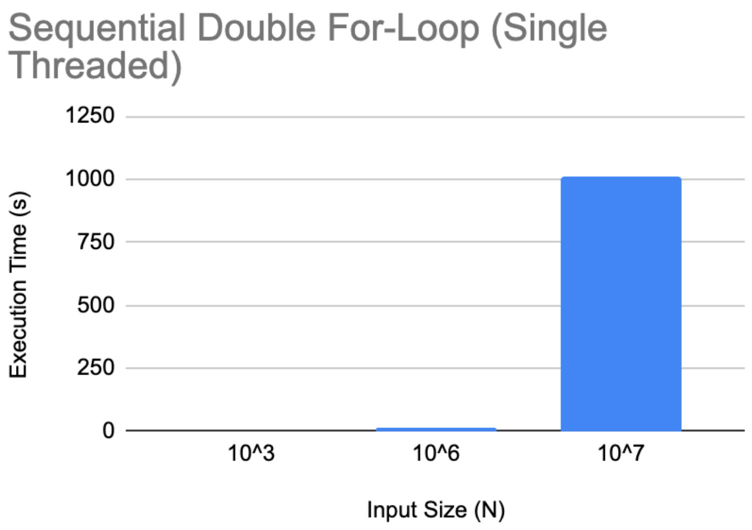
Tal Derei

CSE 411: Project 1
TBB For Parallelism

Sequential Execution (Baseline)

The codebase contained in prime.cc initially calculated whether a single input was prime. Changing the algorithm to calculate all primes in a given range from [0-input] required implementing a doubly-nested for loop. The sequential execution ran in quadratic time-complexity O(n^2), and as the input increased 10x in size from 10^6 to 10^7, the execution time performed 100x slower at 1000 seconds (~17 minutes). The baseline, and subsequent concurrent testing, was done natively on Apple M1 (8-core cpu, 8-core gpu).

Prime Size	Exec Time (M1)	Structure
10^3	0.000158583	O(n^2) for-loop
10^6	11.8367	
10^7	1013.79	



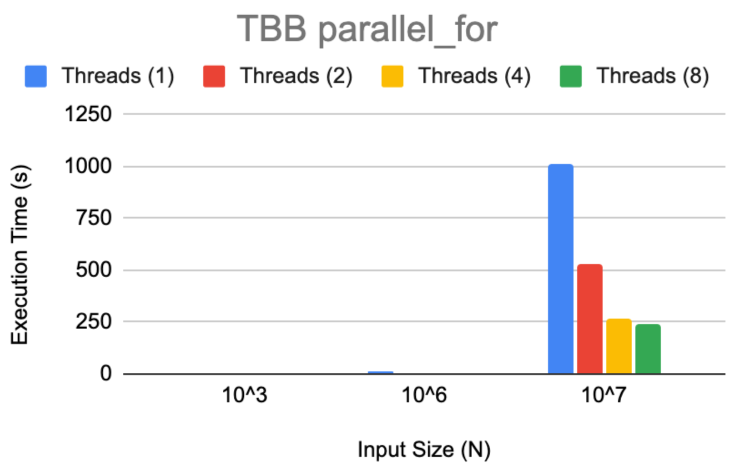
TBB Parallel_For

Intel's TBB Building Blocks library was used to convert the sequential double-for loop into a dynamically parallelized operation. TBB breaks the iteration space into chunks, and runs each chunk on a separate thread. Chunk sizes are automated by auto_partitioner to achieve load balancing across the threads and limit contention overhead. Simple_partitioner gives a more fine-grained control, allowing you to specify the grain-size associated with the task. The smaller the grain size, the larger the chunking overhead. And the larger the grain size, the larger the parallelism overhead (i.e. reduction in parallelism). TBB optimizations include utilizing simple and auto partitioners, as well as fine-tuning the grain sizes as a function of the input size.

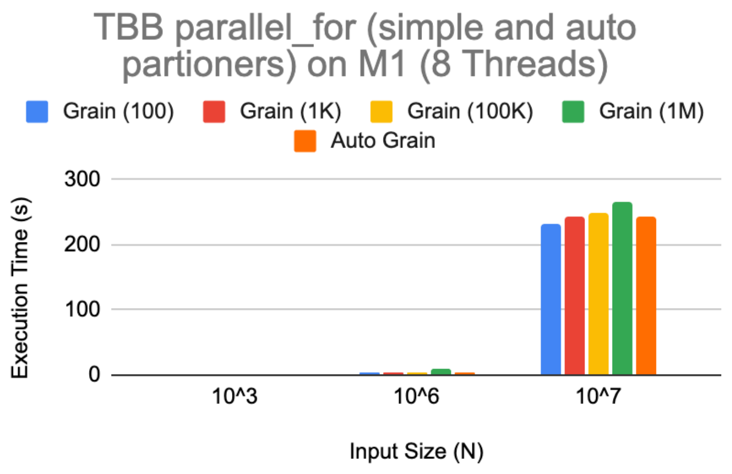
As expected, TBB *improved* the execution performance. Looking specifically at 10^6 and 10^7 sized inputs, the auto partitioner performed the best on 8 threads at 4.3x and 4.19x speedups respectively as compared with the sequential baseline. Note, TBB defaults to 8 threads in the thread-pool, and the task_scheduler_init() object is deprecated.

Of particular interest is a 10^6 input with a simple_partioner and 1M grain (second plot): the execution time was an outlier with a speed of 8.83s. Since the grain size was 1M executing on 8 threads, and the input size was the same as the grain size, there was a significant reduction in parallelism since this is effectively a single-thread sequential execution.

Prime Size	Threads (1)	Threads (2)	Threads (4)	Threads (8)
10^3	0.000158583	0.0304612	0.0358365	0.00348992
10^6	11.8367	6.47832	3.60892	2.74948
10^7	1013.79	525.659	269.965	242.724



Prime Size	Grain (100)	Grain (1K)	Grain (100K)	Grain (1M)	Auto Grain	Partitioner	Structure	Thread
10^3	0.021954	0.00318521	0.00418879	0.004517	0.00348992	Simple/Auto	TBB parallel_for	8
10^6	2.82796	2.81203	3.16057	8.82537	2.74948			
10^7	233.368	244.218	249.579	266.737	242.724			

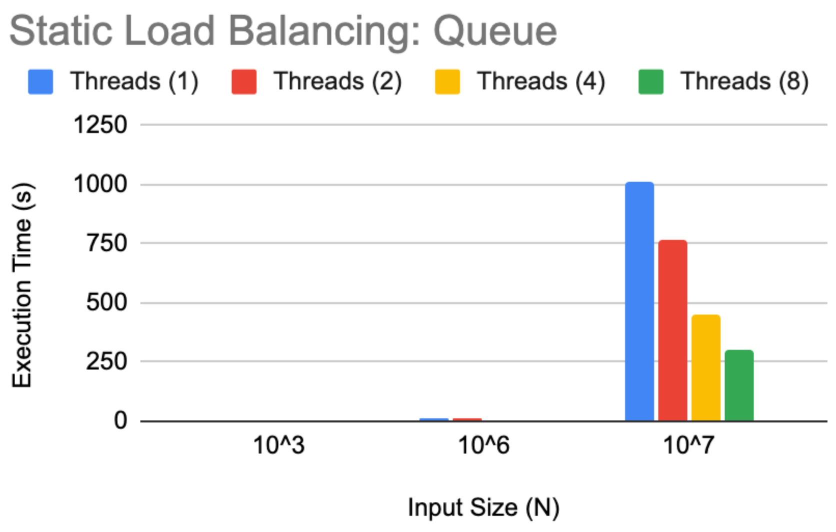


Static Load Balancing:

For static load balancing, I implemented a struct ‘queue_struct’ containing a queue and mutex for locking the queue. A queue was implemented to statically partition and buffer the input before processing it. Included in the execution time is the time spent initializing and populating the queue with integers. The ‘spawn_threads()’ function declares a vector of threads and calls the ‘tasks()’ function for each thread. The tasks() function: [1] locks the global queue, [2] acquires an integer from the front of the queue, [3] pops the integer from the front of the queue, [4] unlocks the queue, and [5] calls the ‘calculate()’ function for calculating the prime values in the specified range asynchronously for each thread. Static Load Balancing statically partitions the queue into workloads equivalent to queue.size() / # THREADS (i.e. if your input is 1000 and 8 threads, every thread will calculate the prime numbers for a 125 number range in the queue). Load balancing is necessary to split up the workload amongst the threads in order to avoid threads being idle, and make sure the other threads aren’t bottlenecked by a single thread.

The results clearly show that the performance of static load balancing is *equivalent* to the optimizations made by TBB on 8 parallel threads.

Prime Size	**Threads (1)**	**Threads (2)**	**Threads (4)**	**Threads (8)**	**Structure**	**Implementation**
10^3	0.000219792	0.000345916	0.000311791	0.000458875	Static LB	Queue
10^6	11.863	9.05548	5.65396	3.50773		
10^7	1012.02	762.707	454.207	305.04		

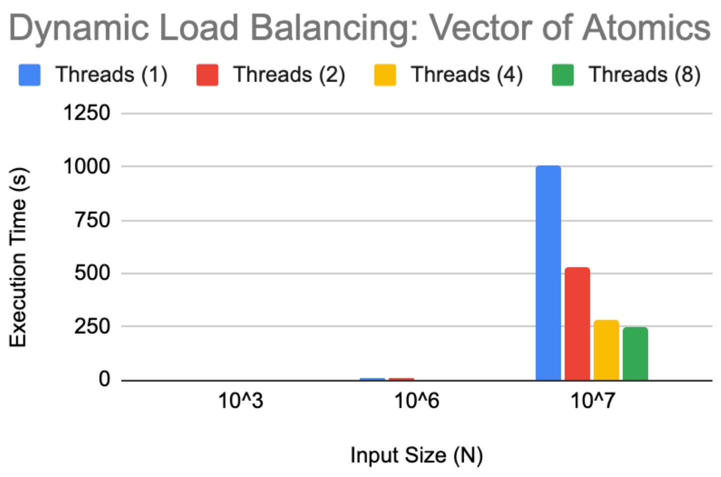


But in this implementation, the bottleneck is the queue itself! There’s a global queue locked by a single lock, and every thread must wait to access the queue and retrieve their ‘tasks’ before calculating the prime numbers asynchronously. So with only one queue, only one thread can access and lock the shared queue at a time. Although the time spent by every thread in the queue is relatively short in association with the time spent calculating the prime numbers, an interesting solution might be rather than having all 8 threads accessing the same shared queue, each thread will access their individual queue in parallel. Since the queue and lock were implemented in a struct, it would be as simple as instantiating 8 separate struct objects (i.e. 8 queues and 8 locks), and assigning each to a separate thread. Calling the calculate() function asynchronously to calculate the prime numbers will remain unchanged. Another solution might be to simply just partition the array itself rather than implementing a queue.

<u>Dynamic Load Balancing:</u>
For dynamic load balancing, I rewrote my blocking queue from static load balancing by using atomic operations to eliminate the mutex around the critical section of code. The new data structure is a *vector of atomics,* meaning multiple threads can access the same data structure at the same time, but only one thread can access a particular element at a time. Since you cannot have a vector of std::atomic<int> because it's not copyable/movable, I implemented a vector of unique_ptrs to atomic<int>, essentially created a vector of pointers to atomics (i.e. std::vector<std::unique_ptr<std::atomic<int>>> vector). There are a number of advantages to this: [1] Every operation is passed by 'pointer' rather than by 'value', and [2] vector elements are protected by atomics rather than blocking locks. This in turn makes the execution times faster, in theory, than static load balancing. Other improvements include: [1] randomizing the vector elements to ensure every thread accesses "equally" difficult possible prime numbers, [2] atomic counter uses fetch_add(std::memory_order_relaxed) for greater optimization by relaxing the memory ordering constraints. Moreover, store() and load() operations associated with atomics weren't necessary since we're pre initializing the vector sequentially.

The results clearly show that the performance of dynamic load balancing is *<u>better</u>* than the static load balancing for all thread counts.

Prime Size	Threads (1)	Threads (2)	Threads (4)	Threads (8)	Structure	Implementation
10^3	0.000308667	0.000992875	0.000378708	0.000706333	Dynamic LB	Vector of Atomics
10^6	12.1229	6.18775	3.65246	2.79452		
10^7	1005.02	526.818	277.978	247.203		

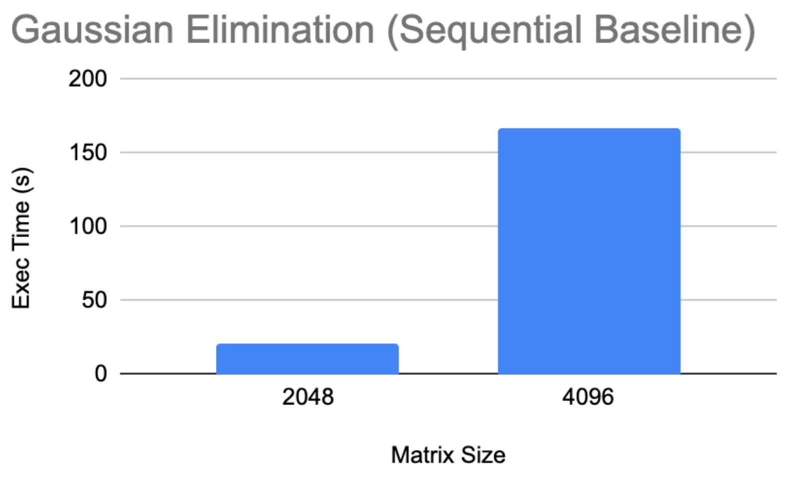


Some potential optimizations in the future include implementing the tbb::concurrent_vector data structure, alongside some memoization techniques for saving intermediary computations. These techniques weren't tested in this report.

<u>Gaussian Elimination (Sequential Execution):</u>
The sequential execution runs in cubic time O(n^3) in a triple-nested for loop. The baseline, and subsequent concurrent testing, was done in an isolated Ubuntu distribution of linux in a docker container.

It's important to note that the verification initially failed on any non-linux non-isolated environment, but worked in a linux distribution inside of docker when casting the range to a *double*. This was the case for the *<u>SEQUENTIAL</u>* execution. But the verification issue persisted for the *<u>PARALLEL</u>* execution.

Matrix Size	**Exec Time (Linux)**	**Structure**
2048	20.7742	O(n^3) loop
4096	166.395	



As expected, doubling the matrix size results in an 8x time-differential (slowdown) in execution time.

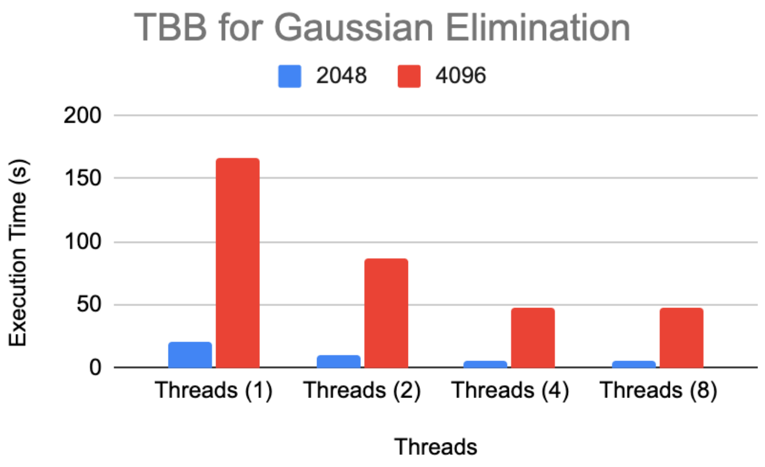
<u>Gaussian Elimination (Parallel/Concurrent Execution):</u>
In regards to optimization techniques employed to the codebase, exploiting the multicore architecture was achieved by: TBB's parallel_for to parallelize the triple-nested for loop. In regards to TBB, parallelizing the for loops in the initialization of the arrays (i.e. InitializeFromSeed() function) was pointless since there's more thread contention and overhead than execution speed. This was verified by placing probes in the function, and comparing the parallelized double for loop in that function against the baseline execution, and no speedup was achieved. And you can't parallelize the for-loops in check() because there's heavy thread

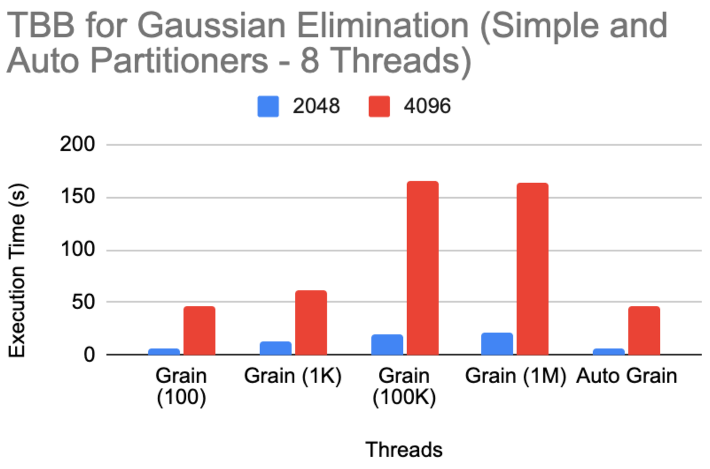
synchronization that you'd need to implement. All verifications across all threads would need to synchronize their solutions, and to verify that at least one of the threads didn't fail to verify their chunk of the solution. And can't parallelize the second double-for loop in gauss() function since there's a dependency that may lead to data race.

The majority of the TBB optimizations occur in gauss() since that's where the matrix multiplication occurs, and that's where we're timing the execution of the program. Parallelizing ONLY the outer loop in the triple-nested for loop yielded the best performance, since parallelizing all 3 loops individually resulted in slower execution time.

Matrix Size	**Threads (1)**	**Threads (2)**	**Threads (4)**	**Threads (8)**	**Structure**
2048	20.7742	11.0368	5.79981	6.13229	TBB parallel_for
4096	166.395	86.3239	47.8327	47.2416	

Matrix Size	**Grain (100)**	**Grain (1K)**	**Grain (100K)**	**Grain (1M)**	**Auto Grain**	**Partitioner**	**Structure**	**Threads**
2048	6.39312	12.8448	20.1927	20.791	6.13229	Simple/Auto	TBB parallel_for	8
4096	47.2021	61.4753	164.89	164.408	47.2416			





The results above confirm a few assumptions: matrix multiplication is a highly parallelizable task, where the speed up in execution time is the greatest for a 4096x4096 matrix operated by 8 concurrent threads. For 8 threads, the 2048x2048 and 4096x4096 matrices performed at 3.4x and 3.53x speed ups respectively in the first plot. Note, the variance between the sub-trials was non-negligible.

The second plot also shows that manually tuning grain size for matrix multiplication on 8 concurrent threads yields WORSE results than simply allowing the auto_partitioner() to determine the grain size dynamically for each iteration of the matrix. Manually tuning the grain size with simple_partitioner yielded worse performance results due to awkward chunking. Moreover, other c++ optimizations such as loop unrolling techniques were employed to improve locality in the for loops. I was experimenting with some unrolling sizes and tuning it for the best performance. The best iterations seem to be about half of the number of threads running (i.e. with 8 threads, increment += 4). I'm not sure why, but it seems to accelerate the computation time.

Lastly, SIMD operations were ***NOT implemented as an optimization technique since there were some issues associated with the header files on my Mac M1 machine.***