HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
LAB 10: OPENMP (PART I)

This lab gives an introduction to shared-memory parallel programming using OpenMP.

The following aspects are included in the lab:

(1) Creating threads using the `omp parallel` directive

(2) Controlling the number of threads used in an OpenMP program

(3) Providing different input data to threads in an `omp parallel` code section

(4) Dividing work between threads

(5) Checking what speedup we get when varying the number of threads used

(6) Creating a portable code, that can be compiled even if OpenMP is not supported by the compiler used

(7) Using the `omp critical` directive to achieve safe access to shared data

(8) Nested parallelism

Documentation: There is lots of useful information on the official OpenMP web page, here:

`http://openmp.org/`

In particular, there is a page with complete specifications for different versions of the OpenMP API:

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

It may seem confusing that there are several versions there, but don't worry; the newer versions generally include the functionality from previous versions, it is just that new things have been added. The OpenMP features we use in this course have been around at least since version 3.1, which includes the most important features and is supported by most compilers. A nice 2-page overview of the main OpenMP functionality can be found on the "Summary Card" for version 3.1, here:

`http://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf`

Note however that you definitely do not need to know all of that in order to use OpenMP. As we will see in this lab, you can do a lot with just a few OpenMP directives and function calls.

Documentation for the OpenMP functionality in gcc can be found here:

Date: November 30, 2021.

https://gcc.gnu.org/onlinedocs/libgomp/

Note about compiler optimization flags: As you know, our end goal in this course is to achieve the best performance we can. However, some of the example codes in this lab are so simple that the compiler may just optimize away everything if we turn on compiler optimizations. Therefore, for the purposes of this lab, it is OK to compile without compiler optimization. Just remember that in real cases, and in assignments in this course, you should aim for the best possible performance and then of course use both compiler optimization flags and your own code optimizations and parallelization efforts.

1. Main Tasks

Task 1

The code for this task is in the `Task-1` directory.

Look at the code in `hello_openmp.c`. This code uses the `#pragma omp parallel` OpenMP directive to create threads, and inside the parallel block of code each thread is doing a printf statement.

First compile the code like an ordinary C program, without any compiler options:

```
gcc hello_openmp.c
```

Then run it. What happens? Note that if OpenMP is not enabled, the compiler will ignore the `#pragma omp` line.

Now compile the code using the `-fopenmp` compiler flag:

```
gcc -fopenmp hello_openmp.c
```

Then run it again. Do you get several lines of output now?

The number of threads used is now set using the `num_threads(5)` clause in the `#pragma omp parallel` directive. Try changing the number of threads from 5 to some other number, to verify that we really can set the number of threads in that way.

When you have seen how that works, try removing the `num_threads(5)` clause so that the parallel directive simply looks like this:

```
#pragma omp parallel
```

Then run the code again. How many threads does it use now?

When the number of threads to use has not been explicitly set by the program, the OpenMP runtime system will use a default number of threads that is determined based on, among other things, environment variables.

In Lab 1 we had a brief look at environment variables; we saw for example how the `PATH` environment variable is used. For OpenMP, the environment variable `OMP_NUM_THREADS` can be used to set the default number of threads used by OpenMP programs. To see if this environment variable is set, we can simply use the `echo` command like this:

```
echo $OMP_NUM_THREADS
```

Do that. If it gives you a number, then that means that the `OMP_NUM_THREADS` environment variable is set to that value, and then OpenMP programs should by default use that number of threads.

Now test this by setting `OMP_NUM_THREADS` to a value you want, for example:

```
export OMP_NUM_THREADS=3
```

Then do "`echo $OMP_NUM_THREADS`" again to see that it is really set to the value you wanted.

Then run the program again. Does it work? Note that you do not need to recompile, since the value of the environment variable is checked each time the program runs. You can change the value of the environment variable by simply using the `export` command again, and you can remove it using the `unset` command, like this:

```
unset OMP_NUM_THREADS
```

There is also a OpenMP function called `omp_set_num_threads()` that can be used from within the program to set the number of threads that will be used. Test this: add a call to `omp_set_num_threads()` in the beginning of `main()`, before the `#pragma omp parallel` directive, for example:

```
omp_set_num_threads(6)
```

To get access to function declarations for OpenMP functions like `omp_set_num_threads()` you need to include the `omp.h` header file:

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

Now compile and run again. What happens? Does it use the number of threads you set when calling the `omp_set_num_threads()` function?

Now we have looked at several different ways that the number of threads used in OpenMP programs can be decided:

- Using the `num_threads` clause in the `#pragma omp parallel` directive
- Using the `omp_set_num_threads()` function
- Using the `OMP_NUM_THREADS` environment variable
- Relying on the default used if `OMP_NUM_THREADS` is not set

Those different ways have decreasing priority in the above order. So, for example, if a program calls `omp_set_num_threads()` then that overrides the value of `OMP_NUM_THREADS`. And using the `num_threads` clause in the `#pragma omp parallel` directive in turn overrides the setting from `omp_set_num_threads()`. Try a few different combinations to see that it really works like that.

Task 2

The code for this task is in the `Task-2` directory.

Here you are given a code that uses a `#pragma omp parallel` directive just like in the previous task, but this time a function is called and that function contains a loop with many iterations, so it is doing some work that requires some CPU time.

Start by rewriting the code so that the number of threads to use is given as a command-line input argument to the program. So, instead of

```
#pragma omp parallel num_threads(3)
```

you write

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

where `n` is a variable that you have set based on the input arguments to the program. You can also modify the number of loop iterations in the `do_some_work()` function if you want it to take longer or shorter time to run. Note that the loop variable is declared as "`long int`" instead of just "`int`". Do you understand why? What is the largest value that can be represented by an "`int`"?

Then run it for different numbers of threads and use the function `omp_get_wtime()` to see how much time it uses. Also look at `top` while the program is running.

In this program each thread is doing the same amount of work, so if each thread gets a separate core to run on, and if they do not disturb eachother in any way, then the wall time should be the same for any number of threads. However, when the number of threads is larger than the number of cores available, the wall time should increase. Does it behave this way in your tests? Do you understand what is happening?

Task 3

The code for this task is in the `Task-3` directory.

So far in this lab we have seen how we can create threads with OpenMP, but so far all threads have been doing exactly the same thing. Of course, if our multi-threaded program is to do something meaningful, we typically want different threads to be doing different things, so that we can divide work between them in some way.

In previous labs, when we were using Pthreads, we could give each thread its own input data by giving the threads different pointers when we called `pthread_create()`. Now, with OpenMP, we do not call any function like `pthread_create()` to create each thread; threads are created automatically for us. So how can we get our threads to do different things?

One way of getting our OpenMP threads to do different things is to use the functions `omp_get_thread_num()` and `omp_get_num_threads()`. The `omp_get_thread_num()` function returns the thread ID number for the current thread, and the function `omp_get_num_threads()` returns the active number of threads. There is also function `omp_get_max_threads()`. What is the difference between the functions `omp_get_num_threads()` and `omp_get_max_threads()`?

Modify the code in `thread_func()` in `program.c` so that it calls `omp_get_thread_num()` and `omp_get_num_threads()`, saving those values in some variables and then does printf to print the values. When running the program you should then see that if the total number of threads is n the thread ID number ranges from 0 to $n-1$. Try running the program using different numbers of threads. Does it work as expected?

So, now we have a way of letting the threads do different things: we can let each thread determine what it should do based on its thread ID number. For example, if we have two threads we can decide that thread 0 is going to handle the first half of the work, while thread 1 is going to take care of the second half. Or whatever; of course we can decide to split the work in any way we want.

Task 4

The code for this task is in the `Task-4` directory.

Now we will look at how we can give our threads different input data to work with, and how we can let them store results in different places.

The code `omptest.c` first sets up an array of structs. This is done in the beginning of main(), before the the `omp parallel` directive. Then, that array will be shared; all threads in the parallel block will have access to that same array.

Then, inside the parallel block, the code calls the function we looked at earlier to get the ID number of the current thread, and calls a function that takes a pointer to one of the structs in the array as input.

Look at the code and try to understand what it is doing. Then add some printf statements inside `thread_func()` to print out the contents of the struct that the input argument p points to. Then you should see that each thread gets its part of the input data that was prepared in main(). Does it work?

This code is an example of how we can give each thread a different set of input parameters to work with. Perhaps the values A, B, x, and y in the struct are all input values to the computation that the thread is going to do.

Now let us assume that the computation each thread is doing gives a result that we want the main() function to be able to access after the parallel code block. Add another member in the struct called "result", and let the code in `thread_func()` do some computation with A and B and store the value that gives in the "result" member in the struct. Then, in the main() function after the parallel code block, print the "result" values for `arr[0]` and `arr[1]`. Does it work as expected?

Note that what we have done now is very similar to how a Pthreads program might work, giving each thread a separate pointer to the data that thread should work with. The difference is just that now, with OpenMP, we do not create each thread explicitly but instead we get the thread ID and based on that we decide which pointer to give to the function. An advantage with OpenMP in this case is that we can use the correct pointer type directly, we do not need to cast pointers to/from the `(void*)` type as we did when using Pthreads.

Another advantage compared to Pthreads is that we do not need to call any function like `pthread_join()` for each thread to be sure that all threads have completed their work. With OpenMP that is handled automatically; we know that our program will not continue execution after the parallel block until all threads have completed the parallel block.

Task 5

The code for this task is in the `Task-5` directory.

In this task, we will investigate how the wall time and CPU usage for our program is affected when the work is divided between two threads in different ways.

The program `Ntest.c` does a very simple computation, dividing the work between two threads. The amount of work each thread gets is determined by the `NA` and `NB` constants. If you change the `NA` and `NB` values so that the sum `NA`+`NB` remains the same, the program performs the same work in total but divided in different ways.

Look at the code and make sure you understand how it works. Then run it for different choices of `NA` and `NB`, measure timings using the `omp_get_wtime()` function and monitor the CPU usage using `top`. Does it behave as you expect? Do you understand what is happening?

Do you get the expected speedup of 2 when the work is split equally compared to the case when one thread does all the work?

One thing we can check using this example code is what happens with OpenMP threads that reach the end of the parallel block before other threads. Do those threads still consume CPU resources? You can check this for example by giving one of the threads twice as much work as the other, and then monitor the CPU usage with `top`. Do you see the CPU usage drop when the first thread reaches the end of the parallel block?

Recall that with Pthreads, when a thread has finished its thread function it no longer uses any CPU time. Here we can check if threads reaching the end of an OpenMP parallel block also behave in that way.

Task 6

Here you will use OpenMP to parallelize a simple prime numbers computation, the same problem you worked on with Pthreads in Lab 9.

Start from a serial code that uses the most straightforward algorithm for determining how many prime numbers there are in a given range of numbers, from 1 to M where M is an input argument to your program. Simply use a loop where you naively test how many numbers each number is divisible by. (If you want, copy your existing serial code from Lab 8, Task 4).

Then parallelize your program using OpenMP: divide the work between two threads.

Note that in this kind of prime numbers computation, the work required to check if a given number is a prime number increases, so the parallelization does not work so well if you just give the lower half of the numbers to the first thread and the rest to the other thread. This is an example of a *load balancing* problem. Can you think of a way to improve the load balancing and thereby get a better speedup?

When you have the two-threads version working, modify it so that it uses a variable number of threads. Let the program accept two input arguments: the first argument is M as above, and the second argument is the number of threads to use.

When your program is working, try running it for different numbers of threads and measure the timings. Ideally, you should see a speedup up to the number of cores available on the computer. For example, if there are 4 cores available, the runtime should be reduced to 25 % when using 4 threads. Also try running your program with larger numbers of threads, more than the number of cores available. Do you see any further speedup in that case?

Task 7

Portability

Since not all compilers support OpenMP, it can be desirable to write a code in a way that allows it to work as a serial code if OpenMP is not supported, and as a multi-threaded code if OpenMP is supported.

The `#pragma` directives will anyway be ignored by the compiler if it does not support them, so those lines in the code are no problem. However, the `#include <omp.h>` can be problematic since the compiler may not be aware of any "`omp.h`" file. For example, trying to compile an OpenMP program using the clang compiler version 3.8.0 gives this error message:

```
$ clang program.c
program.c:2:10: fatal error: 'omp.h' file not found
```

A way to fix this is to check for the `_OPENMP` preprocessor macro, like this:

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

Then we only try to include "`omp.h`" if OpenMP is supported. In the same way, we can do our calls to OpenMP library functions like `omp_get_thread_num()` only if OpenMP is supported:

```
#ifdef _OPENMP
    myID = omp_get_thread_num();
#else
    myID = 0;
#endif
```

Do this for your prime numbers computation code from the previous task, and verify that you can then compile it both with and without OpenMP support. To test it without OpenMP support, try with clang if you have it on the computer you are using, or otherwise just use gcc without the `-fopenmp` option.

Task 8

Critical sections

The code for this task is in the `Task-8` directory.

In this task we look at essentially the same thing as in the corresponding task in Lab 8, except that now we are using OpenMP instead of Pthreads.

Look at the code in `ompthreadtest.c`, compile and run it and try to understand what is happening. If you run the program several times, do you get the same result each time? If not, can you understand why?

Note that the variable `sum` is declared at global scope, so it is accessible by all threads, and all threads try to change its value simultaneously in the loop in `the_thread_func()`.

In the Pthreads case, we used mutex lock and unlock functions to protect access to the shared variable. Now, with OpenMP, there is a simpler way, we can just add one line:

```
#pragma omp critical
```

Insert that line right before the code line modifying the `sum` variable. This means that only one thread at a time is allowed to execute that section of code. Does it work – do you get the same result each time you run the program now?

Task 9

Nested parallelism

The code for this task is in the `Task-9` directory.

It is possible to use nested parallelism in OpenMP. However, this is a feature that can be enabled or disabled. You can check if it is enabled by calling `omp_get_nested()`

which will return 1 or 0 meaning enabled or disabled. You can enable or disable the feature yourself by calling `omp_set_nested()`.

When nested parallelism is enabled, that means that you can have a parallel block inside another parallel block. So for example, you can have a team of two threads and then let each of them start a new team of two threads, giving a total of 4 threads.

Write a small code to test this: let your code have one outer `#pragma omp parallel` directive, and then inside that another `#pragma omp parallel` directive. Use printf statements to see how many threads are created. Also check how the program's behavior changes when you enable/disable nested parallelism. What happens with the inner parallel block if nested parallelism is disabled?

Now that you have seen how nested parallelism works, apply it to parallelize the merge sort program given in the `Task-9` directory. This is the same merge sort code that you worked with in Lab 5. Since the merge sort algorithm splits the work into two parts in each recursive call, it seems convenient in this case to use a `#pragma omp parallel` directive with two threads each time.

However, at the same time you want the program to use at most a given number of threads input by the user (typically chosen according to the number of cores available). To make that work, you can add an extra argument `nThreads` to the `merge_sort()` function, specifying how many threads it should use. Then each time it splits the work between two threads, the number of threads to use at the next level is divided by two.

For example: the user specifies that 4 threads should be used for the sort operation. Then the `merge_sort()` function is called from the main program with the `nThreads` argument set to 4. Then, inside the `merge_sort()` function, a parallel block is used so that the work is split between two threads. Each of those threads calls `merge_sort()` recursively for its part of the work, but now with the `nThreads` argument set to 2, since each of those calls is allowed to use 2 threads. Inside each of those operations the work is again split between two threads, and `merge_sort()` is called with the `nThreads` argument set to 1. Then, when the `nThreads` argument is 1, all lower levels of the `merge_sort()` operation are done by a single thread.

Implement the merge sort parallelization outlined above. Are you able to make it work? What speedup do you get?

Note that the merge sort code you were given here does two malloc calls each time; as we have seen previously in the course we can get a performance improvement by reducing that to one malloc call. When the program is multi-threaded, this may be more important than for the serial case because the memory management system must do some synchronization when several different threads are doing many malloc/free calls. Change the code so that it uses one malloc call instead of two. Do you see a greater parallelization speedup after that change? If so, can you understand why?