MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

OpenMP - Part 2

Daniel Playne
`d.p.playne@massey.ac.nz`

Reminder

In the last lesson we started looking at some of the code directives in OpenMP.

- `parallel`
- `shared/private/reduction`
- `parallel for`

OpenMP - `parallel sections`

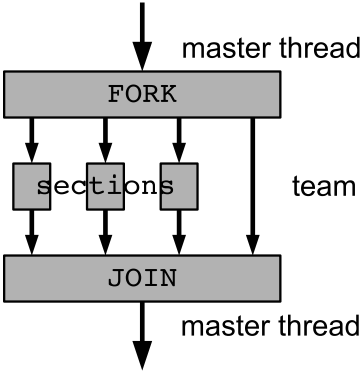
`sections` are a non-iterative work-sharing construct, they specify a number of different sections of code that can be divided between threads in the team.

Each `section` is executed once by a thread in the team

Depending on the number of threads/sections, the same thread may execute multiple sections.

OpenMP - `parallel sections`

`sections` construct



OpenMP - parallel sections

```
int main(int argc, char *argv[]) {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      std::cout << "A: " << omp_get_thread_num() << std::endl;
    }
    #pragma omp section
    {
      std::cout << "B: " << omp_get_thread_num() << std::endl;
    }
    #pragma omp section
    {
      std::cout << "C: " << omp_get_thread_num() << std::endl;
    }
  }
}
```

Output (maybe):

```
B: 1
A: 0
C: 2
```

OpenMP - `parallel single`

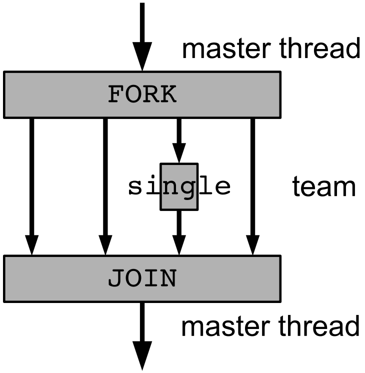
`single` is a construct that will be executed by a single thread from the team.

The `single` construct has an implicit barrier at the end of the region.

The other threads that do not execute the `single` region will wait at the end.

OpenMP - `parallel single`

`single` construct



OpenMP - parallel single

```
int main(int argc, char *argv[]) {
   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num();
      std::cout << "parallel " << tid << std::endl;
      #pragma omp single
      {
         std::cout << "single " << tid << std::endl;
      }
   }
   std::cout << std::endl;
}
```

Output (maybe):

```
parallel parallel 2parallel parallel 03



1
single 2
```

OpenMP - Synchronisation

As with any shared-memory multithreading library, OpenMP needs to provide some way to protect against race conditions.

With explicit threading library we used mutexes and semaphores to protect critical sections.

OpenMP uses a similar approach to how it describes parallelism in a program.

OpenMP - `critical`

The `critical` directive can be used to instruct the compiler that the following region of code should not be executed by multiple threads at the same time.

This is similar to the way that OpenMP allows us to express that certain pieces of code may be executed in parallel without defining the low-level details

We can designate a piece of code as a critical section without specifying the exact mechanism for protecting it.

OpenMP - critical

```
int main(int argc, char *argv[]) {
   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num();

      #pragma omp critical
      {
         std::cout << tid << ": ";
         for(int i = 0; i < 3; ++i) {
            std::cout << i << " ";
         }
         std::cout << std::endl;
      }
   }
}
```

Output (maybe):

```
3:  0 1 2
2:  0 1 2
0:  0 1 2
1:  0 1 2
```

OpenMP - `critical`

`critical` directives may include a name to identify them, this allows separate critical sections to be defined that may execute at the same time as each other.

Unnamed `critical` sections have limited value in large programs as essentially they define one global lock for every critical section.

Named `critical` directives allow different critical sections to be defined that protect separate pieces of shared data etc.

OpenMP - critical

```
int main(int argc, char *argv[]) {
    std::string str;
    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();
        #pragma omp critical(a)
        {
            std::cout << tid << " running." << std::endl;
        }
        #pragma omp critical(b)
        {
            str += std::to_string(tid) + " ";
        }
    }
    std::cout << str << std::endl;
}
```

Output (maybe):

```
0 running.
2 running.
3 running.
1 running.
0 2 3 1
```

OpenMP - `atomic`

Another construct OpenMP provides is the `atomic` construct which ensures that a particular memory location is accessed atomically.

Essentially `atomic` is a restricted, light-weight version of `critical` that can be used for updating a shared variable.

`atomic` is limited to a single statement updating (or reading) from a memory location.

OpenMP - `atomic`

`atomic` is restricted to a single statement of the following form (depending on version).

- `v = x` (read)
- `x = expr` (write)
- `x++`
- `++x`
- `x--`
- `--x`
- `x <binop>= expr`
- `x = x <binop> expr`
- `x = expr <binop> x`

OpenMP - atomic

```
int main(int argc, char *argv[]) {
   int total = 0;
   int subtotal = 0;
   #pragma omp parallel shared(total), private(subtotal)
   {
      subtotal = 0;
      #pragma omp for
      for(int i = 0; i < 1000; i++) {
         subtotal += i;
      }

      #pragma omp atomic
      total += subtotal;
   }

   std::cout << "Total: " << total << std::endl;
}
```

Output:

```
Total: 499500
```

OpenMP - Locks

Critical sections of code can be protected using `critical` directives and by using different names, different critical sections may execute at the same time.

However, the names for the critical sections are set during compilation and are not always sufficient for all use-cases.

The most common case of this is when we want to protect a data item rather than a block of code.

OpenMP - Locks

For example, consider a collection of different data structures.

Different threads may safely add or remove items from different data structures at the same time.

They can safely execute the same section of code provided the data they operate on is different.

OpenMP - Locks

For this reason, OpenMP also provides ***locks*** and associated functions in its library.

These locks can be used by the programmer to enforce mutual exclusion for critical sections.

The functions for simple locks are:

```
void omp_init_lock   (omp_lock_t* lock_p);
void omp_set_lock    (omp_lock_t* lock_p);
void omp_unset_lock  (omp_lock_t* lock_p);
void omp_destroy_lock(omp_lock_t* lock_p);
```

OpenMP - Locks

Given that there are three different methods for providing mutual exclusion, which one should you use?

In general, `atomic` provides the potential to be the fastest (but obviously for limited use-cases).

However the OpenMP specification does allow for mutual exclusion to be enforced across all `atomic` directives, even if they operate on entirely different memory locations.

Exact behaviour may depend on the implementation used.

OpenMP - Locks

Both named and unnamed `critical` directives are easy to use and for many implementations there is a not a significant performance difference between using `critical` and locks.

`critical` directives are more in keeping with the OpenMP approach than using locks and in general should be used when applicable.

As we have seen though, there are cases where locks are necessary for protecting data rather than code sections.

OpenMP - Mutual Exclusion

There are some restrictions on mutual exclusion in OpenMP

- Don't mix different methods of mutual exclusion for the same critical section.
- OpenMP does not guarantee ***fairness*** in mutual exclusion.
- Mutual exclusion directives can be nested but should be used carefully as it may end up in deadlock.

OpenMP - `barrier`

`barrier` is a construct that can be used to synchronise all the threads in a team within a `parallel` region of code.

This construct provides a way to synchronise threads without needing to end the `parallel` region.

It can often be helpful when threads may have some intermediate values they wish to keep after synchronisation.

OpenMP - `barrier`

```
int main(int argc, char *argv[]) {
   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num();
      #pragma omp critical
      std::cout << tid << " started." << std::endl;

      #pragma omp barrier
      #pragma omp critical
      std::cout << tid << " finished." << std::endl;
   }
}
```

Output:

```
0 started.
2 started.
1 started.
3 started.
0 finished.
3 finished.
2 finished.
1 finished.
```

Example - Game of Life

Now we can look at how to implement our Game of Life example using OpenMP.

We will continue to use the *phase-parallel* method to structure our simulator.

Unlike our C++ implementation we will not need to worry about explicitly creating threads.

Example - Game of Life

Quick reminder of our serial implementation:

```
unsigned char *buffer[2];
buffer[0] = new unsigned char[N*N];
buffer[1] = new unsigned char[N*N];

int r = 0; // Read/Write Index
int w = 1;

for(int k = 0; k < N*N; ++k) {   // Initialise
   buffer[r][k] = rand() % 2;
}

for(int ig = 0; ig < G; ++ig) {          // For each Generation
   for(int iy = 0; iy < N; ++iy) {       // For each Row
      for(int ix = 0; ix < N; ++ix) {    // For each Column
         // Count neighbours
         int count = count_neighbours(buffer[r], ix, iy, N);

         // Update
         buffer[w][iy*N + ix] = gol_update(buffer[r][iy*N+ix], count);
      }
   }

   // Swap buffers
   r = !r;
   w = !w;
}
```

Example - Game of Life

To implement a phase-parallel update for this program we need to determine which section of code should be marked as parallel and which work-sharing directive to use.

We will target the same loop for parallelisation as we did in the C++ threads implementation:

```
for(int iy = 0; iy < N; ++iy) {
```

Parallelising this loop will divide the rows between different threads.

Example - Game of Life

```
unsigned char *buffer[2];
buffer[0] = new unsigned char[N*N];
buffer[1] = new unsigned char[N*N];

int r = 0; // Read/Write Index
int w = 1;

for(int k = 0; k < N*N; ++k) {   // Initialise
   buffer[r][k] = rand() % 2;
}

for(int ig = 0; ig < G; ++ig) {          // For each Generation
   #pragma omp parallel for
   for(int iy = 0; iy < N; ++iy) {       // For each Row
      for(int ix = 0; ix < N; ++ix) {    // For each Column
         // Count neighbours
         int count = count_neighbours(buffer[r], ix, iy, N);

         // Update
         buffer[w][iy*N + ix] = gol_update(buffer[r][iy*N+ix], count);
      }
   }

   // Swap buffers
   r = !r;
   w = !w;
}
```

Example - Game of Life

With the addition of a single directive:

```
#pragma omp parallel for
```

We have a parallel implementation of the Game of Life simulation.

The threads will implicitly synchronise at the end of the parallel for loop.

Example - Game of Life

There are some possibilities for tuning this work-sharing construct, specifically we can change the schedule used to divide the iterations.

For a system size of 2048^2 and 100 generations, the different schedules give the following approximate results.

schedule	time (seconds)
`static(128)`	$\approx$ 0.330
`static(64)`	$\approx$ 0.325
`static(32)`	$\approx$ 0.366
`dynamic(32)`	$\approx$ 0.305
`dynamic(16)`	$\approx$ 0.297
`dynamic(8)`	$\approx$ 0.293
`guided`	$\approx$ 0.291

OpenMP Limitations

As shown in the Game of Life example, OpenMP can provide a very simple API to convert a program to a multi-threaded implementation.

Many applications (or sections of them) can be parallelised in this simple manner.

There are some limitations with the constructs we have seen so far.

OpenMP `for`

The `parallel for` directive can be used very effectively to parallel `for` loops in a serial program.

However, the directive is limited to just that - `for` loops. Not `while` loops (or `do-while` loops).

More specifically the `parallel for` directive can only be used to parallelise `for` loops where the number of iterations can be determined.

OpenMP `for`

The `for` loops must have the following form:

$$\texttt{for}\left(\begin{array}{ccccc} & & & \texttt{index}++ \\ & & & ++\texttt{index} \\ & & \texttt{index} < \texttt{end} & & \texttt{index}-- \\ & & \texttt{index} <= \texttt{end} & & --\texttt{index} \\ \texttt{index} = \texttt{start} & ; & \texttt{index} >= \texttt{end} & ; & \texttt{index}+=\texttt{incr} \\ & & \texttt{index} > \texttt{end} & & \texttt{index}-=\texttt{incr} \\ & & & & \texttt{index} = \texttt{index} + \texttt{incr} \\ & & & & \texttt{index} = \texttt{incr} + \texttt{index} \\ & & & & \texttt{index} = \texttt{index} - \texttt{incr} \end{array}\right)$$

- The `index` variable must be an integer or pointer type.
- `start`, `end` and `incr` must be compatible type.
- `start`, `end` and `incr` must not change during the loop.
- The `index` variable can only be modified by the increment expression.

OpenMP Nesting Directives

There are many restrictions on what type of directives / regions may be nested inside one another.

For example - a work-sharing region may not be nested inside another work-sharing region, `critical`, `master` etc.

Compiling OpenMP

Compiling OpenMP programs is not always straightforward, especially when a compiler may just ignore `#pragma` directives and compile a sequential program.

First you must have a compiler that supports OpenMP. You must also include the `omp` library and find the appropriate flags to enable OpenMP.

compiler	flags
`gcc`	`-fopenmp`
`clang`	`-fopenmp`
`icc`	`-openmp`

Compiling OpenMP

Depending on your compiler and system, some other flags may be necessary.

For example, on my machine I am using macOS with the Clang compiler and macports to manage packages.

The compiler flags I use to compiler an OpenMP program are:

```
g++ -Xpreprocessor -fopenmp -I/opt/local/include/libomp/ -L/opt/local/lib/libomp/ -lomp -O3
<source>.cpp -o <executable>
```

OpenMP Implementations

There are several parts of the OpenMP specification that are left open to avoid overly restricting implementations.

One risk of this approach is that developers produce code that relies on a certain implementation or behaviour that is not required by the specification.

This code may or may not work correctly on a different OpenMP implementation.

OpenMP Specification

For more information on OpenMP and additional constructs / directives / library functions see:

`https://www.openmp.org/specifications/`

Summary

- `sections` / `section`
- `single`
- `critical`
- `atomic`
- locks
- `barrier`

Note: OpenMP has a number of other features/restrictions we haven't fully discussed.