Performance Profile of Synchronization Constructs

Problem Statement: We discussed various synchronization primitives such as busy-waiting, mutexes, conditional variables, barriers, and read-write locks. You are required to design experiments (or test cases), execute them and measure the performance of each of these synchronization constructs. Please note that the performance overheads depend on multiple factors such as library version, hardware and system capabilities, computation performed on each thread, number of cores, number of threads and so forth. Therefore, the performance measures are not scalar numbers, but instead is a profile over varying quantities. Use the performance metrics such as speedup, and efficiency discussed in the class if you find them useful. Since there is significant run-to-run variations in multi-threaded programs, ensure you repeat the same experiment at least 5 times while measuring the performance. Report the maximum, minimum, and average time taken for every study you perform.

Kaushal Kishore
111601008

Contents

Section 1

Hardware Specifications

CPU Info: (Used the linux command `**lscpu**`)

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPUS(s)	4
Thread(s) per core	2
CPU family	6
Model name	Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	3072K

Memory Info: (used the command `dmidecode -t memory`)

Total Width	64 bits
Data Width	64 bits
Size	4096 MB
Type	DDR3
Speed	1600 MHz

OS Info:

Base System	Debian GNU/Linux 9 (stretch) 64-bit
Kernel Info	4.9.0-6-amd64 #1 SMP Debian 4.9.88-1+deb9u1 (2018-05-07) x86_64 GNU/Linux

Programming Language and Libraries:

C++ STL multithreading libraries

Section 2

Benchmark: Sequential

Before moving on to the multithreaded programs we will first measure the performance of the sequential program. The program chosen for this experiment is to compute the global sum of the array (of type double).

Input Parameters:

- Size of the array

Some important things to note here is that the global sum might cause overflow hence we will try to generate the elements of the array in a way so that they do not cause any overflow and hence (if by any chance) will not be the cause of any performance loss/gain.

The table given below records the result of the experiment. All time has unit **ms**.

Array Size	**T1**	**T2**	**T3**	**T4**	**T5**
100000	0.649	**1.530**	0.655	1.538	1.093
1000000	6.433	6.839	**9.628**	8.964	7.966
10000000	**73.601**	71.968	70.227	71.783	71.895

Section 3

Benchmark: Mutex

Moving on to the multithreaded programs we will repeat the same experiment of computing the global sum using more than one threads, with different array size. As mentioned in previous section all the time are given in **ms** units.

For thread count = 2

Array Size	T1	T2	T3	T4	T5
100000	**0.518**	0.437	0.423	0.415	0.390
1000000	3.490	3.436	3.483	4.664	**6.984**
10000000	**46.933**	42.783	40.289	37.078	36.419

For thread count = 4

Array Size	T1	T2	T3	T4	T5
100000	1.407	0.498	0.528	1.398	**1.633**
1000000	3.716	3.538	3.561	2.582	**4.148**
10000000	**36.347**	27.066	33.033	31.933	34.240

For thread count = 6

Array Size	T1	T2	T3	T4	T5
100000	**0.639**	0.453	0.426	0.483	0.464
1000000	**4.180**	3.029	3.505	2.885	2.936
10000000	**35.656**	34.978	30.885	34.652	33.952

For thread count = 8

Array Size	T1	T2	T3	T4	T5
100000	**0.688**	0.597	0.541	0.572	0.604
1000000	**3.601**	3.022	2.971	3.425	2.702
10000000	24.516	**37.132**	33.512	32.700	36.033

SpeedUp

Size ↓ Threads→	2	4	6	8
100000	**2.503**	1.000	2.217	1.820
1000000	1.806	2.270	2.409	**2.533**
10000000	1.766	**2.210**	2.113	2.193

Section 3

Benchmark: Busy-Wait

As expected we did see a tremendous amount of performance gain in the previous section. Let's repeat the same experiment using busy wait. Please note that the program still uses some use of mutex for IO operations in VERBOSE mode, but since this mode is turned off by default for the experiment which is common with the previous program and hence won't cost to performance.

For thread count = 2

Array Size	T1	T2	T3	T4	T5
100000	**0.514**	0.441	0.424	0.398	0.408
1000000	4.953	4.904	3.968	**6.132**	4.713
10000000	**53.471**	39.297	38.823	41.133	39.026

For thread count = 4

Array Size	T1	T2	T3	T4	T5
100000	**0.637**	0.557	0.483	0.563	0.478
1000000	2.699	**6.773**	3.695	2.673	2.684
10000000	33.221	33.454	32.802	35.672	**36.666**

For thread count = 6

Array Size	T1	T2	T3	T4	T5
100000	0.636	0.528	**0.685**	0.574	0.539
1000000	4.220	4.172	3.833	**12.127**	7.143
10000000	**43.719**	42.471	38.049	35.767	34.157

For thread count = 8

Array Size	T1	T2	T3	T4	T5
100000	0.948	3.857	2.637	**44.852**	10.367
1000000	20.752	13.594	**21.141**	13.895	4.012
10000000	47.487	47.460	35.261	39.979	**70.993**

SpeedUp

Size ↓ Threads→	2	4	6	8
100000	**2.501**	2.011	1.845	0.0872
1000000	1.614	**2.150**	1.265	0.543
10000000	1.698	**2.092**	1.851	1.490

Section 4

Benchmark: Condition Variable

What are Condition Variables? Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. In the busy-wait experiment a thread with a rank `**my_rank**` waits in a loop until a the `**flag**` becomes equal to its `**my_rank**`, one can easily observe that conditional variables are natural for this purpose. The question is which method is better condition variable or busy-wait?

For thread count = 2

Array Size	T1	T2	T3	T4	T5
100000	0.700	0.455	0.438	0.420	0.434
1000000	3.534	3.472	3.514	4.957	6.996
10000000	42.542	40.209	39.693	39.937	39.656

For thread count = 4

Array Size	T1	T2	T3	T4	T5
100000	0.505	0.407	0.433	0.495	0.492
1000000	3.030	3.457	2.966	4.261	8.131
10000000	37.458	38.495	42.079	36.569	42.204

For thread count = 6

Array Size	T1	T2	T3	T4	T5
100000	0.534	0.522	0.611	0.616	0.689
1000000	6.977	5.409	3.343	3.919	3.291
10000000	37.199	37.696	36.771	42.045	38.015

For thread count = 8

Array Size	T1	T2	T3	T4	T5
100000	0.689	0.638	0.690	0.709	0.836
1000000	4.021	3.909	4.414	7.856	3.278
10000000	43.371	36.241	38.205	38.295	35.473

SpeedUp

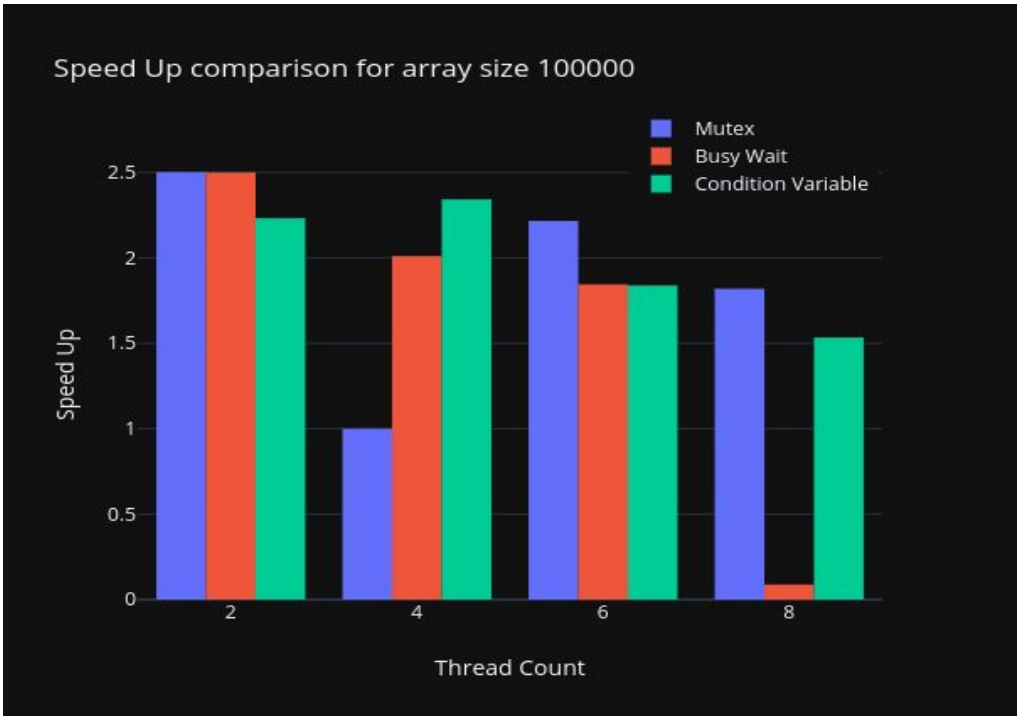
Size ↓ Threads→	2	4	6	8
100000	2.233	**2.343**	1.839	1.534
1000000	1.772	**1.823**	1.736	1.696
10000000	1.779	1.827	**1.875**	1.875

Section 5

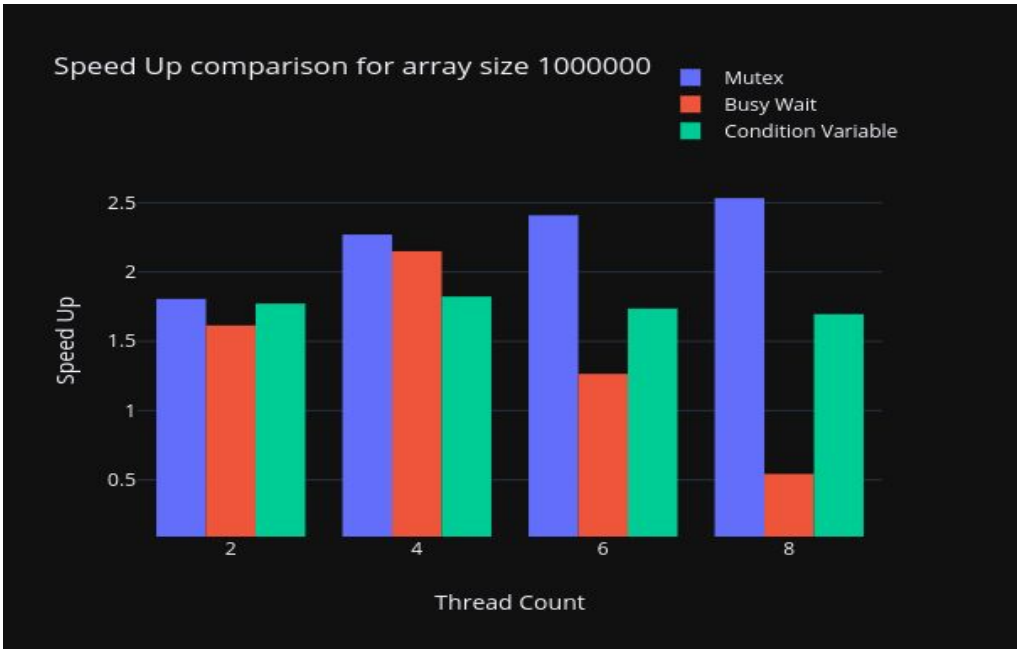
Comparison of mutex, busy wait and condition variables

Given below are the barplots for different array sizes which summarize the experiment results.

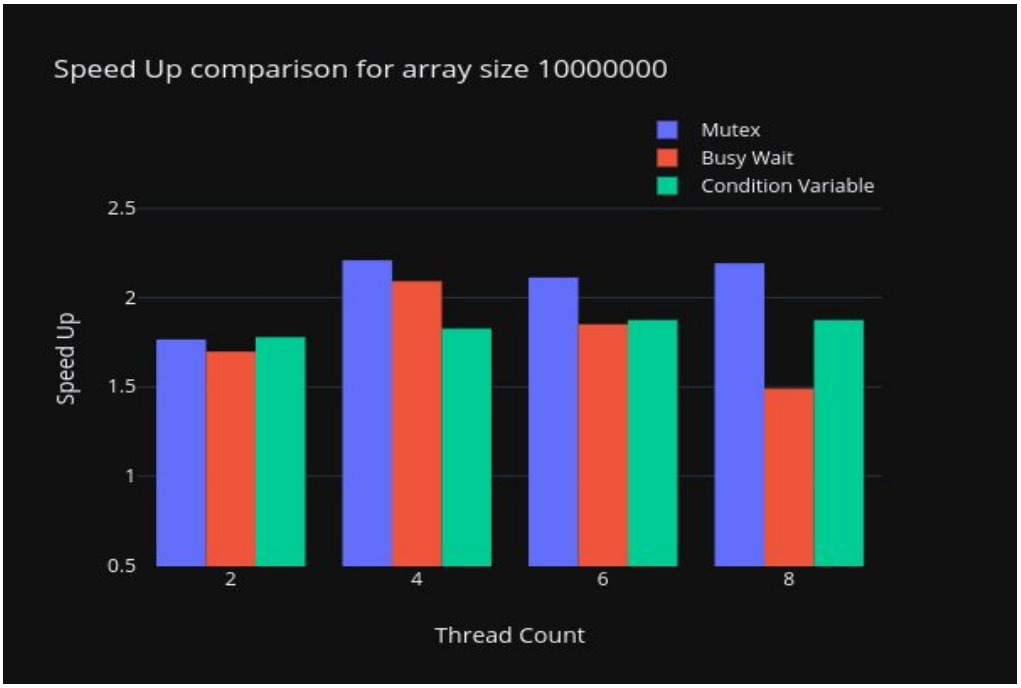
For array size of 100000



For array size of 1000000



For array size of 1000000



The table given below summarises the final results of the comparison between Mutex(**M**), Busy-Wait(**BW**) and Condition Variable (**CV**).

Size ↓ Performance→	**Low**	**Medium**	**High**
100000	BW	M	**CV**
1000000	BW	CV	**M**
10000000	BW	CV	**M**

Hence, the winner for this particular experiment is Mutex. One more important thing to note here is that the performance of Condition Variable remains nearly constant across all thread counts whereas the performance of Busy Wait fluctuates very much.