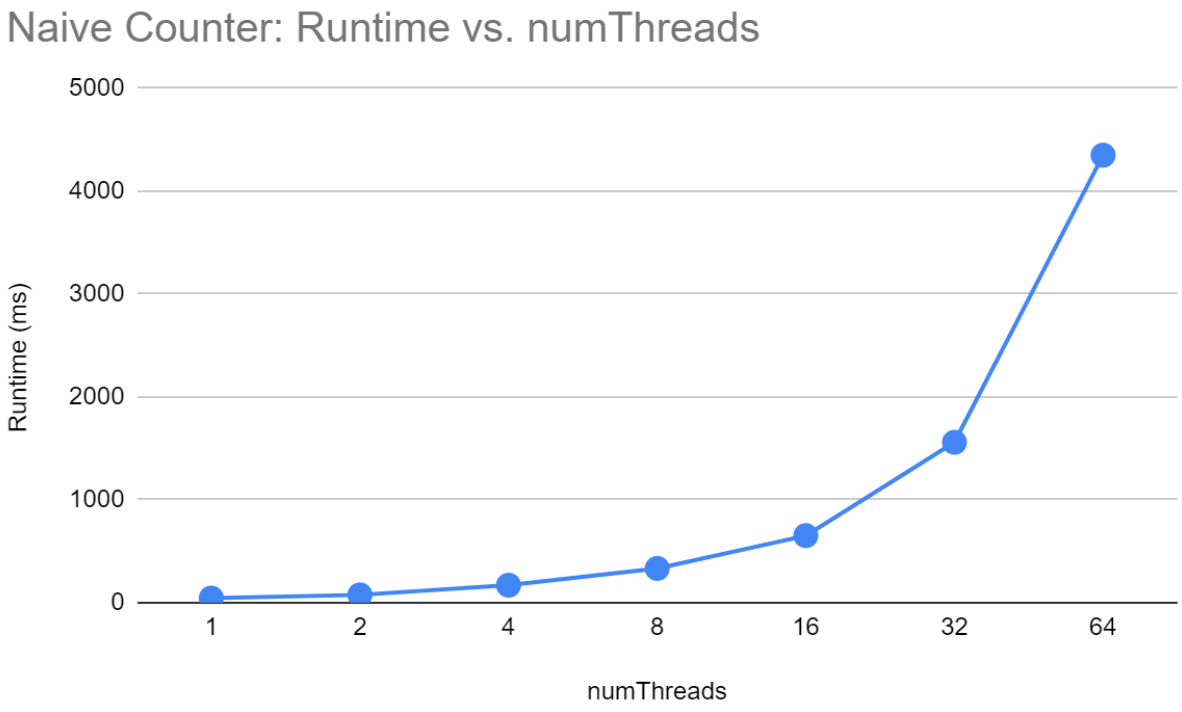
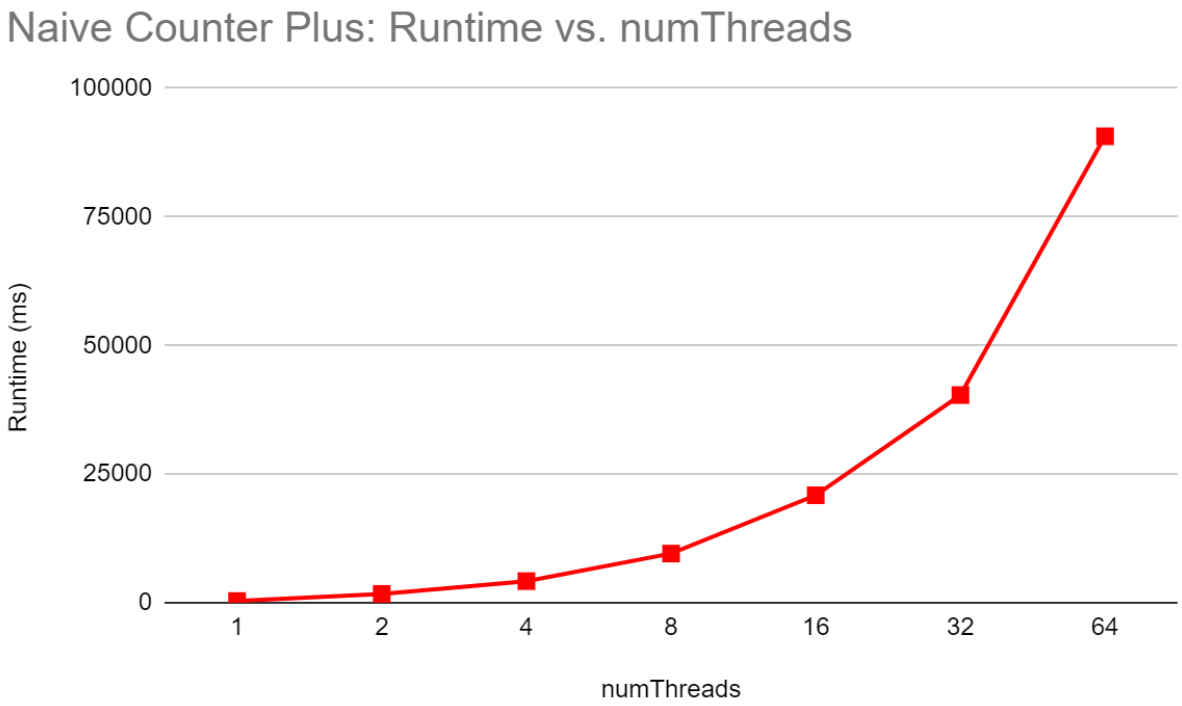
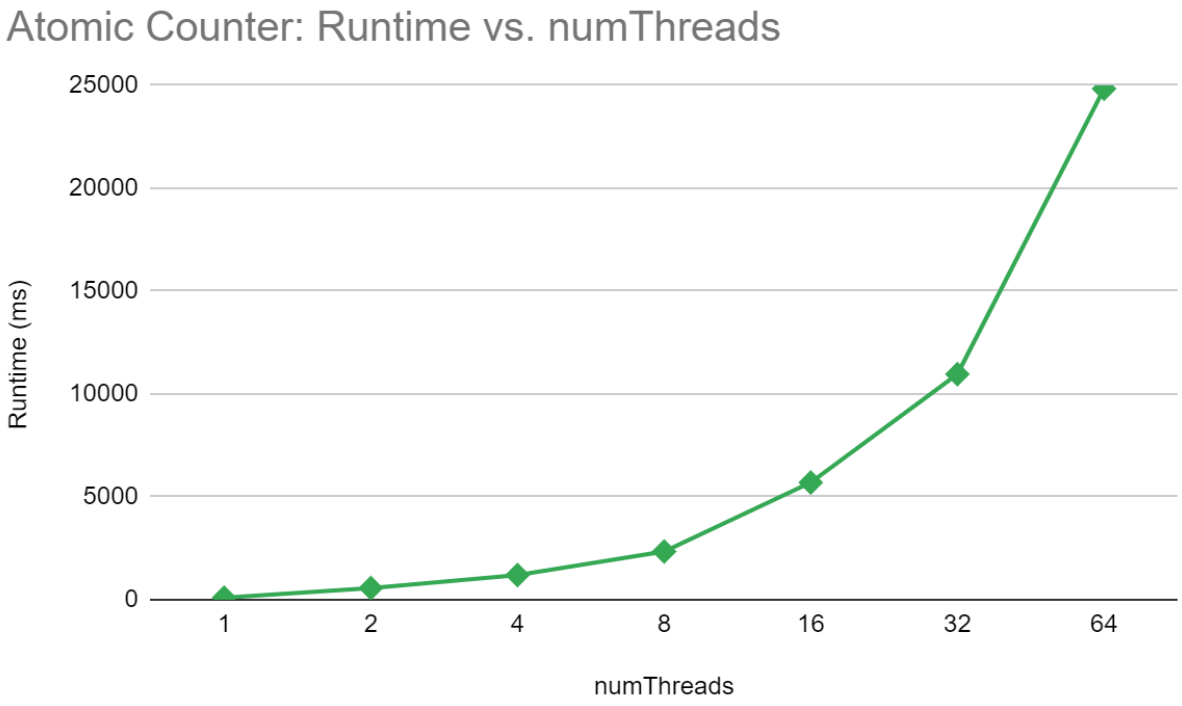
Operating Systems 416 - Project 3
Sagnik Mukherjee (sm2367)
Prof. Nath
November 16, 2021

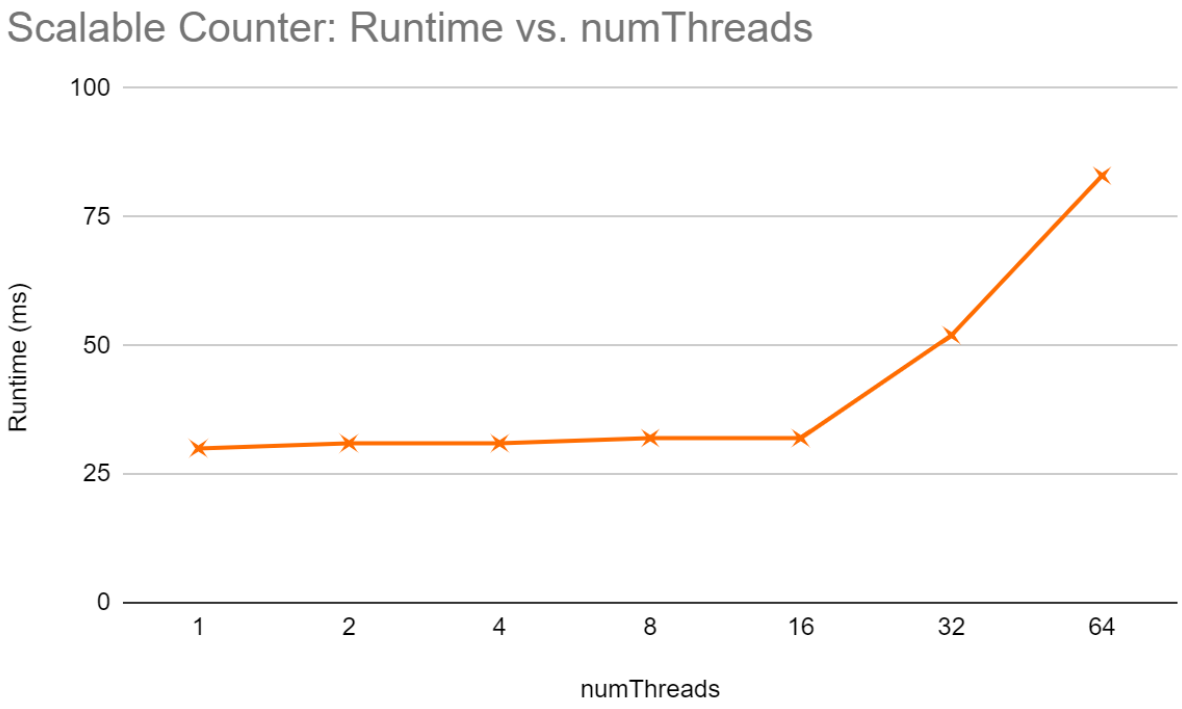
Performance tests were run on iLab2.cs.rutgers.edu using x2Go Client, SSH protocol.

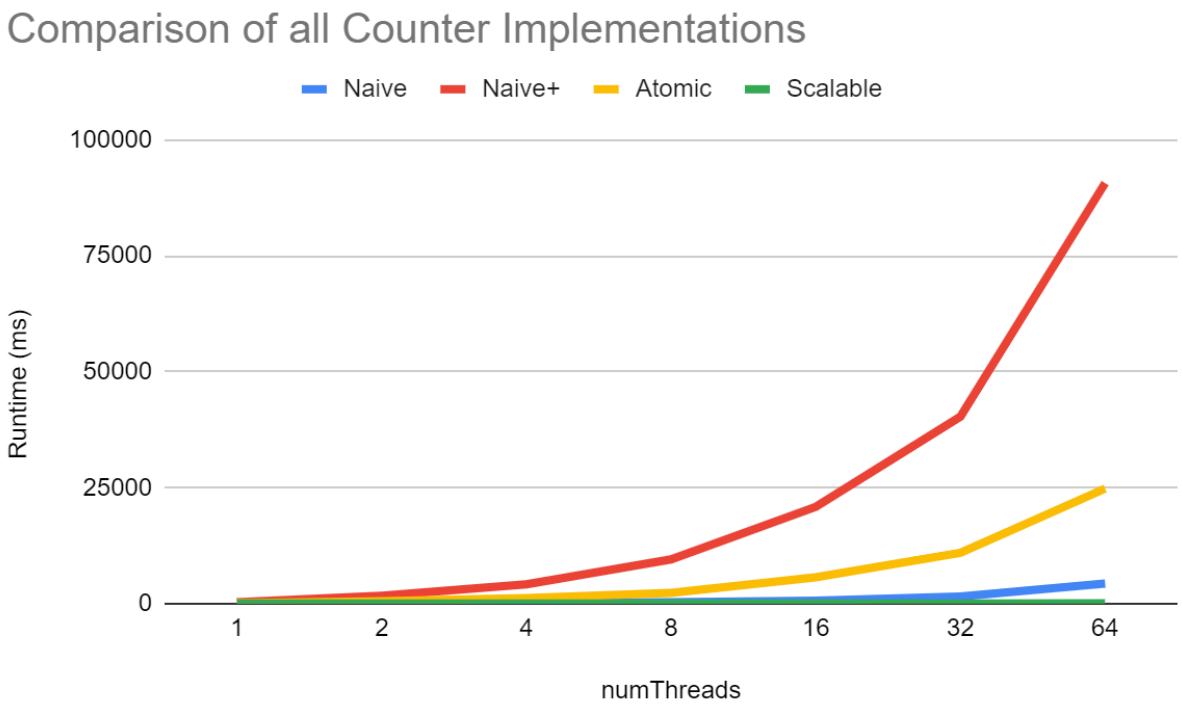
Counter Type	**Naive Counter**	**Naive +**	**Atomic**	**Scalable**
numThreads	runtime (ms)	runtime (ms)	runtime (ms)	runtime (ms)
1	45	322	97	30
2	76	1714	565	31
4	170	4192	1197	31
8	333	9550	2341	32
16	652	20887	5696	32
32	1559	40362	10959	52
64	4350	90667	24833	83




Naive Counter Plus: Runtime vs. numThreads
100000
75000
50000
25000
0
Runtime (ms)
1
2
4
8
16
32
64
numThreads


Atomic Counter: Runtime vs. numThreads
25000
20000
15000
10000
5000
0
Runtime (ms)
1
2
4
8
16
32
64
numThreads


Scalable Counter: Runtime vs. numThreads
100
75
50
25
0
Runtime (ms)
1
2
4
8
16
32
64
numThreads


Comparison of all Counter Implementations
Naive
Naive+
Atomic
Scalable
100000
75000
50000
25000
0
Runtime (ms)
1
2
4
8
16
32
64
numThreads

Question 1: Why does the Naive Counter have a huge difference from the correct value?

Without proper synchronization of threads and assuring atomicity when incrementing the globally shared counter variable, multiple threads access the pointer value haphazardly, and update the current value of the global counter several times, but only one might actually get saved. A thread reads the counter value, modifies, and then writes the new value. Many threads get and increase the value, but when writing the new value, it's as though only one thread actually had any effect, so the final result yielded by the naive counter is significantly off, usually a smaller value, than the expected result.

Question 2: Why would Atomic Counter be superior to Naive Counter Plus?

The atomic counter increments the global counter in an atomic, uninterrupted fashion. It accesses the counter value in register storage, and blocks other threads from accessing that same register while it increments the counter value. In contrast, the naive-plus implementation inefficiently uses mutual exclusion locking every single time it wants to perform a single read-modify-write increment to the global counter. Locking/unlocking every single loop iteration in the threadFunc, by design, will result in a massive and unnecessary waste of time, which the atomic implementation avoids entirely.

Question 3: Why is Atomic Counter still not able to achieve better performance than Naive Counter?

Despite not having the "stop-start" limitation of mutexes like naive-plus, the atomic counter will still be limited, in the long run, by the actual act of incrementing the global counter's value at a given register. One thread accesses the register at a given moment. The naive implementation, though it is noticeably inaccurate, allows multiple threads to read-modify-write on the same counter memory address, without locking to avoid race conditions, so no atomic access means a faster overall runtime as threads run together, and the overall program is able to run to completion in a fraction of the time of the atomic counter. This does not seem like a fair comparison though, since the naive counter yields a wildly inaccurate result in the end, so its shorter runtime is, honestly, pointless. When designing an implementation for a given technical problem, a program should first strive to achieve an accurate result (or an acceptable one, with reasonable error bounding), then it can be further improved on from there, in the interest of linear or linearithmic time-complexity.