Lecture 10X—A Principled View of OpenMP (extra)

ECE 459: Programming for Performance

February 11, 2013

What is OpenMP?

A portable, easy to use parallel programming API.

Combines:

- Compiler directives;
- Runtime library routines; and
- 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.

Either a single statement or a compound statement { } goes after the directive.

Most clauses have a **list** as an argument.

- A **list** is a comma-separated list of **list items**.
  A **list item** is simply a variable name (for C/C++)

Part I

Data Terminology

Three Keywords for Variable Scope and Storage

- private;
- shared; and
- threadprivate.

Private Variables

Declared with `private` clause in OpenMP.

Creates new storage (does not copy values) for the variable.

Scope extends from the start of the region to the end. Destroyed afterwards.

Pthread pseudocode for private variables:

```
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.

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 the thread in any parallel region.

OpenMP code:

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

maps to this Pthread pseudocode:

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

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

Contents of Clauses

A variable may not appear in **more than one clause** on the same directive.

There's an exception for `firstprivate` and `lastprivate`, which we'll see later.

By default, variables declared in regions are private and outside are shared (exception: anything with dynamic storage is shared).

Part II

Directives

Parallel

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

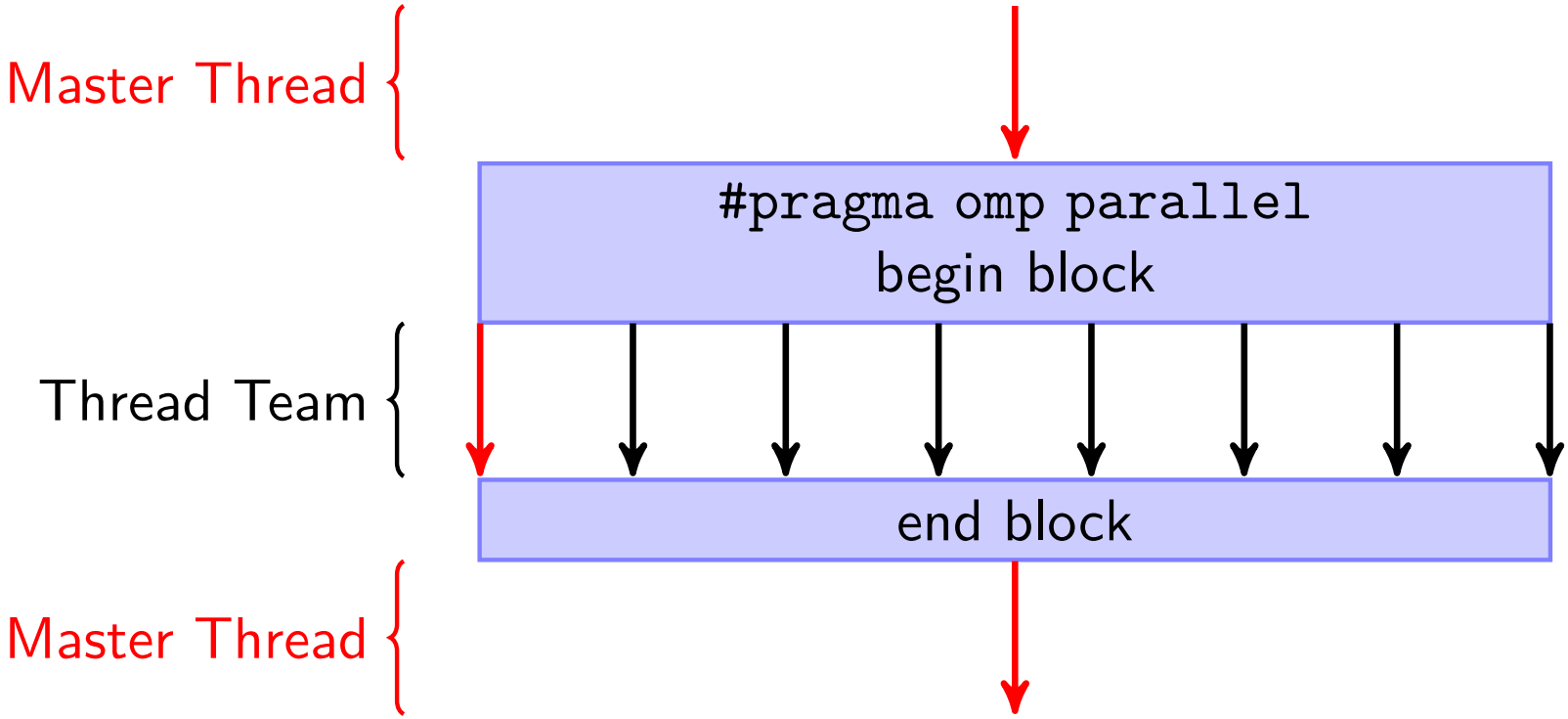
This is the most basic directive in OpenMP.

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**.

Visual Explanation of Parallel



- 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 4, this produces:

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

`if` and `num_threads` Clauses

if(*primitive-expression*)

- If primitive-expression `false`, then only one thread will execute.

If the parallel section is going to run multiple threads (e.g. **if** expression is true), we can specify how many:

num_threads(*integer-expression*)

- Spawns at most **num_threads**, depending on the number of threads available.
- Can only guarantee the number of threads requested if **dynamic adjustment** for number of threads is off and enough threads aren't busy.

`reduction` Clause

reduction(*operator:list*)

Operators (Initial Value)

+	(0)	-	(0)	|	(0)	&&	(1)	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 Pthreads Pseudocode

```
void* run(void* arg) {
    variable = initial value;
    // code inside block---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;
}
```

(For) Loop Clause

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

Iterations of the loop will be distributed among the current team of threads.

Only supports simple "for" loops with invariant bounds (bounds do not change during the loop).

Loop variable is implicitly private; OpenMP sets 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; 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 by dividing the loop into large chunks, and decreases the chunk size as fewer iterations remain.

`collapse` and `ordered` Clauses

collapse(n)

This collapses n levels of loops.

n should be at least 2, otherwise nothing happens.

Collapsed loop variables are also made private.

ordered

Enables the use of `ordered` directives inside loop.

Ordered

`#pragma omp` **ordered**

Containing 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.

Invalid Use of Ordered

```
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);

  // Each iteration of the loop has 2 "ordered" clauses!
  #pragma omp ordered
  work(i + 100);
}
```

Valid Use of Ordered

```
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) {
    // two ordered clauses are mutually-exclusive
    #pragma omp ordered
    work(i+100);
  }
}
```

Valid Use of Ordered

```
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) {
    // two ordered clauses are mutually-exclusive
    #pragma omp ordered
    work(i+100);
  }
}
```

- **Note:** if we change `i > 10` to `i > 9`, use becomes 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: output is determinstic; program will run two threads as long as thread limit is at least 2.

Parallel Loop

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

Basically 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**

- Sections distributed among current team of threads.
- **Sections** statically limit parallelism to the number of sections lexically in the code.

Parallel Sections

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

Again, basically 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.

Not 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—a synchronization point.

Loops, Sections, Single have an implicit barrer at the end of their region (unless you use **nowait**).

Cannot appear naked as a then-clause or else-clause; wrap the barrier with a { }.

Also available in pthreads as `pthread_barrier`.

Master

```
#pragma omp master
```

Similar to the **single** directive.

Master thread (and only the master thread) is guaranteed to enter this region.

No implied barriers, no clauses.

Critical

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

The enclosed region is guaranteed to only run one thread at a time (on a per-name basis).

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?)

Atomic Capture

read expression: `v = x;`

write expression: `x = expr;`

update expression: `x++; x--; ++x; --x;`
`x binop= expr; x = x binop expr;`

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

capture expression: `v = x++; v = x--; v = ++x; v = --x;`
`v = x binop= expr;`
Performs the indicated update. Also stores the original or final value computed.

Atomic Capture

`#pragma omp` **atomic capture**

structured-block

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

Pthreads 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 executed sequentially.

copyin, copyprivate and default Clauses

- **copyin** like firstprivate, but for threadprivate variables.

Pthreads pseudocode for **copyin**:

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

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

copyprivate is only used with **single**.

- Copies the specified private variables from the thread to all other threads.
- Cannot be used with **nowait**.

default(shared) makes all variables shared;
default(none) prevents sharing by default.

Part III

Runtime Library Routines

Execution Environment

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

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

Locks

Two types of locks:

- Simple: cannot be acquired if it is already held by the task trying to acquire it.
- Nested: can be acquired multiple times by the same task before being released (like Java).

Usage 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();`
  elapsed wall clock time in seconds (since some time in the past).

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

Other Routines

Might see these in later lectures. Included 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();`

Part IV

Internal Control Variables

Internal Control Variables

Control how OpenMP handles threads.

Can be set with clauses, runtime routines, environment variables, or just from defaults.

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

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

nest-var

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

Operation of Parallel Regions (2)

thread-limit-var

- 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

- number of threads requested for parallel regions.
- **Set by:** `OMP_NUM_THREADS` `omp_set_num_threads`
- **Get by:** `omp_get_max_threads`

run-sched-var

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

stacksize-var

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

wait-policy-var

- Controls desired behaviour of waiting threads.
- **Set by:** `OMP_WAIT_POLICY`

Part V

Summary

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`