Parallel Systems HW 1

November 2024

Introduction

For compilation, the makefile found in the initial directory was used, while the source code for each exercise is located under the corresponding directory **ask-X**. For each exercise there is a separate rule and therefore with **make askX** each exercise is compiled with its executable located in its directory. The measurements were made automatically with python scripts in which, to calculate the time of each program, 5 executions were made and their average was obtained. Their results are printed on the terminal in table format and saved in .csv files, which facilitated the process of measurements and production of graphical representations.

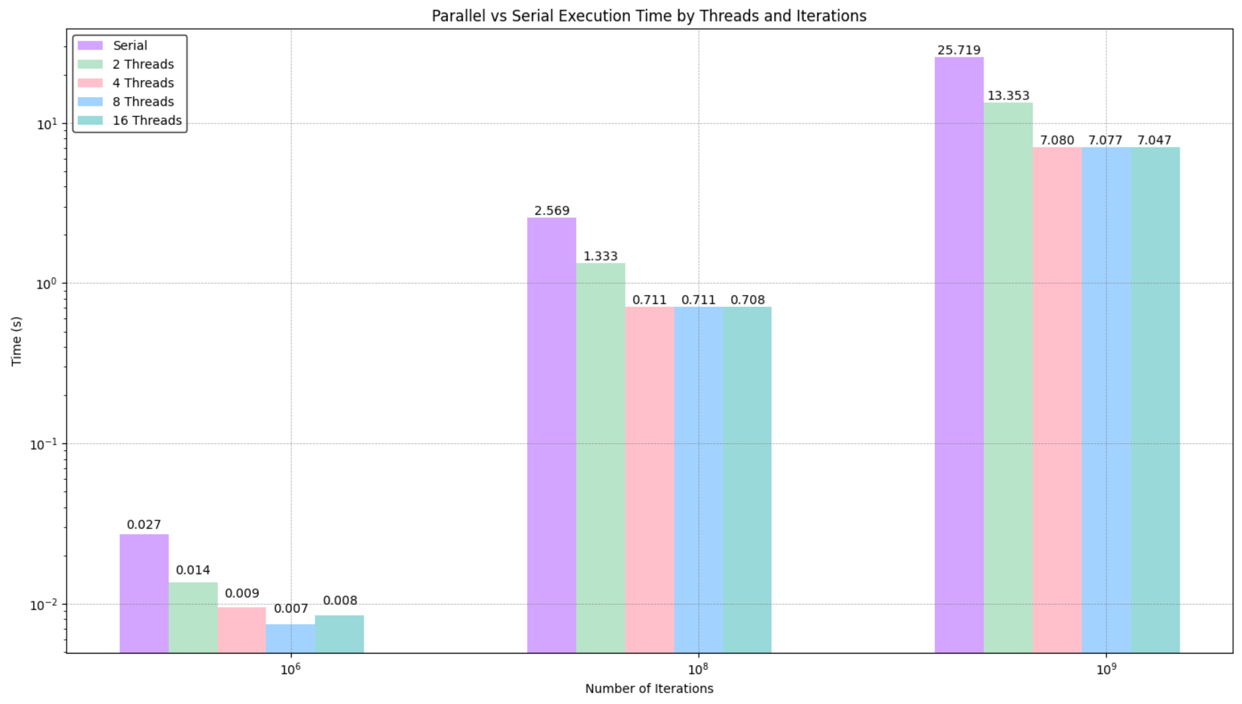
The measurements were made on the department's Ubuntu Linux machines, which have a CPU with 4 cores and an L1 cache-line of 64 bytes::

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
... skip ...
cpu cores       : 4
... skip ...
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual

$ gcc -v
... skip ...
Target: x86_64-linux-gnu
... skip ...
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.6 LTS (Focal Fossa)"
```

Exercise 1.1

No synchronization was used in this program, as there are no shared variables that are updated by the threads. Each thread manages its own independent variables, and at the end of the execution of all threads, their results are combined to produce the final estimate of π.
Running the program on the laboratory machines we obtain the following results:

Throws	Threads	Pi (Serial)	Serial Time (s)	Pi (Parallel)	Parallel Time (s)
$1 \cdot 10^6$	2	3.138712	0.0271	3.14306	0.0136
$1 \cdot 10^6$	4	3.138712	0.026	3.141772	0.0095
$1 \cdot 10^6$	8	3.138712	0.0261	3.140048	0.0074
$1 \cdot 10^6$	16	3.138712	0.0261	3.141908	0.0085
$1 \cdot 10^8$	2	3.141738	2.5691	3.141882	1.3332
$1 \cdot 10^8$	4	3.141738	2.5679	3.141732	0.711
$1 \cdot 10^8$	8	3.141738	2.5686	3.141733	0.7114
$1 \cdot 10^8$	16	3.141738	2.5729	3.14174	0.7078
$1 \cdot 10^9$	2	3.141572	25.7189	3.141597	13.3532
$1 \cdot 10^9$	4	3.141572	25.7097	3.141667	7.0803
$1 \cdot 10^9$	8	3.141572	25.8484	3.141702	7.0774
$1 \cdot 10^9$	16	3.141572	25.7939	3.141661	7.0471



Serial Algorithm:

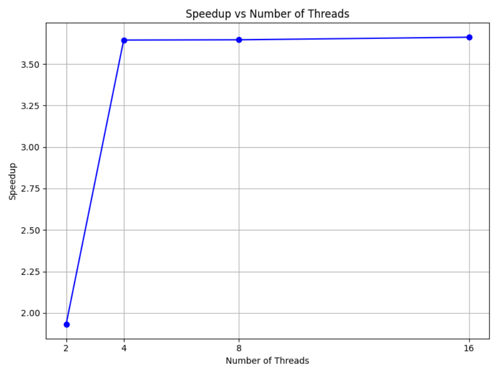
From the above table, we observe that the execution time of the serial algorithm increases as we increase the number of iterations, as is logical. This is due to the increase in the iterations in the main loop of the algorithm. Furthermore, the accuracy of the approximation of p improves with the increase in iterations, since for 10^6 throws we have precision of one decimal digit, while for 10^9 throws we have an accuracy of 4 decimal digits. This increase is expected, as more throws mean better sampling.

Parallel Algorithm:

For better randomness, the pseudo-random number generator of each thread is seeded with the rank of each thread, so that all threads produce the same prediction, while maintaining deterministic results from execution to execution. The time reduction is evident from the very first executions with parallel processing, even for 10^6 throws. Time is reduced compared to the serial algorithm, which is justified by the equal distribution of the workload among the available threads, which execute the throws in parallel. Specifically, for 2 threads the time is halved, while for 4 it is almost quadrupled. It is important to note that by oversaturating the number of threads, i.e. by adding more threads than the available cores (4 on the lab machines) we do not observe a change in time, as the additional threads do not actually run in parallel. The estimate of π here is not deterministic as in the serial algorithm (depending on the number of iterations), but rather varies from run to run due to the randomness of the (x, y) coordinates generated by the threads. However, as in the serial algorithm, we see that as we increase the number of iterations, the approximation of π improves.

Speedup for 10^9 throws:

Threads	T.Serial	T.Parallel	Speedup
2	25.8	13.35	1.932
4	25.8	7.08	3.644
8	25.8	7.077	3.6456
16	25.8	7.047	3.66



We observe that for 2 threads the speedup is quite close to ideal, however from 4 onwards its value remains constant, at around 3.6, as is logical, since our system only has 4 cores. However, from 4 threads onwards it deviates, possibly because the initialization of more threads incurs additional overhead which costs time.

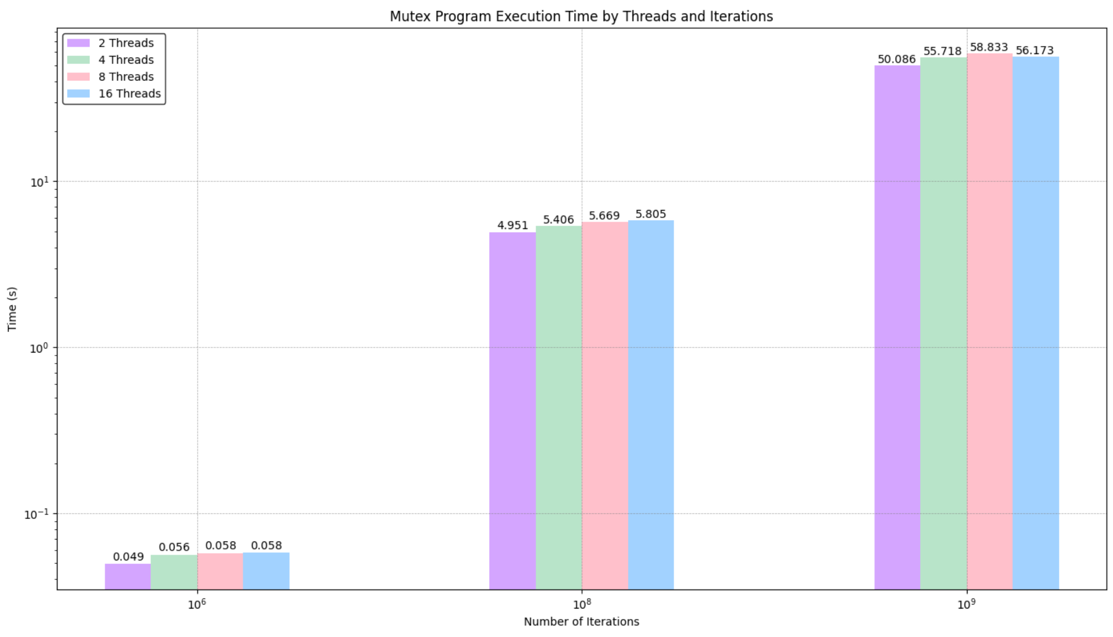
Άσκηση 1.2

In this program, since the threads update a shared variable, synchronization is required to ensure a correct result. One approach uses locks and the other atomic instructions. We attach the table of times and results of the 2 approaches.

Iterations	Threads	Shared (Mutex)	Mutex Time (s)	Shared (Atomic)	Atomic Time (s)
$1 \cdot 10^6$	2	$1 \cdot 10^6$	0.0495	$1 \cdot 10^6$	0.0204
$1 \cdot 10^6$	4	$1 \cdot 10^6$	0.0564	$1 \cdot 10^6$	0.0203
$1 \cdot 10^6$	8	$1 \cdot 10^6$	0.0575	$1 \cdot 10^6$	0.0203
$1 \cdot 10^6$	16	$1 \cdot 10^6$	0.0577	$1 \cdot 10^6$	0.0215
$1 \cdot 10^8$	2	$1 \cdot 10^8$	4.9515	$1 \cdot 10^8$	1.9542
$1 \cdot 10^8$	4	$1 \cdot 10^8$	5.406	$1 \cdot 10^8$	2.1352
$1 \cdot 10^8$	8	$1 \cdot 10^8$	5.6688	$1 \cdot 10^8$	2.0596
$1 \cdot 10^8$	16	$1 \cdot 10^8$	5.8049	$1 \cdot 10^8$	2.1179
$1 \cdot 10^9$	2	$1 \cdot 10^9$	50.0856	$1 \cdot 10^9$	21.2323
$1 \cdot 10^9$	4	$1 \cdot 10^9$	55.718	$1 \cdot 10^9$	21.0862
$1 \cdot 10^9$	8	$1 \cdot 10^9$	58.8334	$1 \cdot 10^9$	21.7636
$1 \cdot 10^9$	16	$1 \cdot 10^9$	56.1733	$1 \cdot 10^9$	21.7712

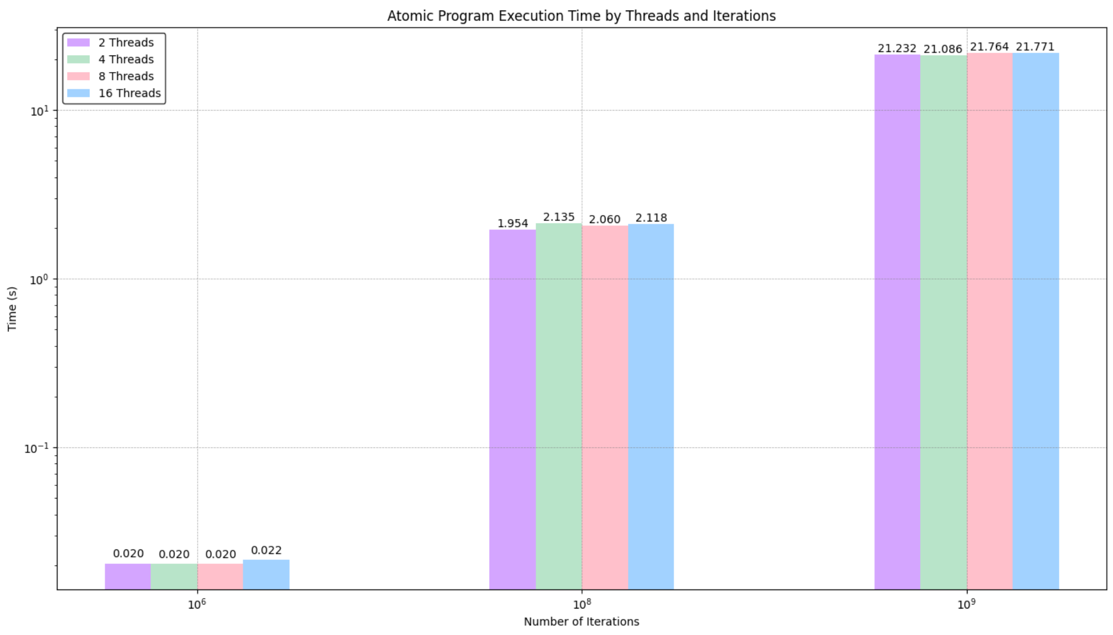
Mutex Locking:

In this version of the algorithm, synchronized operation of threads is achieved through the use of mutex from the pthread library. Before each access and modification of the shared variable, each thread acquires the mutex by calling pthread_mutex_lock, οπότε μπορεί να προχωρήσει στο critical section. at which point it can proceed to the critical section. Within this protected section of code, the thread safely updates the shared variable. Once the update is complete, the thread releases the mutex with pthread_mutex_unlock, allowing other threads to access their own critical section. This ensures that access to the shared variable is mutually exclusive, preventing phenomena such as race conditions.



Atomic Instructions:

This approach uses built-in functions of the compiler to use atomic commands that complete read-add-write in one cycle and therefore even if a thread is preempted by the scheduler at any stage of execution, the integrity of the variable is maintained as long as the writes are executed in full.



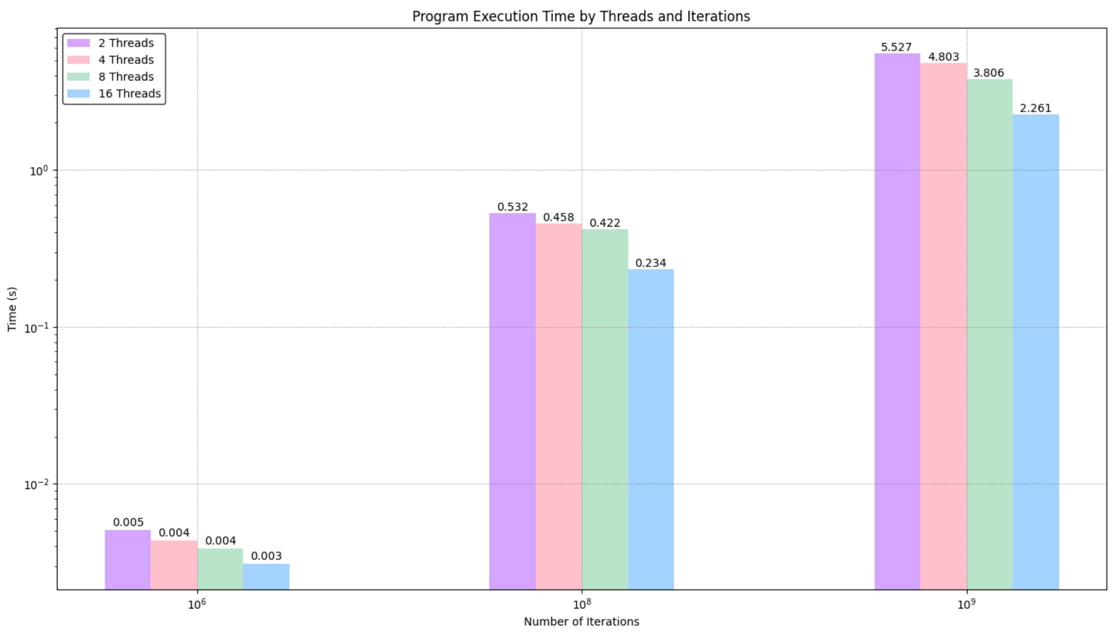
Comparison:

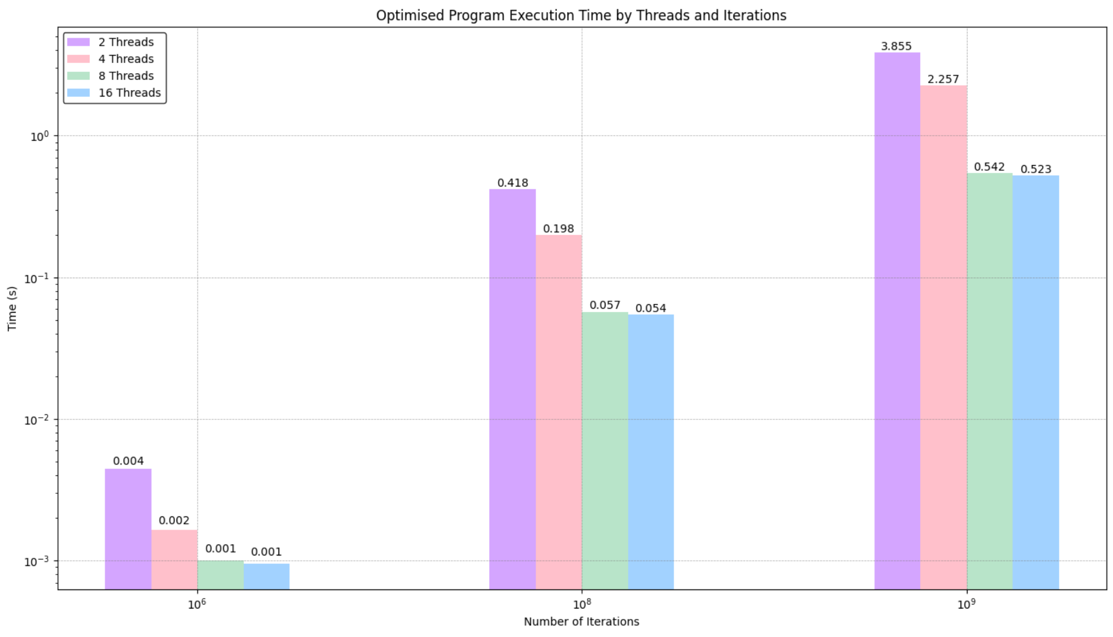
Regarding the execution time, we note that in the mutex version, the gradual increase in the number of threads is accompanied by an increase in the total execution time. This is due to the fact that more threads claim the mutex, resulting in congestion. In addition, the frequent calls of the pthread_mutex_lock and pthread_mutex_unlock functions introduce additional computational overhead. Typically, for 10^9 iterations, using 8 threads instead of 2 leads to an increase in average execution time by about 10 seconds.
On the contrary, in the atomic instructions version we observe that the time is significantly shorter, locking, as the threads no longer compete for the common resource (the mutex), but operate as independently as possible without affecting each other. This independence is also confirmed by the measurements, since for a fixed iteration time, no matter how much we increase the number of threads, the execution time remains almost constant.

Exercise 1.3

In this program, unlike exercise 1.2, thread synchronization is not necessary. This is because each thread updates only a specific location in the array, which is not used by other threads. Therefore, there is no risk of race conditions, as multiple threads do not access the same memory location simultaneously.
We run the program for different numbers of iterations and threads:

Iterations	Threads	Sum	Time (s)	Sum Opt	Time Opt (s)
$1 \cdot 10^6$	2	$1 \cdot 10^6$	0.0051	$1 \cdot 10^6$	0.0045
$1 \cdot 10^6$	4	$1 \cdot 10^6$	0.0044	$1 \cdot 10^6$	0.0016
$1 \cdot 10^6$	8	$1 \cdot 10^6$	0.0039	$1 \cdot 10^6$	0.001
$1 \cdot 10^6$	16	$1 \cdot 10^6$	0.0031	$1 \cdot 10^6$	0.0009
$1 \cdot 10^8$	2	$1 \cdot 10^8$	0.5316	$1 \cdot 10^8$	0.4179
$1 \cdot 10^8$	4	$1 \cdot 10^8$	0.4583	$1 \cdot 10^8$	0.1981
$1 \cdot 10^8$	8	$1 \cdot 10^8$	0.4221	$1 \cdot 10^8$	0.0571
$1 \cdot 10^8$	16	$1 \cdot 10^8$	0.234	$1 \cdot 10^8$	0.0545
$1 \cdot 10^9$	2	$1 \cdot 10^9$	5.5269	$1 \cdot 10^9$	3.8547
$1 \cdot 10^9$	4	$1 \cdot 10^9$	4.8031	$1 \cdot 10^9$	2.2569
$1 \cdot 10^9$	8	$1 \cdot 10^9$	3.806	$1 \cdot 10^9$	0.5421
$1 \cdot 10^9$	16	$1 \cdot 10^9$	2.2611	$1 \cdot 10^9$	0.523





False Sharing

The initial approach introduces false sharing, i.e. even though the threads do not update any common variable because their elements are in an array and therefore side by side with each other, there is a case where 2 elements needed by 2 different threads are in the same cache-line. Because of this, as soon as an element is written, the entire cache-line is marked as dirty and therefore invalid in the cache of other cores running the other threads and therefore to update it they must get the line from L3 cache/MM, thus wasting a lot of time.

Optimization

An optimization that solves the problem is to add padding to each entry:

```
typedef struct cache_line{
    long long int element;
    char padding[64-sizeof(long long int)];
}cache_line;
```

This way, each element that concerns us is located on its own cache-line, thus eliminating the issue of false sharing. The optimization brings a visible improvement in execution time.

Observations

An anomaly becomes apparent in the measurements compared to the previous ones. We previously observed that after 4 threads the execution time reaches a plateau and stops improving, while this is not the case here, as for 10^9 iterations the time continues to drop even after 4 threads. We believe this happens for the following reasons:

- **False Sharing:** As mentioned previously, although the variables are not actually shared, one core invalidates the cache of the other because the values are in the same cache line. In the case of 4 threads, each thread ran in its own core, which creates false sharing. On the contrary, now the cores are oversaturated with threads. Therefore, there is a serious possibility that adjacent threads can be scheduled in the same core. Therefore, these threads now have the same L1 cache, therefore, writes that were side by side are already there. This theory is also confirmed by the optimization times, e.g. for 10^9 iterations and 8-16 threads the time remains the same (after we eliminated false sharing) in contrast to the original version which was reduced again.
- **Load Balancing:**The use of more threads also means a smaller workload per thread. Each thread now has to run a smaller number of iterations and perform fewer operations, which leads to a reduction in the time it takes each thread to complete its work.
- **Cache Locality:** In the case of optimization, although we eliminated false sharing we continue to observe improvement over time with oversaturation. This probably happens because, as we mentioned before, 2 neighboring threads share the same core, and therefore L2 cache. That is, for example, when thread 0 requested cache line 0, the adjacent cache lines came with it to L2. Thus, when it is thread 1's turn to run, it will find cache line 1 already in L2 and will not need to go to MM, saving time

Exercise 1.4

Two tables are provided with metrics from the execution of the program for different configurations and priorities (reader/writer). The first table presents the execution times of the program that uses locks with reader priority, for a different number of threads, while the second table refers to the program that uses locks with writer priority. Also included are graphs comparing the two programs for different rates of read-write operations.

Table 1: Elapsed time for program with Reader priority

Threads	Elapsed Time (s)	Total Ops	Member Ops (%)	Insert Ops (%)	Delete Ops (%)
2	0.5077	500000	99.91	0.04	0.05
4	0.3595	500000	99.90	0.04	0.05
8	0.4019	500000	99.91	0.05	0.05
16	0.4440	500000	99.91	0.04	0.05
2	18.3868	500000	94.99	2.97	2.03
4	19.04725	500000	94.98	3.01	2.00
8	20.14022	500000	95.03	2.97	2.00
16	25.31474	500000	95.03	3.02	1.95
2	36.59843	500000	89.96	5.03	5.01
4	40.26346	500000	89.95	5.03	5.02
8	41.49172	500000	89.99	5.04	4.97
16	47.83583	500000	90.00	5.03	4.97
2	85.08262	500000	79.92	10.04	10.04
4	89.78151	500000	79.99	9.96	10.05
8	90.21368	500000	79.99	10.00	10.01
16	99.38099	500000	79.93	10.07	10.00

Table 2: Elapsed time for program with Writer priority

Threads	Elapsed Time (s)	Total Ops	Member Ops (%)	Insert Ops (%)	Delete Ops (%)
2	0.5035	500000	99.91	0.04	0.05
4	0.3240	500000	99.90	0.04	0.05
8	0.3521	500000	99.91	0.05	0.05
16	0.3644	500000	99.91	0.04	0.05
2	10.2531	500000	94.99	2.97	2.03
4	6.6279	500000	94.98	3.01	2.00
8	7.20119	500000	95.03	2.97	2.00
16	9.1385	500000	95.03	3.02	1.95
2	23.3422	500000	89.96	5.03	5.01
4	16.5116	500000	89.95	5.03	5.02
8	17.1237	500000	89.99	5.04	4.97
16	19.6647	500000	90.00	5.03	4.97
2	60.4564	500000	79.92	10.04	10.04
4	45.7615	500000	79.99	9.96	10.05
8	46.7724	500000	79.99	10.00	10.01
16	52.4362	500000	79.93	10.07	10.00

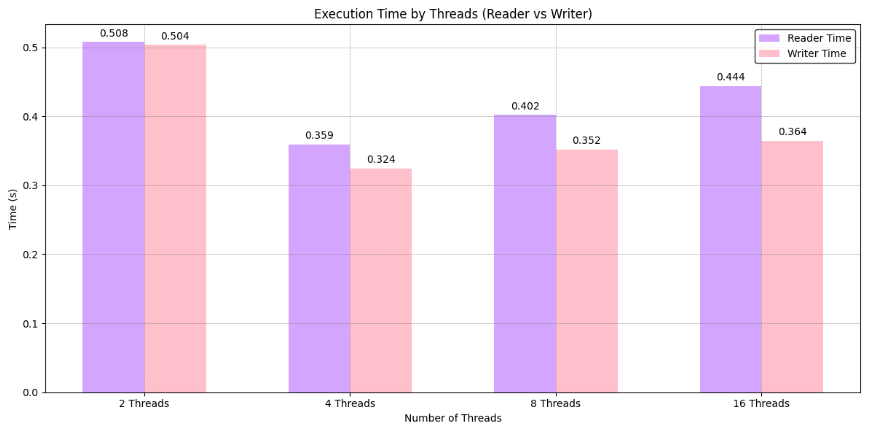


Figure 1: Program Execution time for 99.9% member ops, 0.05% insert ops and 0.05% delete ops

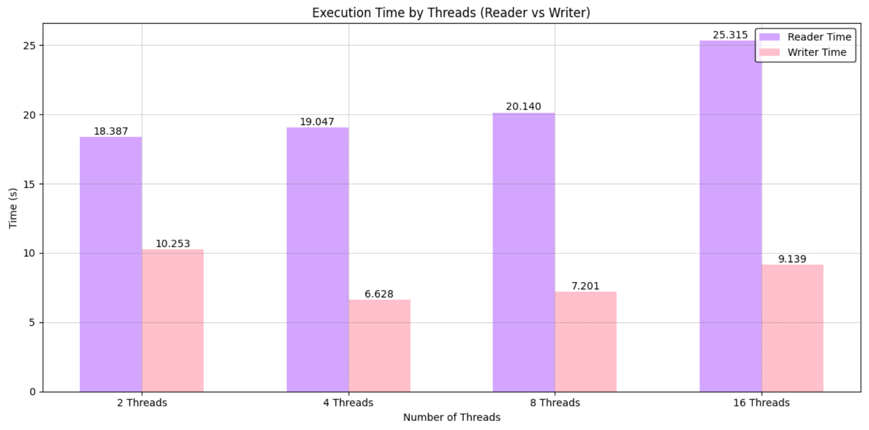


Figure 2: Program Execution time for 95% member ops, 3% insert ops and 2% delete ops

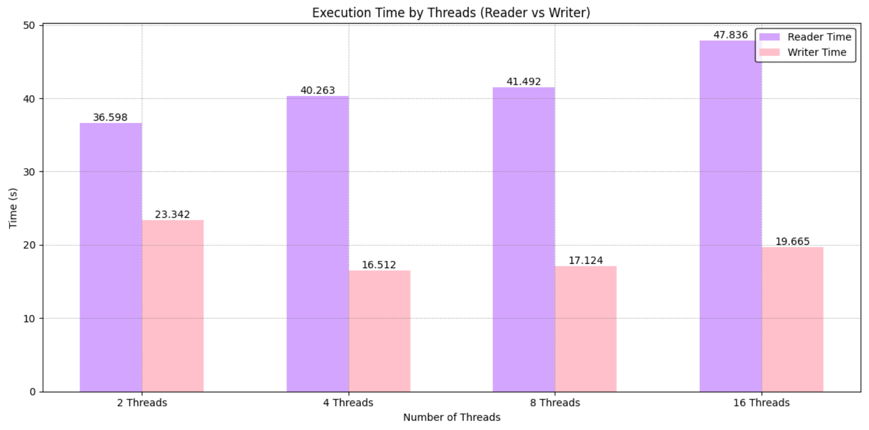


Figure 3: Program Execution time for 90% member ops, 5% insert ops and 5% delete ops

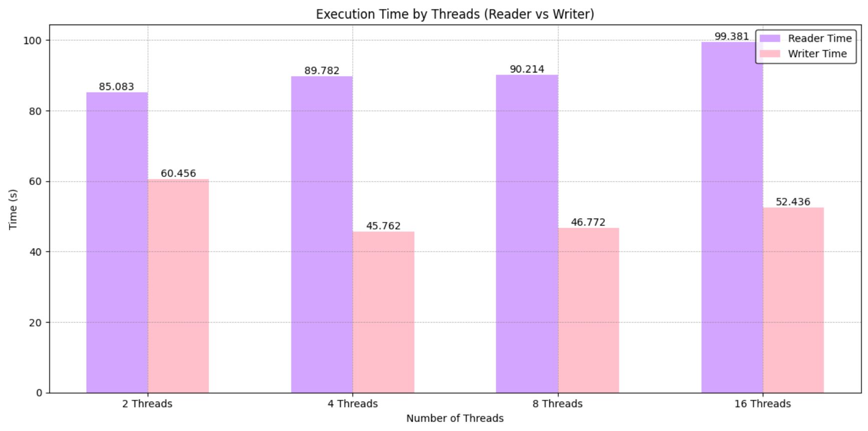


Figure 4: Program Execution time for 80% member ops, 10% insert ops and 10% delete ops

Implementation:

To implement the new lock we defined the structure:

```
typedef struct rw_lock{
    pthread_mutex_t mutex;
    pthread_cond_t cond_rd;
    pthread_cond_t cond_wr;
    int active_rd;
    int wait_rd;
    int active_wr;
    int wait_wr;
}rw_lock;

void rw_init(rw_lock* lock);
void rw_destroy(rw_lock* lock);
void rw_read_acquire(rw_lock* lock);
void rw_write_acquire(rw_lock* lock);
void rw_release(rw_lock* lock);
```

Which contains a mutex for mutual exclusion, 2 condition variables, one for each type of thread, so that they can wait in them until they acquire the lock, and other variables for book keeping.

Reader priority:

- To acquire the lock, the reader only checks if there is an active writer, otherwise all readers can enter simultaneously.
- The writer acquires the lock only when all readers have completed their work.
- When releasing, the type of thread holding the lock is checked. If it is a writer and there are waiting readers, it wakes them up first, otherwise a waiting writer. If it is a reader and it is the last one, it wakes up a waiting writer.

This can lead to writer starvation, since writers may have to wait for a long time if new readers are constantly added.

Writer priority:

- If a reader attempts to enter the critical section and there is either an active or waiting writer, then the reader is blocked.
- For the writer, it is enough that there are no active readers or another active writer.
- When releasing, the type of thread holding the lock is checked. If it is a writer and there is a waiting writer, it wakes it up first, otherwise the waiting readers. If it is a reader and it is the last one, it wakes up the waiting writer.

This management prevents writer starvation, as once the execution of the current readers is completed, the waiting writer immediately enters its critical section (since the next readers are waiting for it), thus ensuring fair switching between readers and writers.

Comparison:

From the above graphs it can be seen that as we increase the percentage of writers, the program with writer priority shows significantly better performance compared to the program with reader priority. Furthermore, increasing the number of writers also leads to an increase in the total execution time of both programs. These phenomena are likely to occur for the following reasons:

1. **Reader Parallelism:** By definition multiple readers can run their CS, therefore the more concurrent readers we have, the more operations can be done in parallel and therefore the time is reduced. This is also confirmed by the times as we see that increasing the percentage of writers increases the overall execution time.
2. **Writer Starvation:** As mentioned above, the approachwith reader priority leads to writer starvation, since some of the limited threads block. That is, it may happen that 3 of the 8 threads of the program become writers and are forced to wait until the end for the other 5 that can be continuously readers. On the contrary, in the second approach with writer priority, writers are guaranteed a shorter wait and therefore after they perform their write-operation, there is a possibility that they also become readers, thus increasing the number of threads that can run in parallel and thus reducing the total time.