Assignment 1

Introduction to the assignment

Assignment 1 is designed to introduce you to the shared memory parallel programming model using a low-level programming language. There are two parts to this assignment. In Part 1, your goal is to parallelize two simple programs using OpenMP. Part 1 is designed to familiarize yourself with OpenMP programming constructs and reason about serial and parallel phases of a program. In Part 2, your goal is to apply software optimizations from the lectures to improve program performance. Part 2 is designed for you to reason about how various hardware and software factors affect the performance of parallel programs, and how they can be optimised.

Part 1: Introduction to OpenMP

Background

Monte-Carlo Integration (MCI) is a technique for numerical integration that approximates the area underneath a curve with a non-deterministic approach. Figure 1 shows an example.

Assume we want to calculate the area of a quarter circle (of radius 1) with MCI. First, we pick an area that is strictly larger than the target integral (in Figure 1, this area is the unit square, bounded by the lines x=1 and y=1). Then, we randomly generate a number of points inside the domain of integration and calculate whether or not each point resides in the *target* area (the quarter circle) or the background area (the square). Therefore, the fraction of points *inside* the targeted area (in blue) to the total number of points (blue and orange) represents the area of the quarter circle.

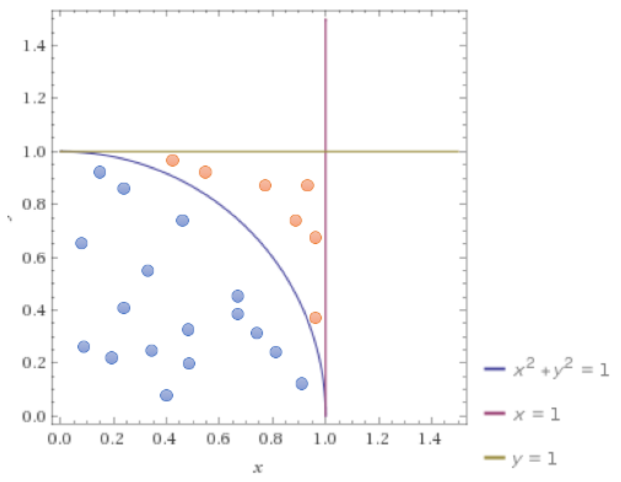

Figure 1 – MCI approach to calculate the area of a quarter circle

Program A

In Figure 1, we can compute an approximation of the constant π by comparing the ratio of area of the quarter circle and the area of the square.

Area of the square (S) = $r^2 = 1$
Area of the quarter circle (A_c) = $\frac{\pi r^2}{4} = \frac{\pi}{4}$

Using the MCI approach, we know $\frac{A_c}{S} = \frac{N_C}{N_S}$, where N_C is the number of points sampled inside the quarter circle and N_S is the total number of points sampled inside the unit square. Therefore,

$$\frac{A_c}{S} = \frac{\pi}{4} = \frac{N_c}{N_S} \Rightarrow \pi = 4 * \frac{N_C}{N_S}$$

Write a parallel program in C to approximate π using this approach, following the steps below:

- Read the number of threads and the total number of random samples to generate from command line parameters.
- Use OpenMP to sample the points and calculate, in parallel, the total number of points that fall inside the quarter circle as described above.
- Print the computed value of π using the above formula and the running time of your program to the standard output.

In the file A1.zip, we have provided `pi.c` which contains a skeleton of how you should structure your program. In the file `pi.c`, please write the function:

```
double calculate_pi (int num_threads, int samples)
```

We will be testing your program's correctness with automated infrastructure, so it **must** comply with both the input and output specifications of our test program, which are:

1. The filename `pi.c` must remain unchanged.
2. The program takes two input arguments. The first argument is the number of threads to use, and the second argument is the total number of samples to generate.
3. Your program must print the output in the exact format as shown below:

```
# /bin/bash

$ ./pi 4 500000
- Using 4 threads: pi = 3.1416045 computed in 0.405s.
```

Program B

You are going to extend Program A to provide a basic numerical integration function for arbitrary functions using MCI. The MCI algorithm to integrate any general function `f()` on the domain [a,b] is slightly different. As we don't know the range of `f()` on [a,b] a priori, and thus cannot simply bound it by a square as we did in Program A, we leverage a well-known approximation from calculus. Say that we generate one pair `(x,f(x))` and the area of the rectangle with height `f(x)` and width `a-b`. We have just generated a rough approximation of the area under the curve, albeit a very poor one. However, Figure 2 shows how in the limit, the average area of a large number of randomly generated rectangles converges to the actual integral of `f()`.[1]

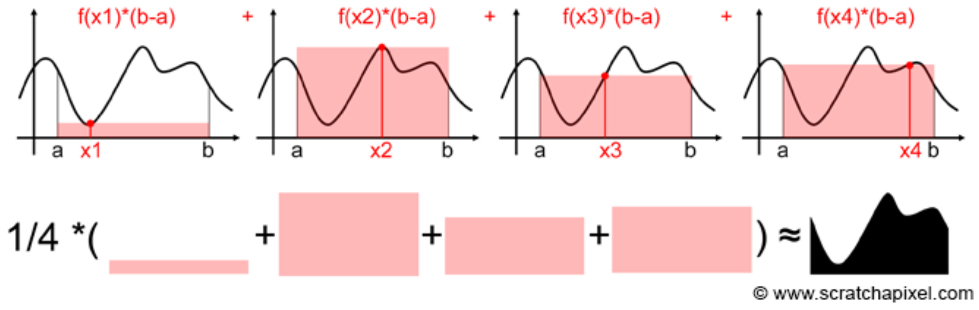

Figure 2 - MCI As a Limit of Rectangular Areas

Program B requires you to implement this type of MCI using an arbitrary function as well as an arbitrary domain. For Program B, your code should be placed in the file `integral.c`, which has the same basic structure as `pi.c`. Instead of embedding the function to be integrated into the program ($x^2 + y^2 = 1$ in program A), you need to update your program to operate with a call to an abstract function `f()`, which is defined in an external file `function.c`. Additionally, the domain [a,b] must be passed to your code at invocation time.

In the file A1.zip, we have provided `integral.c` which contains a skeleton of how you should structure your program. In the file `integral.c`, please write the function:

```
double integrate (int num_threads, int samples, int a, int b,
                        double (*f)(double))
```

To test your Program B, you can make up your own definition of `f()`. Our tester will define the function in `function.c`, so please name your test files accordingly. Please keep in mind that you do not need to rename the already defined `identity_f()` function. We will be testing your program's correctness with automated infrastructure, so it **must** comply with both the input and output specifications of our test program, which are:

[1] For more details on this, you can visit https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.

1. The filename `integral.c` must remain unchanged.
2. The program takes four input arguments. The first argument is the number of threads to use, the second argument is the number of total samples to evaluate, the third and fourth arguments are the lower and upper bounds of the domain of integration. These are the same arguments that get passed to the `integrate()`. Note that `integrate()` has an extra argument: `double (*f)(double)` a function pointer that takes in a double x and returns f(x).
3. Your program must print the output in the exact format as shown below:

```
# /bin/bash

$ ./integral 4 2000 5 9
- Using 4 threads: integral on [5,9] = 26.0460699392424
computed in 0.000184s.
```

Part 2: Optimized Parallel Programming in OpenMP

Program C

For Program C, you will optimize true/false sharing similar to the example presented in the lecture. In the file A1.zip, you will find `sharing.c` that implements a single-threaded version of a simple histogram algorithm that repeatedly generates random numbers and updates the histogram based on the obtained value. Your job is to parallelize the algorithm using OpenMP, study the effects of true/false sharing in the histogram using 100M operations, and optimize them using the techniques explained in the lectures.

We will be testing your program's correctness with automated infrastructure, so it **must** comply with both the input and output specifications of our test program, which are:

1. The filename `sharing.c` must remain unchanged.
2. The program takes three arguments. The first argument is the number of threads to use, the second argument is the number of samples to generate, and the third argument is the number of buckets in the histogram.
3. Your program must print the output in the exact format as shown below:

```
# /bin/bash

$ ./sharing 4 100000000 32
Using 4 threads: 100000000 operations completed in 0.29s.
```

Program D

Reducing Matrix Multiplication (RMM) is a variant of matrix multiplication. In vanilla matrix multiplication, each entry of the output matrix, C[i][j] is determined as the dot product of the row i of matrix A and column j of matrix B. In RMM, each entry of the output matrix C[i][j] is determined by two rows (rows (i*2) and (i*2+1)) and two columns ((j*2) and (j*2+1)). The value of C[i][j] is determined as the sum of the following dot products:

1. Row [i*2] of A * column [j*2] of B
2. Row [i*2] of A * column [j*2+1] of B
3. Row [i*2+1] of A * column [j*2] of B
4. Row [i*2+1] of A * column [j*2+1] of B

Input: Matrix A of dimension M * N and Matrix B of dimension N * K
Output: Matrix C of dimension (M/2) * (K/2)
Note: For the purpose of this assignment, you may assume M, N, and K are even numbers.

For Program D, your task is to implement and optimize the RMM of two matrices. A sample single-threaded unoptimized implementation `rmm.c` is provided in the handout A1.zip. You are required to provide and optimize a multi-threaded implementation using OpenMP. The program must operate as follows:

1. Read the values of M, N and K from the command line arguments.
2. The program then generates and initializes matrices A and B with random values.
3. The program then computes the matrix C as the RMM of matrices A and B.
4. The program finally writes matrix C into a csv file matC.csv and exits.

Step 3 is the main operation and timed. You are required to only optimize Step 3 using OpenMP. You cannot optimize the overall algorithm based on any symmetry, i.e. the overall number of calculations should remain the same as in the single-threaded version. We will be testing your program's correctness with automated infrastructure, so it **must** comply with both the input and output specifications of our test program, which are:

1. The filename `rmm.c` must remain unchanged.
2. Parts 1, 2 and 4 of the program must remain unchanged.
3. The program must write the final value of matrix C into matC.csv, you are not allowed to change the name of the output file.
4. The program takes five arguments. The first argument is the number of threads. The next three arguments are the values of M, N and K respectively. The fifth argument is a debug flag which can be either 0 or 1. Set the flag to 1 to print the matrices into the terminal to make it easier to debug.
5. Your program must print the output in the exact format as shown below:

```
# /bin/bash

$ ./rmm 4 1024 1024 1024 0
- Using 4 threads: matC computed in 0.1397s.
$ ls
matC.csv ...
```

Development infrastructure

In the file `A1.zip`, we have provided the following files:

1. `pi.c` – skeleton of program A
2. `integral.c` – skeleton of program B
3. `function.c` – contains functions to be integrated by program B
4. `sharing.c` – single-threaded unoptimized version of histogram algo (program C)
5. `rmm.c` – single-threaded unoptimized version of RMM (program D)
6. `utility.h` – contains utility functions used by the other files, such as functions for:
    a. Random number generator
    b. Measuring elapsed time between two points in a program
    c. Reading and writing csv files
    d. Initializing and displaying matrices
7. `Makefile` – To compile the programs using the current version of gcc
    a. `make all` to compile all programs
    b. `make <prog>` to compile a specific program (pi/integral/sharing/rmm)
    c. `make clean` to delete all generated executables

Important: Ideally, you should run these programs on Linux platforms and test them on the SCITAS cluster. But if you are temporarily working on an operating system that is non-Linux, you will need to take some extra steps to make your system work with OpenMP. On macOS, you can use Homebrew[2] to install the latest version of gcc (following instructions here[3]). Then,

1. Find the version of gcc installed using `gcc -version`
2. In the Makefile, replace `CC=gcc` with `CC=gcc-x` where `x` is the version of gcc installed

On Windows, you have several options. You can either use WSL/Cygwin to install a Unix-like environment on your computer or use the VirtualBox VM image EPFL provides as a workspace[4] in which to perform your development.

[2] https://brew.sh
[3] https://discussions.apple.com/thread/8336714?sortBy=rank
[4] https://support.epfl.ch/epfl?id=epfl_kb_article_view&sysparm_article=KB0017020&sys_kb_id=56aff0d8973c9618e55e370f2153af54

Deliverables

You need to turn in the following things for this assignment:

1. Completed code for `pi.c`.
2. Completed code for `integral.c`.
3. A parallelized and optimized version of `sharing.c`.
4. A parallelized and optimized version of `rmm.c`.
5. A report that answers the questions listed below. The report name should be a1_GROUPID.pdf

Remember to check if your code complies to the format expected by the automated tester; otherwise, you will receive no points for correctness!

To submit your code and your report, create a tarball archive (**this is the only accepted format!**) called `a1_GROUPID.tgz` and upload it on Moodle[5].

Report

In the report, we expect you to answer the following questions:
For Programs A and B:

1. Describe the algorithm that was implemented. Identify:
    a. Which parts of the program you parallelized and why. If the algorithm has several phases (parallel phases followed by serial phases), identify each of them.
    b. Identify the operations that dominate the execution time of each program phase. You should pick a single type of operation per phase. (i.e., If you believe that your program spends most of its time performing multiplications, then that operation is multiplication).
    c. For each phase, identify the program arguments that affect the number of performance-critical operations in each phase and provide the asymptotic execution time of the program in the big O notation.
    d. Estimate the speedup of the multithreaded program over the single-threaded version of it, using thread counts equal to {1, 2, 4, 8, 16, 32, 48, 64} and draw a graph showing how the execution time scales with the number threads. Ignore any hardware limitations in your estimation.
2. In a table, present the execution times and speedups over the single-threaded version you measured for the same thread counts as in 1d. Then add these results to the previous graph to see how your prediction matches with the actual execution times. Run all of your results on the SCITAS cluster using the job submission system explained in the exercise session, as well as the link on Moodle.
3. Compare the speedups you measured on the SCITAS machines to what you predicted in Q1d. If they are different, why?

[5] If you don't know how, look online for a guide like this one: https://www.howtogeek.com/248780/how-to-compress-and-extract-files-using-the-tar-command-on-linux/

For Program C:

1. Explain how parallelizing the given algorithm impacted performance.
    a. Was there any true/false sharing?
    b. If yes, how did you optimize it?
    c. Report the performance differences before and after these optimizations for thread count {1,2,4,8} and 100M operations.
2. Explain how does the histogram size affects the performance of the unoptimized algorithm.

For Program D:

1. Explain how you parallelized and optimized the RMM algorithm.
    a. List the optimizations you applied and the reasons you chose to apply them.
    b. Note that the program is trivially parallelizable, so we are more interested in the memory optimizations than in parallelizing the program itself.
2. Identify the different ways of splitting the work among threads and which result in more data locality within threads.
3. Identify whether there is an issue with load balancing when dividing the work.
4. Report how much each optimization improves performance and explain the results obtained.
5. Design and run experiments that isolate the effect of each optimization/thread organization you tried out and then report the results.
6. Present the execution time you measured in a graph for the following set of thread counts {1,2,4,8,16,32} for M, N, K = 1024.

The report should be as precise as possible while addressing the questions above. Keep the descriptions of the algorithm implemented and of your reasoning in parallelizing the program short.

Grading

The code will be graded automatically by a script and checked for plagiarism. The script will check how the running time scales with the number of threads and if the results returned are consistent with what was expected. Plagiarized code will receive 0. We will escalate plagiarism cases to the student section. The reports will be read and graded by humans and checked for plagiarism automatically. The grade breakdown is as follows:

• Correctness:	50%
• Report:	50%