1. GETTING STARTED: Example OpenMP Program

- OpenMP was explicitly designed to allow programmers to incrementally parallelize existing serial programs.
- OpenMP provides 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.

  Pragmas in C and C++ start with
  #pragma.

- Pragmas (like all preprocessor directives) are, 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 n.

- Simple example, a “hello, world” program

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

void Hello(void); /* Thread function */

intmain(intargc, char_ argv[]) {
/* Get number of threads from command line */
int thread count = strtol(argv[1], NULL, 10);

# pragma omp parallel numthreads(thread count)
Hello();

return 0;
}/* main*/

void Hello(void) {
int my rank = omp get thread num();
int thread count = omp get numthreads();

printf("Hello from thread %d of %dnn", my rank, thread count);

} /* Hello */
```

Program 1: A “hello,world” program that uses OpenMP

2. Compiling and running OpenMP programs:

- To compile this with gcc we need to include the -*fopenmp option*

```
$ gcc  −g  −Wall  −fopenmp  −o  omp_helloomp_hello . c
```

- To run the program, we specify the number of threads on the command line. Forexample, we might run the program with four threads and type

```
$ . /  omp_hello  4
```

- 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.

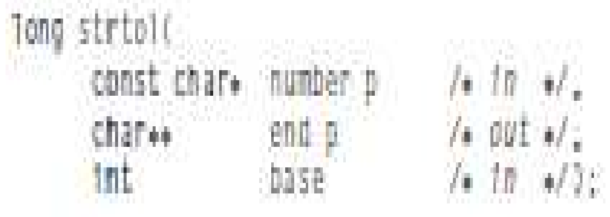
- If we want to run the program with just one thread, we can type

```
$ ./omp hello 1
```

and we would get the output

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

- 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. The OpenMP header file is *omp.h*, and we include it in Line 3.
- we need to specify the number of threads on the command line. In Line 9 we use the *strtol* function from *stdlib.h*to get the number of threads. The syntax of this function is:



```
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— 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 passin a *NULL*pointer.

- 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.
- In Line 11the first OpenMP directive we're using to specify that the program should start some threads.
- Each thread that's forked 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.
- We needed to allocate storage for a special structfor each thread, we used a ***for loop*** to start each thread, and we used another ***for loop*** to terminate the threads.   OpenMP is higher-level than Pthreads.
- We've already seen that pragmas in C and C++ start with

  # pragma
- OpenMP pragmas always begin with

  # pragmaomp
- The first directive is a *parallel* directive 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.
- Thread is a sequence of statements executed by a program. Threads are typically started or ***forked*** by a ***process***, and they share most of the resources of the process that starts them—for example, access to ***stdin*** and ***stdout***—but each thread has its own stack and program counter. When a thread completes execution it joins the process that started it. This terminology comes from diagrams that show threads as directed lines.
- the parallel directive is simply
  # pragmaomp parallel

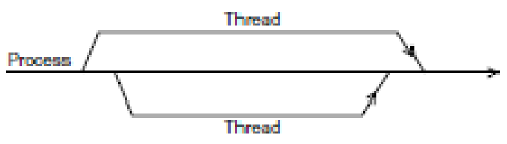


Fig: A process forking and joining two threads

- We'll 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:

  # pragmaomp parallel num threads(thread count)

- 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.

- 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***, the original threadis called the ***master***, and the additional threads are called ***slaves***.
- 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 hascompleted the block of code will wait for all the other threads in the team to completethe block—in our example, a thread that has completed the call to Hello will wait forall the other threads in the team to return.
- When all the threads have completed the block, the slave threads will terminate and the master thread will continue executing the code that follows the block. In our example, the master thread will execute the return statement in Line 14, and the program will terminate.
- 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 functionis called, each thread will get its rank or id and the number of threads in the teamby 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, : : : ,thread count -1. The syntax for these functions is

```
intomp- get -thread -num(void);
intomp-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.

3. Error checking

- 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 supportOpenMP, 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 thecalls to the OpenMP functions. The following modifications to ourprogram.
Instead of simply including ***omp.h*** in the line

```
#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
```

- 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 one.