Lecture 11 - OpenMP Overview

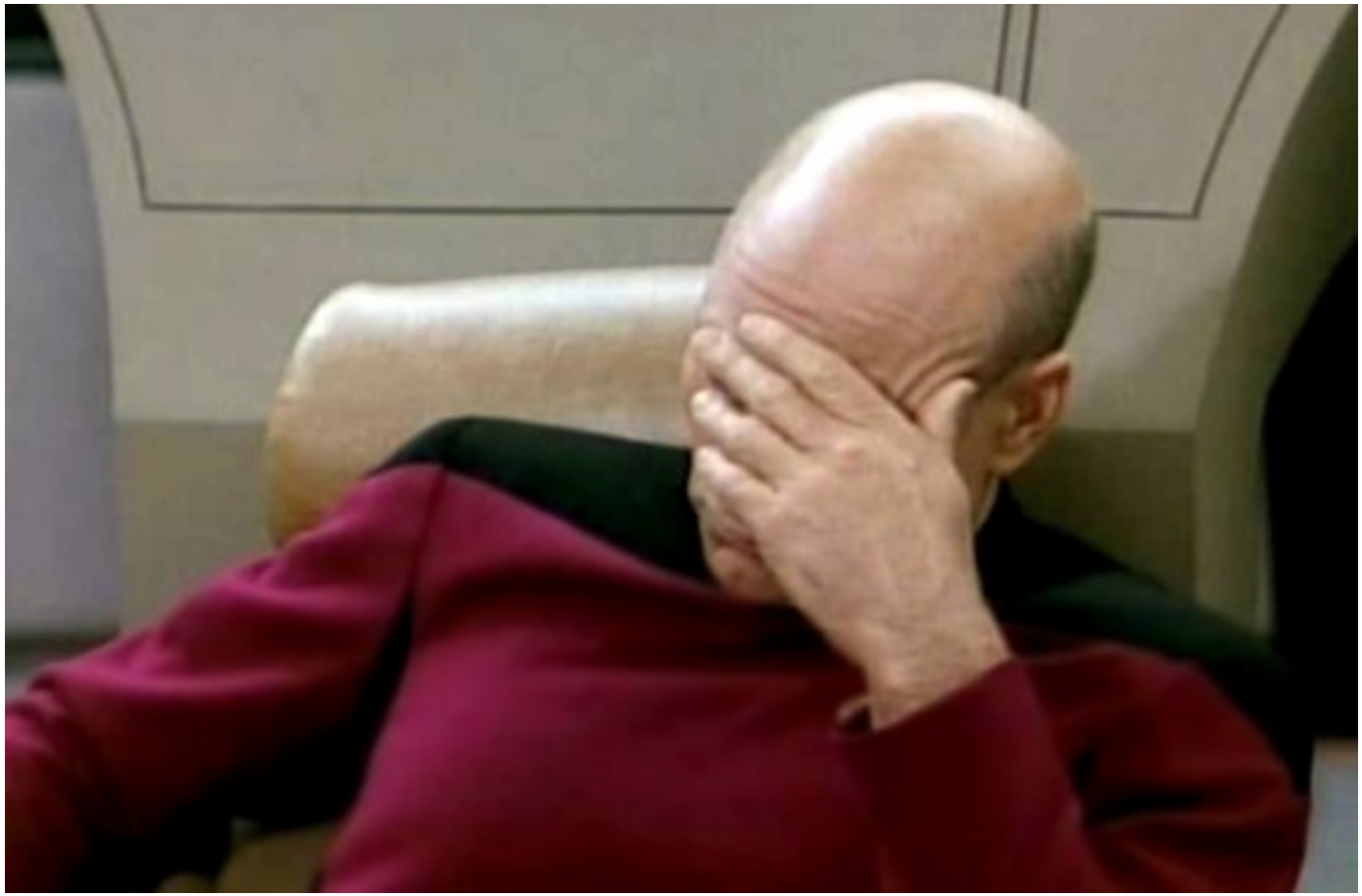
ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 27, 2012

Last Lecture



What is OpenMP?

- A portable, easy to use parallel programming API
- A collection of:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Compiling with OpenMP also defines `_OPENMP` for `ifdefs`

Documentation

`http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

Directive Format

`#pragma omp` *directive-name [clause [[,] clause]*]*

- There are **16** directives
- You can either have a single statment or compond statement { } after the directive

- Most clauses have a **list** as an agrument
  - A **list** is a comma seperated list of **list items**
    - A **list item** is simply a variable name (for C/C++)
- Most clauses have a list of variables as an argument

Data Terminology Keywords

- There are **3** keywords for data types
  - private
  - shared
  - threadprivate

- All of these data types relate to the scope and storage of variables

Private Variables

- Declared with `private` clause in OpenMP
- Creates new storage (does not copy values) for the variable
- Scope is from the start of the region to the end, after it is destroyed

Pthread pseudocode:

```
void* run(void* arg) {
    int x;
    // use x
}
```

Shared Variables

- Declared with `shared` clause in OpenMP
- All threads have access to the same block of data in the region

Pthread pseudocode:

```
int x;

void* run(void* arg) {
    // use x
}
```

Thread-Private Variables

- Declared with `threadprivate` directive in OpenMP
- Each thread makes a copy of the variable
- Variable accessible to a thread in any parallel region

```
int x;
#pragma omp threadprivate(x)
```

Maps to this Pthread pseudocode:

```
int x;
int x[NUM_THREADS];

void* run(void* arg) {
  // use x[thread_num]
}
```

Note for Clauses

- A variable may not appear in **more than one clause** on the same directive
- Exception for `firstprivate` and `lastprivate`, which we'll see later
- By default, variables declared in regions are private and outside are shared (few exceptions, anything with dynamic storage is shared)

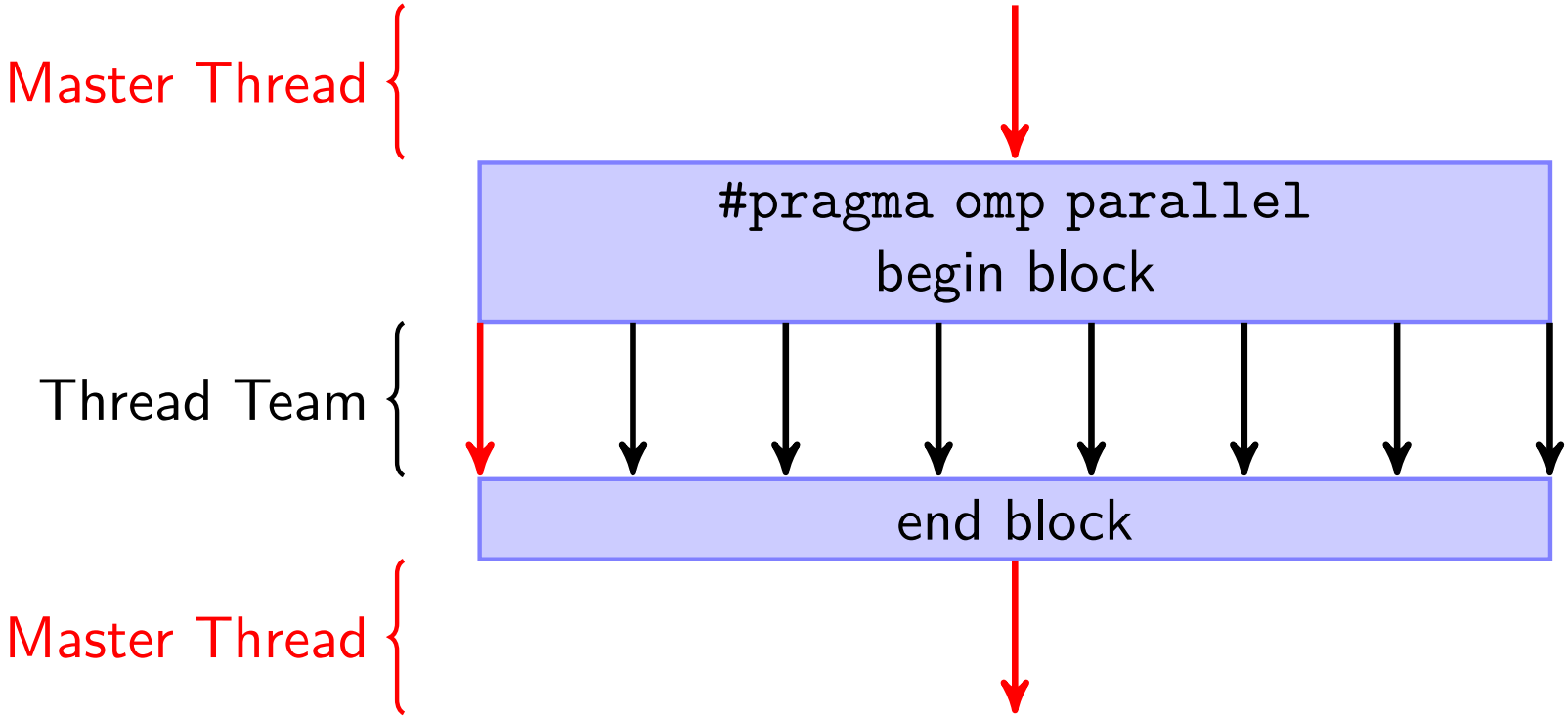
Parallel

`#pragma omp` **parallel** *[clause [[,] clause]*]*

- The most basic directive in OpenMP
- Forms a team of threads and starts parallel execution
- The thread that enters the region becomes the **master** (thread 0)

Allowed Clauses: **if, num_threads, default, private, firstprivate, shared, copyin, reduction**

Parallel Visually



- By default, the number of threads used is set globally automatically or manually
- After the parallel block, the thread team sleeps until it's needed

Parallel Example

```
#pragma omp parallel
{
    printf("Hello!");
}
```

If the number of threads is set to 4, this produces:

```
Hello!
Hello!
Hello!
Hello!
```

`if` and `num_threads` Clauses

- At most, one `if` and `num_threads` clause in a `parallel` directive

if(*primitive-expression*)

- If the expression is `false` then only one thread will execute

num_threads(*integer-expression*)

- This will spawn at most **num_threads** depending on the number of threads available
- The only way you'll guarantee the number of threads requested is if the **dynamic adjustment** for number of threads is off and there's enough threads that aren't busy

`reduction` Clause

reduction(*operator:list*)

Operators (Initial Value)

+	(0)	-	(0)	|	(0)	&&	(0)	max	MAX
*	(1)	&	(~0)	^	(0)	||	(0)	min	MIN

- Each thread gets a private copy of the variable
- The variable is initialized by OpenMP (so you don't need to do anything else)
- At the end of the region, OpenMP updates your result using the operator

reduction Clause Pseudocode using Pthreads

```
void* run(void* arg) {
    variable = initial value;
    // code inside of block which modifies variable
    return variable
}
... later in master thread (sequentially)
variable = initial value
for t in threads {
    thread_variable
    pthread_join(t, &thread_variable)
    variable = variable (operator) thread_variable
}
```

Loop

`#pragma omp` **for** *[clause [[,] clause]*]*

- Says that iterations of the loop will be distributed amoung the current team of threads
- Only supports simple for loops with invariant bounds (the bounds do not change during the loop)
- The loop variable is implicitly private and is set to the correct values

Allowed Clauses: **private, firstprivate, lastprivate, reduction, schedule, collapse, ordered, nowait**

`schedule` Clause

schedule(*kind[, chunk_size]*)

- The **chunk_size** is the number of iterations a single thread should handle at a time
- **kind** is one of:
  - **static**
  - **dynamic**
  - **guided**
  - **auto**
  - **runtime**
- **auto** is obvious (OpenMP decides what's best for you)
- **runtime** is also obvious, and we'll see how to adjust this later

`schedule` Clause `kinds`

static

- Divides the number of iterations into chunks and assigns each thread a chunk in round-robin fashion (before the loop executes)

dynamic

- Divides the number of iterations into chunks and assigns each available thread a chunk until there are no chunks left

guided

- Same as dynamic, except **chunk_size** represents the minimum size
- Starts off dividing the loop into large chunks, and decreases the chunk size as less iterations remain

`collapse` and `ordered` Clauses

collapse(n)

- This collapses n levels of loops
- This value should be at least 2, otherwise nothing happens
- The collapsed loop variables are also made private

ordered

- Enables the use of `ordered` directives

Ordered

`#pragma omp` **ordered**

- Loop must have an **ordered** clause
- OpenMP will ensure that the ordered directives are executed the same way the sequential loop would (one at a time)
- Each iteration of the loop may execute **at most one** ordered directive

Ordered Invalid Use

```
void work(int i) {
    printf("i = %d\n", i);
}
...
    int i;
    #pragma omp for ordered
    for (i = 0; i < 20; ++i) {

        #pragma omp ordered
        work(i);

        #pragma omp ordered
        work(i + 100);
    }
```

Ordered Valid Use

```
void work(int i) {
    printf("i = %d\n", i);
}
...
    int i;
    #pragma omp for ordered
    for (i = 0; i < 20; ++i) {
        if (i <= 10) {
            #pragma omp ordered
            work(i);
        }
        if (i > 10) {
            #pragma omp ordered
            work(i+100);
        }
    }
```

Ordered Valid Use

```
void work(int i) {
    printf("i = %d\n", i);
}
...
    int i;
    #pragma omp for ordered
    for (i = 0; i < 20; ++i) {
        if (i <= 10) {
            #pragma omp ordered
            work(i);
        }
        if (i > 10) {
            #pragma omp ordered
            work(i+100);
        }
    }
```

- **Note:** if we change `i > 10` to `i > 9`, this is now invalid

Tying It All Together

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

int main(int argc, char *argv[])
{
    int j, k, a;
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for collapse(2) ordered private(j,k) \
                        schedule(static,3)
        for (k = 1; k <= 3; ++k)
            for (j = 1; j <= 2; ++j) {
                #pragma omp ordered
                printf("t[%d] k=%d j=%d\n",
                       omp_get_thread_num(),
                       k, j);
            }
    }
    return 0;
}
```

Output of Previous Example

```
t[0]  k=1  j=1
t[0]  k=1  j=2
t[0]  k=2  j=1
t[1]  k=2  j=2
t[1]  k=3  j=1
t[1]  k=3  j=2
```

Note: this will always be our output, and it will run in two threads as long as our thread limit is at least 2

Parallel Loop

`#pragma omp` **parallel for** *[clause [[,] clause]*]*

Basically a shorthand for:

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

    }
}
```

Allowed Clauses: everything allowed by `parallel` and `for`, except **nowait**

Sections

`#pragma omp` **sections** *[clause [[,] clause]*]*

Allowed Clauses: **private, firstprivate, lastprivate, reduction, nowait**

Each **sections** directive must contain one or more **section** directive

`#pragma omp` **section**

- The sections are distributed among the current team of threads
- In **sections**, your parallelism is limited to the amount of sections you have

Parallel Sections

`#pragma omp` **parallel sections** *[clause [[,] clause]*]*

Again, basically a shorthand for:

```
#pragma omp parallel
{
    #pragma omp sections
    {

    }
}
```

Allowed Clauses: everything allowed by `parallel` and `sections`, except **nowait**

Single

`#pragma omp` **single**

- Only a single thread executes the region after **single**
- This isn't guaranteed to be the master thread

Allowed Clauses: **private, firstprivate, copyprivate, nowait**

- Must not use **copyprivate** with **nowait**

Barrier

`#pragma omp` **barrier**

- Waits for all the threads in the team to reach the barrier before continuing
- In other words, it's a synchronization point
- Also available in pthreads as `pthread_barrier`
- Loops, Sections, Single have an implicit barrer at the end of their region (unless you use the **nowait** clause)
- Cannot be used in any conditional blocks

Master

```
#pragma omp master
```

- Similar to the **single** directive
- Master thread is guaranteed to enter this region, and only the master thread
- No implied barriers or clauses

Critical

`#pragma omp` **critical** *[(name)]*

- The enclosed region is guaranteed to only have one thread at a time (for a specific name)
- Same as a block of code in Pthreads surrounded by a `mutex` lock and unlock

Atomic

`#pragma omp` **atomic** [**read** | **write** | **update** | **capture**]
expression-stmt

- Ensures a specific storage location is updated atomically
- More efficient than using critical sections (or else why would they include it?)

read expression: `v = x;`
write expression: `x = expr;`
update expression: `x++; x--; ++x; --x;`
`x binop = expr; x = x binop expr;`

Atomic Capture

- `expr` must not access the same location as `v` or `x`
- `v` and `x` must not access the same location and must be primitives
- All operations to `x` are atomic

capture expression: `v = x++; v = x--; v = ++x; v = --x;`
`v = x binop = expr;`

`#pragma omp` **atomic capture**
structured-block

- The structured blocks are equivalent to the expanded expressions

Other Directives

- **task**
- **taskyield**
- **taskwait**
- **flush**

We'll get into these next lecture

`firstprivate` and `lastprivate` Clauses

Pthread pseudocode for **firstprivate** clause:

```
int x;

void* run(void* arg) {
    int thread_x = x;
    // use thread_x
}
```

Pthread pseudocode for **lastprivate** clause:

```
int x;

void* run(void* arg) {
    int thread_x;
    // use thread_x
    if (last_iteration) {
        x = thread_x;
    }
}
```

- Same value as if the loop exited sequentially

copyin, copyprivate and default Clauses

- **copyin** is only for threadprivate variables

copyin Pthread pseudocode:

```
int x;
int x[NUM_THREADS];

void* run(void* arg) {
  x[thread_num] = x;
  // use x[thread_num]
}
```

- **copyprivate** is only used for a **single** directive it:
  - Copies the specified private variables of the thread to all other threads
  - Cannot be used with **nowait**

- **default** sets the default data-sharing is for variables (private, firstprivate, shared, none)

Execution Environment

To access the runtime library you need to `#include <omp.h>`

- `int omp_get_num_procs();`
  - The number of processors in the system
- `int omp_get_thread_num();`
  - The thread number of the currently executing thread
  - The master thread will return 0
- `int omp_in_parallel();`
  - Whether or not the function executed in a parallel region
- `int omp_get_num_threads();`
  - The number of threads in the current team

Locks

There are two types of locks:

- Simple
  - Cannot be set if it is already owned by the task trying to set it
- Nested
  - Can be set multiple times by the same task before being unset

The routines and usages are very similar to Pthreads:

`omp_init_lock`	`omp_init_nest_lock`
`omp_destroy_lock`	`omp_destroy_nest_lock`
`omp_set_lock`	`omp_set_nest_lock`
`omp_unset_lock`	`omp_unset_nest_lock`
`omp_test_lock`	`omp_test_nest_lock`

Timing

- `double omp_get_wtime();`
  - The elapsed wall clock time in seconds (since some time in the past)

- `double omp_get_wtick();`
  - The precision of the timer

Other Routines

We'll might see these in later lectures, they're mainly here for completeness:

- `int omp_get_level();`
- `int omp_get_active_level();`
- `int omp_get_ancestor_thread_num(int level);`
- `int omp_get_team_size(int level);`
- `int omp_in_final();`

Internal Control Variables

- These variables control how OpenMP handles threads
- Variables can be set with clauses/runtime routines/environment variables or just default values
- Routines will be represented as all lower case, environment variables as all upper case
- Clause > Routine > Environment Variable > Default Value
- All values (except 1) are implementation defined

Operation of Parallel Regions (1)

dyn-var

- Whether dynamic adjustment of the number of threads is enabled
- **Set by:** `OMP_DYNAMIC` `omp_set_dynamic`
- **Get by:** `omp_get_dynamic`

nest-var

- Whether nested parallelism is enabled
- **Set by:** `OMP_NESTED` `omp_set_nested`
- **Get by:** `omp_get_nested`
- Default value: `false`

Operation of Parallel Regions (2)

thread-limit-var

- The maximum number of threads in the program
- **Set by:** `OMP_NUM_THREADS` `omp_set_num_threads`
- **Get by:** `omp_get_max_threads`

max-active-levels-var

- Maximum number of nested active parallel regions
- **Set by:** `OMP_MAX_ACTIVE_LEVELS` `omp_set_max_active_levels`
- **Get by:** `omp_get_max_active_levels`

Operation of Parallel Regions/Loops

nthreads-var

- The number of threads requested for encountered parallel region
- **Set by:** `OMP_NUM_THREADS` `omp_set_num_threads`
- **Get by:** `omp_get_max_threads`

run-sched-var

- The **schedule** that the runtime schedule clause uses for loops
- **Set by:** `OMP_SCHEDULE` `omp_set_schedule`
- **Get by:** `omp_get_schedule`

Program Execution

bind-var

- Controls the binding of threads to processors
- **Set by:** `OMP_PROC_BIND`

stacksize-var

- Controls the stack size for threads
- **Set by:** `OMP_STACK_SIZE`

wait-policy-var

- Controls the desired behavior of waiting threads
- **Set by:** `OMP_WAIT_POLICY`

Summary

- Main concepts
  - **parallel**
  - **for** (**ordered**)
  - **sections**
  - **single**
  - **master**
- Synchronization
  - **barrier**
  - **critical**
  - **atomic**
- Data sharing: **private**, **shared**, **threadprivate**
- Should be able to use OpenMP effectively with a reference

Reference Card

`http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf`