# Performance Profile of Synchronization Constructs

(Parallel Programming Assignment 2)

**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**

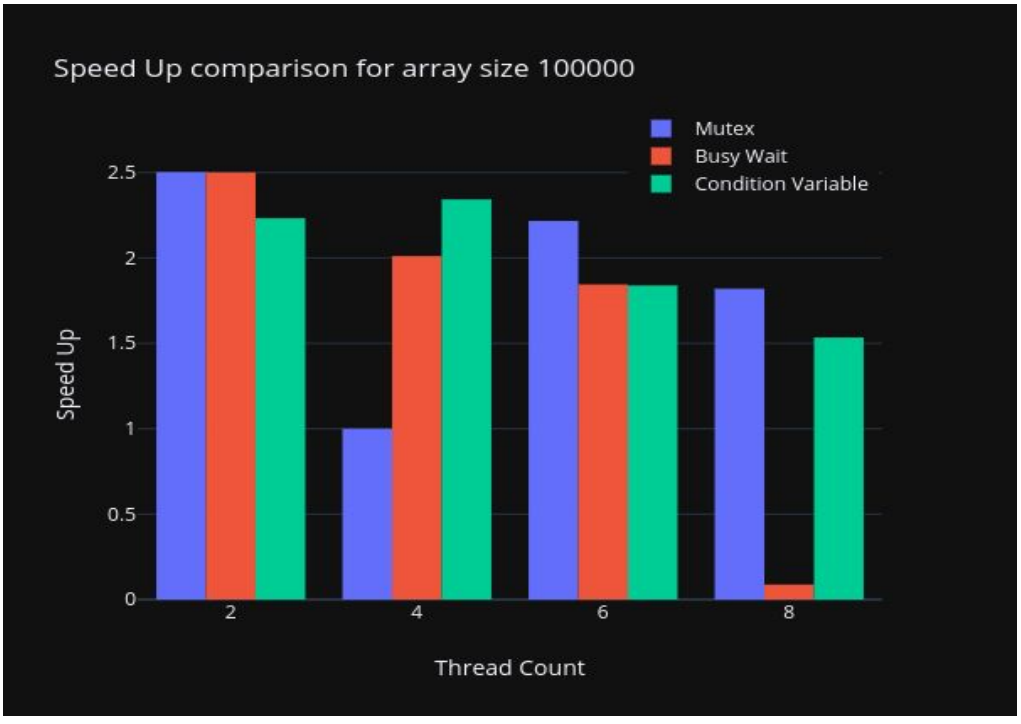
| Size ↓ Threads→ | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| **100000** | 2.233 | **2.343** | 1.839 | 1.534 |
| **1000000** | 1.772 | **1.823** | 1.736 | 1.696 |
| **10000000** | 1.779 | 1.827 | **1.875** | 1.875 |

# Section 5

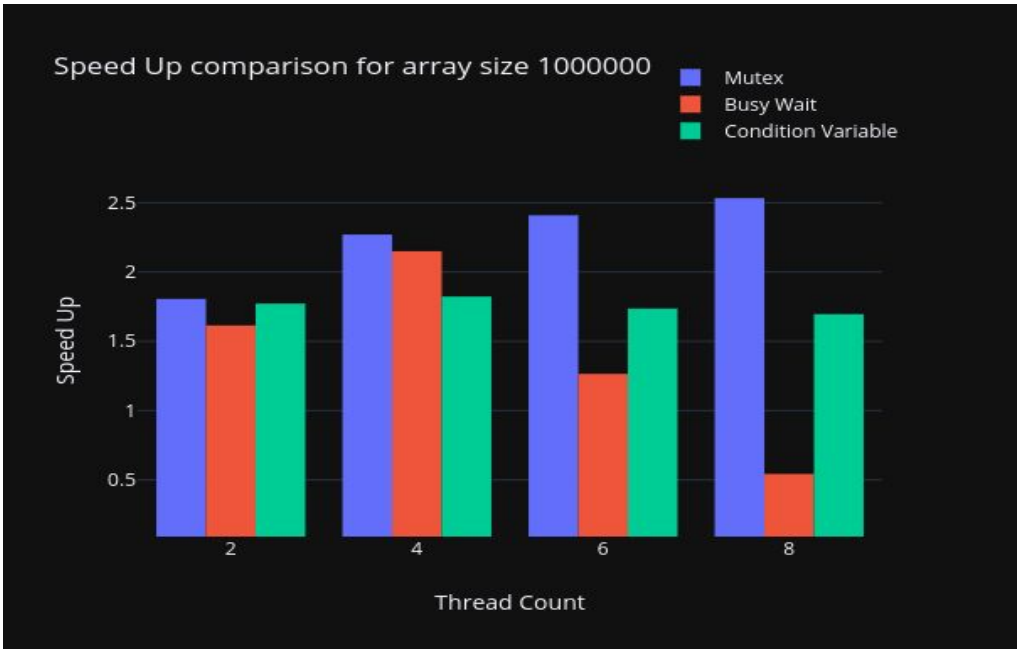
## Comparison of mutex, busy wait and condition variables

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

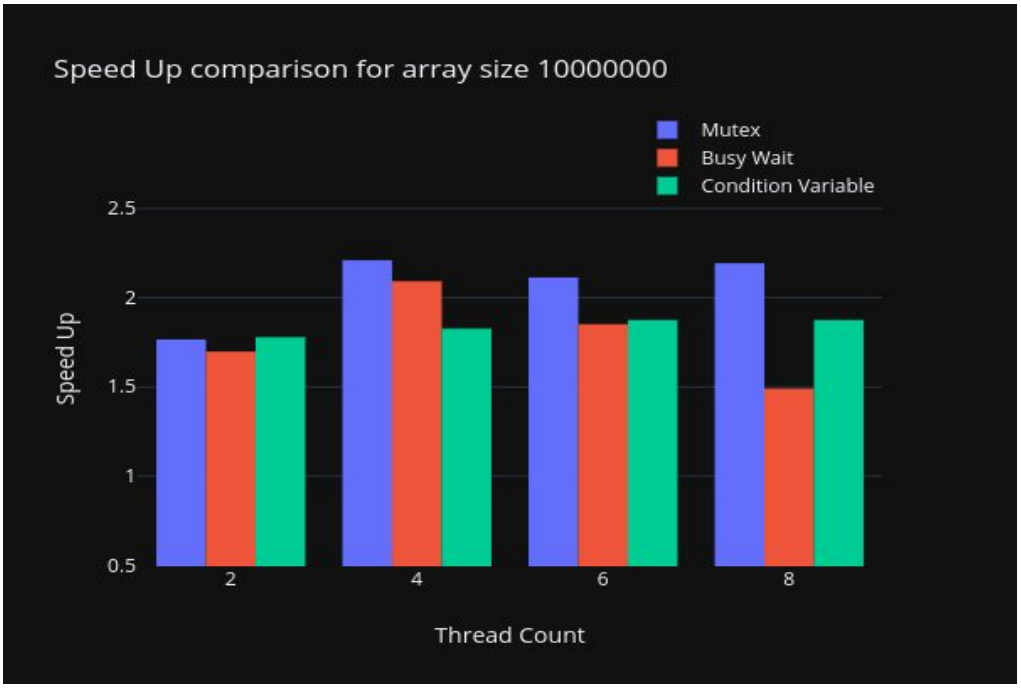
**For array size of 100000**



**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.

# Section 6

## Benchmark: Read / Write Lock

C++ supports read like lock using the `**std::shared_mutex**`. The shared_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a shared_mutex has two levels of access:

- **shared** - several threads can share ownership of the same mutex.
- **exclusive** - only one thread can own the mutex.

Shared mutexes are usually used in situations when multiple readers can access the same resource at the same time without causing data races, but only one writer can do so.

We design an experiment to compare the performance of the shared mutex with the mutex in terms of speed up. The experiment involves array of sizes of 100000, 10000 and 1000 and there can be three operations: update the value at any index in the array using a predefined constant time function, read the value at any index in the array, sort the array. The reason why we have designed the above experiment in such a way is that the smaller array sizes will give more simultaneous read-write operations for the same array index.

As already mentioned we have three approaches for this problem:

1. Sequential: the naive solution of one operation at a time (no parallelism)

2. Mutex: mutex for each array index which will give exclusive read/write permission
3. Shared Mutex: shared mutex for each array index which will allow multiple threads to read the data simultaneously while in case of any write operation it will block all other mutexes.

Please note that this problem can also be solved using busy-wait, but as already seen in previous section that it fluctuates too much just to read an array and compute the global sum and stood at the bottom of the performance list we are not letting it to participate in this experiment.

Given below is the results of the sequential approach:

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | 2601.3 | 2392.3 | 2896.8 | 2337.5 | 2230.0 |
| **10000** | 209.7 | 174.3 | 197.3 | 170.4 | 201.5 |
| **1000** | 18.0 | 16.2 | 17.0 | 17.8 | 17.7 |

Given below is the results of the mutex approach:

**For thread count = 2**

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | 1692.1 | 1947.4 | **1980.7** | 1874.0 | 1893.5 |
| **10000** | 190.4 | 174.0 | 179.5 | **216.1** | 179.5 |
| **1000** | **22.9** | 19.5 | 22.3 | 19.0 | 21.6 |

**For thread count = 4**

| Array Size | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| **100000** | 2203.1 | 1688.9 | 2116.9 | 2225.3 | **2435.1** |
| **10000** | 159.6 | **197.2** | 180.3 | 151.7 | 165.0 |
| **1000** | 20.7 | 19.9 | **22.0** | 20.6 | 21.9 |

**For thread count = 6**

| Array Size | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| **100000** | 1991.3 | 1891.8 | **2109.1** | 2078.4 | 1632.0 |
| **10000** | 160.2 | 167.1 | 180.9 | 162.7 | **193.1** |
| **1000** | **23.2** | 18.8 | 19.4 | 21.6 | 20.0 |

**For thread count = 8**

| Array Size | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| **100000** | 1711.6 | 1724.9 | 1874.9 | **2135.4** | 1968.7 |
| **10000** | 160.0 | 163.5 | 150.9 | **188.1** | 163.5 |
| **1000** | 21.9 | **23.3** | 19.8 | 21.0 | 20.8 |

**SpeedUp**

| Size ↓ Threads→ | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| **100000** | **1.327** | 1.167 | 1.284 | 1.323 |
| **10000** | 1.014 | 1.116 | 1.103 | **1.154** |
| **1000** | 0.823 | 0.825 | **0.842** | 0.812 |

Given below is the results of the shared mutex approach:

**For thread count = 2**

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | 1747.1 | 2056.4 | 2010.0 | 2022.3 | **2167.2** |
| **10000** | 190.1 | 194.9 | 204.7 | 203.5 | **207.1** |
| **1000** | **25.2** | 22.2 | 20.8 | 20.9 | 20.5 |

**For thread count = 4**

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | 1931.7 | 2022.4 | 2011.5 | 2190.5 | **2335.3** |
| **10000** | 176.8 | **190.4** | 172.2 | 179.1 | 183.4 |
| **1000** | 20.9 | 20.1 | **23.5** | 22.5 | 20.9 |

**For thread count = 6**

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | 1847.0 | 2034.2 | **2186.9** | 2054.4 | 2075.2 |
| **10000** | 198.0 | 160.0 | 186.2 | 188.5 | **218.4** |
| **1000** | 20.6 | **24.7** | 21.4 | 22.6 | 22.8 |

**For thread count = 8**

| **Array Size** | **T1** | **T2** | **T3** | **T4** | **T5** |
|---|---|---|---|---|---|
| **100000** | **1986.2** | 1963.8 | 1828.8 | 1614.0 | 1945.7 |
| **10000** | 198.1 | 189.3 | 195.7 | **200.7** | 194.5 |
| **1000** | 20.9 | 22.8 | **23.8** | 21.4 | 21.2 |

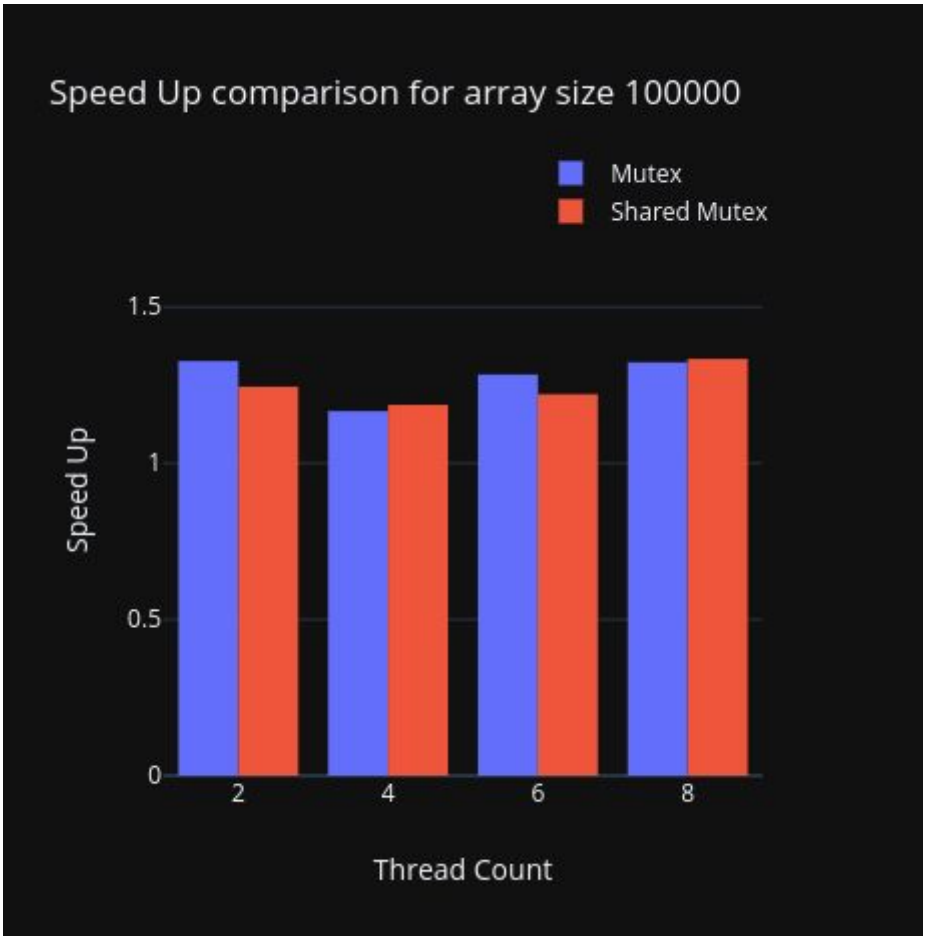
**SpeedUp**

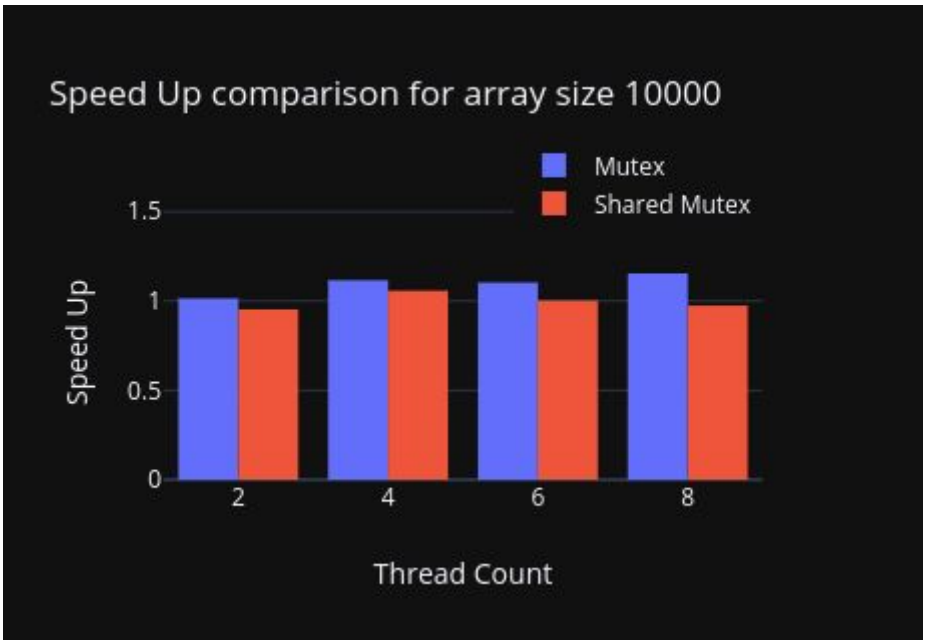
| **Size ↓ Threads→** | **2** | **4** | **6** | **8** |
|---|---|---|---|---|
| **100000** | 1.245 | 1.187 | 1.221 | **1.334** |
| **10000** | 0.953 | **1.057** | 1.002 | 0.974 |
| **1000** | 0.791 | **0.803** | 0.773 | 0.787 |

# Section 7

## Comparison of mutex and read / write lock

The bar plots given below summarise the results of the experiment.




Speed Up comparison for array size 10000
Mutex
Shared Mutex
1.5
1
0.5
0
Speed Up
2
4
6
8
Thread Count


Speed Up comparison for array size 1000
Mutex
Shared Mutex
1
0.8
0.6
0.4
0.2
0
Speed Up
2
4
6
8
Thread Count

The table given below summarises the final results of the comparison between three approaches: Sequential(**S**), Mutex(**M**), and Shared Mutes (**SM**).

| **Size ↓ Performance→** | **Low** | **Medium** | **High** |
|---|---|---|---|
| **100000** | S | SM | **M** |
| **10000** | S | SM | **M** |
| **1000** | SM | M | **S** |

Hence, the winner for this particular experiment is Mutex.