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 it distributes 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.

Summary

1 Create a directory for this week's work, then move gettimeofday to after fscanf.

2 Secondly, modify the supplied Makefile so that the C compiler (gcc) and linker "know about" the OpenMP library.

3 Your next task is to learn about OpenMP. 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

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

In Detail

1 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 code solution from the previous lab. Whichever route you choose, do all your work in this week's directory using only the three files: findvals.c, utils.h and utils.c.

Parallelism will not speed up the inputting of the array (the fscanf() loops). The only improvement from parallelism will be the loops that search the array. Therefore, we should start our timing clock *after* the array has been read. Move the lines of code from last week's program that took the first reading of the clock (gettimeofday()) to after the input loops and before the loops that search the array.

2 Using a Makefile is a huge sanity-saving features of modern code development. If you have not been using a makefile please get in the habit of it now. A Makefile allows you to run a single command, "make", to compile your program. I have provided a file called 'Makefile' which you should copy and use (requires some changes for M1 Mac users, lab TA's can help with this).
So that the C compiler (gcc) and linker knows about OpenMP, you should append -fopenmp to both the CFLAGS and LDFLAGS entries of the file so that they include the OpenMP libraries.

3 Parallelism, also known as multi-threading, essentially allows us to split the workload of a program across multiple different CPU cores. While extremely useful, it can require fine-tuning to make sure you are efficiently using the extra CPU cores. Luckily, OpenMP is a nice, high-level library that can give us some of that leverage for not too much pain.

An easy tutorial on OpenMP can be found at http://bisqwit.iki.fi/story/howto/openmp.
Please read up to the section "Controlling which data to share between threads".
Read carefully the difference between *parallel* and *parallel for* as described in the section "What are: parallel, for and a team".

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.

The variable *count* that gets incremented every time a hit is found is shared by all threads and so inevitably it is going to fall foul of the race condition that we introduced critical sections with.

Three solutions to this problem come to mind:

- the most complicated way to solve the problem with the count variable is to make it critical section
- a somewhat more straightforward solution would be to keep a private version of count for each thread, then adding them up for the combined count
- there is an even more simple solution that OpenMP provides through the use of *private*, as mentioned in the tutorial previously linked.

Once you include OpenMP code, you will get a compiler error unless you include the omp.h header file at the top of your program:

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

4 Using the following piece of code, we can find what thread finds a match:

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

As always, please use the sample executable as an exact specification of how your output should look (for M1 Mac users, use findvals_arm64 instead).

- Submit the following files: findvals.c, utils.c, utils.h and your executable.
- Submission is due next Week, **Thursday (24th November 2022) at 18:00** – 7 Days