Group-46

Md Tahseen Anam (19941202-T678)

Nafi Uz Zaman (19930919-T550)

Exercise 1: Numerical Integration

a

Run **make ./exercise1** in the terminal to compile the code **exercise1.cpp** . The **exercise1.cpp** file can also be compiled by running the command: **g++ -std=c++11 -Wall -pthread exercise1.cpp -o exercise1** in the terminal. After compiling the code, run **./exercise1** with **-h** argument to view the instructions.

b

Threads	Trapezes	Integral	Runtime
1	1	3	0.000284462
1	2	3.1	0.000427918
1	3	3.12308	0.000339177
2	1	3.1	0.000319763
2	2	3.1	0.0004033
2	3	3.1235	0.000482106
3	1	3.12308	0.000149073
3	2	3.12308	0.000569444
3	3	3.12308	0.000632
4	1	3.13118	0.00020495
4	4	3.13118	0.00170501
4	50	3.14153	0.000490248
4	100	3.14158	0.000574039

4	200	3.14159	0.00111817

As we can see, our computed integral slightly varies as we vary the number of trapezes and threads. In most of the cases, if we increase the number of trapezes than our computed integral also gets increased. This indicates that as we increase the number of trapezes, we go closer to the actual value of the area under the curve. The run time also varies as we vary the number of trapezes and threads. Most of the time, we have observed that the run time increases if we increase the number of trapezes.

c

In our code, we have distributed the workload between threads by dividing the number of trapezes by the number of threads. So if there are two threads and one trapeze, that one trapeze will be divided into two trapezes and equally distributed to those two threads. In this way, we are ensuring that no thread gets more workloads than the others.

Exercise 2: Sieve of Eratosthenes

Run **make ./sieve** in the terminal to compile the code **sieve.cpp** . The **sieve.cpp** file can also be compiled by running the command: **g++ -std=c++11 -Wall -pthread sieve.cpp -o sieve** in the terminal.

Synchronization used by using standard **pthread_mutex_t** in this algorithm to compute the prime numbers. There is only one shared variable, in this case the “seeds”, that are used to compute the prime numbers from the chunks. Data is only read from the “seeds” by the threads but as all the threads are trying to access “seeds” concurrently so synchronization has been used to avoid any exception. As data is only read from the “seeds”, it also reduces the communication between threads.

Work to threads were distributed based on the number of cores and the value of *Max*. After computing the seeds we are left with *Max* - √*Max* values to compute primes from. The difference divided by the number of cores gives us the number of chunks and this is equal to the number of threads that we will need. Depending on the value of *Max*, it could be the case that all threads work on chunks of equal size or only the last thread works on a chunk that is slightly bigger than the others.

Example:

Max = 12

Cores=2

√*Max* = 3

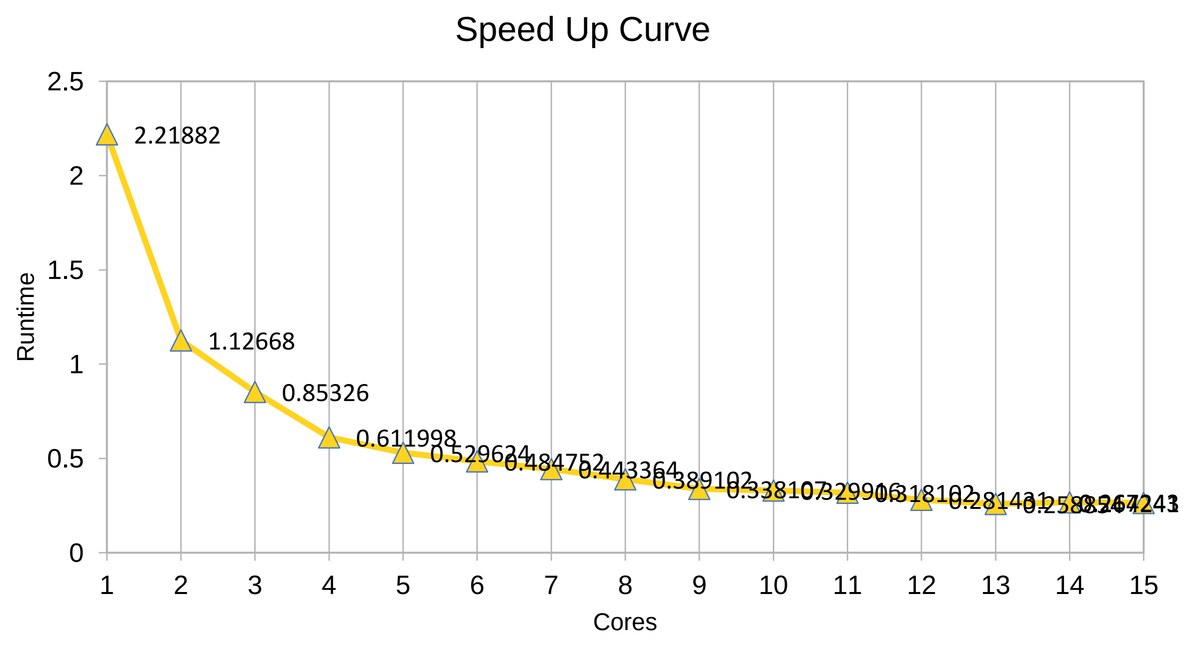
Chunks = (12-3)/2 = 4

Threads = 4

Average number of data to work on = (12-3)/4 = 2

Therefore, the first 3 chunks are of size 2, and the last chunk is of size 3. So, we can see that one thread will have to compute primes from a larger chunk compared to the other threads. Below we demonstrate the runtime for different values of core.

Cores	**Runtime**
1	2.21882
2	1.12668
3	0.85326
4	0.611998
5	0.529624
6	0.484752
7	0.443364
8	0.389102
9	0.338107
10	0.329916
11	0.318102
12	0.281431
13	0.258854
14	0.267243
15	0.264241



Exercise 3: Mutual Exclusion Algorithms

1. A **mutual exclusion** (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

   The protocol satisfies mutual exclusion. If we consider 2 threads (thread_1 & thread_2) running concurrently, if both threads call the lock function, this protocol does not let both threads to acquire the lock simultaneously. The protocol will only let one thread to acquire the lock and enter into the critical section while the other threads will keep executing the while loop until the loop condition breaks and the loop condition will only break when the unlock function gets called by the thread that is holding the lock. So, thread_2 waits for the thread_1 to set a shared variable to a specific value which will allow thread_2 to execute the remaining part of the critical section. The protocol ensures the fact that both thread_1 and thread_2 can not enter the critical section simultaneously and start working on it.

2. **Starvation** occurs when one or more threads in the program are blocked from gaining access to a resource and, as a result, cannot make progress while some other threads get executed and make progress.

   The protocol is not starvation free. If we consider 2 threads (thread_1 & thread_2) running concurrently and trying to acquire the lock, neither thread waits for a long time for a shared variable to complete execution. If we consider the following: thread_1 calls the lock function and enters the first loop and then enters the second loop and exits the second loop. Thread_1 acquires the lock and enters the critical section. In the meantime, thread_2 will begin executing the code in the second loop and has to wait for the thread_1 to set the condition in this case “busy” to false before it also enters the critical section, now if thread_1 does not call the unlock function or keeps executing in the critical section, then thread_2 will never get the chance to enter in the critical section. As a result, starvation will occur.

3. **Deadlock** occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.

   The protocol is not deadlock free. There is a possibility that all the threads are in the while loop inside the lock and waiting for other thread to set “busy” to false. For example, thread_1 enters the second while loop, sets “turn” to “me”, exits the loop, sets “busy” to true. By that time, thread_2 enters the second while loop of the lock function, sets “turn” to “me” and keeps executing in the while loop as “busy” is true. Now, as “turn” is not equal to thread_1, thread_1 will again enter the while loop and it will also keep executing in the while loop as “busy” is true. So, both threads will wait for each other to set “busy” to false. As a result, deadlock will occur.