Assignment 2 - Introduction to Parallel Programming

Bernat Xandri Zaragoza

1. **Exercise 1**: Numerical Integration (3 points)

a) Implement a parallel program in C11 or C++11 for numerical integration of the following function. Your program should accept the total number of threads and trapezes with appropriate command-line arguments. Also, the command line argument -h should print a help message and exit.

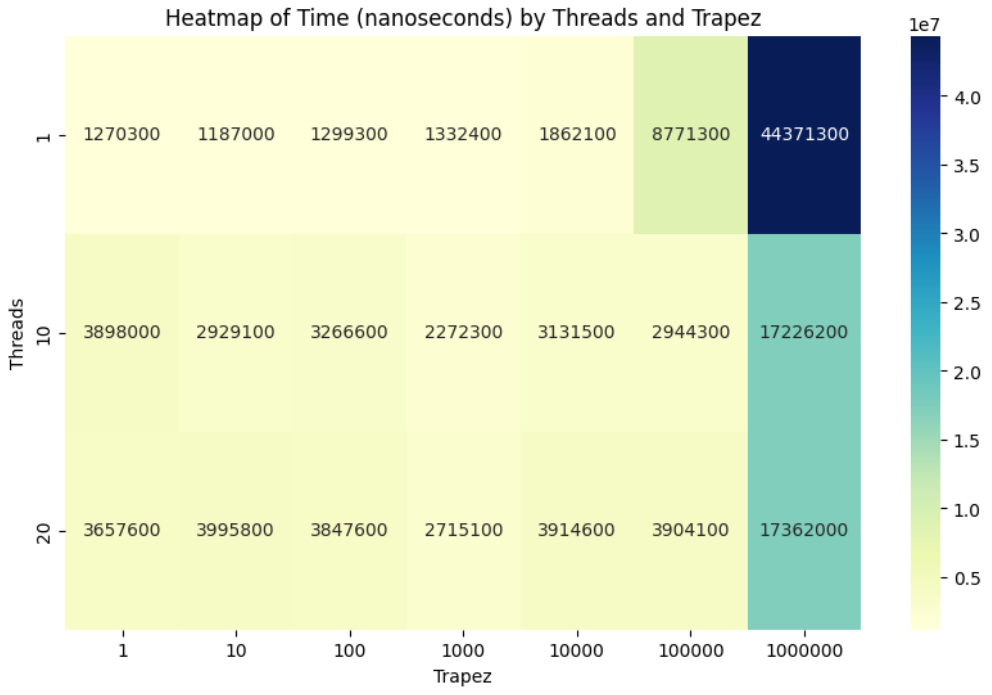
For this solution, I created a vector of threads, in which each thread is created with a function that works as the integration function, which divides the total area of the function of interest in the number of chunks that is indicated through the variable trapezes. Once we have all the chunks, the proportional part of these chunks is assigned to each thread in an even distribution (except the last one, that might have less in case of a non-divisible number). Once all the threads are finished (we use a Thread Synchronization), the value of all the partial trapezes is added, to get the estimated integral.

b) Test your program and measure the runtime of the computation (i.e., without setup time) for various configurations (number of threads, number of trapezes, each starting at 1). However, make sure that you also measure for a large number of trapezes (in the order of tens or hundreds of thousands). Document your results in tabular form and analyse. By the way, did you notice something concerning the value of the computed integral?

Threads	Trapezes	Time (nanoseconds)	Integral
1	1	1270300	3
1	10	1187000	3.13993
1	100	1299300	3.14158
1	1000	1332400	3.14159
1	10000	1862100	3.14159
1	100000	8771300	3.14159
1	1000000	44371300	3.14159
10	1	3898000	3
10	10	2929100	3.13993
10	100	3266600	3.14158
10	1000	2272300	3.14159
10	10000	3131500	3.14159
10	100000	2944300	3.14159
10	1000000	17226200	3.14159
20	1	3657600	3
20	10	3995800	3.13993
20	100	3847600	3.14158
20	1000	2715100	3.14159
20	10000	3914600	3.14159
20	100000	3904100	3.14159
20	1000000	17362000	3.14159

We can observe that the value of the computed integral tends to pi, which is the correct value of this integral, when calculated through symbolic methods.

c) Evaluate and document different methods of distributing the work among threads.



Heatmap with python

We can observe that augmenting the number of trapezes causes and increase in computation time, which makes sense since more trapezes means more individual computations, and therefore a higher resolution.
Also, incrementing the number of threads makes the computation time decrease, since more calculations can be made in parallel. This only applies to a certain number of threads, probably due to hardware limitations.

2. **Exercise 2**: Sieve of Eratosthenes (3 points)

a) Implement a parallel version of the Sieve of Eratosthenes.
Besides your code, you need to provide:

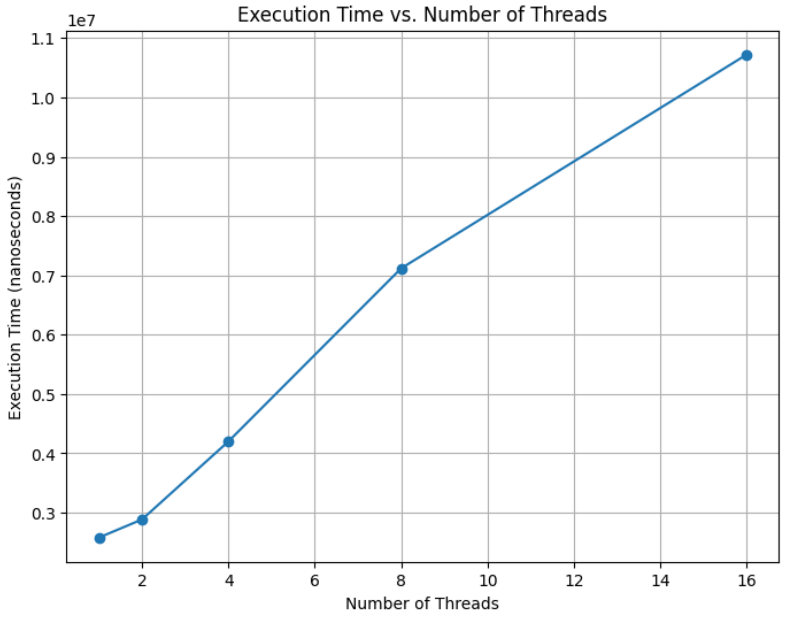
- A brief report that explains your solution (e.g., did you use synchronization between threads and why, how did you distribute the work to threads, how did you minimize communication, how did you achieve load balance, etc.)
- Reports the speedup curve you get as the number of cores is increased.
- You should also try to vary the size of Max to gauge the impact of program size on parallel performance. Make sure to pick large numbers for Max (e.g., in the millions to ensure there's enough work for threads to do).
  Needless to mention, make sure your code is correct.

We used vectors instead of arrays, since the array in C++ can use shared bytes between positions, so there could be trouble when multiple threads are accessing positions on the array next to each other. Even though vectors are not thread-safe, no locks or mutex are needed since we can ensure that two threads will not try to modify the same position in the vector at the same time. We used synchronization between threads to make sure that the main execution waits for all threads to be done before continuing with the program.

The speedup curve that we obtain when running this program with different number of threads is as follows:

Threads	Max Number	Execution Time (nanoseconds)
1	10000	2580600
2	10000	2887200
4	10000	4201800
8	10000	7118600
16	10000	10720600

With these results, we can plot the following graph:



Graph with python

As we can observe, this speedup curve is not what one would expect. In fact, we observe a few problems in this code, that I will try to explain in the following part;

One problem that we encounter with this parallelization, is that the algorithm stops working (at least as far as I understood the instructions given for this assignment). When splitting the vector of numbers to be checked in smaller chunks (*Given p cores, build p chunks of roughly equal length covering the range from √Max + 1 to Max , and allocate a thread for each chunk.*) the issue arises that all threads will start the Sieve of Eratosthenes by the first number of its own chunk, and therefore assume that number to be a prime number, which is a condition that it's not true. Therefore, causing to have some not prime numbers not being crossed out.
This happens because the other threads may contain a number that is a divisor of this first number, but because they are only crossing numbers in its own chunk (*Each thread uses the sequentially computed "seeds" to mark the numbers* ***in its chunk***.), other threads can not benefit of the knowledge of other threads.
Another issue that I identified, is that because of the limitation in the algorithm to test numbers only until the power of two of this number is bigger than the max (*until k^2 is greater than Max*), threads will stop working before they have detected all prime numbers in its chunk, since this condition only make sense in the regular form of the Sieve of Eratosthenes, where it is starting from 2.

In order to try to solve this, I have made some small changes to the instructions given in the assignment, which allowed me to improve the result slightly, but since I didn't want to change the algorithm too much, my final solution is still not fully functional (it doesn't manage to detect all prime numbers).

I understand this might be just caused by me not understanding the instructions of the assignment for the parallelization of the algorithm, but by following said instructions to the best of my understanding, I find this issues that make my code not work properly. I hope that, in case this was not the desired outcome of this exercise, my explanation and analysis of the problems, can compensate for the problems themselves.

3. **Exercise 3**: Mutual Exclusion Algorithms (3 points in total)

```
class Flaky implements Lock {
      private int turn;
      private boolean busy = false;
      public void lock () {
int me = ThreadID.get();
            do {
do {
            turn = me;
} while (busy);
            busy = true;
      } while (turn != me);
      }
      public void unlock() {
busy = false;
      }
}
```

a) Does this protocol satisfy mutual exclusion?

We can see that the presented code implements a Java class called Flaky that inherits the class lock. In this implementation, the protocol attempts to achieve mutual exclusion by means of using the private variables turn and busy, that will act as a simple version of a token access to a shared resource.
This protocol works as follow, when a thread summons the lock method, it sets his own ThreadID as the me variable, and enters the nested do-while loop. Here, it stays on the inner loop trying to set the variable turn as itself, until the shared resource is available. This way, the protocol manages to ensure that while a thread is accessing the resource, all other threads that might try to access it are set to busy-wait in this inner loop. Once this resource is available, a randomly selected thread (or I assume it would be randomly selected, since there's no queuing protocol implemented) manages to set the variable turn as itself, and then sets busy as true again, forcing all other threads to continue waiting until he's done with this shared resource. After setting the turn as itself, this thread can continue and access the resource.
Once it is done with the resource, the thread will summon the method unlock to set busy as false, so that the next randomly selected thread can access the resource.
This protocol achieves mutual exclusion, since this 'token' access implementation ensures that only one thread can access the locked chunk of the code.
It is worth mentioning that the protocol relies on busy-waiting, which is not considered a good practice in Java, due to its inefficiency.

b) Is this protocol starvation-free?

Starvation-free refers to the ability of a protocol to ensure that a thread that is trying to access a locked resource will eventually be allowed to access said resource. In the case of this protocol, this is not achieved, for the following reasons.
In this protocol, since a queueing system is not implemented, which thread is going to be the next one to access the shared resource is picked randomly, causing then a situation

where starvation can happen in case a thread that is waiting never gets randomly picked. This causes the protocol to not ensure fairness.
Starvation can also happen in case that a thread accesses the locked resources, and just never exits it. In this situation, there's no way of avoiding starvation, since no other threads will be able to access this locked resource anymore. In starvation-free protocols there should be a method implemented to avoid this kind of behaviour, and make sure that if a thread has been waiting for a locked resource for a long time, it is allowed to access it even if another thread is using it.

c) Is this protocol deadlock-free??

Deadlock free is when a protocol ensures that a situation where all threads are waiting and no-one can access the resource, and therefore no progress is being done, can not happen. In this case, this protocol can be considered deadlock-free for the following reasons. Since the resource is accessed as an individual resource, meaning that a thread doesn't need to access another resource at the same time, the deadlock where both codependent resources are accessed at the same time for different threads, causing them to get stuck in a deadlock situation where they are waiting for each other to leave, cannot happen.

Since there's also no circular dependencies, circular deadlocks cannot happen.

As a conclusion, since this is a fairly simple implementation of a lock, and doesn't involve any other locked structures, we can say that it is indeed deadlock-free. The only situation where a complete block of access to the resources could happen is if a thread is accessing the resource and just never leaves it (doesn't do the unlock), but I would consider this to be a starvation situation more than deadlock.

4. **Exercise 3**: Concurrent Data Structure (5 points in total)

Your task is to implement (in C++) and compare thread-safe versions of the sorted list, using the following synchronization mechanisms:

1. coarse-grained locking (for example, in C++, using the std::mutex class from the standard library;
2. fine-grained locking (again, using the std::mutex class from the C++ standard library);
3. coarse-grained locking, using a test-and-test-and-set (TATAS) lock implementation;
4. fine-grained locking, using a test-and-test-and-set (TATAS) lock implementation;
5. fine-grained locking, using a scalable queue lock algorithm, either CLH or MCS—feel free to choosebetween them.

Evaluate, plot, and explain the results/behaviour of the five versions from the point of view of performanceand scalability with respect to:

- the number of threads;
- the performed operations.

Make sure you run your experiments on a machine that allows to run at least 16 concurrent threads (e.g., one that has at least 8 physical cores, each with hyperthreading), and run your experiments using a suitable subset of worker threads in that range, making sure you include all powers of two (1, 2, ..., 2 n) in the range of numbers that your machine allows as concurrent threads.

Discuss the advantages and disadvantages of each form of synchronization in terms of: **ease of implementation, lock contention, performance, and scalability**. Identify situations when one version is more suitable than another, e.g. % of read vs. % of update operations.

Dedicate a section of your report to explain the reasoning, challenges and solutions in implementing the five versions of the data structure. Describe both the general solution and technical details. If you ended up getting 'inspiration' from code for some of these locks that you found on the internet, make sure you include a pointer to it in your source and your report. The points of this exercise are split equally into 2.5 points for implementation and 2.5 for the report.

1. Coarse-grained locking mutex

Coarse-grained locking with mutual exclusion (mutex) is a strategy for parallel programming that approaches the problem of concurrency in a simple way, using big locks to make sure that only one thread is using the shared resources at any given moment. This way, it ensures that there will be no data races, but at the same time, because it is a coarse-grained approach, it is also not very efficient, since only 1 thread can access the resources.

One of the advantages of this method is that it is way easier to implement, since it doesn't require to take into account any data-relationship between the different shared resources. Therefore, it is a more straight-forward implementation.

Since it is a general lock for all the shared resources, we would expect more lock contention, since we will have more threads waiting to own the mutex in order to access the shared resources. This is because, since it's a coarse-grained lock, parallel access to different resources is not implemented.

The performance is also going to be worse, since the waiting times are going to grow. The use of only one lock for all resources can be seen as a bottleneck.

The scalability is reducing, since at some point the waiting times will be too long for this solution to be optimal.

The implementation of this strategy is in the file **sorted_list_cglm.hpp** and is thread-safe for concurrent access. Multiple threads can safely call its methods without causing data races, but only one thread can execute any method at a time, as explained before.

2. Fine-grained locking mutex

Fine-grained locking differs from the previous method in that instead of using one big mutex to lock all shared resources, it can use multiple mutex for the different shared resources, to make sure that it is more efficiently used. In this case, we have to implement a lock for each position on the sorted list, so that all positions can be accessed in parallel, but the data integrity of each position is preserved with the lock.

This solution is not as easy to implement, since it requires to consider how the parallel access to different resources may behave in a concurrent program, and consider possible dead-locks

that might happen due to this concurrent access. In this case, since all positions are independent, these situations will probably not happen. Still, it takes more work to implement the locks for each position of the list instead of just locking the whole method with a general mutex.

It allows us to get a lower lock contention, since only threads that are competing for the same position will be forced to do a busy-waiting, threads trying to access different positions will be allowed to work in parallel.

This allows this solution to perform better, achieving a higher efficiency. It is also a solution with better scalability, but also a more difficult scalability, because all codependent data structures need to be considered.

Fine grained is especially useful for operations that require reads, or access to different parts of the shared data structure, like in this case, since this way we can reduce the number of competing threads to access other methods that might be more sensible, such as writing or deleting. It doesn't apply to our exercise, since we only have 3 operations on the data structure, but usually fine-grained solutions would require to consider corner cases like adding a new position at the beginning of the list.

3. Coarse-grained locking, using a test-and-test-and-set

Coarse-grained locking using a TATAS approach consists on a protocol where threads keep checking and setting a variable until they are allowed to access the resource (own the lock/access token). Since it is a coarse-grained locking, it should be a general lock that allows only one thread to access the linked list at any given time.

As explained in the first case, Coarse-grained locking protocols are usually linked to more easy implementations as well as less lock contention, because of bigger portions of code being locked at a time.

As in the first case, the performance wouldn't be great, and in the case of using many threads it will be prone to bottlenecks, since only one thread will be accessing the linked list at a time.

4. Fine-grained locking, using a test-and-test-and-set

Fine-grained locking using a TATAS, in opposite to its Coarse version, provides really good scalability and performance, at a cost of a more difficult and case dependant implementation.

5. fine-grained locking, using a scalable queue lock algorithm

Fine-grained locking using a scalable queue lock algorithm characterizes for using a queue approach to managing access to the locks. In these algorithms, each thread wanting to access a lock, joins a queue, and they wait their turn to own the lock. This queuing mechanism ensures fairness and avoids the problem of thread starvation.

This algorithm has a more difficult implementation, but provide a more efficient and fairer implementation, as well as a better performance.