Assignment 2

Bjørn Christian Weinbach Thameez Bodhanya

September 24, 2021

Instructions

For compiling the code that was provided in the assignment as well as the code that was edited for solving the exercises, the makefile provided was used. In some cases, the programs would not compile with -std=c++11 flag. This was mentioned in the lectures. If that is the case. Replace the flag with -std=c++0x flag.

We have used *siegnbahn.it.uu.se* for checking that the code runs. The code is structured in the following way:

- Q1
  - *question1.cpp*
  - Makefile
- Q2
  - *question2.cpp*
  - Makefile
- Q4
  - Coarse-grained locking.
    - *benchmark_example.cpp*
    - *benchmark.hpp*
    - *sorted_list.hpp*
    - Makefile
  - Fine-grained locking.
    - Same as above
  - Coarse-grained locking TATAS
    - Same as above
  - Fine-grained locking TATAS

    - Same as above
- Queue-grained locking TATAS
    - Same as above

There is a makefile for every question and subquestion. To check a question just navigate to the corresponding code and run:

*Make PROG=**filename***

Where ***filename*** is the name of the C++ file to compile.

Numerical Integration

To numerically calculate a integral of a function f. We can use the trapezoidal rule [Wikipedia contributors, 2021b]. The trapezoidal rule approximates the area under a graph of a function as a trapezoid and calculates the area. This can be done for more and more trapezoids to increase accuracy but also coming at a computational cost. The trapezoidal rule can approximate a definite integral with riemann sums by splitting the interval $[a, b]$ such as $a < x_0 < x_1 < \cdots < x_{N-1} < x_N$ and perform the following calculation

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\left(f(x_0) + 2f(x_1) + \cdots + 2f(x_{N-1}) + f(x_N)\right) \tag{1}$$

The sum in this calculation does not depend on each other and can therefore be parallelized and calculated independently and summed up when they are complete. This has been implemented in our code by creating a C++ program that takes in the command line arguments N for the no of threads and T for the no of trapezoids. All threads gets $\frac{N}{T}$ trapezoids besides the last unlucky thread that gets $\frac{N}{T} + T \bmod N$ due to ease of implementation. All the threads operate this function and work on a shared variable to calculate the sum, hence the mutexes.

```
void *calculateFactorial(void *conf)
{
  Config *cfg = (Config *)conf;
  double localResults;

  for (int i = 0; i < cfg->numTrapPerThread; i++)
  {
    int pos = cfg->startI * cfg->numTrapPerThread + i;
    if (pos == 0 || pos == numTrapezes)
    {
      continue;
    }
    localResults = localResults + 2 * function(a + ((pos)*w));
  }

  results_mutex.lock();
  results += localResults;
```

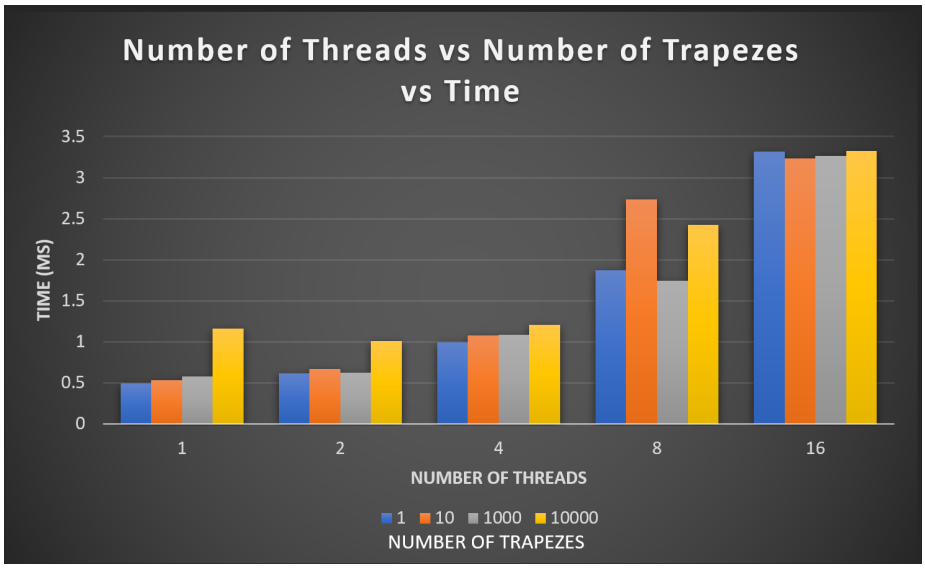

Figure 1: Different simulations for $T < 10000$ and $N \in \{1, 2, 4, 8, 16\}$

```
18    results_mutex.unlock();
19
20    pthread_exit(0);
21 }
```

Listing 1: non-determinism.cpp

Experimental Results

As threads increased for low amount of trapezes, the time to compute actually increases shown in figure 1. This is due to the overhead that parallelization adds to the application (thread-creation, waiting on threads to complete and lock-waits). However as we reach medium amount of trapezes (10^6 in figure 2) or high (10^9 in figure 3) we see that the time decreases significantly as at this point the cost to parallelize is less than the benefit it offers. While the number of trapezes increase, so too did the accuracy of the result.

Evaluation

This solution takes a very naive approach to parallelizing the program. It attempts to split the number of trapezes by the number of threads (with the surplus going to the last thread). This means that:

- the work load is not evenly split (in terms of the data partitioning)
- the time taken for each thread to complete varies based on the effort required to complete the calculation for a given trapeze.

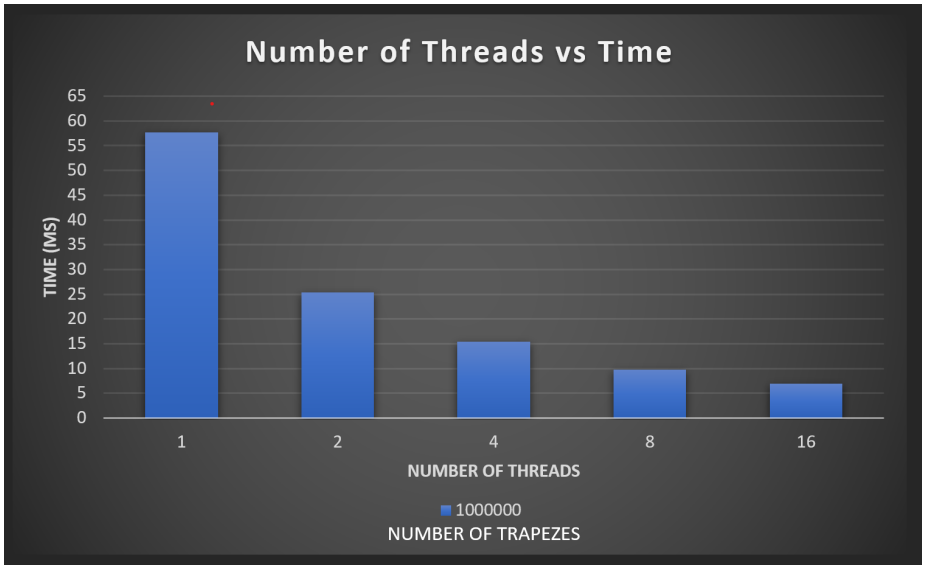

Figure 2: Different simulations for $T = 10^6$ and $N \in \{1, 2, 4, 8, 16\}$

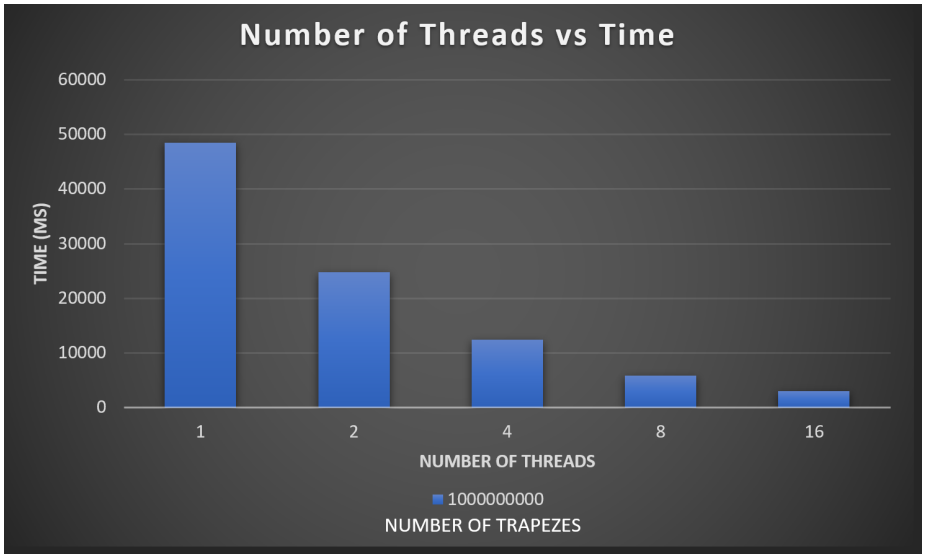

Figure 3: Different simulations for $T = 10^9$ and $N \in \{1, 2, 4, 8, 16\}$

An improvement to speed can occur if a thread queueing system was implemented instead, where each thread worked on a single trapeze (until max threads). Thereafter as a thread completed, a new thread would spin up and compute the next trapeze. This would ensure that no thread would be wasted, as until the sum was complete, all threads would be working (in a more equal and more efficient manner). However the queuing solution could potentially have a mutex related bottle neck as result would need to be written after computing a single trapeze area (vs in our solution, it is only written at the end of a set of trapezes).

Sieve of Eratosthenes

The sieve of Eratosthenes is an ancient algorithm for finding all primes up to a upper limit called Max. This algorithm is very old and was first described to Eratosthenes of Cyrene, who lived in the 3rd century BC. [Wikipedia contributors, 2021a] This is done by iteratively calculate multiples of a iterator k and marking these multiples. This iterator is is 2 in the beginning and when all multiples of 2 is marked, find the first unmarked number i.e 3 and repeat until $k^2 > \text{Max}$.

This implementation attempted to stay as true to the original algorithm as possible and thus first computes the primes from 2 to $\sqrt{max}$ in a serial fashion. Once complete these 'seed' primes are stored in an array. The remaining numbers (that being from $\sqrt{\text{max}}$ to max) are then split amongst the number of threads. This is done by dividing the amount of remaining numbers by the number of threads and flooring the result. If the aforementioned remaining list is divisible by numberOfThreads without remainder, work is distributed equally (and therefore fully *balanced*). If however the result is not fully divisible, the final thread picks up the extra cycles (thus leaving it in an almost equal split for all but one of the threads) and completes the calculation.

Mathematically given the fact the partition size is constant (for almost all cases) and the seed values are constant, the work for each thread is almost equal (as the checks would equate to *for each number in range, check if number is divisible by any number in seeds*). Therefore the max-checks for each thread is *rangeLength * numberOfSeeds*. A potential inequality argument may be made that the actual checks done may be slightly quicker in one thread vs another (as one thread may hold a range with many numbers divisible by 2, which would result in far quicker tests). This argument is valid, however testing has not shown this to be impactful in any meaningful way.

Furthermore no synchronization between threads were utilized in this solution. The simple reasoning behind this is that no thread operates on the same block of memory that another thread operates on. As the results array is a contiguous block of memory, with each element getting a distinct memory address, it therefore follows that each element can be interacted with independently of each other. Since the algorithm partitions the remaining numbers into distinct chunks with no overlap, no thread contains or interacts with numbers out of its chunk, which therefore follows that no thread interacts with elements

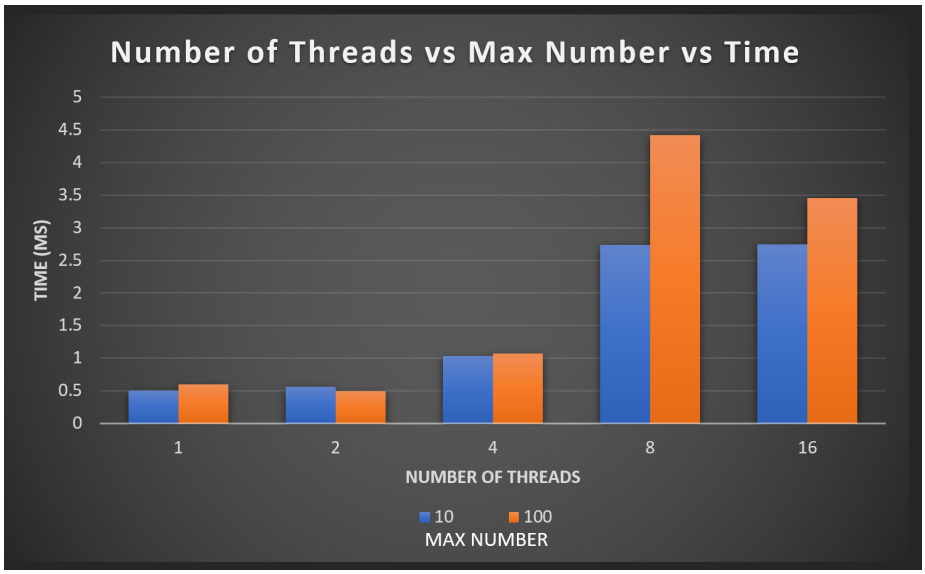

Figure 4: Max $\in \{10, 100\}$ and $T \in \{1, 2, 4, 8, 16\}$

being touched by another thread (in terms of the 'result' array.) One *could* add a mutex lock on writes to results, however the overhead of the lock checks would provide no real benefit as the algorithm ensures that each block works independently.

The only piece of common-shared memory between the threads is the 'seeds' array. This common interaction is only done in a **read-only** fashion. And since the seeds array is computed once in a serial fashion *before* the threads spin-up, we know that its value is written once and never altered after. Therefore there is no issue with multiple threads reading its value at the same time. For safety's sake we could add a read-write lock at this point, however the overhead of the lock will slow down the program for a case that is impossible given the design of the algorithm. As mentioned above, there is no real communication between sub-threads of the main thread (as they operate independently), with the only real communication occurring between the main thread and individual sub-threads, where sub-threads would update distinct shared-memory locations and for the main-thread to wait for sub-threads to complete and join to the main thread.

Results

As seen in the figure 4, at lower maxNums (that being 10 or 100), we see that the time actually increase as we add more threads. This seems to be due to the system overhead of managing threads (creation, tracking and freeing) and the design of the algorithm (which for smaller numbers creates many more partitions

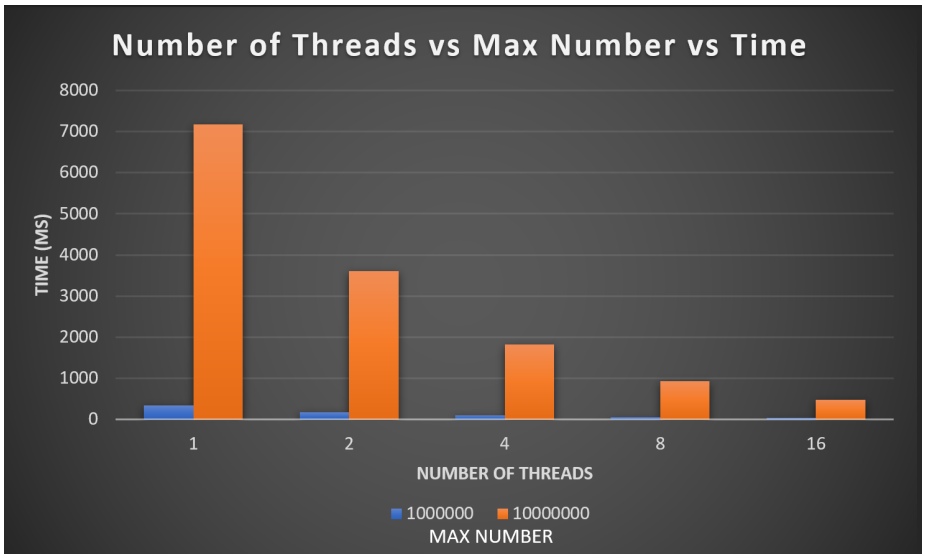

Figure 5: Max $\in \{10^6, 100^7\}$ and $T \in \{1, 2, 4, 8, 16\}$

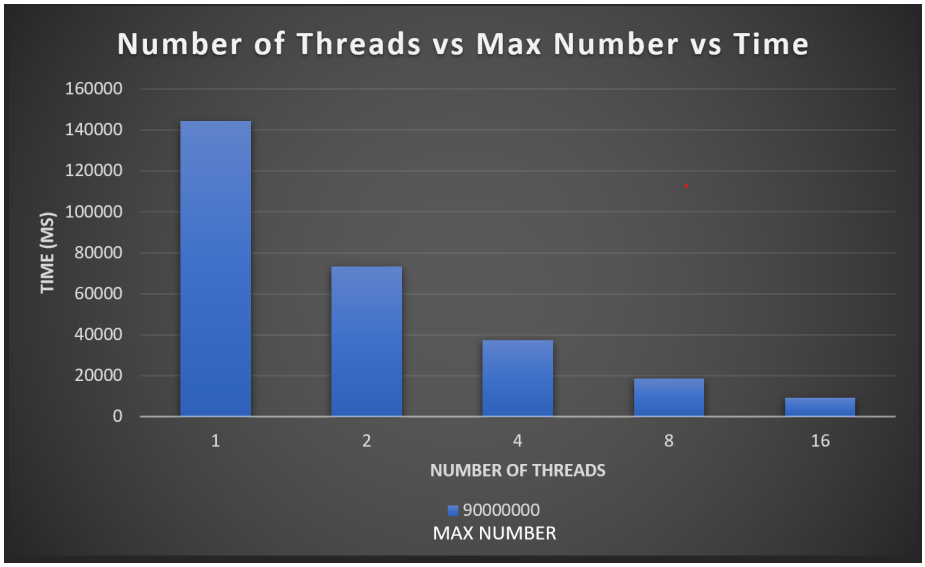

Figure 6: Max $\in \{10, 100\}$ and $T \in \{1, 2, 4, 8, 16\}$

that is actually required, thus threads are barely doing any work). However as we increase maxNum into the millions (in our testing 10^6 and 10^7) as seen in figure 5, we see that the time taken to compute the primes scales linearly as we increase the amount of threads (that is 2 threads takes about 50% of the time that 1 thread takes, with 4 threads taking 25% of the time).This result is further demonstrated when looking at the computational results when maxNumber is 9×10^7 as seen in figure 6. In conclusion, this exercise demonstrates the impact that an algorithm and chosen parameters ultimately decides when parallelism can make a difference (and by how much) to your application.

Mutual Exclusion Algorithms

Question 1

Does this protocol satisfy mutual exclusion?

Yes (in a dangerously fragile manner since it is not deadlock free. See question 3). An example of this can be see with two threads.

1. Thread 1 runs the 'lock' method. This in turn sets 'turn' to its ID and sets 'busy' to true. The lock method then returns and Thread 1 goes into the critical section.
2. Thread 2 now attempts to go into the critical section. Thread 2 runs the 'lock' method. 'turn' is now set to Thread2's ID, and it loops as it fails the 'busy=true' test.
3. Thread 1 now completes the critical section, and thus runs the 'unlock' method. This sets 'busy' to false.
4. Thread 2 now fails the 'busy=true' test and thus exits the loop. It sets 'busy=true' and fails the 'while (turn != me)' test, thus exiting the lock method.
5. Thread 2 now enters the critical section.

Question 2

Is this protocol starvation-free?

Yes in the sense that no priority is given to any of the waiting threads. And thus the quickest thread to set 'turn=me' when 'busy = false' will get the lock (could this leave one thread waiting for longer than others? Yes, but that would be due to the thread being slow in the race.) An example of this can be seen when looking at three threads (and ignores deadlocks).

1. Thread 1 arrives to the critical section first, it runs the method 'lock', sets 'busy' to true and returns from the method.

2. Thread 2 arrives at the critical section and attempts to run the method 'lock'. It sets 'turn=me' and passes the 'busy=false' check thus looping and waiting for the lock

3. Thread 3 arrives at the critical section and it too attempts to run the method 'lock'. It sets 'turn=me' and passes the 'busy=false' check thus looping and waiting for the lock.

4. Therefore Thread 2 and 3 are both waiting on the lock to release (i.e 'busy=false') and are equally setting 'turn' to their IDs.

5. Thread 1 completes the critical section and sets 'busy=false'

6. Thread 3 wins the race and fails the 'busy=false' check before thread 2 can change the 'turn' variable. Thus Thread 3 exits the loop and holds the lock.

7. Thread 1 now attempts to enter the critical section once again and it too attempts to run the method 'lock'. It sets 'turn=me' and passes the 'busy=false' check thus looping and waiting for the lock.

8. At this point we now have Thread 1 and Thread 2 waiting on the lock to release and are equally setting 'turn' to their IDs.

9. Thread 3 completes the critical section and releases the lock

10. The quicker thread between 1 and 2 grabs the lock and the other lock waits.

11. This continues as more threads attempt to enter the critical section.

Question 3

Is this protocol deadlock-free?

No. Due to the do while approached used, the 'turn' variable can be changed between setting 'busy=true' and the check 'turn != me'. This will occur when two threads try to execute the 'lock' method simultaneously

1. Thread 1 and Thread 2 attempt to enter the critical section.

2. Thread 1 is slightly quicker and sets 'turn=me' and 'busy=true'

3. At this point Thread 2 enters the second do-while loop and sets 'turn = me' .

4. Thread 2 will now constantly pass the 'while (busy)' check (leaving it to loop).

5. While Thread 1 will constantly pass the 'while (turn !=me)' check (leaving it to loop).

6. Thus both threads will never return from the 'lock' as each depends on the other, thus leading to a deadlock

Concurrent Data Structures

In this exercise we are supposed to take a sequential implementation of a linked list and make this concurrent. This will demand a sophisticated use of locks to properly synchronize the list.

Coarse Grained Locking using Mutex

To implement coarse grained locking. A lock on the whole list was implemented. This means that if any of the methods **insert**, **remove** or **count** is called then the list will become unavailable for other threads while this operation runs. This obviously will make it possible to have multiple threads operating on the list but it also leaves threads waiting while another one operates on it.

To implement this in C++, we updated the sorted list class to have a mutex as a class variable.

```
template<typename T>
class sorted_list {
  std::mutex li;
  node<T>* first = nullptr;
/* ... class continues ...*/
```

Listing 2: Sorted list with std::mutex

And the lock on the list is locked and unlocked like so:

```
    /* insert v into the list */
    void insert(T v) {
        li.lock();
        /* first find position */
        node<T>* pred = nullptr;
        node<T>* succ = first;
        while(succ != nullptr && succ->value < v) {
            pred = succ;
            succ = succ->next;
        }

        /* construct new node */
        node<T>* current = new node<T>();
        current->value = v;

        /* insert new node between pred and succ */
        current->next = succ;
        if(pred == nullptr) {
            first = current;
        } else {
            pred->next = current;
        }
        li.unlock();
    }
```

Listing 3: Coarse-Grained Insert

With a similar implementation for the other methods, the whole method is locked and the same locked is used for all methods, in essence, the whole list is

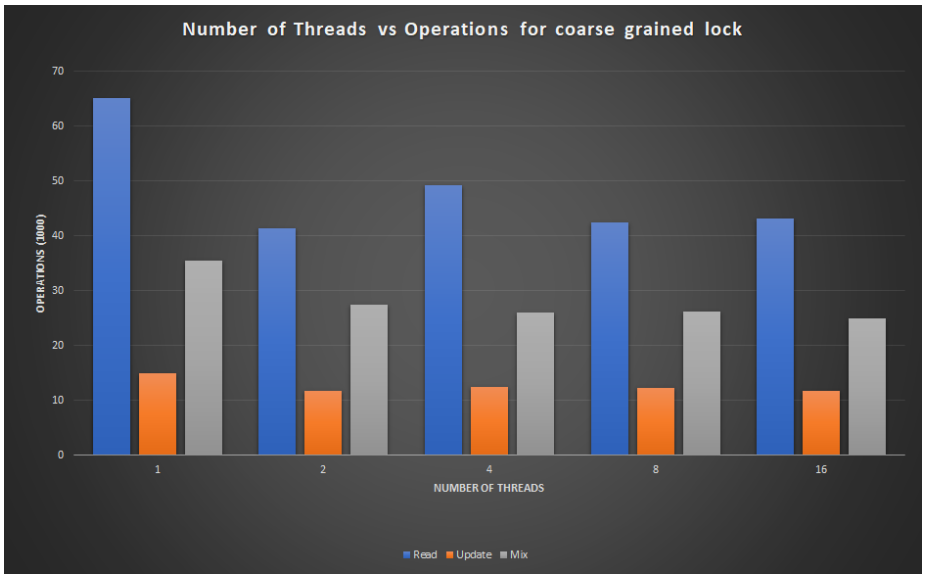

Figure 7: Coarse-grained locking with $t \in \{1, 2, 4, 8, 16\}$

owned by a single thread. This implementation works and multiple threads are able to run. From simulations we see that the number of operations decrease as more and more threads are added. This makes sense since there now is introduced overhead from threads as well as the locking locks out every single thread other than the one that is currently in the critical section. This can bee seen in figure 7.

Coarse-grained locking with tatas

Here we do the same implementation of locking but the lock we will use is a Test-And-Test-And-Set (TATAS) lock. This has been implemented by introducing a class called TATAS

```
1  class tatas {
2    private:
3      std::atomic<bool> l;
4
5    public:
6      tatas() {
7        l = false;
8      }
9
10     void lock() {
11       do {
12         while (l) continue;
13       } while (l.exchange(1));
14       return;
15     }
```

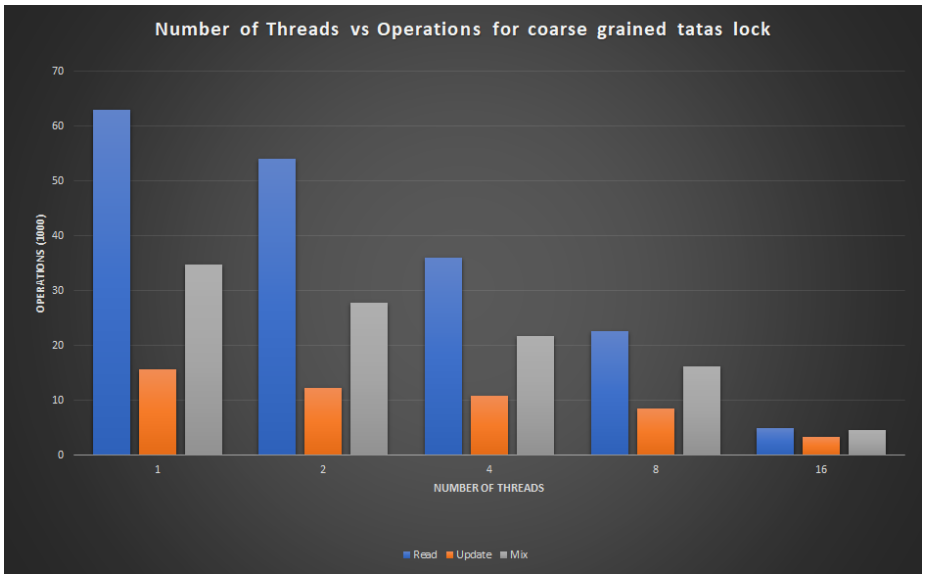

Figure 8: Coarse-grained TATAS locking with $t \in \{1, 2, 4, 8, 16\}$

```
16
17      void unlock() {
18        l.exchange(0);
19      }
20  };
```

Listing 4: TATAS Class

The implementation for course-grained TATAS just replaces the mutex in the linked list with this class and calling its lock and unlock method. The locking works by having a atomic boolean. When a thread calls 'lock' it continually observes the atomic bool l and when this becomes false. It checks the value of the boolean with the exchange method. This method returns the value of the atomic boolean, then tries to change it. So if it has changed to true in the time before the first check. The do-while loop catches this and it goes back. Unlocking is simple and just calls exchange without any further checks.

This implementation works. But when we visualize the performance of this we see that the performance gets even worse. This is due to the even more extensive waiting that threads have to go through in this program as well as the locking being coarse grained. This is shown in figure 8.

Fine-grained locking with Mutex

Our understanding of fine-grained locking is as follows:

- To address the challenges seen in the previous sections, another way of

calling threads and to synchronies them

- Fine-grained has been interpreted by us to mean that each node in the list should have their own locks, and locking nodes that is operated on.
- This should allow different threads to concurrently operate on different parts of the linked list independently.
- This is a non-trivial task since when doing the search through the list. The locking and unlocking of threads must be done correctly to ensure that no nodes are locked when they should not be.
- This is a non-trivial task since our implementation must ensure that the data structure remains intact and in order.
- Our idea. When searching through the list, one points to two nodes at a time and update these as one moves through the list. When the node that is last of the two nodes has served its purpose (we no longer need to read its data), this should be unlocked.
- When inserting a node. The two nodes that it will be inserted into must be locked.

We have tried to do this by introducing mutexes in each node:

```
/* struct for list nodes */
template<typename T>
struct node {
  T value;
  node<T>* next;
  std::mutex l;
};
```

Listing 5: Node with lock

And here is how we implemented the insert method:

```
/* insert v into the list */
void insert(T v)
{
  /* first find position */
  node<T> *pred = nullptr;
  l.lock();
  node<T> *succ = first;
  if(succ) succ->l.lock();
  l.unlock();
  /* Check that first is not nullptr*/
  while (succ != nullptr && succ->value < v)
  {
    pred = succ;
    succ->l.unlock();
    succ = succ->next;
    if (succ != nullptr)
    {
      succ->l.lock();
```

```
    }
  }

  /* construct new node */
  node<T> *current = new node<T>();
  current->value = v;

  /* insert new node between pred and succ */
  current->next = succ;
  if (pred == nullptr)
  {
    first = current;
  }
  else
  {
    pred->next = current;
  }
  if (succ) succ->l.unlock();
}
```

Listing 6: Fine grained insert method attempt

The other methods implementation can be found in the attached code. Whilst the code compiles and executes successfully with a single thread, upon introducing ¿1 thread, data-races were present due to the incorrect order of locking and unlocking. Due to time limitations we were unable to debug this section further. However, based on what was observed in the test with a single thread, the performance of the program should increase in a noticeable fashion as the linked list can be interacted with by multiple threads simultaneously (provided operations are occurring on different sections of the list)

Fine-grained locking with TATAS

Similar to our results in the Fine-grained locking with mutex in section . The other methods implementation can be found in the attached code. And once again, whilst the code compiles and executes successfully with a single thread, upon introducing ¿1 thread, data-races were present due to the incorrect order of locking and unlocking.

Due to time limitations we were unable to debug this section further. However, based on what was observed in the test with a single thread, and with a knowledge of how TATAS locking functions, we can assume that the performance would be quicker than all coarse-grained implementations as threads increase, however it would be worse than the mutex lock above. This is presumed to be due to the extra locks and cache-busts which occur when a lock is released. The code that we have implemented is shown below.

```
template<typename T>
class sorted_list {
  tatas l;
  node<T>* first = nullptr;
```

```
/* ... class continues ...*/
```

Listing 7: Sorted list tatas lock

```
/* insert v into the list */
void insert(T v)
{
  /* first find position */
  node<T> *pred = nullptr;
  l.lock();
  node<T> *succ = first;
  if(succ) succ->l.lock();
  l.unlock();
  /* Check that first is not nullptr*/
  while (succ != nullptr && succ->value < v)
  {
    pred = succ;
    succ->l.unlock();
    succ = succ->next;
    if (succ != nullptr)
    {
      succ->l.lock();
    }
  }

  /* construct new node */
  node<T> *current = new node<T>();
  current->value = v;

  /* insert new node between pred and succ */
  current->next = succ;
  if (pred == nullptr)
  {
    first = current;
  }
  else
  {
    pred->next = current;
  }
  if (succ) succ->l.unlock();
}
```

Listing 8: Fine grained attemot with tatas locks.

Fine-grained locking with scalable queue lock

Similar to our results in the Fine-grained locking with mutex in section or Fine-grained locking with TATAS in section The other methods implementation can be found in the attached code. In this implementation we gote a segmentation fault. Due to time limitations we were unable to debug this section further. However, based on what was observed in the test with a single thread, and with a knowledge of how queue-locks operate, we can assume a substantial speed-up as the number of threads increase (superseding all other implementations thus

far). The cost does occur in the complexity of code written. Our implementation is listed below.

```
struct qnode {
  std::atomic<bool> locked = ATOMIC_VAR_INIT(false);
};

class CLHLock {
  private:
    qnode *tail;
    qnode *pred;
    qnode *mynode = new qnode();
  public:
    void lock() {
      pred = tail;
      tail = mynode;
      while (pred->locked.load()) {}
    }

    void unlock() {
      mynode->locked.store(false);
      mynode = pred;
    }
};

```

Listing 9: The sholution we had in mind for queue lock

References

[Wikipedia contributors, 2021a] Wikipedia contributors (2021a). Sieve of eratosthenes — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sieve_of_Eratosthenes&oldid=1040549678. [Online; accessed 22-September-2021].

[Wikipedia contributors, 2021b] Wikipedia contributors (2021b). Trapezoidal rule — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Trapezoidal_rule&oldid=1036645689. [Online; accessed 19-September-2021].