Lecture 13 — OpenMP

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 24, 2019

From pthreads to OpenMP

So far: Pthreads and automatic parallelization.

Next: “Manual” parallelization using OpenMP (Open Multiprocessing).

You specify parallelization; compiler implements.

“I want 10 turnstiles.”

Make It So, Number One

You use OpenMP by specifying directives in the code.

Directives look like `#pragma omp ...`

There are 16 directives; each applies to the immediately-following statement.

Some have a list as an argument (a list of variable names)

Modern compilers support OpenMP

All major compilers have OpenMP: (gcc, clang, Solaris, Intel, Microsoft).

Benefits:

- Switch parallelism on and off via compile option
- Make the compiler do hard work
- Makes algorithms easier to read
- Apply them only in parts where needed, incrementally

Simple For Loop Exampe

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

The directive instructs the compiler to parallelize the loop.

Developer's responsibility to make sure this is safe!

You don't need to declare `restrict`, but it's a good idea.

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

Source: Feuerwehrmagazin.de

`#pragma omp parallel for`

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

The runtime library starts a number of threads & assigns a subrange of the range to each of the threads.

Loops of a Certain Kind

```
for (int i = 0; i < length; i++) { ... }
```

Can only parallelize loops which satisfy these conditions:

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

Note: these restrictions thus also apply to automatically parallelized loops.

Parallelized Code Generated

Get together a team!

1. Compiler generates code to spawn a team of threads; automatically splits off worker-thread code into a separate procedure.
2. Generated code uses `fork-join` parallelism; when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back.
3. Afterwards, the master thread continues running, while the worker threads wait for more work .

Parallelized Code Generated

You can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable.
(You can also adjust by calling `omp_set_num_threads()`).

- Solaris compiler tells you what it did if you use the flags `-xopenmp -xloopinfo`, or `er_src`.

Variable Scoping

OpenMP includes three keywords for variable scope and storage:

- private
- shared
- threadprivate

Private Variable

Private: new storage, on a per-thread basis, for the variable.

The scope from the start of the region to the end of that region.

The variable is destroyed afterwards.

Thread equivalent:

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

Shared Variable

Shared: the opposite of a private variable.

All threads have access to the same block of data when accessing such a variable.

Thread equivalent:

```
int x;

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

Threadprivate

Threadprivate: This is like private, 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:

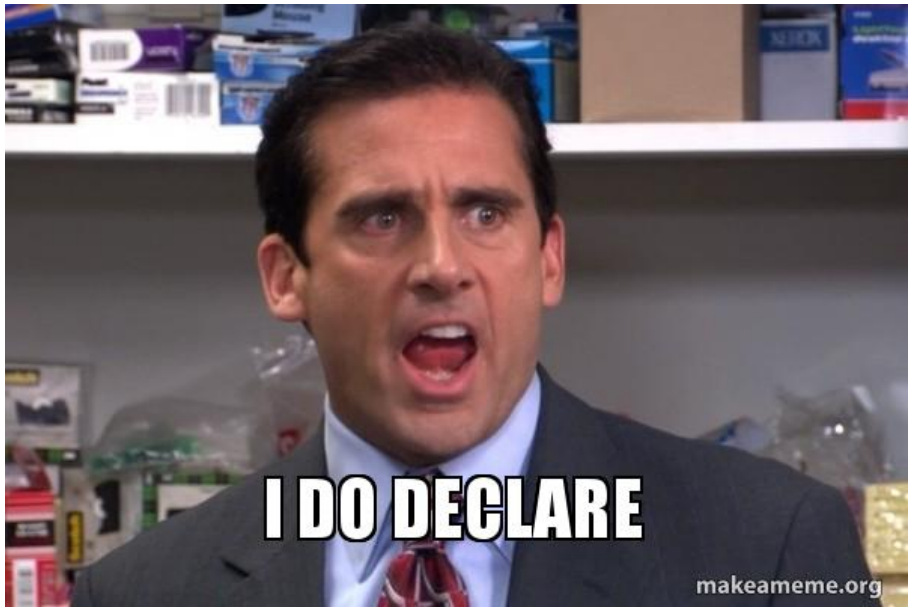
```
int x;
#pragma omp threadprivate(x)
```

maps to this Pthread pseudocode:

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

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

I Do Declare...



A variable may not appear in **more than one clause** on the same directive. (There's an exception for `firstprivate` and `lastprivate`)

Default: vars declared in regions are private; those outside regions are shared.

(An exception: anything with dynamic storage is shared).

Solaris, why...

Let's look at the defaults that OpenMP uses to parallelize the `parallel-for` code:

```
% er_src parallel-for.o
     1.    <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.    }
```

Override!

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

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

Directives...

Oh.... Wait... Directives, not Directors.

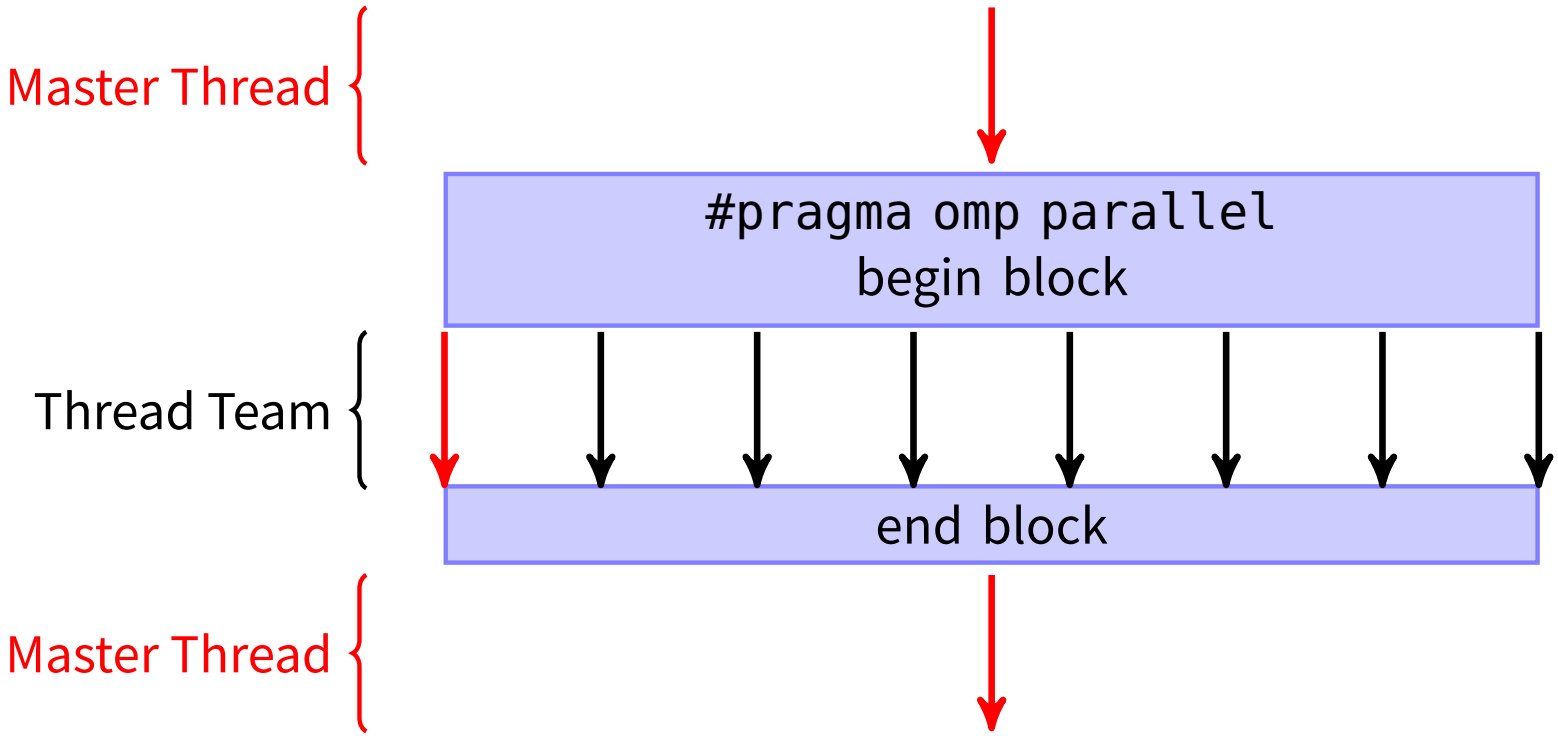
Directives: Parallel

`#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**.


Master Thread
#pragma omp parallel
begin block
Thread Team
end block
Master Thread

Parallel Code Example

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

If the number of threads is 4, this produces:

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

If...

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.)

Be Specific

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.

Directives: Reduction

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)
```

Reduction Operators

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

Directives: For

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.

Loop variable is implicitly private; OpenMP sets the correct values.

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

Schedule it in...

More Control: schedule

schedule(*kind[, chunk_size]*)

The **chunk_size** is the number of iterations a thread should handle at a time.

kind is one of:

- **static**
- **dynamic**
- **guided**
- **auto**
- **runtime**

OpenMP tries to guess how many iterations to distribute to each thread in a team.

Schedule Example

The default of equal number of iterations isn't always optimal:

```
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;
}
```

Telling OpenMP to use a *dynamic schedule* can enable better parallelization.

Collapse

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

```
#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.

Ordnung muss sein

This use of Ordered does not 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);
}
```

Ordnung muss wirklich sein

Here is a valid use of the ordered clause:

```
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.

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

Remember Parallel

This directive is shorthand:

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

We could equally well write:

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

    }
}
```

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

Directives: Sections

`#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**.

All the Single Threads

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... Apprentice... Heartborne... 7th Seeker

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.

Man the Barricades

```
#pragma omp barrier
```

Waits for all the threads in the team to reach the barrier before continuing.

Loops, sections, and single all have an implicit barrer at the end of their region (unless you use **nowait**).

A barrier inside an `if` statement must be `{ }` separated.

This mechanism is analogous to `pthread_barrier` in pthreads.

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?)

Oh, Think Twice...

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
}
```