Lecture 13 — OpenMP

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 5, 2018

photo: Arnold Reinhold via Wikimedia Commons

From pthreads to OpenMP

So far: Pthreads and
automatic parallelization.

Next: “Manual” parallelization
using OpenMP.

OpenMP allows you to specify parallelization.

OpenMP = Open Multiprocessing.

You specify parallelization;
compiler implements.

“I want 10 turnstiles.”

Modern compilers support OpenMP.

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

Use OpenMP[1] by specifying
directives in source code.

C/C++: pragmas of the form
`#pragma omp ...`

[1] More information: `https://computing.llnl.gov/tutorials/openMP/`

OpenMP separates parallelization implementation from algorithm implementation.

OpenMP uses compiler directives—
compile same codebase for serial or parallel.

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

Enables incremental parallelization of the program.
(Start with the hotspots.)

Without OpenMP:
Could compiler parallelize this automatically?

#pragma forces the compiler to parallelize the loop.

- It does not look at loop contents, only loop bounds.
- It is your responsibility to make sure the code is safe.

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

You don't need to declare `restrict`, but it's a good idea. Need `restrict` for auto-parallelization (non-OpenMP).

`#pragma omp parallel for` contains three parts.

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

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

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

OpenMP can parallelize certain loops.

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

Can only parallelize loops which satisfy these conditions:

- must be of the 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.

OpenMP generates parallelized code.

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 .

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`.

You can control variable scoping in OpenMP.

Concept: thread-local variables (private)
vs shared variables.

- Writes to private variables:
  visible only to writing thread.
- Writes to shared variables:
  visible to all threads.

In example, `length` could be shared or private.

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

- if `length` was private,
  you'd have to copy in appropriate initial value.
- `array` variables must be shared.

OpenMP has sensible default scoping.

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
```

Solaris tells you OpenMP parallelization information.

```
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.    }
```

Default Variable Scoping Rules: loop vars private, parallel vars private, others shared.

- Loop variables are private.
- Variables defined in parallel code are private.
- Variables defined outside the parallel region are shared.

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)
```

What is appropriate scope for reduction variable?

Consider a reduction, e.g.

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

What is the appropriate scope for `total`?

What is appropriate scope for reduction variable?

Consider a reduction, e.g.

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

What is the appropriate scope for `total`?
Well, it should be shared.

- We want each thread to be able to write to it.

What is appropriate scope for reduction variable?

Consider a reduction, e.g.

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

What is the appropriate scope for `total`?
Well, it should be shared.

- We want each thread to be able to write to it.
- But, is there a race condition? (of course)

Aha! OpenMP deals with reductions as a special case:

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

specifies that the `total` variable is the accumulator for a reduction over the + operator.

OpenMP lets you access private data outside a parallel region (on request).

Sometimes you want private variables,
but want them initialized before the loop.

Consider this (silly) code:

```
int data=1;
#pragma omp parallel for private(data)
for (int i = 0; i < 100; i++)
    printf ("data=%d\n", data);
```

- `data` is private, so OpenMP will not copy in initial 1.
- To make OpenMP copy data before threads start, use `firstprivate(data)`.
- To publish a variable after the (sequentially) last iteration of the loop, use `lastprivate(data)`.

OpenMP supports thread-private data.

You might have a global variable,
for which each thread should have a persistent local copy
(lives across parallel regions.)

- Use the `threadprivate` directive.
- Add `copyin` for something like `firstprivate`.
- There is no `lastprivate`: data accessible after loop.

Thread-Private Data Example (1)

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

int tid, a, b;

#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    printf("Parallel #1 Start\n");
    #pragma omp parallel private(b, tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    printf("Sequential code\n");
```

Thread-Private Data Example (2)

```
    printf("Parallel #2 Start\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    return 0;
}
```

```
% ./a.out
Parallel #1 Start
T6: a=6, b=6
T1: a=1, b=1
T0: a=0, b=0
T4: a=4, b=4
T2: a=2, b=2
T3: a=3, b=3
T5: a=5, b=5
T7: a=7, b=7
```

```
Sequential code
Parallel #2 Start
T0: a=0, b=0
T6: a=6, b=0
T1: a=1, b=0
T2: a=2, b=0
T5: a=5, b=0
T7: a=7, b=0
T3: a=3, b=0
T4: a=4, b=0
```

Atomic ensures a storage location is updated atomically.

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

Atomic is more efficient than
using critical sections.
(or else why would they include it?)

Atomic expressions include read, write, update, 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.

OpenMP can collapse loops.

Normally, it's best to parallelize the outermost loop. But, consider this code:

```
#include <math.h>
int main() {
    double array[2][10000];
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < 10000; j++)
        array[i][j] = sin(i+j);
    return 0;
}
```

- Would parallelizing this outer loop benefit us? What about the inner loop?

OpenMP supports *collapsing* loops:

- Creates a single loop for all the iterations of the two loops.
- Outer loop only enables the use of 2 threads.
- Collapsed loop lets us use up to 20,000 threads.

Better scheduling can improve performance.

OpenMP's default mode: *Static scheduling*.

- Assumes each iteration takes same running time.

Does that assumption hold for this 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][100];
    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;
}
```

Example gives sublinear scaling with static scheduling.

- Earlier iterations are faster than later iterations.
  Result: sublinear scaling—
  must wait for all iterations to finish.

- Turn on *dynamic schedule* mode
  by adding `schedule(dynamic)` to the pragma:
  - Breaks the work into chunks;
  - Distributes the work to each thread in chunks;
  - Higher overhead;
  - Default chunk size of 1
    (can modify, e.g. `schedule(dynamic, n/50)`).

OpenMP supports yet more scheduling options: `guided`, `auto`, and `runtime`.

- `guided` changes the chunk size based on work remaining.
  - Default minimum chunk size = 1 (can modify)
- `auto` lets OpenMP decide what's best.
- `runtime` doesn't pick a mode until runtime.
  - Tune with `OMP_SCHEDULE` environment variable

Why more than for?

So far, we've seen how to parallelize (some) `for` loops.

Less powerful than Pthreads. (Also harder to get wrong.)

Reflects OpenMP's scientific-computation heritage.

Today, we need more general parallelism, not just matrices.

OpenMP supports parallel sections—an example.

Parallel sections: purely-static mechanism for specifying independent work units which should run in parallel.

Linked list example:

```
#include <stdlib.h>

typedef struct s { struct s* next; } S;

void setuplist (S* current) {
  for (int i = 0; i < 10000; i++) {
    current->next = (S*) malloc (sizeof(S));
    current = current->next;
  }
  current->next = NULL;
}
```

OpenMP can process two linked lists in parallel with parallel sections.

(Exactly) 2 linked lists:

```
int main() {
  S var1, var2;
  #pragma omp parallel sections
  {
    #pragma omp section
    { setuplist (&var1); }
    #pragma omp section
    { setuplist (&var2); }
  }
  return 0;
}
```

Parallelism structure explicitly visible.

Finite number of threads.

(What's another barrier to more general parallelism?)

Parallel sections can help avoid collapsing loops.

Sometimes you don't want to collapse loops.

Example: (better example in PDF notes!)

```
#pragma omp parallel sections
{
  #pragma omp section
  {
      #pragma omp parallel for
      for (int i = 0; i < 1000; i++) { ... }
  }
  #pragma omp section
  {
      #pragma omp parallel for
      for (int i = 0; i < 1000; i++) { ... }
  }
}
```

To enable nested parallelism, call `omp_set_nested(1)` or set `OMP_NESTED`. (Runtime might refuse.)

OpenMP tasks support irregular parallelism.

Tasks: main new feature in OpenMP 3.0.

`#pragma omp task`:
code splits off and scheduled to run later.

More flexible than parallel sections:

- can run as many threads as needed;
- tasks do not need to join (like detached threads).

OpenMP does the task-to-thread mapping—
lower overhead.

Examples of tasks

Two examples:

- web server
  unstructured requests
- user interface
  allows users to start concurrent tasks

Boa webserver main loop example

```
#pragma omp parallel
  /* a single thread manages the connections */
  #pragma omp single nowait
  while (!end) {
    process any signals
    foreach request from the blocked queue {
      if (request dependencies are met) {
        extract from the blocked queue
        /* create a task for the request */
        #pragma omp task untied
          serve_request(request);
      }
    }
    if (new connection) {
      accept_connection();
      /* create a task for the request */
      #pragma omp task untied
        serve_request(new connection);
    }
    select();
  }
```

Other OpenMP qualifiers

`untied`: lifts restrictions on task-to-thread mapping.

`single`: only one thread runs the next statement (not *N* copies).

`flush` directive: write all values in registers or cache to memory.

`barrier`: wait for all threads to complete. (OpenMP also has implicit barriers at ends of parallel sections.)

OpenMP also supports critical sections (one thread at a time), atomic sections, and typical mutex locks (`omp_set_lock`, `omp_unset_lock`).

OpenMP Directives Warning: Write code so that eliding OpenMP 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
}
```