ECE459: Programming for Performance Winter 2013

Lecture 10 — February 7, 2013

Patrick Lam *version 1*

We are going to go and present OpenMP in more detail. This will help you use it in Assignment 2.

OpenMP is a portable, easy-to-use parallel programming API. It combines compiler directives with runtime library routines and uses environment variables for setup. Note that you can detect its presence in your code with `#ifdef _OPENMP`.

For exhaustive documentation, see `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

Directive Format. All OpenMP directives follow this pattern:

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

There are **16** OpenMP directives. Each directive applies to the immediately-following statement, which is either a single statement or a compound statement **{** · · · **}**.

Most clauses have a *list* as an argument. A list is a comma-separated list of list items. For C and C++, a list item is simply a variable name.

Data Terminology

OpenMP includes three keywords for variable scope and storage:

- private;
- shared; and
- threadprivate.

Private Variables. You can declare private variables with the `private` clause. This creates new storage, on a per-thread bases, for the variable—it does not copy variables. The scope of the variable extends from the start of the region where the variable exists to the end of that region; the variable is destroyed afterwards.

Some Pthread pseudocode for private variables:

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

Shared Variables. The opposite of a private variable is a `shared` variable. All threads have access to the same block of data when accessing such a variable.

The relevant Pthread pseudocode is:

```
int x;

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

Thread-Private Variables. Finally, OpenMP supports `threadprivate` variables. This is like a `private` variable in that each thread makes a copy of the variable. However, the scope is different. Such variables are accessible to the thread in any parallel region.

This example will make things clearer. The 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()]
}
```

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; those outside regions are shared. (An exception: anything with dynamic storage is shared).

Directives

We'll talk about the different OpenMP directives next. These are the key language features you'll use to tell OpenMP what to parallelize.

Parallel

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

This is the most basic directive in OpenMP. It tells OpenMP to form a team of threads and start parallel execution. The thread that enters the region becomes the **master** (thread 0).

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

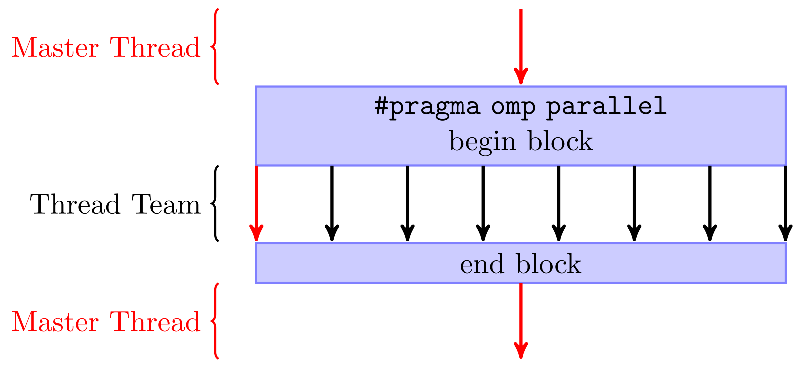

Figure 1: Visual view of `parallel` directive.

Figure 1 illustrates what `parallel` does. By default, the number of threads used is set globally, either automatically or manually. After the parallel block, the thread team sleeps until it's needed again.

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

If the number of threads is 4, this produces:

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

`if` and `num_threads` Clauses. Directives take clauses. The first one we'll see (besides the variable scope clauses) are `if` and `num_threads`.

if(*primitive-expression*)

The `if` clause allows you to control at runtime whether or not to run multiple threads or just one thread in its associated parallel section. If *primitive-expression* evaluates to `0`, i.e. `false`, then only one thread will execute in the parallel section. (It's what you would expect.)

If the parallel section is going to run multiple threads (e.g. `if` expression is true; or if there is no `if` expression), we can then specify how many threads.

num_threads(*integer-expression*)

This spawns at most **num_threads**, depending on the number of threads available. It 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.** We saw reductions last time. Here's another look.

reduction(*operator:list*)

Operators (and their associated initial value)

+	(0)	-	(0)	|	(0)	&&	(1)	max	MAX
	(1)	&	(˜0)	ˆ	(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.

```
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 Directive

Inside a parallel section, we can specify a for loop clause, as follows.

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

Iterations of the loop will be distributed among the current team of threads. This clause 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 thread should handle at a time. **kind** is one of:

- **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. This starts by dividing the loop into large chunks, and decreases the chunk size as fewer iterations remain.
- **auto**: obvious (OpenMP decides what's best for you).
- **runtime**: also obvious; we'll see how to adjust this later.

`collapse` and `ordered` Clauses.

collapse(n)

This collapses n levels of loops. Obviously, this only has an effect if $n \geq 2$; otherwise, nothing happens. Collapsed loop variables are also made private.

ordered

This enables the use of `ordered` directives inside loop, which we'll see below.

Ordered directive

`#pragma omp ordered`

To use this directive, the 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.

Let's see what that means by way of two examples.

Invalid Use of Ordered. This doesn't work.

```
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. This does.

```
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`, the use becomes invalid because the iteration $i = 9$ contains two `ordered` directives.

Tying It All Together. Here's a larger example.

```
#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. And here's what it does.

```
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: the output will be determinstic; still, the program will run two threads as long as the thread limit is at least 2.

Parallel Loop Directive

This directive is shorthand:

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

We could equally well write:

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

but the single directive is shorter; this idiom happens a lot.

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

Sections

Another OpenMP parallelism idiom is 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. They statically limit parallelism to the number of sections which are lexically in the code.

Parallel Sections. Another common idiom.

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

As with parallel for, this is basically shorthand for:

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

    }
}
```

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

Single

When we'd be otherwise running many threads, we can state that some particular code block should be run by only one method.

`#pragma omp` **single**

Only a single thread executes the region following the directive. The thread is 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, this constitutes a synchronization point. Loops, sections, and single all have an implicit barrer at the end of their region (unless you use **nowait**). Note that barrier *cannot* be used in any conditional blocks.

This mechanism is analogous to `pthread_barrier` in Pthreads.

Master

Sometimes we do want to guarantee that only the master thread runs some code.

`#pragma omp` **master**

This is similar to the **single** directive, except that the master thread (and only the master thread) is guaranteed to enter this region. No implied barriers.

Also, no clauses.

Critical

We turn our attention to synchronization constructs. First, let's examine critical.

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

The enclosed region is guaranteed to only run one thread at a time (on a per-name basis). This is the same as a block of code in Pthreads surrounded by a `mutex` lock and unlock.

Atomic

We can also request atomic operations.

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

This ensures that a specific storage location is updated atomically. Atomics should be 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;`

  `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;`

  Capture expressions perform the indicated update. At the same time, they also store the original or final value computed into location `v`.

Atomic Capture

There's also a directive for atomic capture where you specify a more complicated block to be run atomically.

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

Other Directives

We'll get into these next lecture.

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

More Scoping Clauses: `firstprivate`, `lastprivate`, `copyin`, `copyprivate` and `default`

Besides the `shared`, `private` and `threadprivate`, OpenMP also supports `firstprivate` and `lastprivate`, which work like this.

Pthreads pseudocode for `firstprivate` clause:

```
int x;

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

Pthread pseudocode for the `lastprivate` clause:

```
int x;

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

In other words, `lastprivate` makes sure that the variable `x` has the same value as if the loop executed sequentially.

`copyin` is 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]
}
```

The `copyprivate` clause is only used with `single`. It copies the specified private variables from the thread to all other threads. It cannot be used with `nowait`.

Defaults. **default(shared)** makes all variables shared; **default(none)** prevents sharing by default (creating compiler errors if you treat a variable as shared.)

Runtime Library Routines

To use the runtime library and call OpenMP functions (rather than using pragmas), you need to `#include <omp.h>`.

- `int omp_get_num_procs();`: return the number of processors in the system.

- `int omp_get_thread_num();`: return the thread number of the currently executing thread (the master thread will return 0).
- `int omp_in_parallel();`: returns true if currently in a parallel region.
- `int omp_get_num_threads();`: return the number of threads in the current team.

Locks in OpenMP. OpenMP provides 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).

Lock usage is 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. You can measure how long things take, which is useful for domain-specific profiling.

- `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. Wight 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();`

Internal Control Variables

OpenMP uses internal variables to control how it handles threads. These 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. We can control how parallel regions work with the following variables.

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`

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. These apply to both parallel regions and 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. We can also control 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`

Summary

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

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