Report

Assignment 1

Ioannis Koutoulas and Alexandros Tsagkaropoulos

December 2, 2024

Exercise 1.1

Introduction

The exercise aims at computing the value of π with the Monte Carlo method. Specifically, the method counts the number of throws on a square, with side length of 2 m, that reside in the inscribed circle, with radius of 1 m. When the throws inside the square follow the uniform distribution this number, divided by the total number of throws, approaches the constant $\pi/4$. Therefore, in order to compute the value of π, a scale factor of 4 should also be applied to the fraction.

Methods

The Monte Carlo method is executed with both serial and parallel programs. The throws on both are simulated with the re-entrant version of the pseudo-random number generator (`rand_r`) offered by `libc`. On the serial program, the `seed` is always set to zero. On the parallel program, the `seed` on each thread is equal to its rank. All the work is distributed evenly across all threads, by dividing the total number of throws by the number of threads. Also, a mutex is utilized to synchronize the access of threads to the global variable that stores the value of π. To reduce the need of synchronization between threads, the value of π each one calculates is stored in a local variable and only one time, before it joins the main thread, it locks the mutex to update the global variable. The timing measurements utilized the `timer.h` header file that was given in Assignment's 1 supplementary material [1]. The timing measurements do not take into account the initialization of variables and structures of any kind prior to the execution, following the [2].

The system on which tests were executed was part of the Linux Lab. The host name of the system was `linux03` and its operating system was Ubuntu 20.04.6 LTS. The CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Lastly, the compiler used to create the binary executables was the GCC 9.4.0 with the optimization flag `-O2` set.

Results

All the experiments were automated by the script attached to the deliverable. Each experiment was run ten times and simulated 10^9 throws to calculate the value of π. Only the first three fractional points of π were successfully calculated in all cases, so the values were deemed unnecessary to reference. For the parallel program the number of threads varied from one to eight. The results for timing measurements are depicted on Figure 1. In addition, the timing measurements and the speedup, efficiency metrics are gathered on the Table 1.

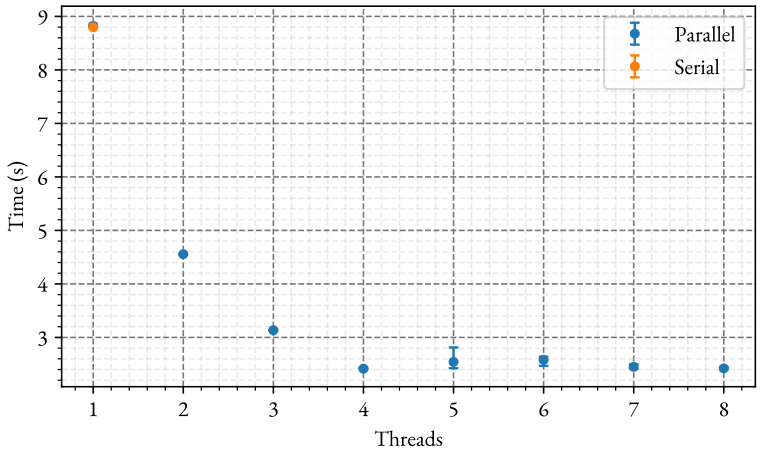


Figure 1. Results from experiments performed with 10^9 throws. Each mark-point in the Figure represents the mean of ten runs with uniformly-distributed input data. Error bars indicate the minimum and maximum values obtained across all runs.

1.1 Discussions

The first observation on the data points of Figure 1 is that execution time is less when the number of threads is no greater than the number of processor's cores. Notably, the decrease in execution time is more

Threads	Time (s)	Speedup	Efficiency
–	8.82	1.00	1.00
1	8.80	1.00	1.00
2	4.56	1.93	0.97
3	3.14	2.81	0.94
4	2.42	3.64	0.91
5	2.54	3.46	0.69
6	2.58	3.41	0.57
7	2.45	3.59	0.51
8	2.42	3.63	0.45

Table 1. Results from experiments performed with 10^9 throws. Each time value in the second column represents the mean of ten runs with uniformly-distributed data. The speedup and efficiency metrics are calculated with respect to the first data row, which corresponds to the serial execution.

significant when two threads are used. Each subsequent increase in the number of threads offers improved but demising returns. After the thread number gets bigger than the number of cores in the host system, the execution time stays flat, or even increases. This behavior is expected, since the resources are utilized more efficiently when threads are no more than the cores of the machine. Then, the kernel's scheduler can assign each thread to a distinct core and let them run concurrently in parallel. This is lucid from the efficiency metric of Table 1, where efficiency stays above 0.91%, when the thread numbers is below five. On the other hand, efficiency drops substantially when the host computer gets over-subscribed and reaches the 0.45 mark when the number of threads is double the core count. Even the deviation of the results increases, as seen in Figure 1, when the system is over-committed. The non-deterministic scheduling of waiting threads in cores introduces variations in timing measurements, that are not detected in the other case.

The speedup approaches the optimal when the number of threads is bound from above by the core count. The data reported on Table 1 indicate that the speedup is no further than 9% from the optimal value, in this range. However, when the system gets over-subscribed the speedup remains still or even declines. This is due to the overhead of managing more threads than the cores of the host.

The serial program execution is 0.2% slower than the parallel program when only one thread is executed. This result is unexpected and it might be due to the compiler's optimizations. Specifically, we believe that in the serial program the final multiplication and division, which are performed to calculate π, both employ floating point arithmetic. Conversely, in the parallel program the multiplication is performed with integers and the final division with floating point numbers.

Exercise 1.2

Introduction

In this exercise the goal is to create a pthread program, where each thread updates a common global variable. Specifically, each thread executes a for loop of some number of iterations and it adds 1 to the value of the variable. If race conditions are avoided with some synchronization mechanism the final value of the variable has to be deterministic. Supposing the initial value of the variable is 0, if the number of threads is P and the number of the for loop iterations within each thread function is I then 1 is added to the value I times for P threads, which means that at the end of the program the value of the variable must be: $I \cdot P$.

Methods

The exercise proposes two ways of implementation. The first is by using pthread locks, which are the mutexes of the `pthread` library and the second is by using atomic commands provided by the GNU GCC compiler. With the mutex lock implementation a single mutex is used to define a critical section. The critical section is the "add-one" operation on the common variable from each thread. The mutex is locked by the first available thread which adds one to the value of the variable and then unlocks the mutex, so that other threads can do the same. In the atomic commands implementation we leveraged the provided atomic functions of the `stdatomic` library. More specifically for the current program the `atomic_fetch_add()` function implements the addition of one to the common variable. As an atomic operation it performs the fetch, addition and storage of the new value of the variable in uninterrupted and indivisible steps, thus avoiding race conditions and also the overhead of the mutex lock and unlock functions.

Both mutex lock and atomic operations programs are designed so that they take as command line arguments the number of execution threads and the number of iteration for the for loop of the function of each thread. The programs measure and print the execution time on the command line. Execution time is measured by storing "snapshot" values of the time before the thread creation and after the termination of all of them and then by computing their difference. The timing measurements do not take into account the initialization of variables and structures of any kind prior to the execution, following the [2]. The timing measurements utilized the `timer.h` header file that was given in Assignment's 1 supplementary material [1]. The programs also print the expected common variable value and its actual value after execution. The executables are built from C code using the make build system and their build and execution are performed by a dedicated python script. The goal is to compare the performance of the two approaches for different numbers of threads and iterations. Other python scripts automate this process by running the programs for varying arguments ten times over, so as to capture the execution times, produce their average and store it in appropriate files. This scripts also assure that the expected and actual value of the variable after execution are the same, or else errors are generated.

All programs where ran on a Linux Operating System (Ubuntu 20.04.6 LTS) on a PC that is part of the Linux Lab (more specifically the one with host name `linux25`). The system's CPU model is Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Each core has cache memory with cache line size of 64 bytes. The sharing of the system with other users or other processes was avoided, as much as possible. The compiler used was GCC with version 9.4.0, with the lowest optimization settings (`-O0`).

Results

To compare the performance of the mutex lock program implementation to that of the atomic operations program implementation the average execution time over ten executions is measured for various thread counts and number of for loops iterations of the threads' function. The number of threads varies from 1 up to 8 and the number of for loop iterations varies from $10^3 = 1000$ up to $10^6 = 1\,000\,000$ exponentially (for each execution we make the number of iterations ten times larger). Note that the scripts found that for all executions the expected and the actual common variable values are identical and therefore the programs are correct.

All the results are summarized in the presented Tables and the Figure. Table 2 presents the timing results for the mutex lock implementation, while Table 3 presents the same results for the atomic operations implementation. The largest number of iterations (10^6) showcases the expected behavior in the best possible way, since the overhead of thread creation and termination is insignificant compared to the whole execution time. Thus Figure 2 shows how the execution time changes for the two implementations for 1 000 000 iterations as the thread count is rising.

Iterations	Thread Number							
	1	2	3	4	5	6	7	8
1000	329	282	396	633	752	934	1169	1471
10000	758	3138	3892	4345	6709	8920	10 906	12 614
100000	5754	26 723	29 529	36 684	48 160	55 836	51 896	62 666
1000000	40 700	111 350	141 779	238 300	289 308	337 158	392 150	447 600

Table 2. Mutex lock implementation execution time results (measured in μs) for varying threads and iterations of the thread function for loop that updates the common variable.

Iterations	Thread Number							
	1	2	3	4	5	6	7	8
1000	311	210	236	317	421	525	590	634
10000	453	1639	3316	3617	4942	6661	7978	7113
100000	1881	19 458	26 513	35 196	32 979	39 518	47 994	48 700
1000000	17 414	84 492	118 928	180 935	203 542	262 235	301 651	334 203

Table 3. Atomic operations implementation execution time results (measured in μs) for varying threads and iterations of the thread function for loop that updates the common variable.

Discussions

From the presented results a common behavior can be discerned for both of the programs. First of all, the programs take longer to execute as the number of iterations rises. The execution time rises by one order of magnitude as the iterations become ten times larger. This is expected since the "add-one" operation inside the thread function for loop is executed more times. The execution time is also rising as the thread count increases. However, the rise in execution time differs between the two implementations.

In the mutex lock implementation the execution time ascends faster than the atomic operations implementation, as the iterations number grows. An almost linear rise in execution times is detected with the rise of the thread count for either implementation, but the atomic operations program consistently has smaller execution times, as it is clearly demonstrated by Figure 2. The reason for that is that the mutex lock and unlock functions that the `pthread` library provides introduce significant overhead to the execution. This is due to stack calls and system calls to change the mutex value to set the critical region. For larger number of iterations and threads this lock and unlock calls are executed more frequently and they have a serious impact on the execution time; let alone when the critical region is blocking for all the other threads, in a way, serializing the execution.

The atomic operations implementation has no lock and unlock mechanisms but it is based on provided atomic functions of the GCC compiler and the support from the ISA (Instruction Set Architecture) and the underling hardware. Those functions are directly compiled to indivisible assembly instructions. This means that if they start execution there is no way to block them or stop them before they terminate. Therefore race conditions are entirely avoided when multiple threads attempt to modify the one common variable. Since the execution is blocking there are many threads that are blocked as one thread modifies the common value and there is still task switching between the threads; this is why the single threaded execution remains better. By avoiding the lock-unlock overhead, however, the atomic functions implementation results in considerably better execution times.

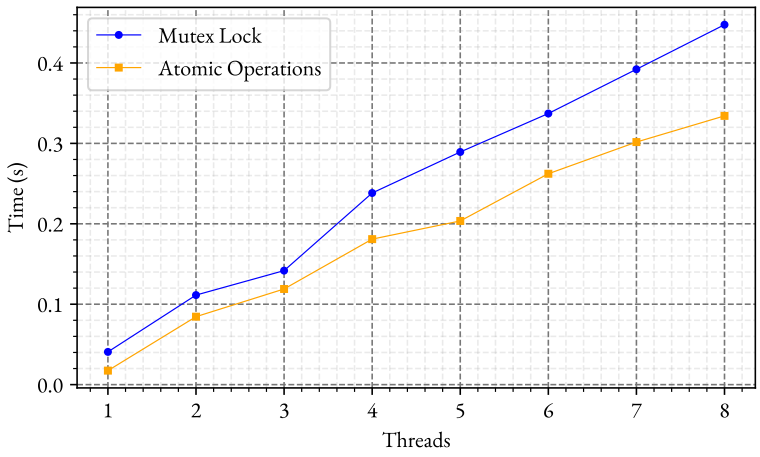


Figure 2. Timing results of the two programs for 10^6 threads' function for loop iterations and rising count of execution threads.

Exercise 1.3

Introduction

This exercise is expanding on Exercise 1.2 by having a common array between all threads. Each thread updates the value of one element of the array and the array size is the same as the number of threads. Specifically, the array values are initialized to zero and each thread executes a for loop where they add 1 to the value of the element that has the same index with their rank. The final value of all the elements must be deterministic as well as their sum. For P number of threads and I number of iterations the expected value of each element, after all threads finish executing their function, will be exactly the number of iterations I. The sum of all the elements of the array who all will have the value I will be $I \cdot P$.

In this exercise there are no critical sections, since every thread updates a different element of the array, but the problem of cache coherence arises. Whenever a thread accesses an element of the array it looks wether the element value already resides in the cache memory. If the value cannot be found in the cache a whole cache line of data is fetched from system memory which contains the element's value and the values of all the neighboring elements from the consecutive memory addresses. For C programs and GCC compiler with no optimization an array of elements has its values stored continuously in the memory. In a multi-threaded application in a multi-core system it is very likely that many threads execute in different cores and each core has its own cache memory. If all the threads update a common structure, like an array in this case, even if they do not update the same memory address, they have to communicate the changes in cache lines containing similar elements, in case some other thread needs the updated elements in the near future and will fetch the wrong value from the out-of-date cache lines. This is achieved by a mechanism of invalidating cache lines that have updated elements.

This mechanism however creates a serious problem in this Exercise: with every single "add-one" operation for every thread its cache line is invalidated and so are all the cache lines of other threads with neighboring array elements. What makes matters worse is that it is not unusual for a cache line to be able to fit the entire array in it. In this case one thread update means invalidation of all the other thread's cache line. This phenomenon is referred as *false sharing*. Even if the threads only ever access and update one element of the array they will always need to fetch its value from main memory making the execution incredibly inefficient. The goal is to study this behavior and propose a solution.

Methods

For this Exercise three programs are submitted. The first one is the direct implementation of the problem without any optimization to battle false sharing. The other two are programs with approaches to avoid or directly account and compensate for false sharing.

The first solution is to use as many local variables as possible. For this particular problem the function that the threads execute is redesigned so that the initial value of each element of the array is copied to a local variable of the function. Then the for loop with the "add-one" operation acts on the local variable with the copied value. In the end the resulting value must be written in memory. The first thread that reaches this point will invalidate the cache line of every other thread. In the end false sharing is not overcome but it is avoided for every "add-one" operation in the thread function's for loop. For a large number of iterations this should yield better efficiency.

The second solution is to eliminate false sharing by spacing out the array's values by the size of the system's cache line. This approach is very intrusive on the initial code. First of all the correct size of the system's cache line length must be input as a command line argument. Then the idea is that each element in the array must be in the beginning of a cache line, and the rest of the cache line will be just junk values. If the array originally contained elements of some type with size in bytes T and it had P elements, then in the initial program the memory block allocated by the array had size $T \cdot P$. However for the proposed approach an array of size $T \cdot P \cdot C$ must be allocated, where C is the cache line size. Then the elements of interest starting points in memory must be spaced out by the size of the systems cache line. This means that their indexes in the array are not consecutive no more, but instead they must be scaled by a constant. This constant is precisely C/T. One peculiarity of this is that the cache line length must exactly divide the size of the stored data types and also the cache line length must be bigger than the size of the data types. With this approach whenever a thread fetches or updates a value of interest from the array the entire cache line will remain local to the thread and will not be invalidated by other threads and this remains true throughout the execution of all the threads.

All the programs are designed so that they take as command line arguments the number of execution threads and the number of iteration for the for loop of the function of each thread. The second solution program with the junk value addition approach also takes as command line argument the system's cache

line length. The programs are designed to measure and print the execution time on the command line and also the expected values of the elements of the array and their sum, as well as their actual values after the termination of execution. Execution time is measured by storing "snapshot" values of the time before the thread creation and after the termination of all of them and then by computing their difference. The timing measurements do not take into account the initialization of variables and structures of any kind prior to the execution, following the [2]. The timing measurements utilized the `timer.h` header file that was given in Assignment's 1 supplementary material [1]. The executables are built from C code using the make build system and their build and execution are performed by dedicated python scripts. The goal is demonstrate some improvement in the efficiency of the two solutions compared to the initial approach, for different numbers of threads and iterations. Other python scripts automate this process by running the programs for varying arguments ten times over, as so to capture the execution times and produce their average value and store it in files. These scripts also capture the expected and the actual values and it compares them, to move on with the storing of the execution times or generate errors if they differ.

From the average execution times the efficiency of the programs can be computed. Efficiency is defined as the ratio of the average execution time of the single-threaded execution to the product of the average multi-threaded execution time and the number of threads. As expected the single-threaded execution will always have efficiency equal to 1, while the multi-threaded execution with efficiency as close to 1 as possible indicates good performance.

All programs where ran on a Linux Operating System (Ubuntu 20.04.6 LTS) on a PC that is part of the Linux Lab (more specifically the one with host name `linux25`). The system's CPU model is Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Each core has cache memory with cache line size of 64 bytes, which means that for all the current problem's executions the entire element array can fit inside one cache line (the array has elements of type `long long int`). The sharing of the system with other users or other processes was avoided, as much as possible. The compiler used was GCC with version 9.4.0, with the lowest optimization settings (`-O0`).

Results

The performance of the three programs is evaluated by the efficiency metric, as was described in the previous section. The efficiencies stem from the average timing results over ten executions of the programs for various thread counts and for loop iterations of the threads' function. The number of threads varies from 1 thread up to 8 threads and the number of for loop iterations varies from $10^3 = 1000$ up to $10^6 = 1\,000\,000$ exponentially (for each execution we make the number of iterations ten times larger).

Iterations	Thread Number							
	1	2	3	4	5	6	7	8
1000	1.000	0.654	0.288	0.204	0.123	0.093	0.073	0.059
10000	1.000	0.590	0.205	0.141	0.088	0.075	0.067	0.048
100000	1.000	0.199	0.080	0.041	0.032	0.029	0.022	0.015
1000000	1.000	0.185	0.093	0.064	0.039	0.035	0.033	0.027

Table 4. Initial simple non-optimized implementation's measured efficiencies for varying threads and iterations of the thread function for loop that updates an array element value.

Iterations	Thread Number							
	1	2	3	4	5	6	7	8
1000	1.000	0.484	0.136	0.183	0.096	0.086	0.068	0.083
10000	1.000	0.574	0.302	0.202	0.102	0.106	0.088	0.055
100000	1.000	0.456	0.297	0.128	0.110	0.069	0.075	0.060
1000000	1.000	0.476	0.263	0.152	0.089	0.080	0.063	0.056

Table 5. Thread local variable implementation's measured efficiencies for varying threads and iterations of the thread function for loop that updates an array element value.

Iterations	Thread Number							
	1	2	3	4	5	6	7	8
1000	1.000	0.418	0.247	0.158	0.091	0.079	0.052	0.058
10000	1.000	0.521	0.331	0.202	0.134	0.113	0.090	0.064
100000	1.000	0.516	0.316	0.171	0.110	0.118	0.097	0.057
1000000	1.000	0.620	0.380	0.244	0.133	0.112	0.095	0.066

Table 6. Junk value injection implementation's measured efficiencies for varying threads and iterations of the thread function for loop that updates an array element value.

All the results are summarized in the presented Tables and the Figure. Table 4 presents the efficiency values of the initial non-optimized problem, where no care is taken against false sharing of cache lines. Table 5 contains the efficiency values for the first solution to false sharing which was the use of a local variable to store the "add-one" results of the threads' function for loop. Finally Table 6 shows the efficiencies of the second solution, which was the injection of junk values to a larger array to space out the values of interest so that every fetch of a value provides a different cache line for every thread. The largest number of iterations (10^6) showcases the expected behavior in the best possible way, since the overhead of thread creation and termination is insignificant compared to the whole execution time. Also for the initial plain implementation a large number of iterations will probably result in more cache misses compared to the two solution programs. Therefore Figure 3 shows how the efficiency changes for the three implementations for 1 000 000 iterations as the thread count is rising. Note that the scripts found that for all executions the expected and the actual common variable values are identical and therefore the programs are correct.

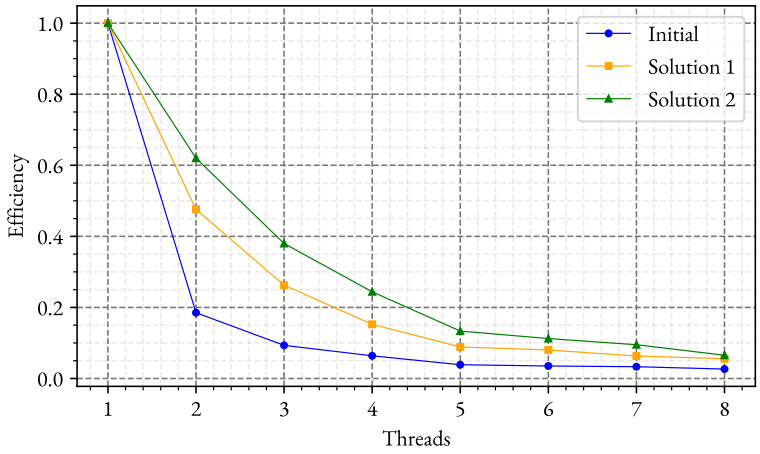


Figure 3. Efficiencies results of the three programs for 10^6 threads' function for loop iterations and rising count of execution threads.

Discussions

The presented results show some similarities and some differences in the way that the thread count and the number of iterations impact the efficiency of the three programs. First of all it is expected for the efficiency to be 1 for the single threaded execution, by definition. Also it is inevitable for the efficiency to drop with the rise in thread count for all three programs. This is because more threads means larger overhead for their creation and termination (time measuring snapshots are before and after thread creation and termination respectfully). For single threaded execution there is also no task switching.

However the efficiency does not drop by the same rate for all three programs. Note also that the efficiency diminishes with the rise of iterations for the initial unoptimized implementation, which is not the case for the two solutions. The plain unoptimized implementation presents better efficiencies for the smallest number of iterations (10^3). This is due to the simplest structure and less instructions in the thread function. There is false sharing but the number of iterations is so small that the thread creation and termination overhead along with the decreased function calls result in a slightly faster execution compared to

the more complex solution programs. Although as the number of iterations rises the efficiency drops dramatically, especially for more than 10^5 iterations. This means that false sharing becomes severe for more iterations due to the invalidation of cache lines many times over.

Looking at the first solution program's results, where "add-one" operation's results are stored and performed in a local variable, a different behavior is detected. Efficiency peaks for 10^4 iterations and drops slightly for more, for thread counts up to 4. For more thread the results are more inconsistent. The system where the program was ran has 4 logical cores, so more threads result in worse efficiency and unpredictable results, since some of them are executed on the same core; this means that they share cache lines but they also suffer more from task switching, and no safe conclusions can be drawn. However for more than 10^4 iterations the program has consistently better efficiency than the initial unoptimized program. While it still suffers from false sharing close to the beginning and termination of the threads it definitely avoids it for the middle time of the execution's time interval. There, all the "add-one" operations are performed locally to the execution cores of each thread. For 10^4 iterations the program seems to find an optimal balance between false sharing at the temporal edges of execution and the middle-of-execution local operations.

The second solution implementation, with the addition of junk values in the array to space out the elements in the cache line's length, provides the optimum improvement to the initial unoptimized solution. Purposely ignoring the results for more than 4 threads, for the same reasons with the first solution program (tests where ran on a system with a 4-core CPU), an improvement in efficiency is detected as the number of iterations rises. For small number of iterations this program has more complicated execution compared to the other two, with more operation calls for tasks like array index multiplication, resulting in worse efficiencies. But for large number of iterations, where the other two programs are plagued by false sharing, this program completely avoids it. The best efficiencies are for the largest number of iterations (10^6) where all the "add-one" operations in the threads' function for loops act on elements that always reside in the local caches of each core.

Concluding, for the largest recorded number of iterations (10^6) the trends of the three programs are obvious from the results presented in Figure 3. The initial unoptimized problem has the worst efficiencies. The first solution with the utilization of local variables improves the efficiency, but the false sharing issue remains. The junk injection to space out array elements solution has clearly the best efficiencies of the other two programs. It has more than double the efficiency of the initial program and better efficiencies from the first solution, for any thread count and has much better efficiencies for thread count up to 4, who coincide with balanced execution of each thread on one of the 4 cores of the CPU.

Exercise 1.4

Introduction

The exercise considers the implementation of a custom read-write lock with two policies: (1) reader prioritization and (2) writer prioritization. Both determine which thread to allow to acquire the lock depending on its state (reader or writer). The incentive is to guard a linked list from race conditions while achieving great performance. The read and write operations' percentages on the linked list will vary in kind to evaluate the strengths and weaknesses of both techniques.

Methods

The implementation of the custom read-write lock consists of an opaque data type and four functions. The opaque data type abstracts a C `struct` that consists of a mutex, which guards the execution of any of the functions, and two conditional variables: one for reader threads and one for writer threads. Furthermore, the read-write lock data structure contains counters for waiting and executing readers and writers and a field for the most recent writer. As for the functions, the two of them are responsible for the creation/initialization and destruction of the read-write lock. The memory management is dynamic in the initialization function and thus the destruction function is necessary to free the allocated memory for the lock. The remaining three offer functionality for acquiring of the lock for reading and writing and unlocking, when the holding thread completes its operation. The unlocking function is were the policies are enforced. Specifically, the reader prioritization first unblocks readers and if no reader is waiting on the corresponding conditional variable, it signals any waiting writer. On the contrary, the writer prioritization policy only considers signaling waiting readers if no writer is waiting on the corresponding conditional variable. Specific to the unlocking function is the writer field of the read-write lock that identifies the role (reader or write) of the unlocking thread. Additionally, an implementation detail concerns the `cond_wait` function of pthread library utilized in both locking function. The remedy to *spurious wake-ups* [2] from the function is looping till the condition predicate and the returned type have the expected values. The implementation of the linked list and the main program was taken from the supplementary material of the Assignment 1 [1] and any modification are made onto it.

The system on which tests were executed was part of the Linux Lab. The host name of the system was `linux03` and its operating system was Ubuntu 20.04.6 LTS. The CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Lastly, the compiler used to create the binary executables was the GCC 9.4.0 with the optimization flag `-O2` set.

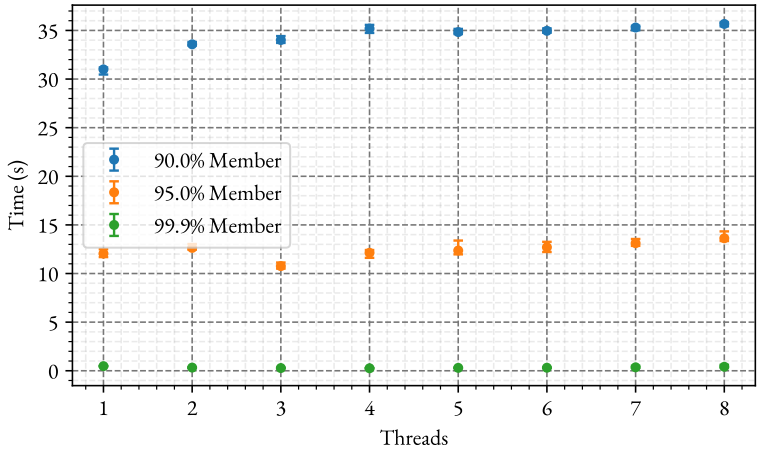
Results

All the experiments were automated by the script attached to the deliverable. It compiles the source code, with the definition of a preprocessor directive that applies the corresponding policy, and runs the experiments ten times each. This operation is repeated for the two policies outlined in the previous Section. Each experiment starts with the linked list being filled with 1000 randomly-distributed keys and the total number of following operation on it being 500 000 in number. When multiple threads spawn to perform the operations the load is split evenly across them. The operations on the list are of three kinds: `Member`, `Insert` and `Delete`. The variable parameters of the experiments are the percentages of the operation kinds. The values of 99.9%, 95% and 90% for `Member` operations are tested. The remaining operations share the same percentages so as all of them to add up to one. The results from the experiments are depicted in the Figure 4.

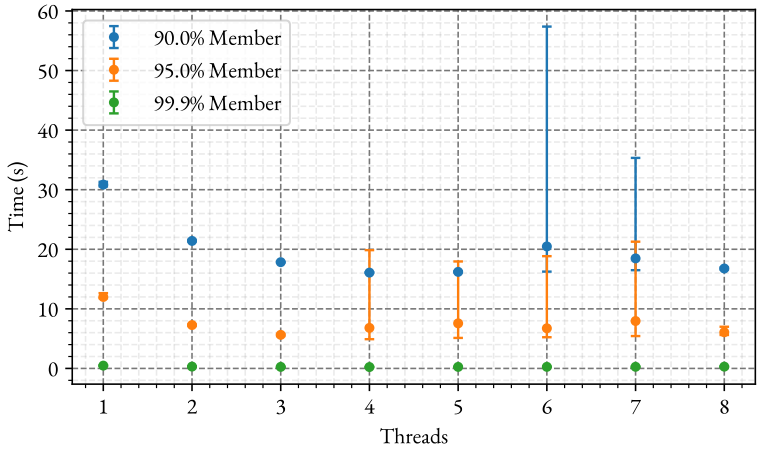
Discussions

The custom read-write lock policies will be compared. It is clear from Figures 4a and 4b that in a multi-threaded application, the write prioritization policy consistently outperforms the read prioritization policy. When readers are prioritized, writer threads starve until no reader is either executing or waiting to acquire the lock. The unblocking of a writer is a relatively rare event due to the majority of operations being `Member` operations in combination with the fact that more than one readers simultaneously being able to execute `Member` operations. Therefore, writer threads remain blocked for extensive time intervals deteriorating the performance of the application. Actually, the timing measurements state that, despite multiple threads operating on the linked list, it might take longer to complete the 500 000 operations than a single thread, by applying the reader prioritization policy.

At the same time, the writer prioritization policy manages to speedup the execution of the application, when there is no over-subscribing. The relatively quick unblocking of writer threads utilizes the resources



(a) Custom read-write locks with reader prioritization policy.



(b) Custom read-write locks with writer prioritization policy.

Figure 4. Each mark-point in the Figures represents the mean of ten runs with uniformly-distributed input data. Error bars indicate the minimum and maximum values obtained across all runs. Initially, the linked list was filled with 1000 keys and measurements took place in the following 500 000 operations, being split evenly among threads.

of the host system more effectively. The effect is more pronounced when write operations are more frequent. In the case of write operations being 0.1% there is no difference in execution times when varying the number of threads or the policies. This is expected since blocked writers occur infrequently and they bare negligible burden to the overall performance. In some cases, the writer prioritization policy experiences significant maximum execution times. This behavior is expected when alternating writers execute write operations and exclude/block any reader from acquiring the read lock, forming a serialization period in execution. These cases become more apparent the higher the percentage of write operations, since the probability of successive writers requesting access to the linked list rises. Lastly, the overhead of context switching and cache thrashing result in worse performance, regardless of the policy applied, in the case of over-subscription.

References

[1] Vasileios Karakostas. *Assignment 1 (Mandatory) – Programming with Pthreads*.

[2] Peter Pacheco, Matthew Malensek. *An Introduction to Parallel Programming*. 2nd Edition.