Chapter 3: Multithreading with OpenMP

More useful features: thread numbers and tasks

In the header `#include <omp.h>` you can find the following functions. This function is useful outside a `parallel` region:

- `omp_get_max_threads()` — Returns the number of threads that OpenMP will use in `parallel` regions by default.

These functions are useful inside a `parallel` region:

- `omp_get_num_threads()` — Returns the number of threads that OpenMP is using in this `parallel` region.
- `omp_get_thread_num()` — Returns the identifier of this thread; threads are numbered 0, 1, ...

Here is a simple example of the use of these functions:

```
a();
#pragma omp parallel
{
    int i = omp_get_thread_num();
    int j = omp_get_num_threads();
    c(i,j);
}
z();
```

Do-it-yourself parallel for

The above functions are enough to implement, for example, parallel for loops! Here is an example:

```
a();
#pragma omp parallel
{
    int a = omp_get_thread_num();
    int b = omp_get_num_threads();
    for (int i = a; i < 10; i += b) {
        c(i);
    }
}
z();
```

This is, in essence, equivalent to the following parallel for loop:

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```

Controlling the number of threads

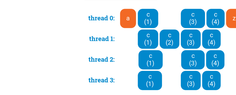
If needed, you can also set the number of threads explicitly.

```
a();
#pragma omp parallel num_threads(3)
{
    int i = omp_get_thread_num();
    int j = omp_get_num_threads();
    c(i,j);
}
z();
```

Single thread only

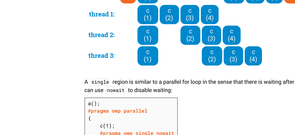
Inside a `parallel` region, you can use the `single` directive to indicate that certain parts should be executed by only one thread:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp single
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



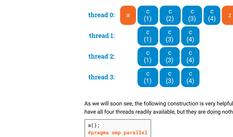
Compare this with a critical section, which is executed by all threads:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp critical
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



A `single` region is similar to a parallel for loop in the sense that there is waiting after it (but not before). You can use `nowait` to disable waiting:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp single nowait
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



As we will soon see, the following construction is very helpful even if it may seem a bit pointless at first. We will have all four threads readily available, but they are doing nothing at the moment.

```
a();
#pragma omp parallel
#pragma omp single
{
    c(1);
}
z();
```


Tasks

Now that we have multiple threads waiting for work to do, we can use the `task` primitive to tell that some part of the code can be executed by another thread. Note that here we create two tasks and hence we will have three threads doing work: the current thread will also continue to do whatever comes next in the program.

```
a();
#pragma omp parallel
#pragma omp single
{
    c(1);
    #pragma omp task
    c(2);
    #pragma omp task
    c(3);
    c(4);
    c(5);
}
z();
```


In general, OpenMP will do the right thing also with a large number of tasks. For example, here tasks `c(2)`, `c(3)`, and `c(4)` get started immediately as there were threads available, while tasks `c(5)` and `c(6)` will wait in the queue until some threads become available.

```
a();
#pragma omp parallel
#pragma omp single
{
    c(1);
    #pragma omp task
    c(2);
    #pragma omp task
    c(3);
    #pragma omp task
    c(4);
    #pragma omp task
    c(5);
    #pragma omp task
    c(6);
    c(7);
}
z();
```