Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Parallel Systems

Assignment 1

Name: Giorgos Nikolaou **sdi:** 1115202000154

Name: Helen Fili **sdi:** 1115202100203

---

1 Exercise 1.1

1.1 Description of the problem

In this problem, the task is to use the **Monte Carlo** method to approximate π by sampling random points in a 2×2 square (centered at $(0, 0)$) and observing how many of them fall within the unit circle. We approach the problem with both serial and parallel implementations (using `PThreads`). As expected, the method produces better approximations as we increase the number of iterations, but that is not efficient when using the serial algorithm. To accelerate the computation, we utilize `PThreads` to parallelize the random sampling among them.

1.2 Brief description of the solution

Description of Algorithms:

- **Serial**: The serial version of the algorithm is executed when the command line argument `<threads>` is set to 1.
- **Parallel**: The parallel version of the algorithm is executed when the command line argument `<threads>` is greater than 1. In this version, the samples of random points from arrow throwing are divided equally among the threads.

`approx_percentage(LONG n)`: In this function, we calculate the number of arrows that landed in the critical area (inside the unit circle). For the point sampling, a thread-safe version of `rand()` is used (`rand_r()`).

`worker(void* args)`: This function utilizes `approx_percentage()` to compute the number of arrows -that landed in the critical area- for the corresponding thread and then adds the result in the global variable `arrows`, which is guarded by a mutex.

`threaded_pi(size_t threads, LONG n)`: This functions is responsible for dividing the total number of arrows among the threads, creating and joining the threads, initializing and destroying the mutex used for sychronization and returning the approximation of π.

Implementation Details:

- Type `long long` is macro-defined as `LONG`.
- The number of sampling points `<n>`, given from command line, must be divisible by the number of `<threads>`.

1.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of threads and sampling points. We also created the corresponding plot where `<n>` is the horizontal axis and presented logarithmically. See: `./monte_carlo/output/plot.png`

Specifically, we executed the program with every combination of the following parameters:

- `<threads>`: $[1, 2, 3, 4, 8, 16, 32]$
- `<n>`: $[10^7, 10^8, 10^9, 5 \cdot 10^9, 10^{10}]$

To conduct this grid search and generate the corresponding plot, we developed the script `plots.py`. To reproduce, execute:

```
$ cd monte_carlo
$ make monte_carlo
$ python ./plots.py
```

Listing 1: Monte Carlo Plotting

To perform manual executions, run:

```
$ cd monte_carlo
$ make monte_carlo
$ ./monte_carlo <threads> <n>
```

Listing 2: Monte Carlo Execution

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

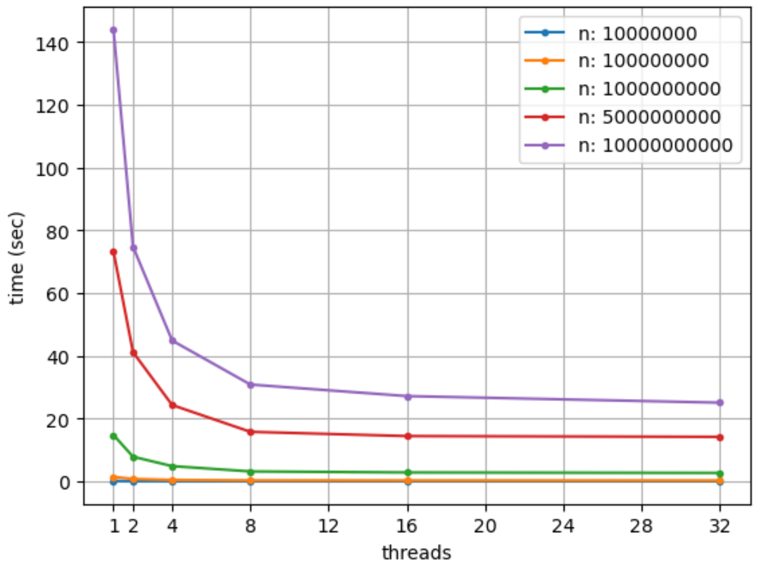


Figure 1: Monte Carlo Results

We make the following observations:

- **Doubling** the number of threads results in **reduction** of the execution time by *half*, as expected due to the shared workload across multiple threads.

- In our configuration, featuring 6 cores with 2 threads each and totaling 12 threads (the maximum number of processor threads), we observe improvement up to the use of 12 threads. However, increasing the thread count beyond 12 does **not** yield further reductions in execution time.

2 Exercise 1.2

2.1 Description of the problem

The goal of this exercise is to implement a parallel **matrix multiplication** program using `PThreads`. An observation made after implementing the initial version of the exercise (`matmul_false_sharing.c`) was the presence of *false sharing*. **False sharing** occurs when multiple threads, running on separate cores, access different variables that happen to reside on the same cache line. To address that issue, we implemented three alternative programs (`matmul_2d.c`, `matmul_pad_var.c`, `matmul_private_var.c`).

2.2 Brief description of the solution

We developed the following 4 implementations:

- `matmul_false_sharing.c`: In this implementation, we **flatten** the $2D$ matrices into $1D$ vectors and perform the multiplication accordingly. The total computations are divided among the threads, with each one handling the computations for a specific number of consecutive rows.

- `matmul_2d.c`: Here we store the matrices as $2D$ constructs, where each row is **allocated independently** in the *heap* and is handled accordingly during the multiplication.

- `matmul_pad_var.c`: In this implementation, we **flatten** the $2D$ matrices into $1D$ vectors but insert a specified amount of **padding rows** between the actual rows of the result matrix in an attempt to eliminate false sharing.

- `matmul_private_var.c`: Matrices are stored in memory and handled as in the false sharing method. This method differs from the previously mentioned, as it uses one local variable in the stack to calculate the sum for a specific row and column. After the sum is computed, the value is assigned to the corresponding position in the array.

Implementation Details:

- The dimension `<m>`, given from command line, must be divisible by the number of `<threads>`.

- To verify the results of the multiplication, compile with `-DVERIFY`, run the desired algorithm, and then execute the script `verify.py` with the appropriate parameters (`<m> <n> <p>`).

2.3 Presentation of experiments and analysis

Below, we provide the results for the parameters mentioned in the assignment. For the padding method, the following pad sizes where used: $[2, 4, 8, 16]$
To perform the grid search described in the assignment and produce the corresponding plots, we created the script `plots.py`. To reproduce, execute:

```
$ cd matmul
$ make all
$ python ./plots.py
```

Listing 3: Matrix Multiplication Plotting

To perform manual executions, run:

```
$ cd matmul
$ make all
$ ./<algo> <m> <n> <p> <pad, when appropriate> <threads>
```

Listing 4: Matrix Multiplication Execution

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

2.3.1 Performance

Generally, when no false sharing is detected in the respective method, neither private variable nor padding methods exhibit improvements in performance (execution time), regardless of the number of threads. (`2d < false sharing` $\simeq$ `private variable` $\simeq$ `padding`). When false sharing **is** observed, except for `2d`, these mitigation methods perform better as the number of threads increases, as expected.

We hypothesize that false sharing will occur when the number of rows in the resulting matrix is sufficiently small, and concurrently, the number of columns and threads are sufficiently large.

The padding size must be sufficiently large to cover the entire cache line with padding elements. In our configuration, where cache lines are 64 bytes (`$ getconf LEVEL1_DCACHE_LINESIZE`), the solution to false sharing requires the following condition: After the last **used** index of one thread, there must be enough padding to fill the remaining space in the cache line. This phenomena becomes particularly evident when $m = 8$ and $p = 1$ (as illustrated below).

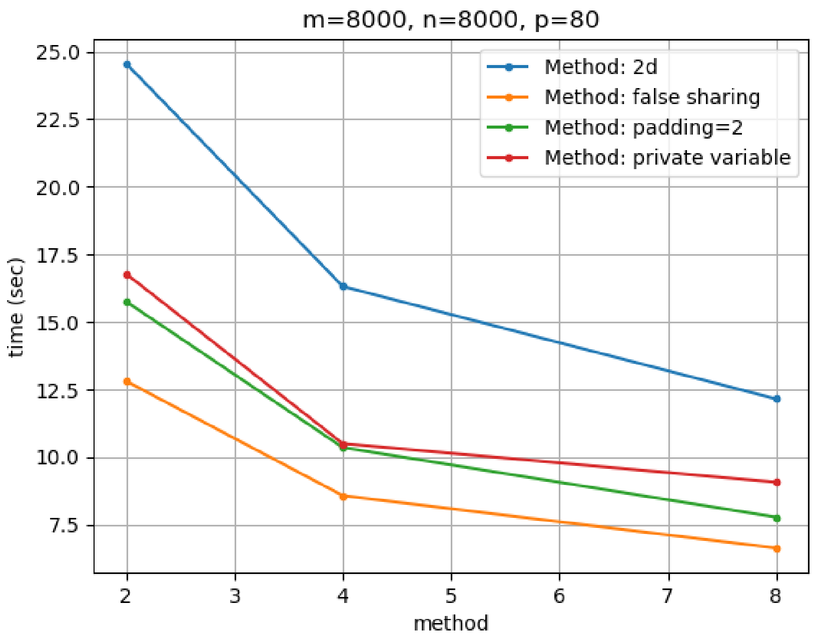
Note: We do not include all of the plots in this report, due to their count. They can be found under `matmul/output/plots/`.

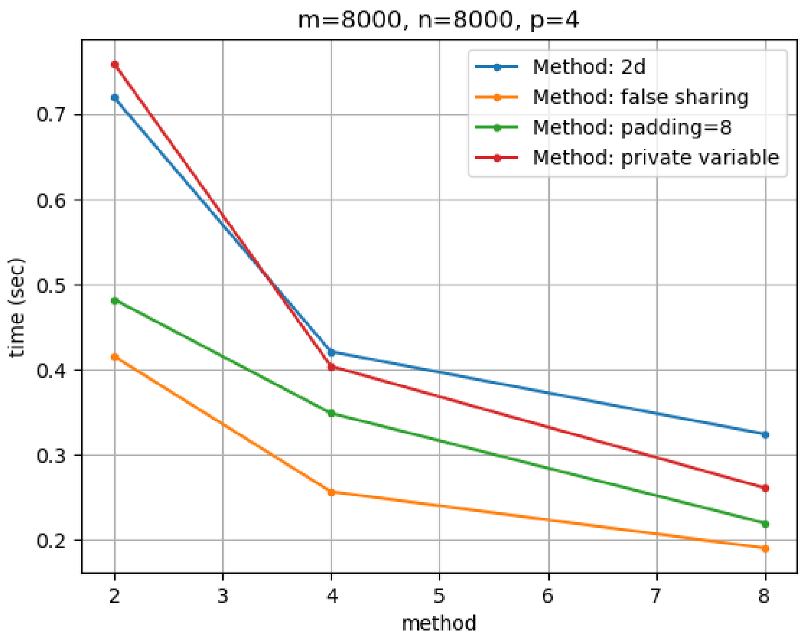
2.3.2 Dimensions: $m = n = 8000, p = 1/2/4/8/80$

It is evident that the effect of any possible *false sharing* is **negligible** across all padding sizes and for all values of `p`. Although false sharing is possible, its frequency is very low, allowing the program to benefit from the increased number of threads and achieve better performance.

With these combination of parameters, the `false sharing` version has the best performance, closely followed be `padding`. `2d` method is significantly lower and is not a good choice!

It is worth noting that for $p = 1/2/4/8$, the plots exhibit similar gradients but different magnitudes, as expected. However, when $p = 80$, the private variable method experiences a significant performance boost when utilizing only 2 threads. This is attributed to the larger number of operations being performed for each private variable.

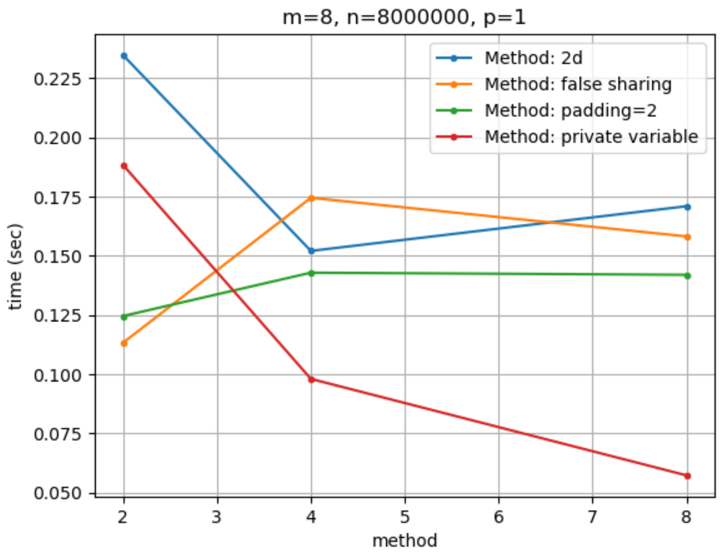


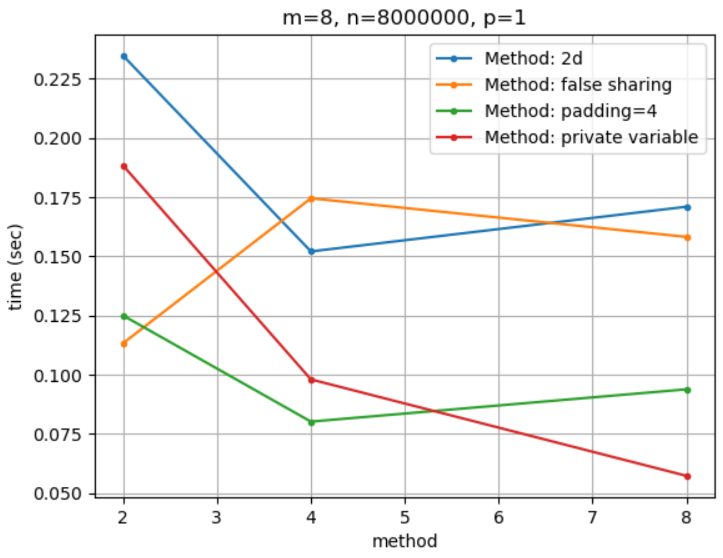


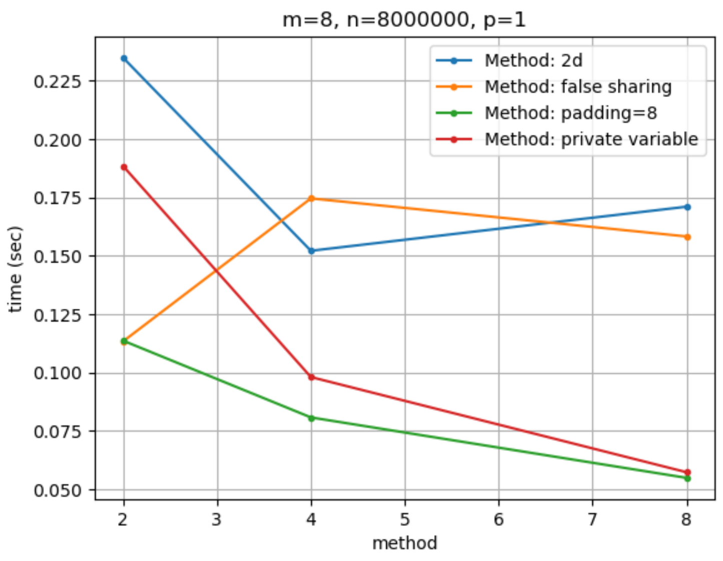
2.3.3 Dimensions: $m = 8, n = 8000000, p = 1/2/4$

In this case, *false sharing* is **noticeable**. Upon examining the plots, we observe the following:

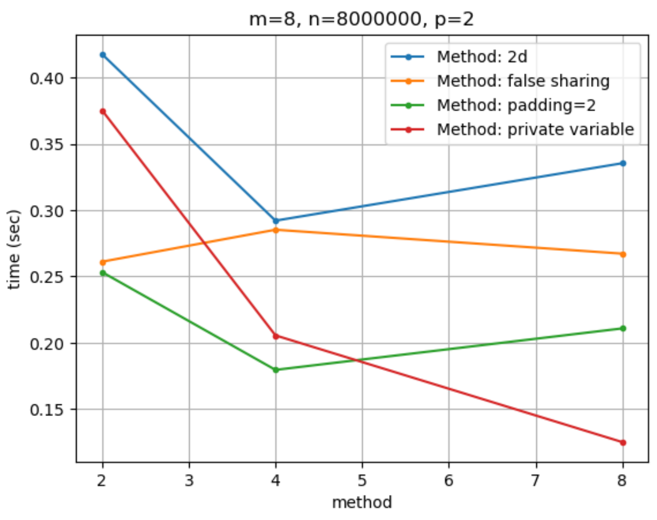
- **False Sharing**: As expected, the naive approach exhibits a substantial amount of false sharing.
- **2D**: Surprisingly, this method did not eliminate false sharing for all parameter combinations. Specifically, false sharing can be noticed when utilizing 8 threads. Additionally, this approach is slower and, therefore, **not** a good choice.
- **Private Variable**: This method is perhaps the simplest solution to make false sharing negligible. It works for every combination of parameters.
- **Padding**: This method also eliminates false sharing when padding is appropriately large. See analysis below.

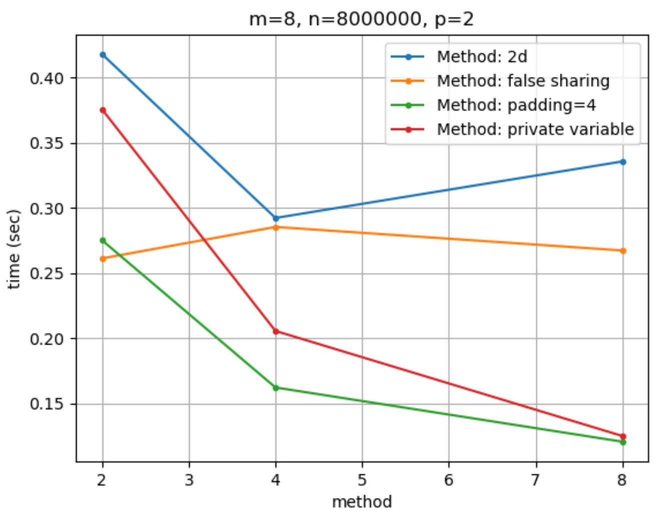


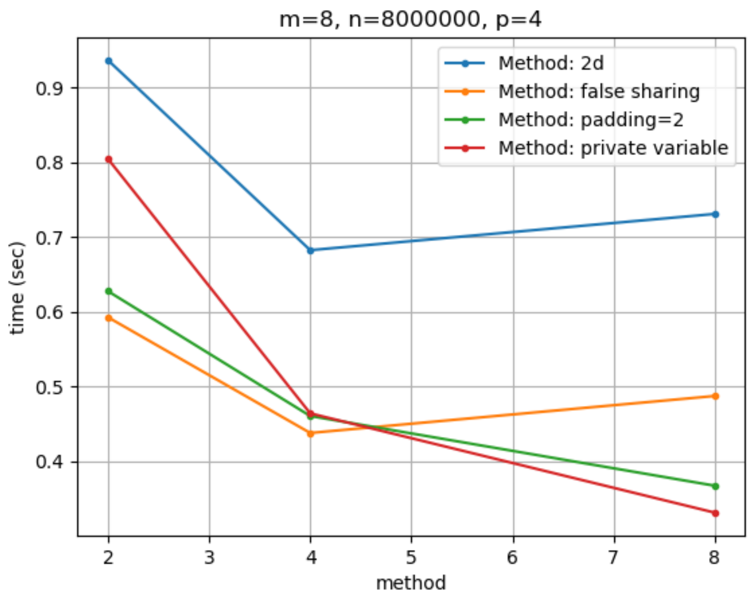




When $p = 1$, to eliminate false sharing, padding size must be at least 8. We get significant performance enchantments from both private variable and padding methods when increasing threads to 4 and 8. Performance is not increased when the number of threads is 2.





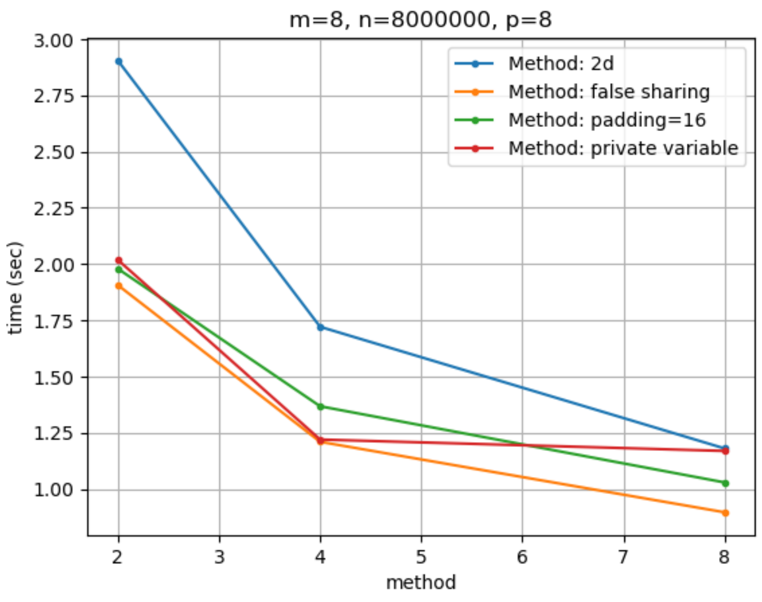


Similar behaviour is observed for both $p = 2$ and $p = 4$.

We observe that, in addition to mitigating false sharing, both the private variable and padding methods are able to significantly lower execution times when increasing the number of threads.

2.3.4 Dimensions: $m = 8, n = 8000000, p = 8$

For these combinations of parameters, similar to 2.3.2, the performance of the algorithms is not affected by false sharing. Additionally, false sharing should not occur here, that is if array is aligned, cache lines should completely fill with elements from only one thread.



3 Exercise 1.3

3.1 Description of the problem

The task of this exercise is to implement our own **read-write** locks to control the respective threads. We need to implement two approaches: locks where *priority* is given to **reading** threads and locks where *priority* is given to **writing** threads. A linked list program from the provided book -which utilizes `PThread` read-write locks- is provided for modification to incorporate our custom locks.

3.2 Brief description of the solution

The read functions (lock-unlock) are employed to safeguard the `Member()` function of the provided linked list module, while the write functions (lock-unlock) are utilized to safeguard the `Insert()` and `Delete()` functions.

3.2.1 Readers Priority

- `read_lock(Args args)`: This function is responsible for handling the entry of read threads into the critical section and utilizes a `mutex` to guard it. If there is an active writer and readers seek access to execute the `Member()` function, the reader threads are suspended on a condition variable (`cond_read`), otherwise, access is granted.
- `read_unlock(Args args)`: This function is responsible for managing the exit of read threads from the critical section and utilizes a `mutex` to guard it. In case there are no readers intending to access the critical section, the reader thread invokes a signal operation on the condition variable `cond_write`, that unblocks the writer threads.
- `write_lock(Args args)`: This function is responsible for handling the entry of write threads into the critical section and utilizes a `mutex` to guard it. If there is an active writer or readers seeking access to execute their critical section, the writer threads are suspended on a condition variable (`cond_write`), otherwise, access is granted.
- `write_unlock(Args args)`: This function is responsible for managing the exit of write threads from the critical section and utilizes a `mutex` to guard it. In case there are readers intending to access the critical section, the writer thread invokes a broadcast operation on the condition variable `cond_read`, otherwise if there are writer threads waiting for access, the writer thread invokes a signal operation on the condition variable `cond_write`.

3.2.2 Writers Priority

- `read_lock(Args args)`: This function is responsible for handling the entry of read threads into the critical section and utilizes a `mutex` to guard it. If there is an active writer or writers seek access to execute the `Insert()` or `Delete()` function, the reader threads are suspended on a condition variable (`cond_read`), otherwise, access is granted.
- `read_unlock(Args args)`: This function is responsible for managing the exit of read threads from the critical section and utilizes a `mutex` to guard it. In case there are no active readers, the reader thread invokes a signal operation on the condition variable `cond_write`, that unblocks the writer threads.
- `write_lock(Args args)`: This function is responsible for handling the entry of write threads into the critical section and utilizes a `mutex` to guard it. If there is an active

writer or readers seeking access to execute their critical section, the writer threads are suspended on a condition variable (`cond_write`), otherwise, access is granted.

- `write_unlock(Args args)`: This function is responsible for managing the exit of write threads from the critical section and utilizes a `mutex` to guard it. In case there are writers intending to access the critical section, the writer thread invokes a signal operation on the condition variable `cond_write`, otherwise if there are reader threads waiting for access, the writer thread invokes a broadcast operation on the condition variable `cond_read`.

3.3 Presentation of experiments and analysis

Below, we provide the results for various percentages of read and write operations. Specifically, we executed the program with every (valid) combination of the following parameters:

- **Read Opertions**: $[0.9, 0.95, 0.999]$
- **Insert Opertions**: $[0.001, 0.005, 0.05]$
- **Threads**: $[2, 4, 8, 16]$

Initial list size is fixed to 1000 and the number of operations to 500000.
To perform this grid search and produce the corresponding plot, we created the script `plots.py`. To reproduce, execute:

```
$ cd readers_writers
$ make all
$ python plots.py
```

Listing 5: Matrix Multiplication Plotting

To perform manual executions, run:

```
$ cd readers_writers
$ make all
$ ./<priority> <thread_count>
```

Listing 6: Matrix Multiplication Execution

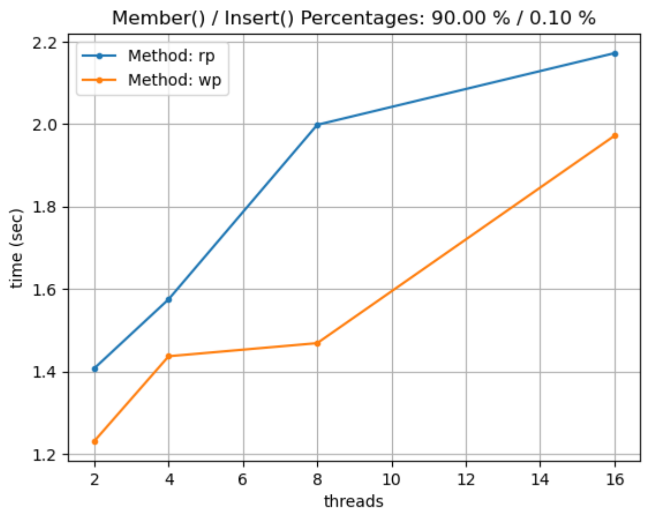
Experiments were conducted locally, 4 times each, using the following specifications:

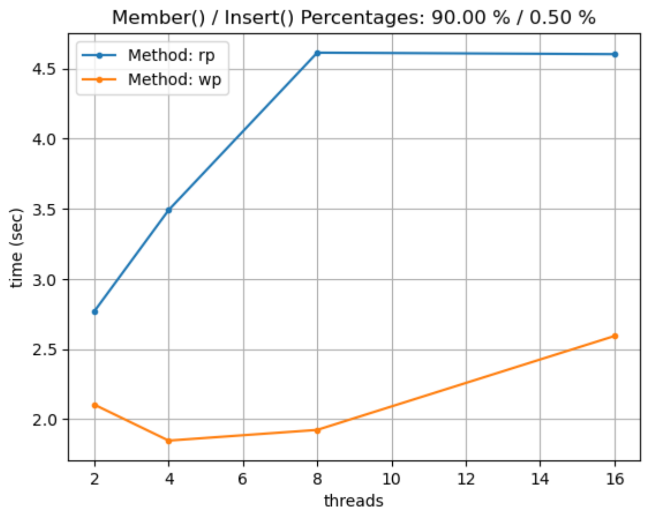
- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

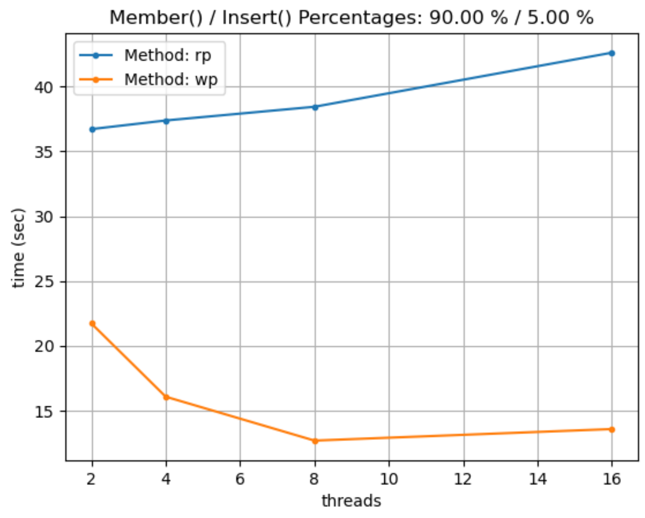
Our initial assumption was that prioritizing reading, given that the majority of operations performed are reads, would be beneficial for the overall execution time of the program. However, after implementing both algorithms and conducting various experiments, our assumption was proven incorrect.

The plots for each pair of read and write operations are presented below.

3.3.1 Read Percentage: 90 %



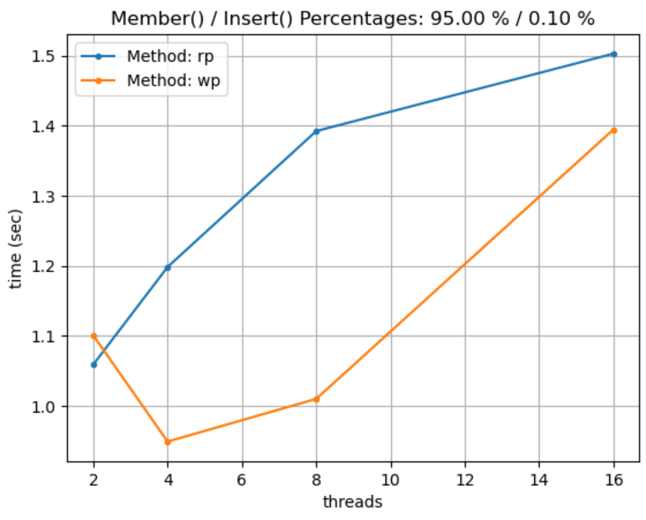


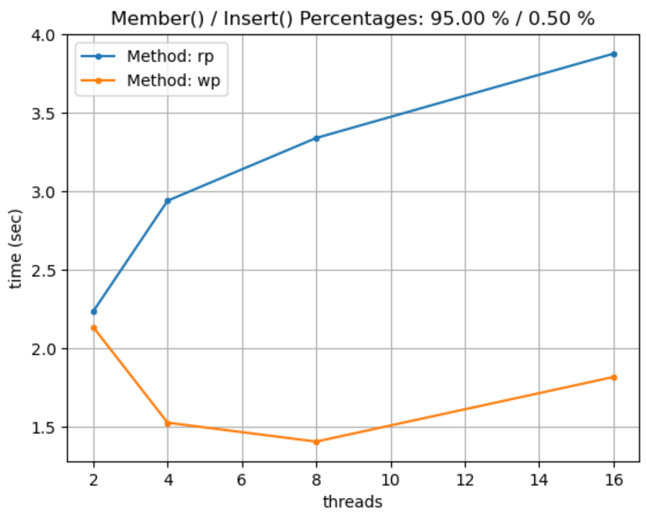


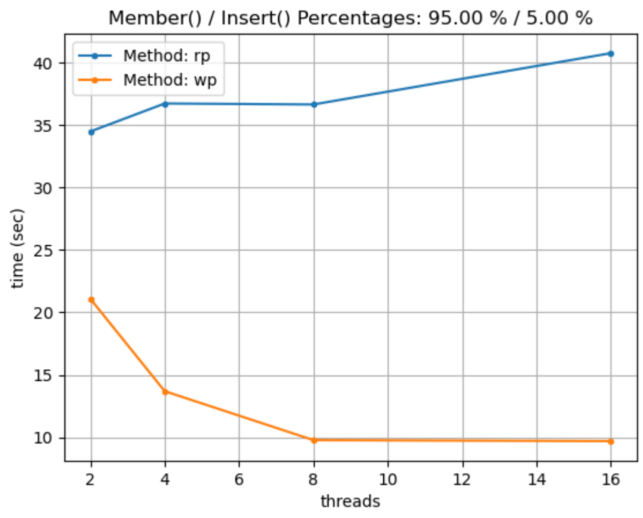
Insertion and deletion, as implemented in a sorted list, are resource-intensive operations with linear time complexity. This implies that writing threads become the bottleneck in performance. Given the initial list size is relatively small, when the ratio of insertions to deletions approaches zero, a notable portion of the list is removed. This enables faster insertions, allowing writing threads to perform their operations more quickly, resulting in similar performance for both priority types.

Conversely, as the aforementioned ratio approaches one, the list size increases, causing writing threads to take even longer to finish. In this scenario, giving reading threads priority creates the following issue: Threads that are currently waiting to write will wait a significant amount of time before doing so, leaving them unable to perform the reading operations they are intended to do later. Prioritizing writing threads resolves this issue by allowing write operations to finish as soon as possible, enabling **all** threads to concurrently perform their reading operations, instead of some of them waiting too long to write something before reading. In other words, writers priority enables better parallelization.

3.3.2 Read Percentage: 95 %

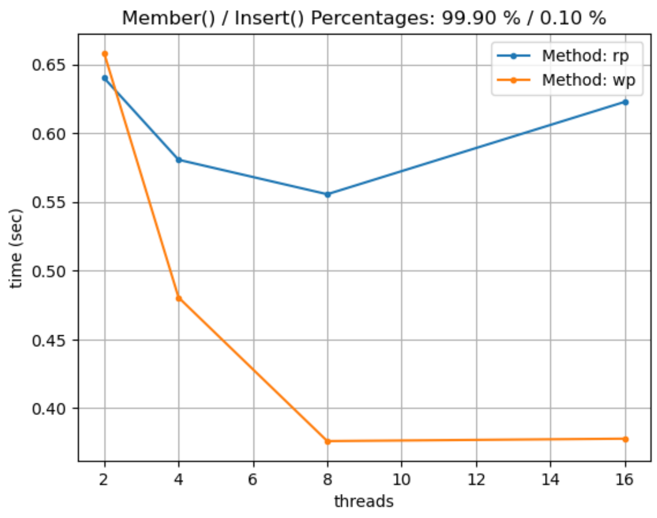






The results substantiate the previous analysis.

3.3.3 Read Percentage: 99.9 %



The results substantiate the previous analysis.