159.341 Programming Languages, Algorithms & Concurrency

OpenMP - 1

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

OpenMP

OpenMP is designed for shared-memory parallel computing and the computer system is viewed as a collection of cores or processors that all have access to memory.

The intention of OpenMP was to provide an API that allowed shared-memory programs to be developed at a higher-level than an explicit threading library.

It was specifically designed to allow programs to be *incrementally* parallelised.

OpenMP

OpenMP is not an explicit threading library and is not intended to explicitly define the behaviour of each thread.

Instead it allows the programmer to designate certain blocks of code that may be safely executed in parallel with *code directives*.

The exact mapping of tasks to threads is determined by the compiler and run-time system.

OpenMP

In C and C++, the OpenMP directives are preprocessor instructions called `pragmas`.

Pragmas in C/C++ are start with the line:

```
#pragma
```

Pragmas are used by other libraries as well, OpenMP pragmas start with the line:

```
#pragma omp
```

OpenMP

In addition to the *code directives*, OpenMP also provides a library of functions and macros, these can be used by including the OpenMP header file.

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

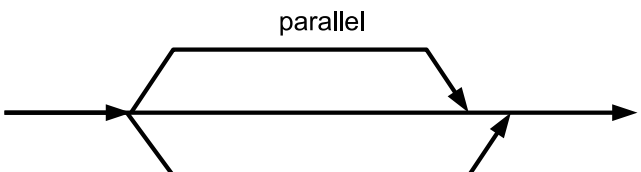
The library contains a number of useful functions that we will use in our examples.

OpenMP - `parallel`

The directive `parallel` signifies a block of code that will be executed by multiple threads in parallel.

This is one of the most fundamental compiler directives in OpenMP.

Number of threads determined by the system (usually matches the number of cores).



OpenMP - parallel

Example 0:

```
#include <omp.h>

int main() {
   #pragma omp parallel
   {
      std::cout << omp_get_thread_num() << " ";
   }
   std::cout << std::endl;
}
```

Output (maybe):

```
094 111 1312 15 5 28146 10 3 7
```

*`omp_get_thread_num()` is an OpenMP library function that returns the id of the executing thread.

OpenMP - `parallel`

In OpenMP terminology, the threads that execute the parallel section of code is a ***team*** consisting of the original thread ***master*** and additional threads ***slaves***.

At the end of the block of code there is an implicit ***barrirer***.

All the threads in the ***team*** must reach the end of the block before the ***master*** thread continues its execution.

OpenMP - `parallel`

The directive `parallel` has a set of parameters that can be used to control whether variables are shared between threads or whether each thread has a separate copy.

- `if`
- `num_threads`
- `private`
- `shared`

OpenMP - `parallel if`

The parameter `if` evaluates condition clause and determines whether it evaluates to true or not.

The clause evaluates to true then the parallel region will be executed using a team of threads.

If not the region will be executed in serial by the master thread.

OpenMP - parallel if

Example 1:

```
#include <omp.h>

int main(int argc, char *argv[]) {
   for(int i = 0; i < 2; ++i) {
      #pragma omp parallel if(i > 0)
      {
         std::cout << omp_get_thread_num() << " ";
      }
      std::cout << std::endl;
   }
}
```

Output (maybe):

```
0
1215 0314 8 4216 5 7 9 13 10 11
```

OpenMP - `parallel num_threads`

The parameter `num_threads` allows the programmer to define how many threads should be in the team executing a parallel region of code.

OpenMP does not guarantee that exactly this many threads will be used to execute the region.

However, since most systems support hundreds or thousands of threads we will almost always get the requested number.

OpenMP - parallel num_threads

Example 2:

```
#include <omp.h>
#include <iostream>
#include <cmath>

int main(int argc, char *argv[]) {
   for(int i = 0; i < 3; i++) {
      #pragma omp parallel num_threads((int)pow(2, i+1))
      {
         std::cout << omp_get_thread_num() << " ";
      }
      std::cout << std::endl;
   }
}
```

Output (maybe):

```
0 1
3 0 2 1
0 1 3 2 7 4 6 5
```

OpenMP - Data Scope Attribute Clauses

Because OpenMP is based on a shared-memory programming model, most variables in a parallel section are shared by default.

Some variables will be private to each thread by default - loop index variables and local variables in functions called from a parallel region.

Data scope attributes allow the programmer to define how other variables should be treated.

OpenMP - Data Scope Attribute Clauses

Some of the data scope attributes are:

- `private (list)`
  variables are private to each thread
- `shared (list)`
  variables are shared between all threads in team
- `default`
  define a default scope for all other variables
- `reduction`
  performs a reduction operation on variables in list

OpenMP - `parallel private`

The data scope attribute `private` takes a list of variables, their behaviour is as follows:

- A new object of the same type is declared for each thread in the team
- All references to the original object are replaced by references to the new object
- Each new object should be assumed to be uninitialised for each thread.

OpenMP - parallel private

Example 3:

```
int main(int argc, char *argv[]) {
   int i = 0;
   #pragma omp parallel private (i)
   {
      std::cout << i << " ";
      i = omp_get_thread_num() + 1;
   }
   std::cout << std::endl << "After: " << i << std::endl;
}
```

Output (maybe):

```
327663276632766 3276632766 327663276632766 03276632766 3276632766 32766 32766 32766
After:  0
```

OpenMP - parallel shared

Example 4:

```
int main(int argc, char *argv[]) {
   int i = 0;
   #pragma omp parallel shared(i)
   {
      i = i + 1;
      std::cout << i << " ";
   }
   std::cout << std::endl << "After: " << i << std::endl;
}
```

Output (maybe):

```
31 45 6798 1210 13 14 42 8 11
After:  14
```

OpenMP - `parallel default`

The `default` data scope attribute set the default scope used for any variables not explicitly scoped.

C/C++ specification only officially supports `shared` and `none` for `default` but implementations may support `private` and others.

The `none` default scope specifies that all variables within the parallel region must be explicitly scoped.

OpenMP - `parallel reduction`

The `reduction` clause performs a reduction operation on the variables that appear in the list.

A private copy of each variable is created and initialised in each thread.

At the end of the region, the reduction operation is used to combine all the private copies and the result is written to the original variable.

OpenMP - parallel reduction

There are some restrictions on the valid reduction operators that can be used with this clause.

Operation	C/C++	Initialisation
addition	+	0
multiplication	*	1
subtraction	-	0
logical AND	&&	true / 1
logical OR	||	false / 0
bitwise AND	&	all bits 1
bitwise OR	|	0
exclusive bitwise OR	^	0
maximum	max	MIN_VALUE
minimum	min	MAX_VALUE

OpenMP - parallel reduction

Example 5:

```
int main(int argc, char *argv[]) {
   int total = 0;
   #pragma omp parallel reduction(+:total)
   {
      total += omp_get_thread_num();
   }

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

Output (maybe):

```
120
```

OpenMP - Data Scope Attribute Clauses

There are two variations on the `private` data scope attribute.

- `firstprivate`
  Variables are `private` but initialised with value of original object

- `lastprivate`
  Variables are `private`, last object is copied back to original object.

OpenMP - Work-Sharing

To divide work between threads in a parallel region, OpenMP provides a set of *work-sharing* directives.

These constructs do not launch any new threads, they simply denote how blocks of code should be divided between threads.

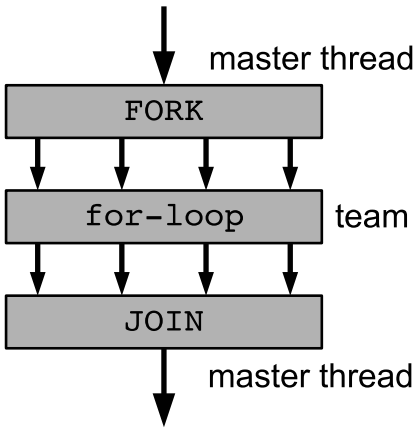
These constructs are:

- `do/for`
- `sections`
- `single`

OpenMP - do/for

The do/for work-sharing construct specifies that the iterations of the following loop should be executed in parallel.

It is assumed that this directive will already be inside a parallel region of code, otherwise the loop will be executed by a single-thread.



OpenMP - parallel for

Example 6:

```
int main() {
   const int N = 64;
   char a[N];
   #pragma omp parallel
   {
      int tid = omp_get_thread_num();
      #pragma omp for
      for(int i = 0; i < N; ++i) {
         a[i] = hexadecimal(tid);
      }
   }
   for(int i = 0; i < N; ++i) { std::cout << a[i]; }
   std::cout << std::endl;
}
```

Output (maybe):

```
0000111122223333444455556666777788889999aaaabbbbccccddddeeeeffff
```

OpenMP - `parallel for`

The `do/for` construct has a number of parameters that may be used to help direct how the work is split between threads.

One of the most significant is the `schedule` parameter which describes how the iterations of the loop are divided.

- `static`
- `dynamic`
- `guided`
- `runtime`
- `auto`

OpenMP - `parallel for`

`schedule(static, chunk)`

The `static` scheduling method will divide the iterations of the loop into pieces of size `chunk` and will be assigned statically to each thread.

If the chunk size is not specified then the loop will be dividing as evenly as possible between the threads.

OpenMP - `parallel for`

`schedule(dynamic, chunk)`

The iterations of the look are divided into sections of size `chunk` (default is `1`) and assigned dynamically to threads.

Once a thread finishes processing it will by dynamically assigned another (until all the iterations are completed).

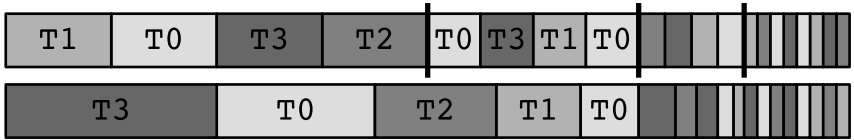
OpenMP - `parallel for`

`schedule(guided, min_chunk)`

Similar to the dynamic scheduling except that the chunk size decreases each time a section of work is assigned to a thread.

The chunk size assigned is proportional to `num_iterations_remaining / num_of_threads`.

May be implemented by the compiler in one of two ways:



OpenMP - `parallel for`

- `schedule(runtime)`

Delegates the scheduling decisions to the runtime environment.

`schedule(auto)`

Delegates the scheduling decisions to the compiler and/or the runtime.

OpenMP - parallel for

Example 6:

```
#pragma omp for schedule(static, 2)
```

```
00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
```

```
#pragma omp for schedule(static, 4)
```

```
000011112222333344445555666677778888999aaaabbbbccccddddeeeeffff
```

```
#pragma omp for schedule(dynamic, 4)
```

```
22220000777722220000777744441111222200007777222233334444111110000
```

```
#pragma omp for schedule(guided)
```

```
cc270fcc270702f1c2707f1c21f2072270d8cb481f270cdcc2c2af12a8df1b47
```

Summary

- OpenMP Code Directives
- `parallel`
- `private`
- `shared`
- `reduction`
- `for/do`