can do the above atomically

OpenMP Barrier

```
#pragma omp barrier
```

- All threads in the current parallel region will synchronize
 - They will all wait for each other at this instruction
- Must appear within a basic block
- Remember that all threads wait for each other at the end of a parallel region anyway
 - i.e., there is an implicit barrier there

OpenMP Loop Scheduling

- The “bread and butter” of OpenMP is loops
- Many developers use OpenMP just to parallelize a few loops here and there
 - Especially non-expert developers who want a performance boost but have no desire/way to take ICS432 and learn all kinds of crazy stuff
 - Think scientists who develop code, rather than people with a CS degree
- Let’s do a simple loop with 4 threads, where each thread merely prints what it’s doing...

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
#pragma omp parallel private(i)
    {
#pragma omp for
        for (i = 0; i < 10; i++) {
            printf("Thread #%d: iteration %d\n",
                omp_get_thread_num(), i);
        }
    }
}
```

Let's look at the output...

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
#pragma omp parallel private(i)
    {
#pragma omp for
        for (i = 0; i < 10; i++) {
            printf("Thread #&#d: iterati
                omp_get_thread_num()
            }
        }
    }
}
```

```
Thread #0: iteration 0
Thread #1: iteration 3
Thread #3: iteration 8
Thread #3: iteration 9
Thread #2: iteration 6
Thread #2: iteration 7
Thread #1: iteration 4
Thread #1: iteration 5
Thread #0: iteration 1
Thread #0: iteration 2
```

Static Loop Scheduling

- The behavior of the previous program is called static scheduling
- Ahead of time, we simply partition the iteration space as evenly as possible across threads
- This is the default behavior in OpenMP, but we could have declared it as:

```
#pragma omp for schedule(static)
```
- So, of course, this means there are other options...

Dynamic Loop Scheduling

- Another option is to do dynamic loop scheduling
- Conceptually: threads “grab” the next iteration to do whenever idle
- This can be done very easily (under the cover) with a lock:

```
while (true) {
    int the_iteration_i_should_do_next; // private to thread
    lock();
    if (next_it < loop_bound) {
        the_iteration_i_should_do_next = next_it;
        next_it++;
        unlock();
    } else {
        unlock();
        break;
    }
    do_iteration(the_iteration_i_should_do_next)
}
```

Dynamic Loop Scheduling

- Another option is to do dynamic loop scheduling
- Conceptually: threads “grab” the next iteration to do whenever idle
- This can be done very easily (under the cover) with a lock:

```
while (true) {  
    int the_iteration_i_should_do_next; // private to thread  
    lock();  
    if (next_it < loop_bound) {  
        the_iteration_i_should_do_next = next_it;  
        next_it++;  
        unlock();  
    } else {  
        unlock();  
        break;  
    }  
    do_iteration(the_iteration_i_should_do_next)  
}
```

Low-key, low-tech
producer-consumer

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
#pragma omp parallel private(i)
    {
#pragma omp for schedule(dynamic)
        for (i = 0; i < 10; i++) {
            printf("Thread #%d: iteration %d\n",
                omp_get_thread_num(), i);
        }
    }
}
```

Let's look at the output...

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
#pragma omp parallel private(i)
    {
#pragma omp for schedule(dynamic)
        for (i = 0; i < 10; i++) {
            printf("Thread #0: iteration %d\n", i);
            omp_get_thread_num();
        }
    }
}
```

```
Thread #0: iteration 0
Thread #0: iteration 4
Thread #0: iteration 5
Thread #0: iteration 6
Thread #0: iteration 7
Thread #2: iteration 3
Thread #2: iteration 9
Thread #0: iteration 8
Thread #3: iteration 1
Thread #1: iteration 2
```

Static vs. Dynamic Scheduling

- Static:
 - 👍 Low overhead (each thread just does its own loop on its own loop index range)
 - 👎 Poor load balancing if iterations are not of the same duration
- Dynamic:
 - 👎 High overhead (use of a critical section to update iteration index)
 - 👍 Good load balancing if iterations are not of the same duration
- Bottom-line so far: if you know all your iterations take the same time, use static!

Dynamic Scheduling Overhead

- If you have many iterations, doing dynamic scheduling where threads do only one iteration at a time could be very wasteful
- Instead, we would like to give out groups of iterations to threads each time
- Real-life metaphor: you have a set of documents to process and 10 workers working on a different floor. Each time one comes by your office asking “what should I do next?” given them a small stack of documents to process, not just one, so as to minimize overhead
- OpenMP allows you to do just that...

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
#pragma omp parallel private(i)
    {
#pragma omp for schedule(dynamic, 3)
        for (i = 0; i < 10; i++) {
            printf("Thread #%d: iteration %d\n",
                omp_get_thread_num(), i);
        }
    }
}
```

Specifies a
“chunk size” of 3

Let's look at the output...

Loop Scheduling Example

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

int main () {
    int i;
    omp_set_num_threads(4);
    #pragma omp parallel private(i)
    {
        #pragma omp for schedule(dynamic)
        for (i = 0; i < 10; i++) {
            printf("Thread #&#d: iteration %d\n",
                omp_get_thread_num(), i);
        }
    }
}
```

```
Thread #1: iteration 0
Thread #1: iteration 1
Thread #1: iteration 2
Thread #2: iteration 6
Thread #2: iteration 7
Thread #2: iteration 8
Thread #3: iteration 9
Thread #0: iteration 3
Thread #0: iteration 4
Thread #0: iteration 5
```

Let's look at the output...

How big a Chunk Size?

- Say you use `schedule(dynamic)` for a loop with iterations with non-uniform work per iteration
- Using a chunk size of 1 is not great, due to high overhead if iterations are short
- Using too large a chunk size is not great, due to poor load balancing if iterations are not uniform
- This begs the question: which chunk size should I use?
- But perhaps using a single chunk size isn't even such a good idea?...

Guided Scheduling

- At the beginning of the execution, using small chunks is not a good idea
 - Load-balancing doesn't really matter then
- At the end of the execution, using big chunks is not a good idea
 - Load-balancing matters a lot then
- Real-life metaphor:
 - You have 10 workers and a million things for them to do
 - It's probably ok to, at first, give them each 10,000 things to do, as if one gets "unlucky" they just won't come back for a while
 - But when you get down to, say, 100 things to do, then you probably want to be careful and give out work in small chunks, so that you won't end up waiting for an "unlucky" worker
- Take-away: It's probably a good idea to **decrease the chunk size** through the execution
- This is called **guided scheduling**...

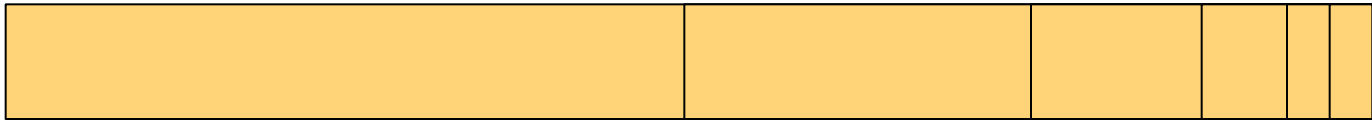
Guided Scheduling

- Say we have 4 threads and this many iterations:



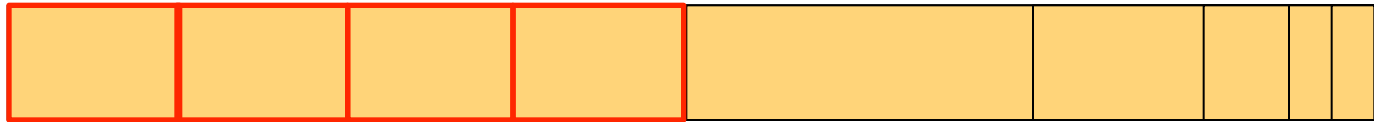
Guided Scheduling

- We split the iterations in halves...



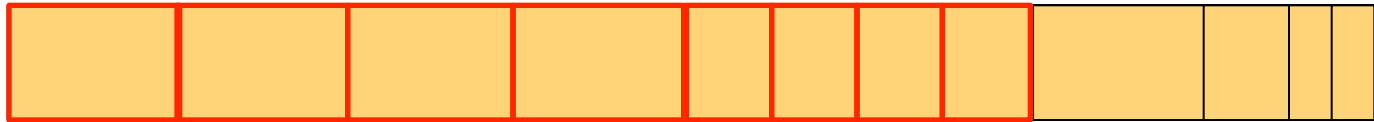
Guided Scheduling

- Then we split each piece into 4 chunks



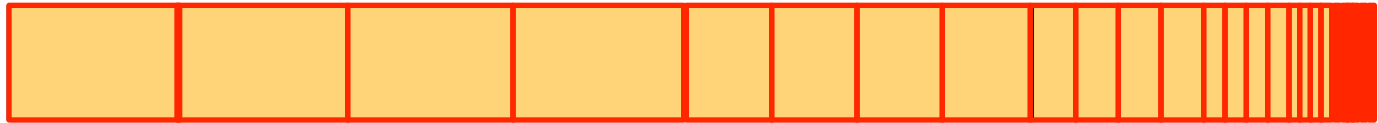
Guided Scheduling

- Then we split each piece into 4 chunks



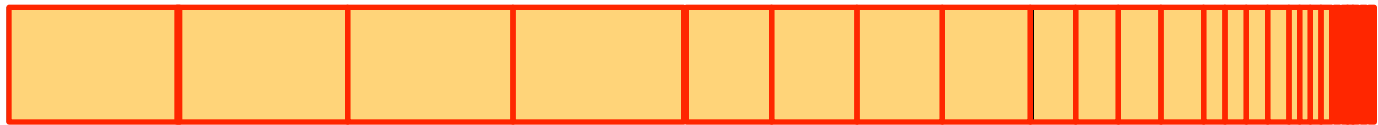
Guided Scheduling

- Then we split each piece into 4 chunks



Guided Scheduling

- Chunks are assigned to threads *dynamically* from left to right order



- We can specify the minimum chunk size
`#pragma omp for schedule(guided,100)`

Picking a Scheduling Strategy

- When parallelizing a loop we now have several scheduling strategies
- Although we can have good guesses on which strategy will work best for extreme situations, it's often a good idea to simply run experiments to determine what works best for our particular program
 - This is because, as usual, performance behavior of complicated, compiled code on a complicated multi-core architecture is hard to fully predict/understand

Nested Loops

- Often we have to parallelize nested loops
- The question then is: which loop do we put the pragma on?

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        // loop body  
    }  
}
```

Nested Loops

- Often we have to parallelize nested loops
- The question then is: which loop do we put the pragma on?

```
#pragma omp parallel private(i,j)
{
    #pragma omp for schedule(???)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            // loop body
        }
    }
}
```



outer loop

Nested Loops

- Often we have to parallelize nested loops
- The question then is: which loop do we put the pragma on?

```
for (i = 0; i < N; i++) {  
    #pragma omp parallel private(i,j)  
    {  
        #pragma omp for schedule(???)  
        for (j = 0; j < N; j++) {  
            // loop body  
        }  
    }  
}
```

inner loop

Collapsing Loops

- If parallelizing the outer loop, assuming a chunk size of 1, threads have to do N “big” iterations
- If parallelizing the inner loop, assuming a chunk size of 1, threads have to do N “small” iterations, N times
- If the body of the loop is relatively expensive to compute and non-deterministic, then we may want to have threads do $N*N$ “small” iterations to improve load balancing
- We could rewrite the loop as a single loop by hand...

Collapsing Loops by Hand

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        // loop body  
    }  
}
```



```
for (x = 0 ; x < N*N ; x++) {  
    i = x/N;  
    j = x % N;  
    // loop body  
}
```

Collapsing Loops with OpenMP

```
#pragma omp parallel private(i,j)
{
    #pragma omp for schedule(???) collapse(2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            // loop body
        }
    }
}
```

- There can be no code between the two for loops
- Loops cannot depend on each other

Conclusion

- Several ways to do multi-threading in C/C++
- OpenMP is popular because it strikes a good compromise between convenience and expressivity
 - You can do a lot with very little code
 - A lot of online material, including this [tutorial](#)
 - There are MANY features I haven't talked about
- Typically, “systems” people use Pthreads, while “applications” people use OpenMP
- Let's look at Homework Assignment #11...