CS4023 Week10 Lab Exercise

Lab Objective: In this week's lab we will continue our focus on the `findvals` of last time. Today we will modify our program from last time so that instead of using Pthreads, it uses `OpenMP` to distribute the running of the outer loop (the loop that iterated over the rows of the array) amongst a number of threads, each working in parallel. This will be done using the OpenMP thread library which is an easy way of taking advantage of threads.

Here's a quick summary of the tasks:

❶ Create a directory for this week's work

❷ We want to watch the speed-up that results from doing the array search in parallel using as many threads as the CPU(s) can provide.

❸ Secondly, in preparation for the main event modify the supplied `Makefile` so that the `C` compiler, `gcc` and linker/loader "know about" the `openmp` library

❹ Your next task is to learn about OpenMP to multithreading, *instead of* using pthreads. Read the tutorial at the supplied web page; I am not asking you to read the entire document but please pay attention as there are a few very relevant points for our work

❺ Modify the working of the `-v` flag to `findvals` so that the *thread* that detected the hit in the array is printed to the screen

In Detail

❶ You can create this week's lab directory with

```
mkdir ~/cs4023/labs/week10
```

As a starting point for this week's lab you can use the source code for the program you wrote last time, or, alternatively, we will give you source code that worked for us. **Note:** we give out source code for the last lab both with and without using pthreads. If you want to use our code, we suggest you use the non-threaded version as a starting point for this lab because OpenMP based code will be very different to the pthread-ed code. We hope you appreciate that writing a multithreaded program with OpenMP is much simpler than with pthreads.

However you choose you should do all your work in this week's directory, `~/cs4023/labs/week10`, and use only the file `findvals.c` and the two utils-related files `utils.[ch]`.

Now change your working directory to this directory since all of our compiling, etc. will be done in this directory.

❷ No amount of parallelism will speed up the inputting of the array (the `scanf()` nested loops). The only improvement that can come from using several threads in parallel is on the loops that search the array. So we should start our timing clock *after* the array has been read in. Move the couple of lines of code from last week's program that took the first reading of the clock (`gettimeofday()`) to after the input loops / before the loops that search the array.

Because of this I have changed – and you should too! – the message that gets printed out at the end to indicate that it is now the *search* time we're reporting on.

❸ Using a `Makefile` is another of those huge sanity-saving features of modern code development. If you have not been using a makefile please get in the habit of it now. I have provided a file called `Makefile.no-openmp` which you should copy and use. So that it is useful for the OpenMP part to follow you will need to modify it so that the `C` compiler, `gcc`, and then the linker/loader knows about OpenMP. You should modify both the `CFLAGS` and `LDFLAGS` entries of the file so that they include the OpenMP libraries. You can do this by appending the string `-fopenmp` to both lines.

Note that `make` assumes by default that the instructions are in a file called `Makefile` so either rename your file to this (Recommended!) or run make with:

```
        make -f Makefile.no-openmp
```

I really do **caution against the latter** because the file contents are now at odds with its name.

❹ We all know that "many hands make light work" and in a computing context having several CPUs[1] available to us to crunch a job should surely be a boon. Nonetheless, it is also a truism that "too many cooks spoil the broth," so part of the art of making your code run fast is knowing how to use the extra CPUs available to you. OpenMP is a nice, high-level library that can give us some of that leverage for not too much pain.

A very readable tutorial on OpenMP can be found at http://bisqwit.iki.fi/story/howto/openmp/. Ignore the details specific to C++, as OpenMP directives are the same both for C and C++. Please read it down to the section "Controlling which data to share between threads" – that is, stop at that section. Read carefully the difference between `parallel` and `parallel for` as described in the section "What are: parallel, for and a team". And just before that section read carefully what it says about running `C` programs.

[1]You can tell exactly how many CPUs are available on your machine by, in linux, browsing the file `/proc/cpuinfo`. This is literally a description of the specs of each CPU known to the OS.

Loop variables in `C` under OpenMP

When a `for` loop is broken up across a number of parallel threads using, for example

```
    #pragma omp parallel for
```

you have to be careful about the loop index variable. Unless you tell OpenMP otherwise, all variables are considered to be shared by all threads. If every thread is working from the same value then chaos will result. What you need is for each thread to keep track of its own part of the range of the looping variable.
You do this by declaring the loop index variables, `r` and `c` to be `private`.

Note, as in the last assignment, you still have to ensure that updates to the variable `count` are mutually exclusive; otherwise, race conditions will result with multiple threads simultaneously trying to update the same variable. OpenMP provides you multiple ways of avoiding that as described in the section “Thread-safety (i.e. mutual exclusion) ”.

- The most complicated way to solve the problem is with the `omp_lock_t` which implements locks (altough, that was what how we implemented our last assignment with pthreads).
- A less complicated solution will be to declare an OpenMP critical section, and update `count` in that section.
- Another solution would be to keep a `private` version of count for each thread and then, having initialised all of them to 0 at the outset, tot them up for the combined count
- Given that we only have to increment a single integer variable, there is an even more simple solution that OpenMP provides and, if you have read the OpenMP tutorial as far as I suggested you would have seen it.

With the `-v` flag enabled we can see what thread found each match with the following piece of code:

```
if (verbose)
  fprintf(stdout, "r=%d, cc=%d: %.6f (thread= %d)\n",
          r, c, rows[r][c], omp_get_thread_num());
```

Once you make the call to the above function you will get a compiler error unless you include the `omp.h` header file:

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

at the top of your program.

As always, please use the sample executable as an exact specification of how your output should look.

With the clock start moved to just before the array search the clock will measure the time spent performing the array search so you should be able to get an accurate picture of the search running time. With the `-v` flag disabled the huge reams of output will be suppressed and this essentially serial job will not affect the timing.

❺ By running the sample executable you can see how the thread that scored the hit is reported to the screen. Please consult the OpenMP documentation to find the function call that identifies itself. That is, what function allows a thread to identify itself. (Yes, the age-old question that has consumed philosophers since the dawn of civilisation, "who am I?" rears its head again.) Please adjust your `printf()` statement accordingly.

To make the point of what the extra CPUs can do for you here's a comparison between running this week's (parallelised) lab and the sample executable provided to you in the skeleton code for last week (NOTE: that sample executable was NOT using pthreads). The machine has 8 CPUs as reported in `/proc/cpuinfo`.

```
[cs4023@CLAWS-29 week10]$ ../week09/findvals -r 5.0 -t 0.5 < mat.1000x1000
# Start time and date: Mon Nov 12 09:56:13 2012
Found 5085 approximate matches.
Elapsed time: 0.280611(s)
[cs4023@CLAWS-29 week10]$ findvals -r 5.0 -t 0.5 < mat.1000x1000
# Start time and date: Mon Nov 12 10:00:15 2012
Found 5085 approximate matches.
Search time: 0.002437(s)
```

However, the difference between the two is also influenced by the positioning of our timing start: last week we started the timer at the "top of the program". This time we started it *after* the array had been read in. Reading the data can take a significant amount of time so the comparison just printed may not be entirely fair.

A comparison – but is it fair?

This is the fifth and final lab that you will be assessed on. To be assessed you will need to have the program written and working properly by 18.00, Thu. Week12.

The command for submitting this lab via the `handin` mechanism is:

```
~cs4023/progs/handin -m cs4023 -p w10
```