CS5005
Parallel Programming
Assignment - 2
(Performance profile of synchronization constructs)

Himanshu Rai - 111601032
Date - 23/02/2019

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.

Experiment - 1 (array_sum)

The first experiment is of calculating sum of numbers in an array. Each thread computes a partial sum of the array. These sums are combined into a global sum using critical section. The synchronisation constructs compared using this experiment are

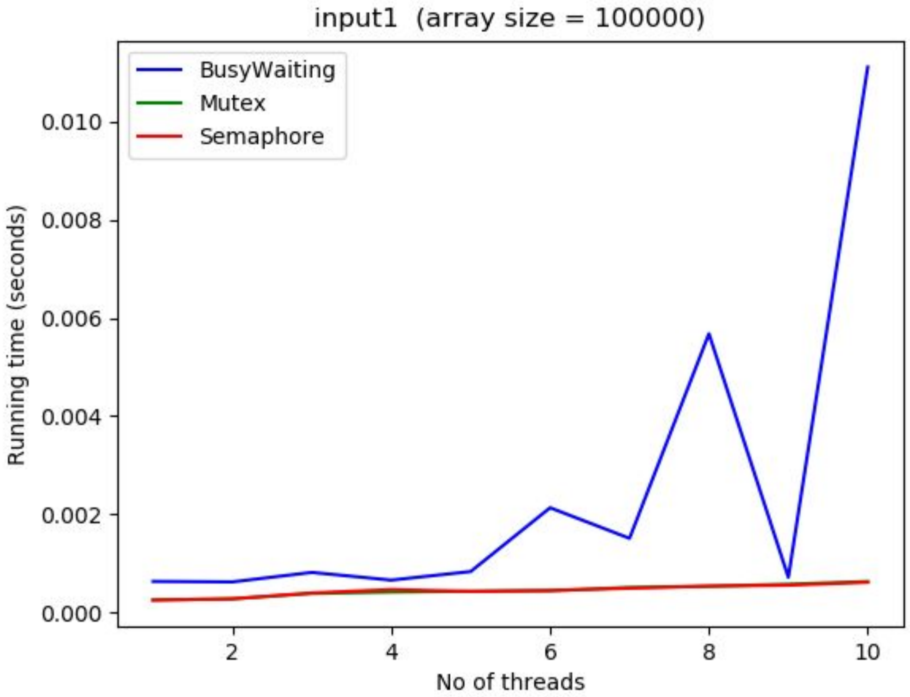
- Busy Waiting
- Mutex
- Semaphore

The synchronisation constructs are compared for different array sizes and also for different number of threads.

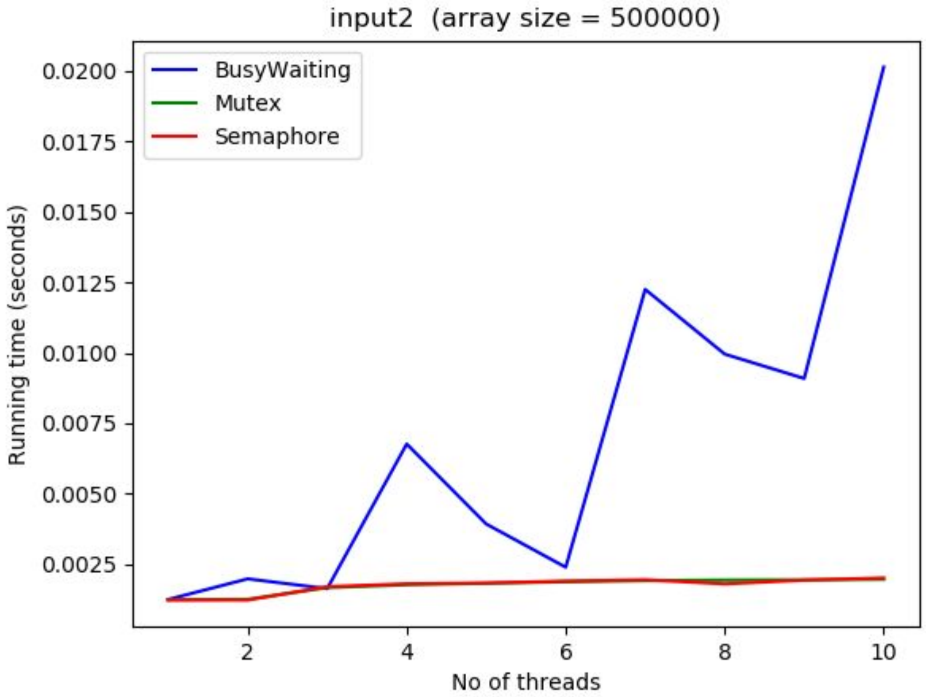
The functions running under the threads for the three constructs are kept similar apart except for the codes specific to them. This ensures uniformity and no extra overhead for any of the three constructs.

Also since the running time for a program depends on multiple factors an average of running time over various runs is taken. Even then some uneven results are likely to be observed.

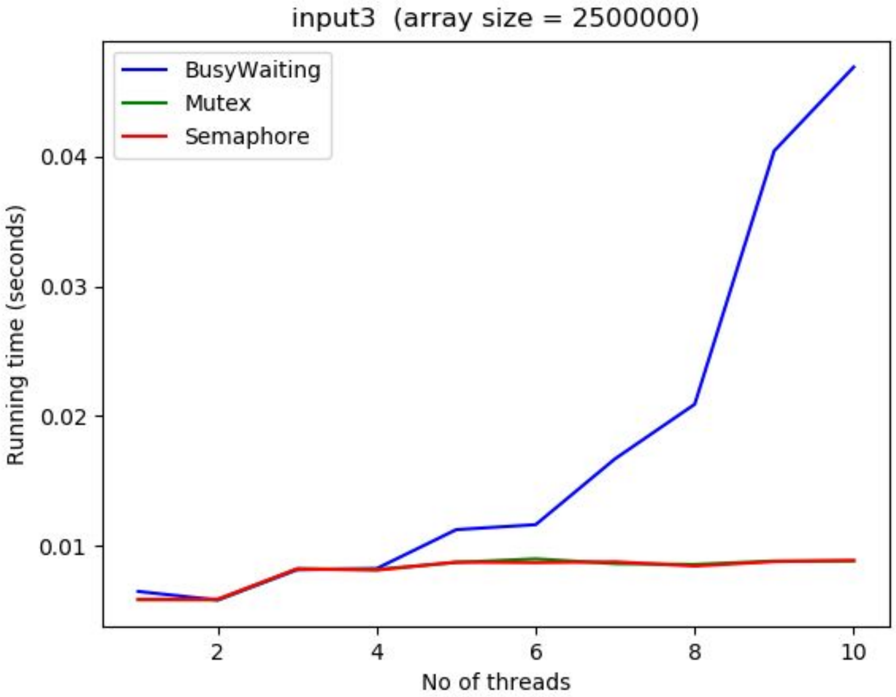
Observations (array_sum)



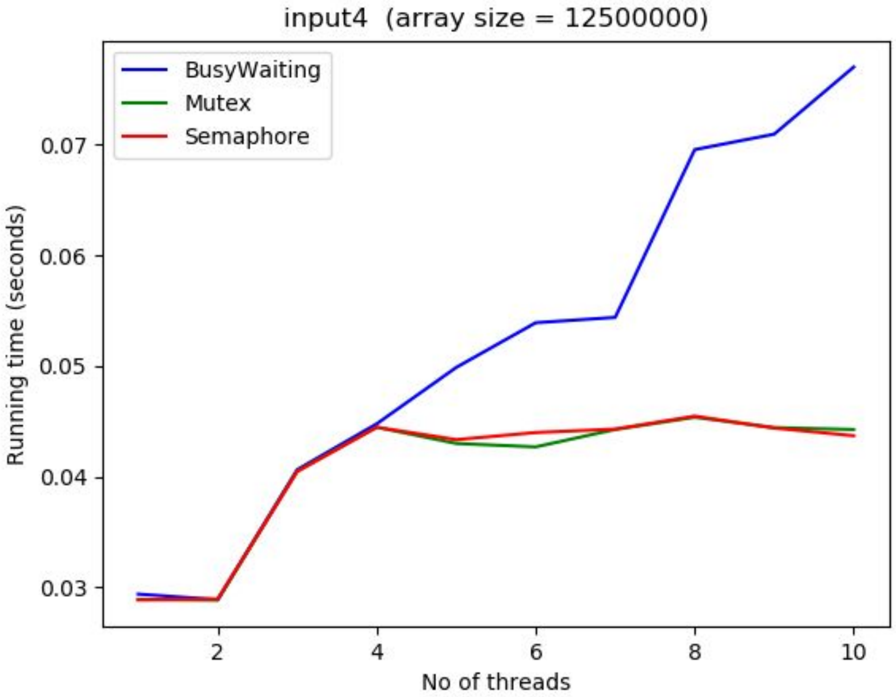
Observations (array_sum)



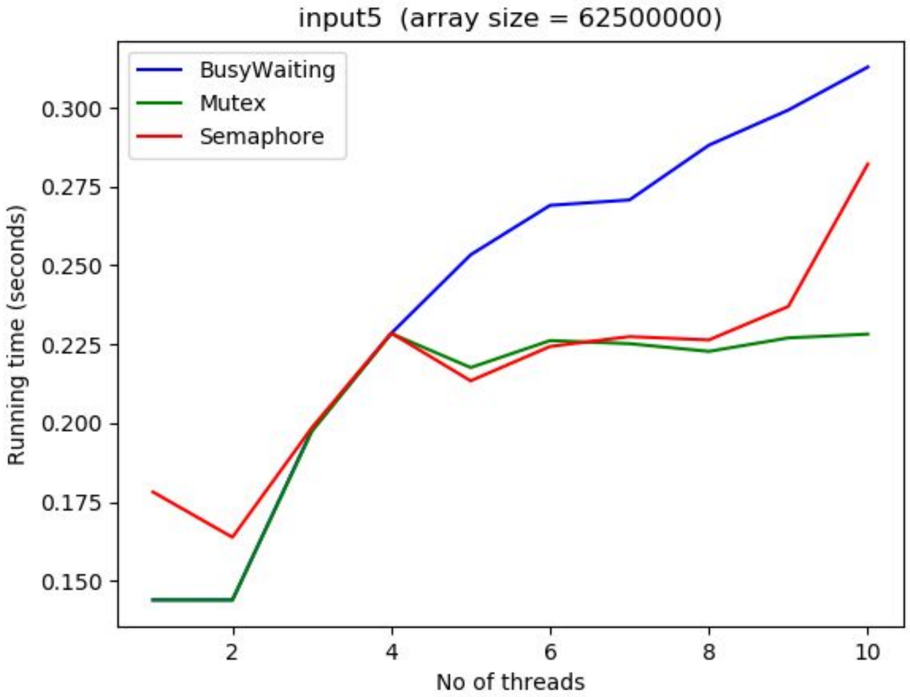
Observations (array_sum)



Observations (array_sum)



Observations (array_sum)



Analysis (array_sum)

As the plots on the previous pages shows, the running time for each of the three constructs - busy waiting, mutex, semaphore first decreases when no of threads increases from 1 to 2, because the experiments were performed on a dual core PC. So, at a time maximum of two threads can be run simultaneously. After this the increasing number of threads increases running time as threads with exception of atmost two, must wait to be scheduled at any point of time.

The effect of increasing number of threads is much more pronounced for busy wait than the other two. This is also clear as a thread which is scheduled but is anyway busy waiting wastes processors time, which is not the case with semaphores and mutexes.

The performance of semaphores and mutexes are similar as seen from the plots.

So, we can conclude that both semaphores and mutexes gives almost similar performance and should be used instead of busy waiting to gain performance.

Experiment - 2 (heat_eqlb)

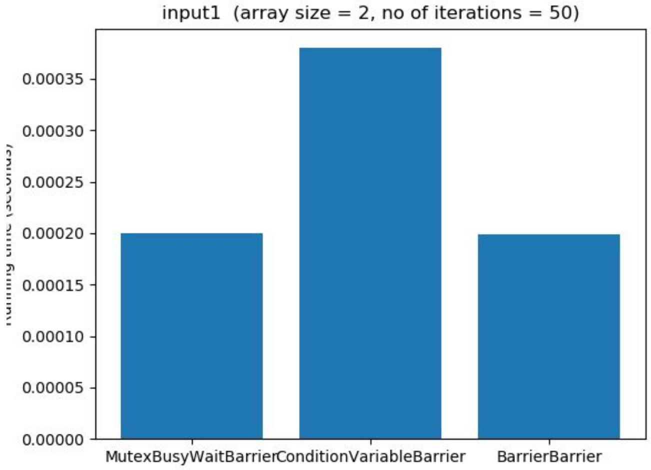
The second experiment is an algorithmic analogy of heat flow through a rod with varying temperature at various points. Ultimately, the rod reaches thermal equilibrium by heat flow from one part of the rod to the other. This experiment simulates the same process using threads. The synchronization constructs compared are

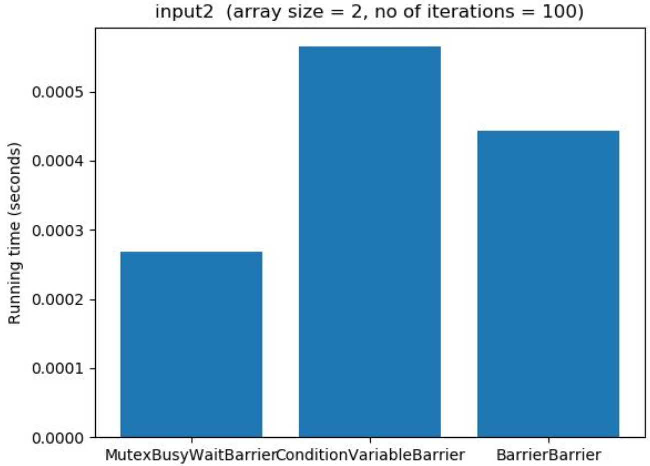
- Synchronisation using mutex and busy waiting
- Condition Variables
- Barriers

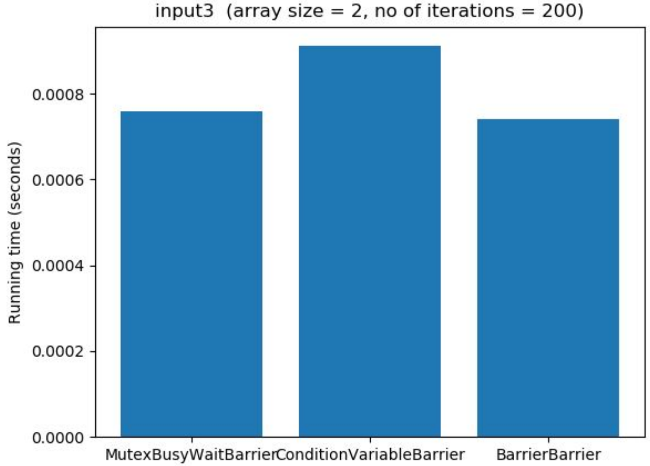
To maintain uniformity each of these is repeated a fixed prespecified number of iterations for an easy comparison.

These are compared for different array sizes and also for different number of iterations. Again the functions running under the threads for the three constructs are kept similar apart except for the codes specific to them. This ensures uniformity and no extra overhead for any of the three constructs. Also an average of running time over various runs is taken.

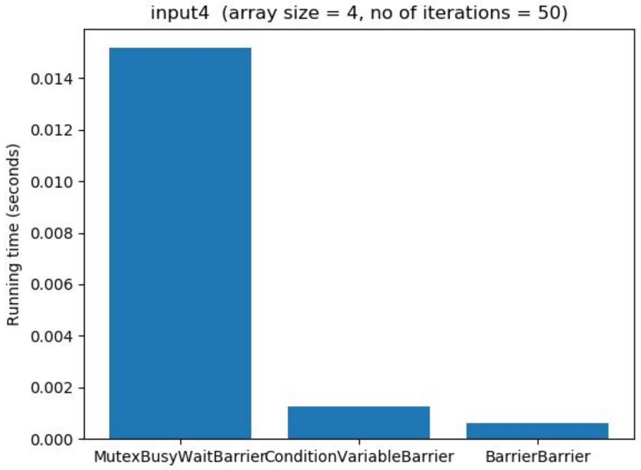
Observations (heat_eqlb)

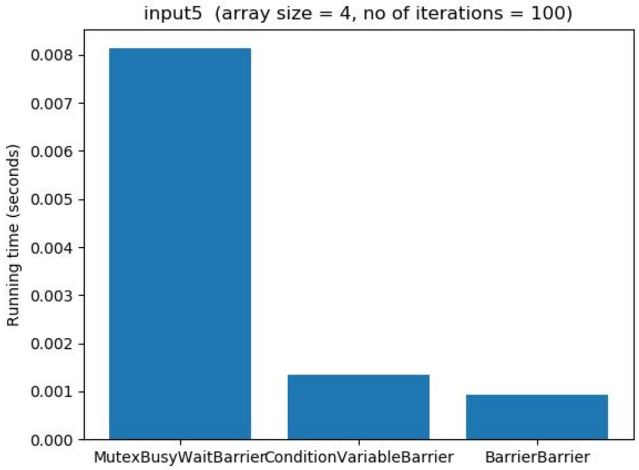


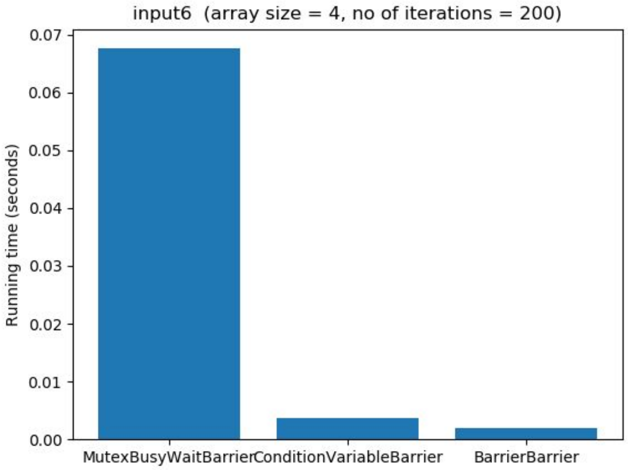




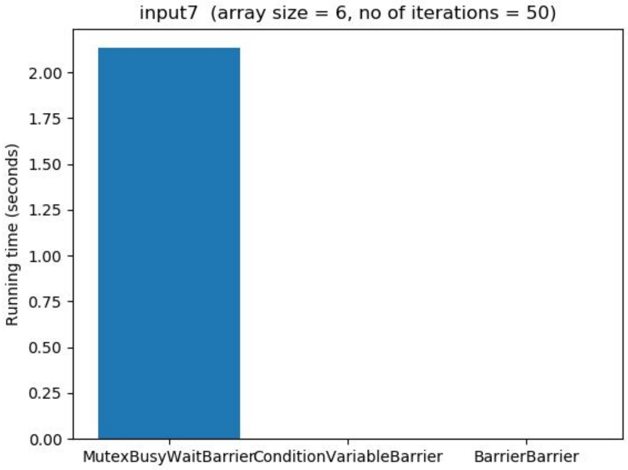
Observations (heat_eqlb)

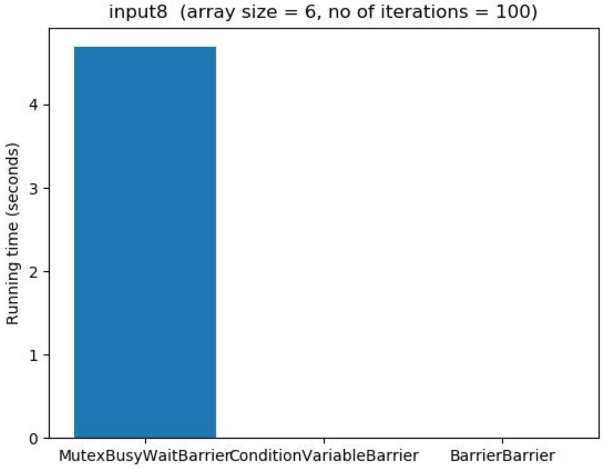


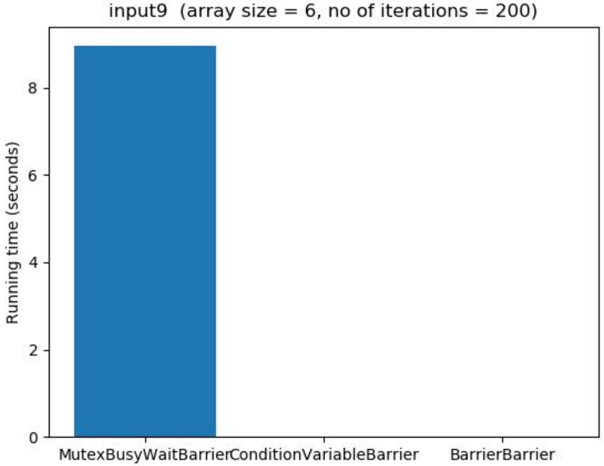




Observations (heat_eqlb)







Analysis (heat_eqlb)

Initially when the number of threads is smaller (2 which is less equal to the number of cores in the PC on which observations was made), the performance of mutex with busy waiting is comparable to that of condition variable and barrier. It is even better than the condition variable. This shows that the overhead involved in implementing the condition variable and barriers is comparable (even we can say it is more) as compared to the wastage of CPU during busy waiting. Also, however this effect is less pronounced as the number of iterations increases and the effect of busy wait becomes apparent.

However, as the number of threads increases (becomes more than available cores in the system) the mutex with busy wait is much badly outperformed by condition variable and barrier. The reason for this is definitely that the threads that are busy waiting also get scheduled by the scheduler, and wastes CPU time.

Also, barriers seem to be performing better compared to condition variables. The reason for this is that condition variables are more general constructs as compared to barriers. So, internal library implementation of condition variables would be much more complex as compared to that of barriers.

So, the conclusion is that if the number of threads involved in a rendezvous is smaller go for mutex with busy wait otherwise use barriers.

Appendix

Raw Data for first experiment

For observation - 1

```
For input1.txt

The sum of the array is: -33408
The size of the array is: 100000

The time spent for computing the array sum (in order of thread no from 1 to 10):
BusyWaiting :
           0.00063     0.00062     0.00081     0.00065     0.00083     0.00213     0.00151     0.00568     0.00071     0.01111

Mutex :
           0.00025     0.00027     0.00038     0.00041     0.00043     0.00044     0.00050     0.00053     0.00057     0.00063

Semaphore :
           0.00025     0.00027     0.00039     0.00046     0.00042     0.00044     0.00049     0.00053     0.00055     0.00061
```

For observation - 2

```
For input2.txt

The sum of the array is: -299601
The size of the array is: 500000

The time spent for computing the array sum (in order of thread no from 1 to 10):
BusyWaiting :
           0.00125     0.00198     0.00163     0.00677     0.00393     0.00239     0.01225     0.00995     0.00909     0.02014

Mutex :
           0.00124     0.00126     0.00167     0.00177     0.00182     0.00188     0.00192     0.00194     0.00194     0.00197

Semaphore :
           0.00123     0.00123     0.00170     0.00181     0.00183     0.00190     0.00195     0.00180     0.00193     0.00201
```

Appendix

Raw Data for first experiment

For observation - 3

```
For input3.txt

The sum of the array is: -1244690
The size of the array is: 2500000

The time spent for computing the array sum (in order of thread no from 1 to 10):
BusyWaiting :
          0.00650    0.00584    0.00817    0.00827    0.01126    0.01164    0.01672    0.02091    0.04041    0.04686

Mutex :
          0.00586    0.00586    0.00824    0.00820    0.00873    0.00901    0.00864    0.00858    0.00884    0.00886

Semaphore :
          0.00590    0.00591    0.00824    0.00814    0.00876    0.00872    0.00878    0.00845    0.00880    0.00889
```

For observation - 4

```
For input4.txt

The sum of the array is: -6177455
The size of the array is: 12500000

The time spent for computing the array sum (in order of thread no from 1 to 10):
BusyWaiting :
          0.02937    0.02887    0.04063    0.04477    0.04987    0.05391    0.05440    0.06957    0.07097    0.07704

Mutex :
          0.02887    0.02883    0.04052    0.04447    0.04299    0.04268    0.04426    0.04537    0.04443    0.04426

Semaphore :
          0.02886    0.02893    0.04045    0.04446    0.04334    0.04398    0.04429    0.04547    0.04439    0.04370
```

Appendix

Raw Data for first experiment

For observation - 5

```
For input5.txt

The sum of the array is: -32168680
The size of the array is: 62500000

The time spent for computing the array sum (in order of thread no from 1 to 10):
BusyWaiting :
          0.14388     0.14391     0.19773     0.22846     0.25336     0.26903     0.27074     0.28811     0.29924     0.31288

Mutex :
          0.14401     0.14392     0.19738     0.22838     0.21760     0.22615     0.22517     0.22273     0.22700     0.22820

Semaphore :
          0.17814     0.16379     0.19853     0.22853     0.21340     0.22429     0.22741     0.22636     0.23697     0.28209
```

Appendix

Raw Data for second experiment

```
For input1.txt
The size of the array is: 2
The number of iterations is: 50

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 0.00020
ConditionVariableBarrier : 0.00038
BarrierBarrier : 0.00020
```

```
For input2.txt
The size of the array is: 2
The number of iterations is: 100

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 0.00027
ConditionVariableBarrier : 0.00056
BarrierBarrier : 0.00044
```

```
For input3.txt
The size of the array is: 2
The number of iterations is: 200

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 0.00076
ConditionVariableBarrier : 0.00091
BarrierBarrier : 0.00074
```

```
For input4.txt
The size of the array is: 4
The number of iterations is: 50

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 0.01517
ConditionVariableBarrier : 0.00128
BarrierBarrier : 0.00060
```

```
For input5.txt
The size of the array is: 4
The number of iterations is: 100

The time spent for reaching equilibrium is
MutexBusyWaitBarrier : 0.00813
ConditionVariableBarrier : 0.00134
BarrierBarrier : 0.00092
```

```
For input6.txt
The size of the array is: 4
The number of iterations is: 200

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 0.06760
ConditionVariableBarrier : 0.00365
BarrierBarrier : 0.00197
```

```
For input7.txt
The size of the array is: 6
The number of iterations is: 50

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 2.13351
ConditionVariableBarrier : 0.00167
BarrierBarrier : 0.00086
```

```
For input8.txt
The size of the array is: 6
The number of iterations is: 100

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 4.68980
ConditionVariableBarrier : 0.00317
BarrierBarrier : 0.00163
```

```
For input9.txt
The size of the array is: 6
The number of iterations is: 200

The time spent for reaching equilibrium is :
MutexBusyWaitBarrier : 8.95425
ConditionVariableBarrier : 0.00627
BarrierBarrier : 0.00310
```

Appendix

Notes

- The experiments were performed on dual core, Intel i5 processor on linux operating system.
- Click here to view the source code used in the analysis.
    - Run “make gentest” to generate test cases file (test case files are not given because of large size).
    - Run “make” to compile the source files.
    - Run “make run” to run the file interactively with standard input.
    - Run “make test” to run the code on all input files (test1 to test5 for array_sum and test1 to test9 for heat_eqlb).
    - Run “make <filename>” to run a specific test file (test1 to test5 for array_sum and test1 to test9 for heat_eqlb).
    - Run “make clean” to remove files produced as a result of running the program.
    - Run “make clean” to remove and files produced as a result of running the program and also the input test files.

Thank You