Systems Programming Assessed Exercise 2

Drew Mackay 2652958M
Joshua McKeown 3067543M

December 18, 2025

1 Introduction

Different programming languages handle concurrency and memory management more efficiently than others. Choosing the right fit for the task at hand is vital for low-level systems where latency matters. Here at *SpaceY*, we are embarking on a brand new project that involves an array of sensors, collecting data about the Earth's atmosphere, from space! This report aims to investigate three programming languages: C, Java, and Python. We summarise the differences between how these languages deal with concurrency, synchronisation, and memory management. Then we empirically test the execution speed of these languages under an applicable workload. Finally, we recommend a language to use on the project going forward, based on the evidence collected.

2 Summary of Concurrency and Synchronisation

Concurrency is when multiple threads or processes make progress during overlapping time periods, this is not necessarily in parallel. Parallelism is when tasks execute at the exact same time on multiple cores. Concurrency is difficult to handle as it introduces challenges such as race conditions, critical sections and non-deterministic execution order. These issues require careful synchronisation to ensure the program runs correctly. Synchronisation exists in concurrent systems to coordinate access to shared resources and prevent incorrect behaviour caused by race conditions. Synchronisation refers to the techniques used to control the execution order of threads or processes; this ensures correct access to shared resources. Without synchronisation, issues may arise such as inconsistent programs and data corruption.

Feature	C	Java	Python
Thread Model	Thread libraries e.g. POSIX threads, follows one-to-one mapping between user threads and kernel threads	Uses native OS threads managed by the JVM, also follows one-to-one mapping	User-level threads managed by the Python interpreter; constrained by the Global Interpreter Lock
Parallelism	Supports parallel execution across multiple cores, suitable for CPU-bound workloads	Supports parallelism, allows multiple threads to execute concurrently on multi-core processors	Limited parallelism; multiple threads cannot execute Python bytecode simultaneously
Ease of use	Low: threads must be created and synchronised explicitly; memory management must be handled carefully	Moderate: high-level abstractions, but the concurrency API can be complex	High: simple syntax and abstractions make concurrent programs easier to write
Performance Implications	High performance with minimal runtime overhead; performance depends on correct synchronisation	High performance with some runtime overhead, acceptable in exchange for safety	Lower performance for CPU-bound tasks
Scalability	Highly scalable, but poor design choices can limit scalability	Good scalability, especially when using thread pools	Poor scalability with threads; improved scalability achievable using multiprocessing
Typical Use Cases	Operating systems, embedded systems, high-performance computing	Server-side systems, large-scale services	Web services, scripting, automation

3 Summary of Memory Management

Memory management refers to the methods used to allocate, organise, and reclaim memory during program execution. It is vital for ensuring efficient use of memory, isolating processes from one another and maintaining stability and performance

throughout the program. In a modern operating system, virtual memory is used to provide every process with an illusion of a large address space, meanwhile transparently handling memory sharing. Using C, Java and Python as points of comparison allows is to highlight different approaches to memory management. C uses manual memory management which requires the programmer to explicitly allocate and deallocate memory. This provides strict control over memory usage but introduces more risk from memory leaks if memory is mismanaged. Python uses automatic memory management which is based on reference counting. This approach abstracts memory allocation and deallocation away from the programmer which decreases risk and improves ease of use at the cost of reduced control and additional runtime. Java also relies on automatic memory allocation and reclamation which is handled by the Java Virtual Machine (JVM) through garbage collection. This method enforces strong safety guarantees and prevents direct memory access. This prevents common errors and introduces a higher runtime.

Feature	**C**	**Java**	**Python**
Memory Management Model	Manual memory management where the programmer explicitly allocates and frees memory	Automatic memory management handled by the Java Virtual Machine using garbage collection	Automatic memory management primarily based on reference counting
Programmer Responsibility	High: the programmer is fully responsible for memory allocation and deallocation	Low: memory allocation and reclamation are managed by the JVM	Low: memory is reclaimed automatically when objects are no longer referenced
Safety	Low: direct memory access allows invalid reads and writes if misused	High: strong safety guarantees enforced by the JVM prevent direct memory access	High: runtime checks and managed memory prevent memory access issues
Performance Implications	Minimal runtime overhead and predictable performance when managed correctly; errors can be costly	Increased runtime overhead due to garbage collection, but mitigated with JVM optimisations	Increased runtime overhead due to reference counting, impacting performance
Interaction with Virtual Memory	Closely aligned with operating system virtual memory mechanisms	Managed transparently by the JVM, which utilises the OS virtual memory system	Managed away from the programmer and instead by the interpreter and operating system

4 Execution Time Comparison

It was attempted to simulate a system where there are 1326 sensors working simultaneously, carrying out computations of varying scale.

The aim is to compare which language is the most viable to support the needs of the *SpaceY* space-based solar power project. The most viable language will be able to run all 1326 sensors concurrently, and scale manageably with computation complexity, with minimal overhead for the creation and management of threads.

On a single machine running Ubuntu 20.04 under WSL[1], code was ran to perform a simple computation involving taking a decimal value, multiplying it by another decimal value, and then adding a decimal value, specifically `x = 1.0 * 1.5 + 1.0`. This computation is repeated a varying number of times, from 1,000 to 100,000,000 times. The computation was ran using 1326 threads to be completed concurrently.

This setup was created for each language under investigation: C, Java, and Python. For each, the time for the computation to execute was measured and repeated three times, and then averaged. The entire code used is shown in Appendix A, Appendix B, and Appendix C.

The results from our experiment comparing the execution times of each language over a range of task sizes are shown in Figure 1. The scales are logarithmic due to the exponentially increasing nature of our computation sizes. Lower execution times are better. Figure 1 shows a trend of Python being slowest at all stages. It shows that for small task sizes, C is the fastest. However, at larger task sizes, Java overtakes C as the fastest.

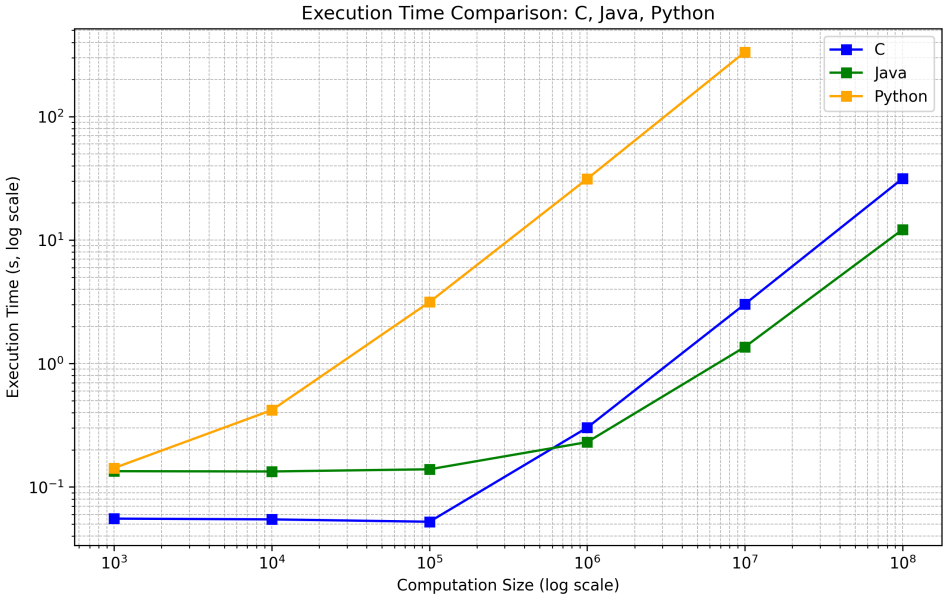


Figure 1: Execution times of a task of varying size, performed in C, Java, and Python.

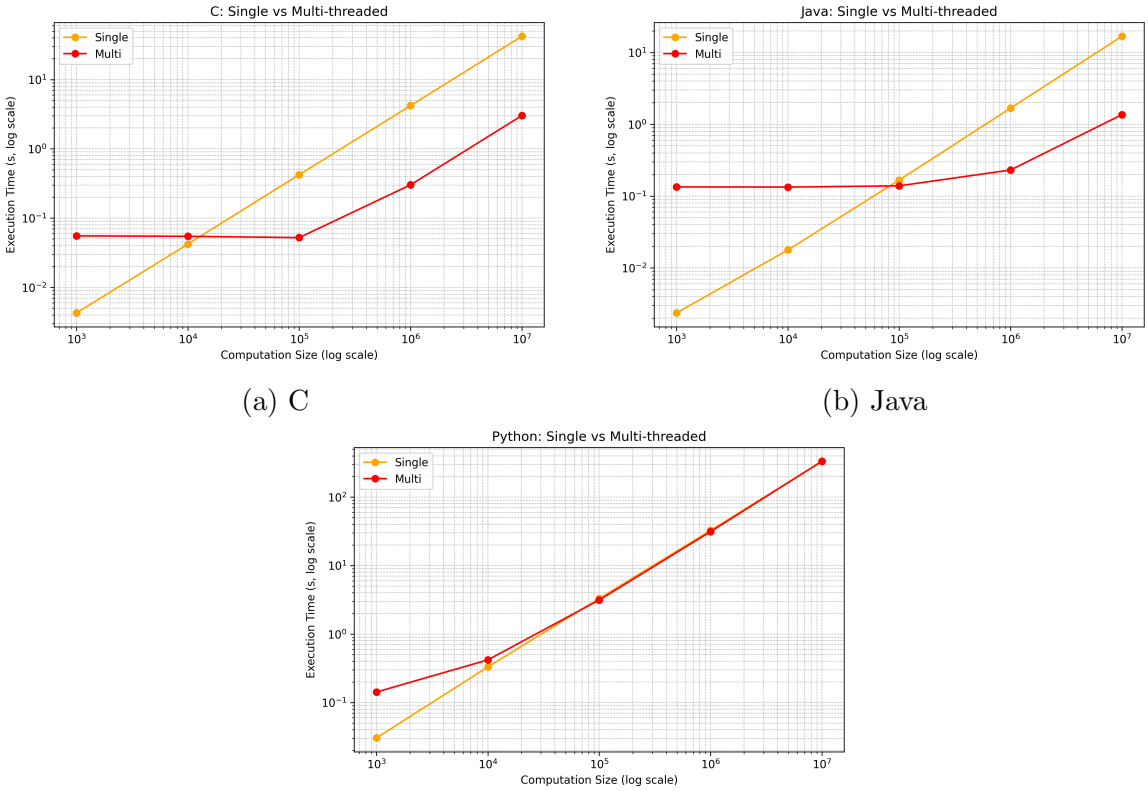
These results were surprising to our pre-conceived notions on these languages. Python was presumed to be the slowest, but it was even slower than expected. Additionally, it was expected that C would be faster than Java at all stages. The overtake at 10^7 was not predicted. The overhead necessary for garbage collection in Java was

expected to make it slower than C at all sizes.

5 Single-threaded vs Multi-threaded

For each language, the setup was performed in a single-threaded form as well, to compare the impact that multi-threading has on the execution time. While single-threaded is not particularly relevant to the goal of the project, it was performed to help explain results we observed with the Python implementation in Section 4.

The results from our experiment comparing the execution times when implemented as single-threaded vs multi-threaded are show in Figure 2. For C and Java, the graphs show a trend of single-threaded being faster for smaller computation sizes, but for larger computation sizes, multi-threaded implementations overtake. For Python, it is similarly seen that single-threaded computation sizes outperform at smaller computation sizes. However, it is not seen that multi-threaded overtakes at larger sizes. Instead, they converge and similar performance is seen.



(a) C

(b) Java

(c) Python

Figure 2: Single vs multi-threaded performance for each language.

For the Python implementation, the threading library[2] was used. Due to the Global Interpreter Lock[3] in Python, only one thread at a time can actually execute code. This explains the results seen in Figure 2, where single-threading is initially quicker, as the creation of threads is omitted. At the larger computation sizes, this static overhead becomes insignificant, causing the tie.

6 C Optimisation Compiler Flags

It was considered that the C code was executing slower than Java due to Java more aggressively optimising the code. The C code was then ran with and without the `O3` compiler flag.[4] The results are shown in Figure 3.

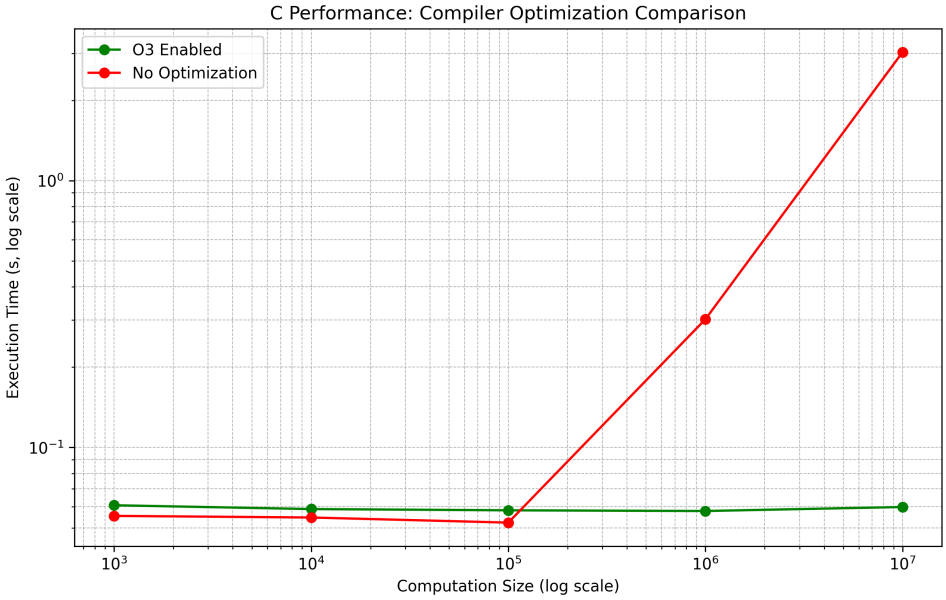


Figure 3: Comparison of C when compiler optimisation is enabled.

This changed things considerably. The code when optimised with the `O3` compiler flag was much faster at the higher computation sizes. In fact, it seemed to no longer scale with any linearity to the computations size. This is not an appropriate reflection of a real-world scenario, as a compiler would not be able to create a method to solve tasks of varying structure in constant time.

7 Java Garbage Collection Compiler Flags

It was considered that manipulating the garbage collection in Java could help explain results seen in Section 4. The experiment was repeated, comparing the use of compiler flags to set the maximum heap size extremely low, to 1MB, which should increase the pressure on the garbage collection and increase the execution time on Java compared to C. However, it was found that changing the heap size to overload the garbage collection did not significantly impact the execution time.

8 Conclusion

The aim was to compare which language support the needs of project's technical needs. The most viable language will be able to run all 1326 sensors concurrently, and scale manageably with computation complexity, with minimal overhead for the creation and management of threads. In Sections 2 and 3, C was to be a very strong

choice, with greater performance provided when implemented correctly. How these languages scale with computation complexity was measured in Section 4. Python is significantly slower than all options and scaled terribly. C and Java are a closer competition, with C outperforming Java at smaller sizes, and Java outperforming at higher sizes. Regardless, we still recommend moving forward with C for the project. One isolated test on one metric is not enough to change the precedence.

References

[1] `https://learn.microsoft.com/windows/wsl`. Accessed 16 December 2025.

[2] `https://docs.python.org/3/library/threading.html`. Accessed 17 December 2025.

[3] `https://docs.python.org/3/glossary.html#term-global-interpreter-lock`. Accessed 17 December.

[4] `https://www.ibm.com/docs/en/aix/7.2.0?topic=implementation-optimization-levels`. Accessed 17 December.

[5] `https://stackoverflow.com/a/53708448`. Accessed 16 December 2025.

Appendix A C code

The timing code is adapted from a StackOverflow answer.[5]

A.1 Single-threaded

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define TASK_LENGTH 10000000

int completed = 0;

void* sensor_task() {
    double x = 1.0;

    for (int i = 0; i < TASK_LENGTH; i++) {
        x = x * 1.5 + 1.0;
    }

    completed++;

    return NULL;
}

enum { NS_PER_SECOND = 1000000000 };

```

```
void sub_timespec(struct timespec t1, struct timespec t2, struct
    timespec *td)
{
    td->tv_nsec = t2.tv_nsec - t1.tv_nsec;
    td->tv_sec  = t2.tv_sec - t1.tv_sec;
    if (td->tv_sec > 0 && td->tv_nsec < 0)
    {
        td->tv_nsec += NS_PER_SECOND;
        td->tv_sec--;
    }
    else if (td->tv_sec < 0 && td->tv_nsec > 0)
    {
        td->tv_nsec -= NS_PER_SECOND;
        td->tv_sec++;
    }
}

int main() {

    int NUM_SENSORS = 1326;

    struct timespec start, finish, delta;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0; i < NUM_SENSORS; i++) {
        sensor_task();
    }

    clock_gettime(CLOCK_MONOTONIC, &finish);
    sub_timespec(start, finish, &delta);

    printf("Sensors: %d\n", NUM_SENSORS);
    printf("Completed: %d\n", completed);
    printf("Time: %d.%.9lds\n", (int)delta.tv_sec, delta.tv_nsec);

    return 0;

}
```

A.2 Multi-threaded

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>

#define TASK_LENGTH 1000

pthread_mutex_t lock;
int completed = 0;

void* sensor_task() {
    double x = 1.0;

```

```
    for (int i = 0; i < TASK_LENGTH; i++) {
        x = x * 1.5 + 1.0;
    }

    pthread_mutex_lock(&lock);
    completed++;
    pthread_mutex_unlock(&lock);

    return NULL;
}

enum { NS_PER_SECOND = 1000000000 };

void sub_timespec(struct timespec t1, struct timespec t2, struct
   timespec *td)
{
    td->tv_nsec = t2.tv_nsec - t1.tv_nsec;
    td->tv_sec  = t2.tv_sec - t1.tv_sec;
    if (td->tv_sec > 0 && td->tv_nsec < 0)
    {
        td->tv_nsec += NS_PER_SECOND;
        td->tv_sec--;
    }
    else if (td->tv_sec < 0 && td->tv_nsec > 0)
    {
        td->tv_nsec -= NS_PER_SECOND;
        td->tv_sec++;
    }
}

int main() {

    int NUM_SENSORS = 1326;
    pthread_t threads[NUM_SENSORS];

    pthread_mutex_init(&lock, NULL);

    struct timespec start, finish, delta;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0; i < NUM_SENSORS; i++) {
        pthread_create(&threads[i], NULL, sensor_task, NULL);
    }

    for (int i = 0; i < NUM_SENSORS; i++) {
        pthread_join(threads[i], NULL);
    }

    clock_gettime(CLOCK_MONOTONIC, &finish);
    sub_timespec(start, finish, &delta);

    printf("Sensors: %d\n", NUM_SENSORS);
    printf("Completed: %d\n", completed);
    printf("Time: %d.%.9ld\n", (int)delta.tv_sec, delta.tv_nsec);

    pthread_mutex_destroy(&lock);

```

```
    return 0;

}
```

Appendix B Java code

B.1 Single-threaded

```
public class JavaSensorTestSingle {
    private static final int TASK_LENGTH = 10000000;
    private static final int NUM_SENSORS = 1326;

    private static int completed = 0;

    static class SensorTask {
        public void run() {
            double x = 1.0;

            for (int i = 0; i < TASK_LENGTH; i++) {
                x = x * 1.5 + 1.0;
            }

            completed++;
        }
    }

    public static void main(String[] args) throws
    InterruptedException {

        SensorTask task = new SensorTask();

        long startTime = System.nanoTime();
        for (int i = 0; i < NUM_SENSORS; i++) {
            task.run();
        }

        long endTime = System.nanoTime();
        long delta = endTime - startTime;

        long seconds = delta / 1_000_000_000L;
        long nanoseconds = delta % 1_000_000_000L;

        System.out.println("Sensors: " + NUM_SENSORS);
        System.out.println("Completed: " + completed);
        System.out.printf("Time: %d.%09ds%n", seconds, nanoseconds);
    }
}
```

B.2 Multi-threaded

```
public class JavaSensorTestMulti {
    private static final int TASK_LENGTH = 10000000;
    private static final int NUM_SENSORS = 1326;
```

```
4
5       private static final Object mutex = new Object();
6       private static int completed = 0;
7
8       static class SensorTask implements Runnable {
9           @Override
10          public void run() {
11              double x = 1.0;
12
13              for (int i = 0; i < TASK_LENGTH; i++) {
14                  x = x * 1.5 + 1.0;
15              }
16
17              synchronized (mutex) {
18                  completed++;
19              }
20          }
21      }
22
23      public static void main(String[] args) throws
    InterruptedException {
24          Thread[] threads = new Thread[NUM_SENSORS];
25
26          SensorTask task = new SensorTask();
27
28          long startTime = System.nanoTime();
29
30          for (int i = 0; i < NUM_SENSORS; i++) {
31              threads[i] = new Thread(task);
32              threads[i].start();
33          }
34
35          for (int i = 0; i < NUM_SENSORS; i++) {
36              threads[i].join();
37          }
38
39          long endTime = System.nanoTime();
40          long delta = endTime - startTime;
41
42          long seconds = delta / 1_000_000_000L;
43          long nanoseconds = delta % 1_000_000_000L;
44
45          System.out.println("Sensors: " + NUM_SENSORS);
46          System.out.println("Completed: " + completed);
47          System.out.printf("Time: %d.%09ds%n", seconds, nanoseconds);
48      }
49  }
```

Appendix C Python code

C.1 Single-threaded

```
1  import time
2
3  TASK_LENGTH = 10000000
```

```
completed = 0

def sensor_task():
    x = 1.0
    global completed

    for i in range(TASK_LENGTH):
        x = x * 1.5 + 1.0

    completed += 1

NUM_SENSORS = 1326
start = time.time()

for i in range(NUM_SENSORS):
    sensor_task()

end = time.time()

print(f"Sensors: {NUM_SENSORS}")
print(f"Completed: {completed}")
print(f"Time: {(end-start):.9f}s")
```

C.2 Multi-threaded

```
import threading
import time

TASK_LENGTH = 1000
completed = 0
lock = threading.Lock()

def sensor_task():
    x = 1.0
    global completed

    for i in range(TASK_LENGTH):
        x = x * 1.5 + 1.0

    with lock:
        completed += 1

NUM_SENSORS = 1326
threads = []
start = time.time()

for i in range(NUM_SENSORS):
    thread = threading.Thread(target=sensor_task)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

end = time.time()
```

```
31
32 print(f"Sensors: {NUM_SENSORS}")
33 print(f"Completed: {completed}")
34 print(f"Time: {(end-start):.9f}s")
```

Appendix D Raw results

Computation size	Repeat	Single-threaded	Multi-threaded	Multi-threaded + O3 flag
1000	1	0.004308	0.056621	0.057655
	2	0.004211	0.056549	0.057652
	3	0.004317	0.053076	0.066909
	Average	0.004278666667	0.05541533333	0.06073866667
10000	1	0.04201	0.053033	0.05861
	2	0.042495	0.056113	0.055644
	3	0.041964	0.05479	0.062062
	Average	0.04215633333	0.05464533333	0.058772
100000	1	0.432981	0.052005	0.05882
	2	0.419277	0.051585	0.058871
	3	0.418471	0.053292	0.05676
	Average	0.4235763333	0.052294	0.05815033333
1000000	1	4.185251	0.287954	0.056204
	2	4.197485	0.303981	0.05936
	3	4.24343	0.315565	0.057836
	Average	4.208722	0.3025	0.0578
10000000	1	42.309822	2.872339	0.057806
	2	42.032935	3.041378	0.059855
	3	42.283346	3.17465	0.061831
	Average	42.208701	3.029455667	0.05983066667
100000000	1		30.116101	
	2		31.816076	
	3		32.548555	
	Average		31.49357733	

Table 1: C raw results

Computation size	Repeat	Single-threaded	Multi-threaded
1000	1	0.002473	0.132203
	2	0.00224	0.132893
	3	0.00238	0.1377
	Average	0.002364333333	0.1342653333
10000	1	0.017954	0.129038
	2	0.017954	0.130714
	3	0.017495	0.140958
	Average	0.017801	0.13357
100000	1	0.168139	0.138044
	2	0.168378	0.14234
	3	0.166433	0.136914
	Average	0.16765	0.1390993333
1000000	1	1.678332	0.227541
	2	1.668823	0.222082
	3	1.691308	0.243183
	Average	1.679487667	0.2309353333
10000000	1	16.860517	1.339528
	2	16.813035	1.389371
	3	16.761765	1.358048
	Average	16.81177233	1.362315667
100000000	1		12.17278
	2		12.000511
	3		12.324234
	Average		12.16584167

Table 2: Java raw results

Computation size	Repeat	Single-threaded	Multi-threaded
1000	1	0.030572	0.141932
	2	0.030249	0.142748
	3	0.030441	0.14244
	Average	0.03042066667	0.1423733333
10000	1	0.329752	0.416306
	2	0.32444	0.420946
	3	0.338508	0.420777
	Average	0.3309	0.419343
100000	1	3.331551	3.137865
	2	3.24767	3.133793
	3	3.333907	3.163765
	Average	3.304376	3.145141
1000000	1	32.261875	31.343857
	2	32.485481	31.253618
	3	33.186811	31.382612
	Average	32.64472233	31.32669567
10000000	1	336.680613	337.351581
	2	328.834883	332.511223
	3	329.082078	330.538815
	Average	331.5325247	333.4672063

Table 3: Python raw results