Getting started

OpenMP provides what's known as a "directives-based" shared-memory API. In C and C++, this means that there are special preprocessor instructions known as pragmas.

- Pragmas are typically added to a system to allow behaviors that aren't part of the basic C specification.
  - Compilers that don't support the pragmas are free to ignore them.
- This allows a program that uses the pragmas to run on platforms that don't support them.
  - So, in principle, if you have a carefully written OpenMP program, it can be compiled and run on any system with a C compiler, regardless of whether the compiler supports OpenMP.
- Essentially not all compilers support pragmas but those that do can utilize them to be more flexible and do things it wouldn't normally od
- If OpenMP is not supported, then the directives are simply ignored and the code will execute sequentially.
- Pragmas in C and C++ start with ..
  - # pragma
- Pragmas (like all preprocessor directives) are, by default, one line in length, so if a pragma won't fit on a single line, the newline needs to be "escaped"—that is, preceded by a backslash \
- The details of what follows the #pragma depend entirely on which extensions are being used.

Example of simple openMP program

Let's take a look at a *very* simple example, a "hello, world" program that uses OpenMP. (See Program 5.1.)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void);  /* Thread function */
6
7  int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 #  pragma omp parallel num_threads(thread_count)
12    Hello();
13
14    return 0;
15 }  /* main */
16
17 void Hello(void) {
18    int my_rank = omp_get_thread_num();
19    int thread_count = omp_get_num_threads();
20
21    printf("Hello from thread %d of %d\n",
22           my_rank, thread_count);
23
24 }  /* Hello */
```

Program 5.1: A "hello, world" program that uses OpenMP.

Note: notice how the pragma is used and how syntactically it works and how it is formatted

How to compile and execute using the example above:

- To compile:

  To compile this with `gcc` we need to include the `-fopenmp` option[1]:

  ```
  $ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
  ```

- To execute:

  ```
  $ ./omp_hello 4
  ```

  - Notice how the 4 is inputted here because of the way our program is written which requires this input
  - Line 9 in example above

If we do this, the output might be

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

However, it should be noted that the threads are competing for access to `stdout`, so there's no guarantee that the output will appear in thread-rank order. For example, the output might also be

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

or

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

or any other permutation of the thread ranks.

If we want to run the program with just one thread, we can type

```
$ ./omp_hello 1
```

and we would get the output

```
Hello from thread 0 of 1
```

The program

- Let's take a look at the source code. In addition to a collection of directives, OpenMP consists of a library of functions and macros, so we usually need to include a header file with prototypes and macro definitions.
  - The openMP header file is omp.h
- In our Pthreads programs, we specified the number of threads on the command line. We'll also usually do this with our OpenMP programs.
  - we therefore use the strtol function from stdlib.h to get the number of threads.

```
long strtol(
      const char*  number_p     /* in  */,
      char**       end_p        /* out */,
      int          base         /* in  */);
```

    - The first argument is a string—in our example, it's the command-line argument, a string—and the last argument is the numeric base in which the string is represented— in our example, it's base 10. We won't make use of the second argument, so we'll just pass in a NULL pointer. The return value is the command-line argument converted to a C long int
- When we start the program from the command line, the operating system starts a single-threaded process, and the process executes the code in the main function.
- However, things get interesting in Line 11. This is our first OpenMP directive, and we're using it to specify that the program should start some threads. Each thread should execute the Hello function, and when the threads return from the call to Hello, they should be terminated, and the process should then terminate when it executes the return statement

  - That's a lot of bang for the buck (or code). If you studied the Pthreads chapter, you'll recall that we had to write a lot of code to achieve something similar: we needed to allocate storage for a special struct for each thread, we used a for loop to start all the threads, and we used another for loop to terminate the threads
    - Thus it's immediately evident that OpenMP provides a higher-level abstraction than Pthreads provides

We've already seen that `pragma`s in C and C++ start with

```
#  pragma
```

  - OpenMP `pragma`s always begin with

```
#  pragma omp
```

    - Our first directive is a parallel directive, and, as you might have guessed, it specifies that the

structured block of code that follows should be executed by multiple threads.

- A structured block is a C statement or a compound C statement with one point of entry and one point of exit, although calls to the function exit are allowed.
- This definition simply prohibits code that branches into or out of the middle of the structured block.

At its most basic the `parallel` directive is simply

```
# pragma omp parallel
```

- As we noted earlier, we'll usually specify the number of threads on the command line, so we'll modify our parallel directives with the num_threads clause
  - A clause in OpenMP is just some text that modifies a directive.
  - The num_threads clause can be added to a parallel directive. It allows the programmer to specify the number of threads that should execute the following block:

```
# pragma omp parallel num_threads(thread_count)
```

  - It should be noted that there may be system-defined limitations on the number of threads that a program can start. The OpenMP Standard doesn't guarantee that this will actually start `thread_count` threads. However, most current systems can start hundreds or even thousands of threads, so unless we're trying to start *a lot* of threads, we will almost always get the desired number of threads.

What actually happens when the program gets to the `parallel` directive? Prior to the `parallel` directive, the program is using a single thread, the process started when the program started execution. When the program reaches the `parallel` directive, the original thread continues executing and `thread_count` − 1 additional threads are started. In OpenMP parlance, the collection of threads executing the `parallel` block—the original thread and the new threads—is called a **team**. OpenMP thread terminology includes the following:

- **master**: the first thread of execution, or *thread 0*.
- **parent**: thread that encountered a `parallel` directive and started a team of threads. In many cases, the parent is also the master thread.
- **child**: each thread started by the parent is considered a *child* thread.

Each thread in the team executes the block following the directive, so in our example, each thread calls the `Hello` function.

- When the block of code is completed—in our example, when the threads return from the call to Hello—there's an implicit barrier.
  - This means that a thread that has completed the block of code will wait for all the other threads in the team to complete the block—in our example, a thread that has completed the call to Hello will wait for all the other threads in the team to return.
  - When all the threads have completed the block, the child threads will terminate and the parent thread will continue executing the code that follows the block.

Since each thread has its own stack, a thread executing the `Hello` function will create its own private, local variables in the function. In our example, when the function is called, each thread will get its rank or ID and the number of threads in the team by calling the OpenMP functions `omp_get_thread_num` and `omp_get_num_threads`, respectively. The rank or ID of a thread is an **int** that is in the range $0, 1, \ldots,$ `thread_count` $- 1$. The syntax for these functions is

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

Since `stdout` is shared among the threads, each thread can execute the `printf` statement, printing its rank and the number of threads.

As we noted earlier, there is no scheduling of access to `stdout`, so the actual order in which the threads print their results is nondeterministic.

- Omp_get_thread_num is used for knowing what rank a thread is
- Omp_get_num_threads is used for knowing how many total threads there are in the team
  - Used in the print statement "_ out of [# of threads] threads"

Error checking

To make the code more compact and more readable, our program doesn't do any error checking. Of course, this is dangerous, and, in practice, it's a *very* good idea—one might even say mandatory—to try to anticipate errors and check for them. In this example, we should definitely check for the presence of a command-line argument, and, if there is one, after the call to `strtol`, we should check that the value is positive. We might also check that the number of threads actually created by the `parallel` directive is the same as `thread_count`, but in this simple example, this isn't crucial.

A second source of potential problems is the compiler. If the compiler doesn't support OpenMP, it will just ignore the `parallel` directive. However, the attempt to include `omp.h` and the calls to `omp_get_thread_num` and `omp_get_num_threads` *will* cause errors. To handle these problems, we can check whether the preprocessor macro `_OPENMP` is defined. If this is defined, we can include `omp.h` and make the calls to the OpenMP functions. We might make the modifications that follow to our program.

Instead of simply including `omp.h`:

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

we can check for the definition of `_OPENMP` before trying to include it:

```
#ifdef _OPENMP
#  include <omp.h>
#endif
```

Also, instead of just calling the OpenMP functions, we can first check whether `_OPENMP` is defined:

```
#  ifdef _OPENMP
      int my_rank = omp_get_thread_num();
      int thread_count = omp_get_num_threads();
#  else
      int my_rank = 0;
      int thread_count = 1;
#  endif
```

Here, if OpenMP isn't available, we assume that the `Hello` function will be single-threaded. Thus the single thread's rank will be 0, and the number of threads will be 1.

The book's website contains the source for a version of this program that makes these checks. To make our code as clear as possible, we'll usually show little, if any, error checking in the code displayed in the text. We'll also assume that OpenMP is available and supported by the compiler.