**ECE459: Programming for Performance** Winter 2020

# Lecture 13 — OpenMP

*Patrick Lam* *2019-12-18*

## OpenMP

OpenMP (Open Multiprocessing) is an API specification which allows you to tell the compiler how you'd like your program to be parallelized. Implementations of OpenMP include compiler support (present in Intel's compiler, Solaris's compiler, `gcc` as of 4.2, and Microsoft Visual C++) as well as a runtime library.

You use OpenMP [Bar15] by specifying directives in the source code. In C and C++, these directives are pragmas of the form `#pragma omp ...`. There is also OpenMP syntax for Fortran.

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

Here are some benefits of the OpenMP approach:

- Because OpenMP uses compiler directives, you can easily tell the compiler to build a parallel version or a serial version (which it can do by ignoring the directives). This can simplify debugging—you have some chance of observing differences in behaviour between versions.
- OpenMP's approach also separates the parallelization implementation (inserted by the compiler) from the algorithm implementation (which you provide), making the algorithm easier to read. Plus, you're not responsible for dealing with thread libraries.
- The directives apply to limited parts of the code, thus supporting incremental parallelization of the program, starting with the hotspots.

Let's look at a simple example of a parallel for loop:

```
void calc (double *array1, double *array2, int length) {
  #pragma omp parallel for
  for (int i = 0; i < length; i++) {
    array1[i] += array2[i];
  }
}
```

This `#pragma` instructs the C compiler to parallelize the loop. It is the responsibility of the developer to make sure that the parallelization is safe; for instance, `array1` and `array2` had better not overlap. You no longer need to supply `restrict` qualifiers, although it's still not a bad idea. (If you wanted this to be autoparallelized without OpenMP, you would need to provide `restrict`.)

OpenMP will always start parallel threads if you tell it to, dividing the iterations contiguously among the threads.

Let's look at the parts of this `#pragma`.

- `#pragma omp` indicates an OpenMP directive;
- `parallel` indicates the start of a parallel region; and
- `for` tells OpenMP to run the following `for` loop in parallel.

When you run the parallelized program, the runtime library starts up a number of threads and assigns a subrange of the loop range to each of the threads.

**Restrictions.** OpenMP places some restrictions on loops that it's going to parallelize:

- the loop must be of the form

```
for (init expression; test expression; increment expression);
```

- the loop variable must be integer (signed or unsigned), pointer, or a C++ random access iterator;
- the loop variable must be initialized to one end of the range;
- the loop increment amount must be loop-invariant (constant with respect to the loop body);
- the test expression must be one of >, >=, <, or <=, and the comparison value (bound) must be loop-invariant.

(These restrictions therefore also apply to automatically parallelized loops.) If you want to parallelize a loop that doesn't meet the restriction, restructure it so that it does, as we did for autoparallelization

**Runtime effect.** When you compile a program with OpenMP directives, the compiler generates code to spawn a *team* of threads and automatically splits off the worker-thread code into a separate procedure. The code uses fork-join parallelism, so when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back. Then the master thread continues running, while the worker threads wait for more work.

You can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable (adjustable by calling `omp_set_num_threads()`), and you can get the Solaris compiler to tell you what it did by giving it the options `-xopenmp -xloopinfo`.

## Variable scoping

When using multiple threads, some variables, like loop counters, should be thread-local, or *private*, while other variables should be *shared* between threads. Changes to shared variables are visible to all threads, while changes to private variables are visible only to the changing thread. Let's look at the defaults that OpenMP uses to parallelize the above code.

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 basis, 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) {
  // int* id = (int*) arg;
  // use x[*id]
}
```

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).

We can ask the Solaris compiler what it did:

```
$ er_src parallel-for.o
     1.   void calc (double *array1, double *array2, int length) {
        <Function: calc>

    Source OpenMP region below has tag R1
    Private variables in R1: i
    Shared variables in R1: array2, length, array1
     2.     #pragma omp parallel for

    Source loop below has tag L1
    L1 autoparallelized
    L1 parallelized by explicit user directive
    L1 parallel loop-body code placed in function _$d1A2.calc along with 0 inner loops
    L1 multi-versioned for loop-improvement:dynamic-alias-disambiguation.
        Specialized version is L2
     3.     for (int i = 0; i < length; i++) {
     4.       array1[i] += array2[i];
     5.     }
     6.   }
```

We can see that the loop variable `i` is private, while the `array1`, `array2` and `length` variables are shared. Actually, it would be fine for the `length` variable to be either shared or private, but if it was private, then you would have to copy in the appropriate initial value. The `array` variables, though, need to be shared.

**Summary of default rules.** Loop variables are private; variables defined in parallel code are private; and variables defined outside the parallel region are shared.

You can disable the default rules by specifying `default(none)` on the `parallel` pragma, or you can give explicit scoping:

```
#pragma omp parallel for private(i) shared(length, array1, array2)
```

## 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.

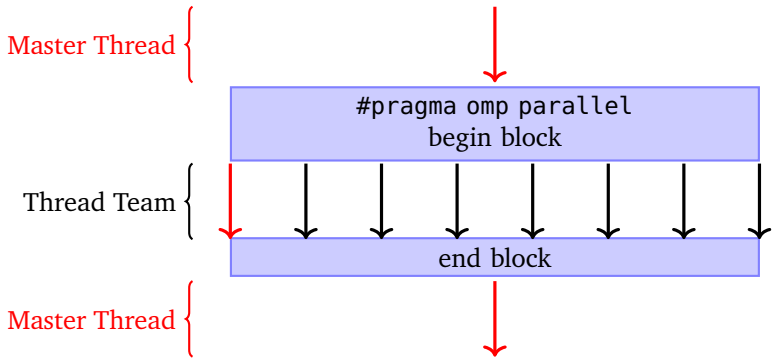
### Parallel

`#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**.

The figure below 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.

## Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++) total += array[i];
```

What is the appropriate scope for `total`? We want each thread to be able to write to it, but we don't want race conditions. Fortunately, OpenMP can deal with reductions as a special case:

```
#pragma omp parallel for reduction (+:total)
```

specifies that the `total` variable is the accumulator for a reduction over `+`. OpenMP will create local copies of `total` and combine them at the end of the parallel region.

**Operators (and their associated initial value)**

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

## (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**.

**The `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.

OpenMP tries to guess how many iterations to distribute to each thread in a team. The default mode is called *static scheduling*; in this mode, OpenMP looks at the number of iterations it needs to run, assumes they all take the same amount of time, and distributes them evenly. So for 100 iterations and 2 threads, the first thread gets 50 iterations and the second thread gets 50 iterations.

This assumption doesn't always hold; consider, for instance, the following (contrived) code:

```
double calc(int count) {
  double d = 1.0;
  for (int i = 0; i < count*count; i++) d += d;
  return d;
}

int main() {
  double data[200][200];
  int i, j;
  #pragma omp parallel for private(i, j) shared(data)
  for (int i = 0; i < 200; i++) {
    for (int j = 0; j < 200; j++) {
      data[i][j] = calc(i+j);
    }
  }
  return 0;
}
```

This code gives sublinear scaling, because the earlier iterations finish faster than the later iterations, and the program needs to wait for all iterations to complete.

Telling OpenMP to use a *dynamic schedule* can enable better parallelization: the runtime distributes work to each thread in chunks, which results in less waiting. Just add `schedule(dynamic)` to the pragma. Of course, this has more overhead, since the threads need to solicit the work, and there is a potential serialization bottleneck in soliciting work from the single work queue.

The default chunk size is 1, but you can specify it yourself, either using a constant or a value computed at runtime, e.g. `schedule(dynamic, n/50)`. Static scheduling also accepts a chunk size.

OpenMP has an even smarter work distribution mode, `guided`, where it changes the chunk size according to the amount of work remaining. You can specify a minimum chunk size, which defaults to 1. There are also two meta-modes, `auto`, which leaves it up to OpenMP, and `runtime`, which leaves it up to the `OMP_SCHEDULE` environment variable.

`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.

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. But if you forget the parallel, you don't get the behaviour you expect: no team of threads!

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**

## 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.

## 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, although it's always good practice to put statements following a `#pragma omp` inside a compound block (`{ ...}`), a barrier inside an `if` statement must be `{ }` separated.

This mechanism is analogous to `pthread_barrier` in pthreads.

### 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?)

## A Warning About Using OpenMP Directives

Write your code so that simply eliding OpenMP directives gives a valid program. For instance, this is wrong:

```
if (a != 0)
    #pragma omp barrier // wrong!
if (a != 0)
    #pragma omp taskyield // wrong!
```

Use this instead:

```
if (a != 0) {
    #pragma omp barrier
}
if (a != 0) {
    #pragma omp taskyield
}
```

## References

[Bar15] Blaise Barney. Openmp, 2015. Online; accessed 12-December-2015. URL: `https://computing.llnl.gov/tutorials/openMP/`.