Ighor Tavares - f.tavares@wustl.edu

Funda Atik - fatik@wustl.edu

**CSE422S: LAB 2 REPORT**

**1. Resources Used**

We used LKD book as a resource.

**2. Module Design and Implementation**

Barrier Design

We used an atomic variable to determine the state of the barrier and a counter to keep how many threads reached the barrier. When a thread reaches the barrier, it increments the counter and checks whether the counter equals the number of threads. Otherwise, the thread regularly checks if the status of barrier state changes. A better design would be using a wait queue so that whenever a thread reaches the barrier and it is not the last one, it releases the lock and is added to the wait queue. When the last thread reaches the barrier, it broadcasts a signal to wake up all threads in the wake queue. We did not use mutex because it is slow and can sleep.

Prime Computation Function

We defined two different mutexes for the non-atomic module. The first mutex is used to protect the section when the current variable is updated and the second mutex is used for protecting accesses to the number array. By doing so, we aim to decrease the contention on mutexes. Each thread calls this function repeatedly until the all prime numbers are found. We defined a global integer variable to keep the current index and used a mutex to protect the critical section in which a new local index is determined according to the global current index. After a thread determined the local index and update the global current index, it performs the cross out operations. For non-atomic version, each access to the number array is protected by a mutex while for the atomic version we removed the mutex and convert the number array into atomic one.

Thread Function

The logic of this function like this:

first\_barrier()

while(!barrier1\_state)

prime\_computation()

second\_barrier()

while(!barrier2\_state)

Initialization

We allocated and initialized the counter and the number array. We initialized the current index variable and other atomic variables. Finally created and woke up all threads.

**3. Module Performance**

**3.1. Atomic Module**

We used 4 different number of threads (1, 2, 4, 8) and 5 different upper bounds (24, 28, 212, 216, 220). We did not experiment with larger upper bounds that 1 million because raspberry does not throw an error. Moreover, we did not use a high number of threads since the raspberry has only 4 cores and we did not observe any performance improvements. We calculated the average of 3 runs for each configuration.

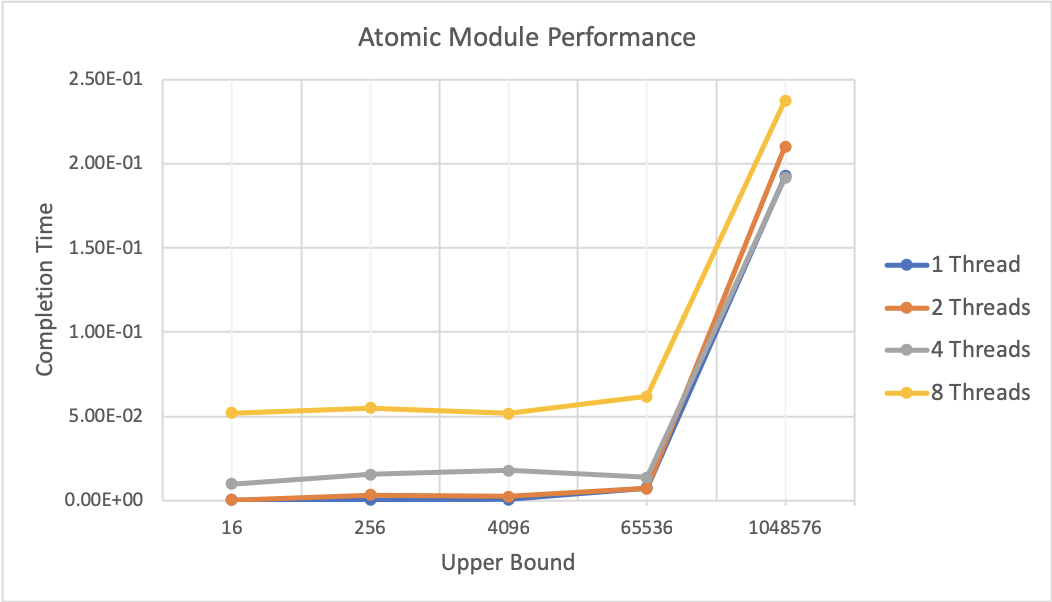


Figure 1. Timing results for atomic module

**3.2. Non-atomic Module**

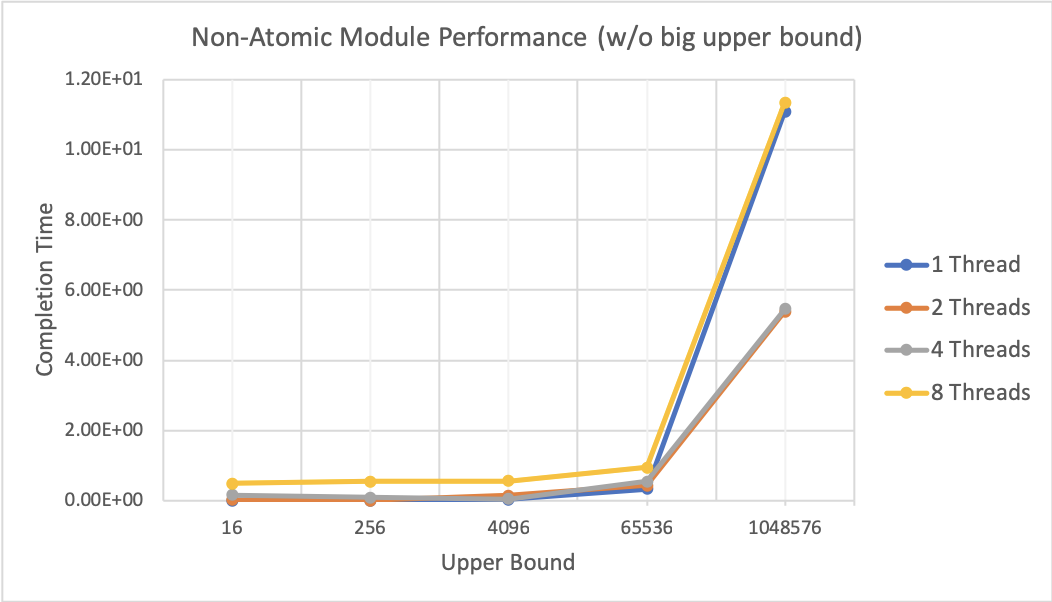
****

Figure 2.Timing results for non-atomic module

As can be seen from Figure 2, for the non-atomic module, all threads perform similar performances until when the upper bound becomes 65536. After that point, although the setting with 1 thread and the setting with 8 threads spend the nearly same time to complete, the difference for the timing results between execution with 2 threads and execution with 4 threads is negligible for all different upper bounds. On contrary, for the atomic module, the setting with 1 thread and the setting with 2 threads are almost identical and execution with 8 threads performs the worst performance among others. Moreover, we observe that the atomic module performs better than the non-atomic module. The reason would be using a mutex is more expensive than using atomic operation when contention is high for the same lock. In other words. for protecting the accesses to the number array, using a mutex will create a contention for competing to acquire the same lock repeatedly which slows down the performance. For non-atomic version, programs run with 2 threads and 4 threads respectively are faster than the programs run with 1 and 8 thread. Finally, we see the benefits of multithreading.

**3.3. Module Efficiency**

For both atomic module and non-atomic module, we compared the unnecessary cross out operations with upper bounds, and we notice that for all different threads spawned, they stay the same. This is expected since we implement that each thread “takes” the next free prime and its multiples, eliminating most of the repetitive check for previous crossed out primes multiples and we used a small number of threads. When we increase the number of threads, we see that module efficiency decreases and the number of the cross out operations might not be the same for given upper bound. Figure 5 is scaled logarithmically (by 10) because the number of cross-outs increases exponentially according to given upper bounds.

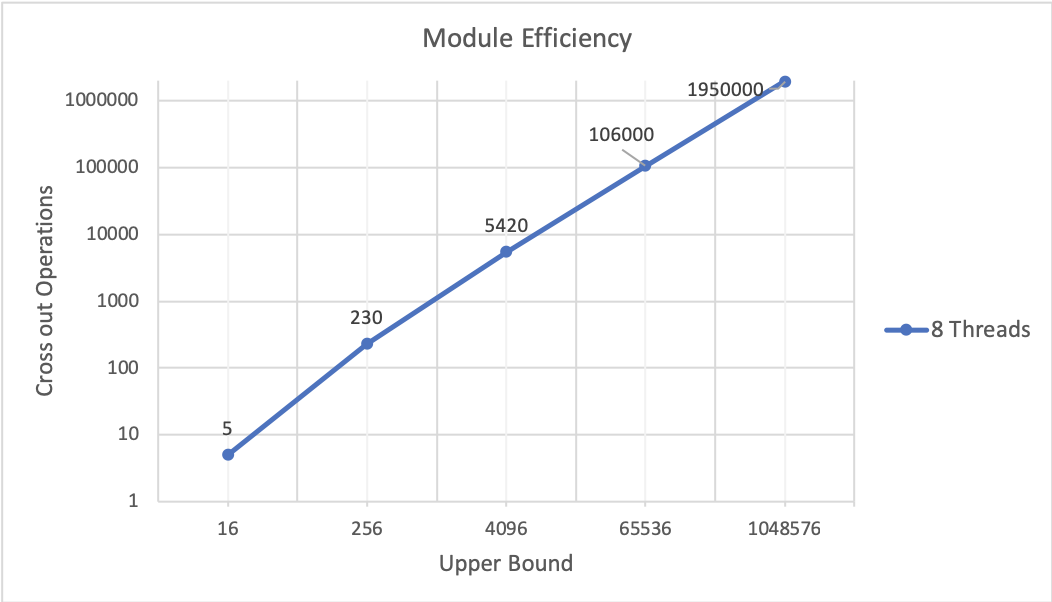
****

Figure 3.The efficiency of both modules

**4. Names of the Files with Plots of the Timing and Efficiency Results**

Figure 1: atomic\_compTime\_UpBound\_v1.png

Figure 2: anonAtomic\_compTime\_UpBound\_v1.png

Figure 3: module\_efficiency.png

Figure 4: atomic\_compTime\_UpBound\_v2.png

Figure 5: nonAtomic\_compTime\_UpBound\_v2.png

**5. Names of the Files with Interesting Screenshots and Traces**

Figure 5 and Figure 6 do not allow us to see any interesting pattern because the completion time for a big upper bound (220) is much higher than the others. Therefore, we excluded the big upper bound from our result and created other figures.

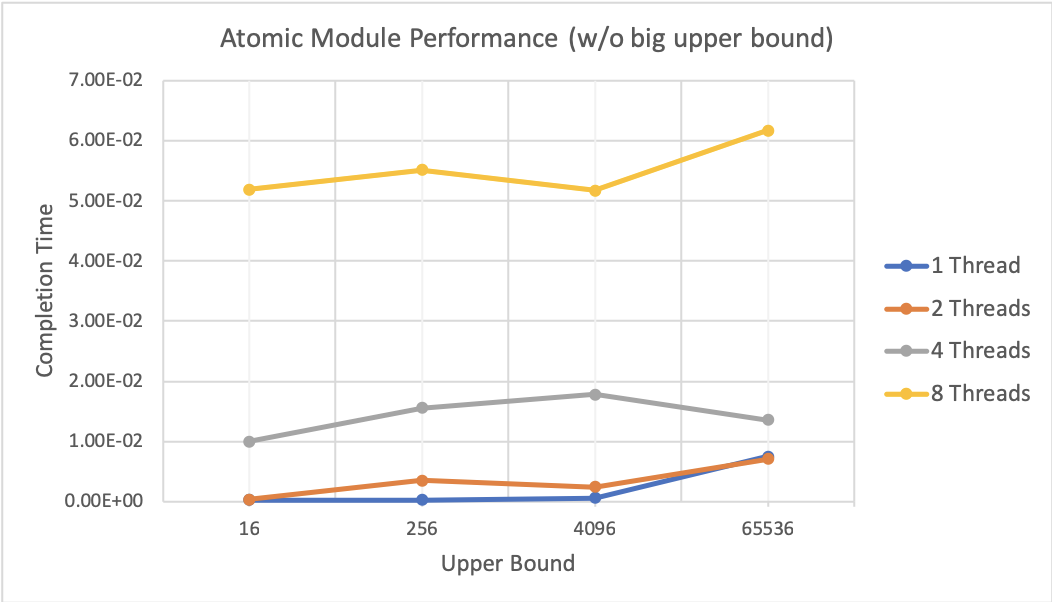


Figure 5. Performance results for atomic module without using big upper bound

One interesting pattern shown in Figure 5, by running the program with 4 threads in the atomic module, it seems to be decreasing in completion time as the number of upper bound increases (212). Also, it is possible to notice when both tests run with 1 and 2 threads reach a larger upper bound, they tend to take fairly the same amount of completion time. One more interesting pattern observed is running the program with a different number of threads, consist of fairly the same “shape”, where there is an alternation of computation time for small upper bound values.

We did not observe any performance improvements when we increase the number of threads. This is a little disappointing since this is multithreading application. Since the upper bounds are small and we used several lock mechanisms which create a contention when a lock is frequently trying to be acquired by many threads. More than one threads will compete with each other at the same time but only one thread can obtain the lock and do a meaningful work.

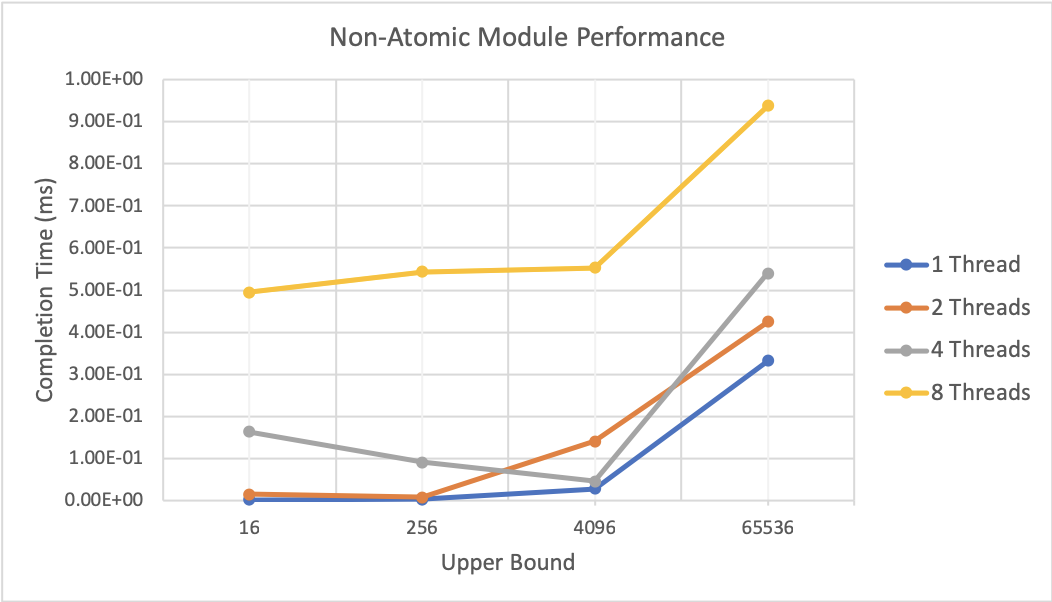
****

Figure 6. Performance results for non-atomic module without using big upper bound

As can be seen from Figure 6, one interesting pattern is observed when the program runs with 4 threads. It is observed for a larger upper bound in our test, all thread cases are expected to take longer to be calculated, however, running with 4 threads first is faster and essentially becomes relatively slower compared to running the program with 1 and 2 threads. Running the program with 1,4 and 8 threads all programs presented similar behavior as upper bound becomes bigger, with the exception of running with 2 threads, which gradually took longer to complete the program with a smaller upper bound. For being a multithreaded application, it was expected to be an increase of performance, however, due to our raspberry pi limitations of 4 cores, we did not notice any performance improvements when increasing the number of threads.

**6. Insights and Questions**

Because our raspberry pi consists of only 4 cores and we used lock mechanism inefficiently running tests with a greater number of threads would not generate a noticeable change in performance. Running the program with 8 threads, with a smaller number for upper bound, there was no performance difference for both atomic and non-atomic modules.

It was possible to test that there is a limit upper bound that can be calculated by our program, where we approximately estimated to be above 1 million.

Our atomic modules provided a noticeably better performance in calculating larger upper bounds than our non-atomic modules. This is expected for with atomic modules, the program would not need to perform “lock” operations which slows down the total completion time as the number of the upper bound increases.

**7. Suggestion for How to Improve the Assignment Itself**

It would be interesting to compare the completion time for these modules with userspace code, where we would perform similar operations (of using locks, and creating threads, etc) and see if there is any noticeable performance difference.

**8. Time Spent**

About 18 hours