Multi-core Synergy: A Study of Performance Improvements Utilizing Multi-core Threading

J. Wolffrom

May 21, 2025

Abstract

Contents

1 Disclosure

Any and all resources related to- and used in this study can be found here:

https://github.com/X1las/AsSynergy

Due to the amount of variables that can affect the performance of a CPU, the results of this study should be taken in comparison to each other, and not as a definitive measure of performance. It is encouraged to run the code snippets on your own machine to get a better understanding of the performance improvements that can be achieved with multi-core threading.

In addition to this, this paper does fall within the scope of the Humanities and Technology department at the University of Roskilde, as it aims to understand how multi-core threading is being used in society today, and how it can be improved. More specifically, it falls within the dimension of Technological Systems and Artifacts (TSA) as we take a look at computer systems at large, and how people are using them- as well as how concepts within them are understood.

2 Introduction

Quoted by Rauber and Rünger 2023, "Moore's law is an empirical observation which states that the number of transistors of a typical processor chip doubles every 18 to 24 months. This observation has first been made by Gordon Moore in 1965 and has been valid for more than 40 years. However, the transistor increase due to Moore's law has slowed down during the last years"

Rauber and Rünger posits that the number of transistors on a microchip used to double rougly every two years, leading to an exponential increase in computing power. However they, in accordance with Mattson 2014, agree that this trend has been dwindling since some time around 2005, as a limit was reached on how many transistors could be placed on a single computer chip without overheating and damaging the circuitry. Even by utilizing the most advanced tricks and optimizations available, the performance improvements we are seeing annually as of 2023 are somewhere around 3.5% faster CPU clock speeds (Rauber and Rünger 2023, p. 11) which is a far cry from the 50% annual increases we were seeing in the early 2000's (Rauber and Rünger 2023, p. 11).

As a result of the processing speed ceiling, the focus has shifted from increasing the clock speeds of CPUs to adding more computational units to them. This has led to a rise in multi-core processors, which contain multiple processing units on a single CPU chipset, often referred to as "cores". These cores can execute multiple orders simultaneiously, which can lead to significant performance improvements in software applications. However, a discrepancy can be found in the communities that rely on these improvements. An article from IGN (Thomas 2025) on picking optimal hardware for computer games describes it best:

> *"While there are some PC games that love CPUs with a dozen or more cores, they're few and far between. Instead, finding an 8-core, 16-thread processor with a high clock speed and a lot of L3 Cache is going to get you further than just adding more CPU cores to the equation".*

A trend of diminishing returns can be seen, or rather felt on a consumer basis. Even people who purchase CPUs know to steer away from something that advertises a high amount of cores, as it's simply not worth it. Whereas the relationship between core-count and performance we should be seeing, in theory, should be an equivalence. It is evident that traditional single-core processors are opting out of the market in place of processors with more and more cores (Smith 2023, January), but the performance improvements are not being felt by the end-user.

2.1 The Problem Area

If multi-core threading is the future, then it is essential to understand why it is not delivering the expected performance improvements it has the potential to deliver. Rauber and Rünger 2023, p. 12 mentions that simulations have shown that superscalar processors with up to four functional units yield substantial benefits over the use of a single functional unit. So in theory, it would seem there is a great potential to gain from utilizing multiple cores. Some of it has been utilized according to Thomas (2025), as we are talking about 8-cores as opposed to one, but it has taken us 20 years to get to this point.

As such, this study aims to address this question of why multi-core optimization is so difficult to accomplish, if and/or how it can be improved, and what the potential benefits of doing so are.

2.1.1 Research Question

This being said, there are limits to what can be achieved in such a short study, therefore we will be focusing on threads and schedulers. The way to implement them, as well as how different programming languages handle them. This will be done by looking at the different threading libraries available in C, C#, Java and Python, as well as the different schedulers that are used to manage them.

The research question that we will be addressing is then as follows:

> *What does multi-core threading performance look like, and how do different schedulers affect it?*

This question can then be further broken down into the following sub-questions:

- What is a thread and how do you use it?
- What is a scheduler and does where does it differentiate between systems and programming languages?
- What performance increases and deficits does multi-core threading incur?
- What are the challenges of multi-core threading, and how can they be overcome?

These questions serve to guide the study as a framework, outlining the key areas we need to understand in order to answer the main research question at large. But with that being said, we will be limiting the scope of this study in order to have a deeper understanding of the subject matter within their short confines.

2.1.2 Limitations

This study is limited in its research and data-collection to only a handful of programming languages: C, C#, Java and Python. This is due to the fact that these languages are the most commonly used in the field of computer science, and therefore provide a lot of insight into the subject matter that is relevant to the readers. Whilst also common, they all handle multi-core threading in wildly different ways, which allows us to trace the remnants of different threading architectures and how they affect the performance of multi-core threading.

This being said, we will not delve into the specifics of each language, but rather focus on the threading libraries and how they are used in practice. This is to ensure that we can provide a comprehensive overview of the subject matter without getting bogged down in the details of each language.

Additionally, this study limits its operating system scope to just Linux, as this was the easiest method of collecting data. This is due to linux having a lot of built-in tools and easy-to-install features that allowed for easy programming in all 4 different languages. This is not to say that the results are not applicable in other operating systems, in fact we encourage the reader to run the code snippets on a windows environment to see if the results are similar- However, we will not be able to provide any data on this, as it was not part of the study.

In addition to this, we will also only be talking about x64 architecture, as this is the most common architecture used in modern computers, and the architecture used in both the Intel and AMD processors tat most Windows and Linux machines run on. For the purpose of this study, we will only be looking at Intel processors, as they are the components readily available to us for testing. This is not to say that the results are not applicable to AMD processors, but we will not be able to provide any data on this, as it was not part of the study.

3 Asynchronous Systems

To understand multi-core systems effectively, we must first differentiate between synchronous and asynchronous execution paradigms. These concepts form the foundation for how tasks are distributed and managed across multiple processing units.

In traditional single-core processing, operations follow a synchronous model where instructions execute sequentially in a predetermined order (Johnson and Dinyo 2015, p. 118). This means that each instruction must complete before the next one begins, creating a linear flow of execution. This model is straightforward and easy to understand, but even on a single-core processor there is still a need the need for running multiple processes at the same time. In addition to this, it's also inefficient to wait for one process to finish entirely before starting the next one, leading to the processor "idling" whilst waiting for I/O operations or resources to become available.

> *"It is much easier to reason sequentially, doing only one thing at a time, than to understand situations where many things occur simultaneously." ... "Thus, while we are "parallel processors", and we live in a world where multiple things happen at the same time, we usually reason by reduction to a sequential world."*
> - Rajsbaum and Raynal 2020

This creates a need for switching the processors attention, or "context switching" between different tasks, which is a key feature of modern operating systems. This allows the processor to switch between different tasks quickly, giving the illusion of everything running at the same time, also known as "concurrency"(Rajsbaum and Raynal 2020). This is a common approach in single-core systems, where the operating system manages the execution of multiple processes by rapidly switching between them. This is often done using a time-slicing technique, where each process is given a small time slice to execute before the processor switches to the next one.

3.1 Subroutines and Coroutines

This is often seen by the programmer in the form of subroutines, which in layman's terms are just functions (Pyeatt and Ughetta 2020), a piece of code that can be called from anywhere in the program. This is a common approach in programming that most, if not all programmers are familiar with, where functions are used to encapsulate a specific piece of functionality and can be called from anywhere in the program, and often multiple times. This allows for code reuse and modularity, making it easier to write and maintain complex programs, but by expanding them to coroutines we can achieve a lot more, namely Concurrency and Asynchronousity.

Coroutines carry an additional layer of complexity, as they allow for the suspension and resumption of execution at specific points (Moura and Ierusalimschy 2009). This means that a coroutine can "yield" control back to the calling code, before finishing its execution, allowing for more complex interactions between different parts of the program. This is often used in asynchronous programming, where coroutines can be used to handle I/O operations without blocking the main process.

Depending on how Coroutines are used, they can be either synchronous or asynchronous. In a synchronous coroutine, the coroutine will yield control back to the calling code, but will not return until it has finished executing. This means that the calling code will wait for the coroutine to finish before continuing. In an asynchronous coroutine, the coroutine will yield control back to the calling code, and will return immediately, allowing the calling code to continue executing.

This suspension and resumption capability is often expressed through 'yield' and 'resume' operations, enabling coroutines to implement cooperative multitasking patterns where execution flow can be transferred between different tasks without requiring them to complete first.

For an example, look at the following python snippet:

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

Which defines an asynchronous main function, or a coroutine that will print "Hello World" interspaced by one second. This shows the prominent 'yield' operation. If we gave the function a target to yield to, we would in turn make the code "synchronous" in nature.

When expanding to multiple processing units on the same chipset (a multi-core processor), we can maintain this synchronous approach for consistency or adopt an asynchronous model that allows processing units to operate with greater independence.

In practice, synchronous and asynchronous models are rarely implemented in their purest forms. Even highly parallel systems require synchronization mechanisms to coordinate access to shared resources. For instance, memory access operations must be coordinated when multiple cores need to read from or write to the same memory locations simultaneously. Even when we believe to have written a purely synchronous program, the underlying hardware and operating system may still introduce asynchronous behavior through context switching and resource management because of the need for concurrency.

4 Threading

A thread is often described as a lightweight process, as well as the smallest unit of execution within a process (Rauber and Rünger 2023). A process often being associated with or connected to having distributed memory(Rauber and Rünger 2023, p. 4, 27) as opposed to threads, which are often associated with shared memory.

The reason why subroutines and coroutines relate to this topic, is because a thread is in practice a coroutine, and as will be demonstrated later, usually is passed by a function. Threads can be suspended mid-execution and yield to other processes akin to coroutines, such that the processor it runs on always has something to do. Threads are often used to implement parallelism in a program(Rauber and Rünger 2023, p. 27), allowing for multiple tasks to be executed simultaneously. This is an achievment that can be achieved in single-core systems as well, by continuous context switching between different threads(Rauber and Rünger 2023) we achieve the illusion of multitasking, also known as "Concurrency". Concurrency is the ability of a system to execute multiple tasks simultaneously, even if they are not actually running at the same time, and it extends to multi-core systems as well, which we will be adressing here as well as how to contribute to it by using various threading libraries.

4.1 Kernel- and User Threads

Threads are divided into two main categories: "User-threads" and "Kernel-threads" (Rauber and Rünger 2023, p. 27). User threads being the threads closest to the user or programmer, and kernel threads being the threads that are created and managed by the operating system itself.

> *"The kernel threads are mapped by the operating system to processors or cores for execution. User threads are managed by the specific programming environment used and are mapped to kernel threads for execution. The mapping algorithms as well as the exact number of processors or cores can be hidden from the user by the operating system."* - (Rauber and Rünger 2023, p. 27)

These user-threads are managed by the afforementioned libraries particular to the pro-

gramming languages used, and are mapped to kernel threads for execution. This means that the operating system is responsible for managing the execution of kernel threads, while user threads are managed by the particular programming environment used. This allows for a high degree of flexibility and control over the execution of threads, but it also means that the performance of user threads can be affected by the performance of the kernel threads they are mapped to, as well as the mapping algorithms used by their own programming environment. This is important to note, as it adds additional layers of complexity to measuring the performance of threads, as the affected performance of user threads can vary depending on the operating system and programming environment used.

4.2 Threading in different languages

In order to create threads in a program, we often use a threading library that is provided by the programming language. Usually, this is done by creating a function or subroutine that pertains the code to be run independently, using the threading library to pass the function to its own thread. This will become clearer in the following sections, where we will be looking language specific methods of threading in the afforementioned programming languages, starting with the POSIX library, as it relates to C and Python.

4.2.1 Posix

POSIX, or "Portable Operating System Interface"(*POSIX (The GNU C Library)* n.d.; *POSIX Documentation* 2025, January), is a set of operating system standards that include features such as a shell, file system, and threading. It is a standard for multi-threading and comes pre-installed with the GCC compiler(*POSIX (The GNU C Library)* n.d.) used for this study. POSIX carries a certain level of overhead, as it is not exclusively a threading library, nor is it a library made for C specifically. It is a standard that is implemented in C, but languages such as Python utilize it as well when using the 'multiprocessing' library. When used in this context of multi-threading, it is referred to as 'pthreads', and will be referred to as such in this paper.

Pthreads are essentially user-threads that are created and managed by the POSIX library, but the library itself also allows for the creation of kernel threads. In the context of the programming languages we will be looking at, posix is the only library that offers the ability to create kernel threads directly. For further reading on the POSIX standard, for further reading see Stevens, W. R., & Rago, S. A. (2005). *Advanced programming in the unix environment.* Addison-Wesley on programming in UNIX environments, as well as Harbour, M. G. (2003). Real-time posix: An overview. *University of Cantabria* an older article on posix functionality that still holds relevance.

4.2.2 Threading in C

As previously mentioned, in order to accomplish multi-core threading in C we make use of the POSIX library, which comes pre-installed in the GCC compiler used to compile C-code on unix systems. This library provides access to pthreads, which we can easily invoke by including the header file 'pthread.h' in our code:

```
#include <pthread.h>
```

After this, we can create a thread by first making a pthread variable to pass on to the pthread_create function, which takes a function pointer as its first argument. This function pointer is the function that will be executed in the new thread, and it must have a specific signature in order to be used with pthreads. The signature is as follows:

```
void *function_name(void *arg) {
    // Code to be executed in the new thread
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, function_name, NULL);

    return 0;
}
```

Posix provides the ability to work with multi-core systems by default, unless the used specifies something different by declaring "AFFINITY":

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>

void *function_name(void *arg) {
    // Code to be executed in the new thread
    return NULL;
}

int main() {
    pthread_t thread;
    cpu_set_t cpuset;

    // Set the CPU affinity for the thread
    CPU_ZERO(&cpuset);
    CPU_SET(0, &cpuset); // Set the thread to run on CPU 0

    pthread_create(&thread, NULL, function_name, NULL);
    pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);

    pthread_join(thread, NULL); // Wait for the thread to finish

    return 0;
}
```

What this code does is create a thread that will run on CPU 0, and then wait for the thread to finish before exiting the program. This is a simple example of how to use pthreads to create a thread in C, but it is important to note that this is not the only way to do it. There are many other functions and options available in the pthreads library that can be used to create and manage threads, such as pthread_join, pthread_detach, and pthread_cancel. These functions allow for more advanced thread management, such

as waiting for a thread to finish, detaching a thread from the main process, or canceling a thread that is no longer needed. For further reading on the pthreads library, see the official documentation: Pthread.h - threads. (n.d.). https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

4.2.3 Threading in C-Sharp

C# just like C, is a compiled language that comes from the C language family like source Python (*General Python FAQ — Python 3.13.3 documentation* n.d.), however where it differs is its striking resemblance to Java and its use of object oriented programming. To get started with threading in C#, the process is different from environment to environment. Using the .NET framework for Unix, threads come pre-installed and ready for use by simple instantiating a thread object and passing it a function to run.

```
class thread_example {

    static void MyThreadFunction() {
        // Code to be executed in the new thread
    }

    static void Main(string[] args) {
        Thread thread = new Thread(new ThreadStart(MyThreadFunction));
        thread.Start();
    }
}
```

Not too dissimilar from how it was carried out in C, we create a function wherein we write the code to be executed in the new thread, and then we create a thread object that takes a function pointer as its argument. This is a simple example of how to use threads in C#, it includes further functionality that we won't get into depths about in this paper, but further reading can be found in the official documentation: Thread class - system.threading. (n.d.). https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-9.0 as well as this article on its implementation: Wagner, B., Poojari, P., Warren, G., & Lee, D. (2022). Using threads and threading - .net — microsoft learn. *Microsoft*. https://learn.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading In addition, it is possible to set the CPU affinity of a thread in C# just like in C, but this is not as straightforward, see this documentation for more information: (*Processor Affinity Property - ProcessThread.ProcessorAffinity - System.Diagnostics* n.d.).

4.2.4 Threading in Python

Threading in python is carried out in a handful of ways, but the most common way is to use the 'threading' library(*Threading — Thread-based parallelism — Python 3.13.3 documentation* n.d.), which comes pre-installed with Python.

```
import threading

def my_thread_function():
    # Code to be executed in the new thread
```

```
    pass

thread = threading.Thread(target=my_thread_function)
thread.start()
```

This is again a simple example of how to use threads, but it is important to note that due to the Global Interpreter Lock (GIL) in Python, native threads can only run on a single core(*Global Interpreter Lock - Python Wiki* n.d.). This doesn't mean there are no ways of achieving multi-core threading in Python, as there are libraries such as 'multiprocessing' that allow for the creation of multiple processes that can run on different cores:

```
import multiprocessing

def my_process_function():
    # Code to be executed in the new process
    pass

thread = multiprocessing.Process(target=my_process_function)
thread.start()
```

Despite being labelled as processes, these are still threads as they utilize run on POSIX under the hood(*Multiprocessing — Process-based parallelism — Python 3.13.3 documentation* n.d.), and are therefore still subject to the same limitations as pthreads. But even so, multi-core threading can be achieved in Python with relative ease, however it's difficult to notice the performance improvements as the GIL is still present(*Global Interpreter Lock - Python Wiki* n.d.). This means that even though we are creating multiple processes, they are still limited by the GIL, and therefore cannot take full advantage of multi-core systems. This is an important consideration when using Python for multi-core threading, as it can lead to performance issues if not handled correctly. For further reading on the threading library, see the official multiprocessing documentation: Multiprocessing — process-based parallelism — python 3.13.3 documentation. (n.d.). https://docs.python.org/3/library/multiprocessing.html

4.2.5 Threading in Java

Java Standard Edition (JSE) comes with a built-in threading library(*Class Thread - Java Platform SE 8* n.d.), usually this functionality is accomplished by inheritance, extending a class from 'Thread' and including a 'run' method that contains the code to be executed in the new thread:

```
class MyThread extends Thread {
    public void run() {
        // Code to be executed in the new thread
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
```

```
    }
}
```

This is slightly different from other languages, as it uses inheritance to create a new thread class that extends the Thread class. This means that the thread objects created from this class will have all its methods and properties included, meaning we don't have to pass a function pointed to the thread object before running it, instead we just call the 'start' method on the thread object. In terms of multi-core threading, Java Virtual Machine (JVM) is able to utilize multiple cores in a program, as they are allotted a pool of Kernel Threads to operate with. For more information on the Java threading library, see the official documentation: Class thread - java platform se 8. (n.d.). https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

5 Parallelism

This ability to execute multiple tasks simultaneously with multiple cores is known as "parallelism". In this section we will be looking at notable terms and concepts that are important to understand when working with multi-core systems.

5.1 Degree of Parallelism

In the field of parallel systems architecture we often talk about the degree of parallelism a system is capable of. This is a measure of how many tasks can be executed at the same time, and is often used to describe the performance improvements that a parallel system provides.

This, in turn, coincides with the concept of "Potential Parallelism", which is the theoretical maximum degree of parallelism that a system can achieve. This is often used to describe the performance improvements that a parallel system can provide, and is often used as a benchmark for comparing different systems.

5.2 Thread Level Parallelism (TLP)

5.3 Schedulers

Schedulers are the part of the operating system that is responsible for managing the execution of threads. They are responsible for deciding which thread to execute at any given time, and for managing the resources that are used by each thread. This includes managing the CPU, memory, and other resources that are used by the threads. The scheduler is responsible for ensuring that each thread gets a fair share of the resources, and that no single thread is allowed to monopolize the resources.

Schedulers are often implemented as part of the operating system kernel, and are responsible for managing the execution of all threads in the system. This includes managing the scheduling of threads, as well as managing the resources that are used by each thread. The scheduler is responsible for ensuring that each thread gets a fair share of the resources, and that no single thread is allowed to monopolize the resources.

5.3.1 OS Schedulers

5.3.2 .NET

The .NET framework is a software development platform developed by Microsoft. It provides a large library of pre-built functions and classes that can be used to create applications for Windows and other platforms. The .NET framework includes a built-in threading library, which allows for the use of multiple threads in a program, managed by the .NET runtime. This means that the .NET framework is able to utilize multiple cores in a program, but it is not as easy to implement as it is in other languages.

5.3.3 JVM

The Java Virtual Machine (JVM) is a software-based execution environment that allows Java programs to run on any platform that has a JVM installed. The JVM is responsible for managing the execution of Java programs, including the creation and management of threads. This means that the JVM is able to utilize multiple cores in a program, but it is not as easy to implement as it is in other languages.

5.4 Parallel Execution Time (PET)

In this context of degrees of parallelism, we often use measures such as "Parallel Execution Time" (PET) to quantitatively describe the time it takes for a parallel system to execute a given task.

This execution time is measured across all functional units, or cores of a system, and can be used to give a tangible measure of the running time of a parallel program. This means that waiting times for memory fetching, writing and plain idle times are included in the measure.

In terms of coding with multi-core threading, this means that "scheduling" time is included in the measure as well, as this is the time it takes for the operating system to assign tasks to different cores. This is important to note, as it means that the performance improvements of a parallel system are not just determined by the number of cores, but also by the efficiency of the scheduling algorithm used by the operating system.

5.5 Load Balancing

The term "Load Balancing" refers to the standard process of distributing tasks across multiple cores in a parallel system, such that tasks are executed equally across all cores. This is important to ensure that all cores are utilized effectively, and that no single core is overloaded with tasks while others are idle.

To this, it is important to note that not all cores are created equal, and that in terms of cpu-architecture we see a lot of variation with "Performance" and "Optional" cores. Performance cores are designed to handle heavy workloads, while optional cores are designed to handle lighter workloads. This means that load balancing is not just about distributing tasks evenly across all cores, but also about ensuring that the right tasks are assigned to the right cores.

Even within the same core type we can see a lot of variation in terms of performance, as some cores are designed to handle specific tasks better than others. This is known

as "heterogeneous computing", and is an important consideration when designing parallel systems, but as mentioned earlier, in this paper we will only deal with intel x64 architecture.

5.6 Idle Time

The time a processor cannot do anything useful but wait for more work.

6 Memory

6.1 Shared Memory

Memory organization where the machine shares memory for all threads.

Synchronization plays a heavy role, for example by keeping threads from reading files before another has written to them.

Often connected to the term "Thread".

6.2 Local Caching

6.3 Memory Access Time (MAT)

Add content if applicable.

7 Further Reading

efficiency cores vs performance cores

8 Data Collection

8.1 Experiment Setup

Before we get into the benchmarks of the different programming languages, it is important to disclose that the data was collected on a computer running Ubuntu 24.04.2 LTS with an intel Core i5-4460 CPU @ 3.2GHz, with 4 cores and 4 threads. The time measured was done so with various clocking methods depending on the programming language over the course of 5 runs, with the average time being taken as the final result. The code snippets used for the benchmarks can be found in the appendix.

To begin with, we clocked the time it took to execute a simple count from 1 to 1 billion in the various languages to get a frame of references. This proved especially difficult for Java, as the JIL converts commonly used methods to bytecode to optimize the execution time. This meant that to get a proper benchmark we had to disable the JIT compiler by using the '-Xint' flag, which in turn made the execution time of Java significantly slower than the other languages, with the exception of Python.

8.2 Results

As expected, the results of running the code snippets in multiple cores were significant performance increases, as comared to the single core. This makes sense, as evidently four cores are faster than one, however some of these results were quite surprising.

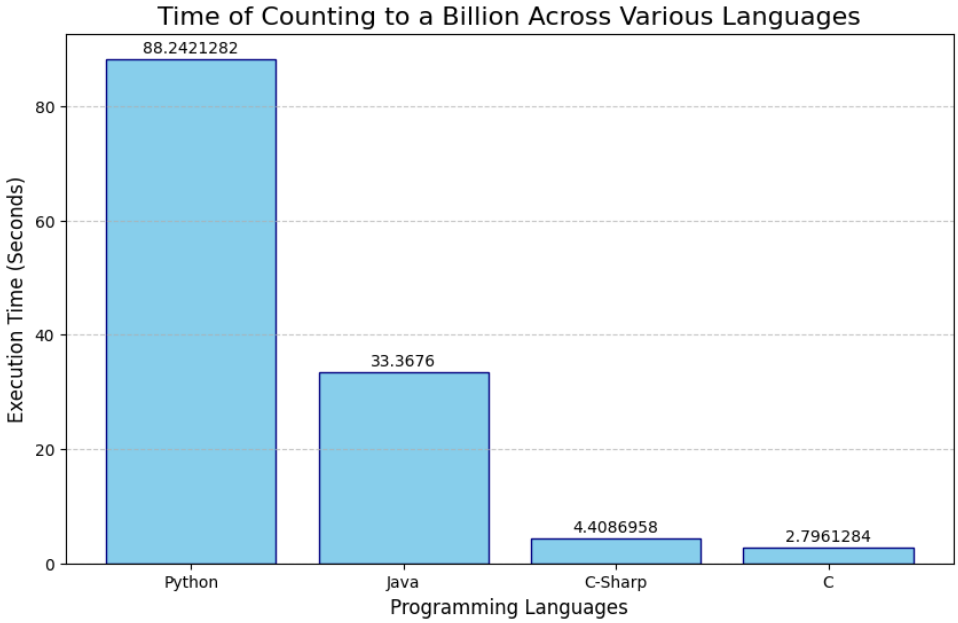

Figure 1: Execution time comparison of different programming languages running synchronously when counting from 1 to 1-billion.

The synchronous execution results above benchmark the time it took to compute the same task in each of the language. We saw expected results within most of the languages, with python having an average run time of 88 seconds, Java at 33, C# 4 and C at a little less than 3 seconds. There was not much to note in these results, other than Java operating surprisingly slow when not using the JIT compiler.

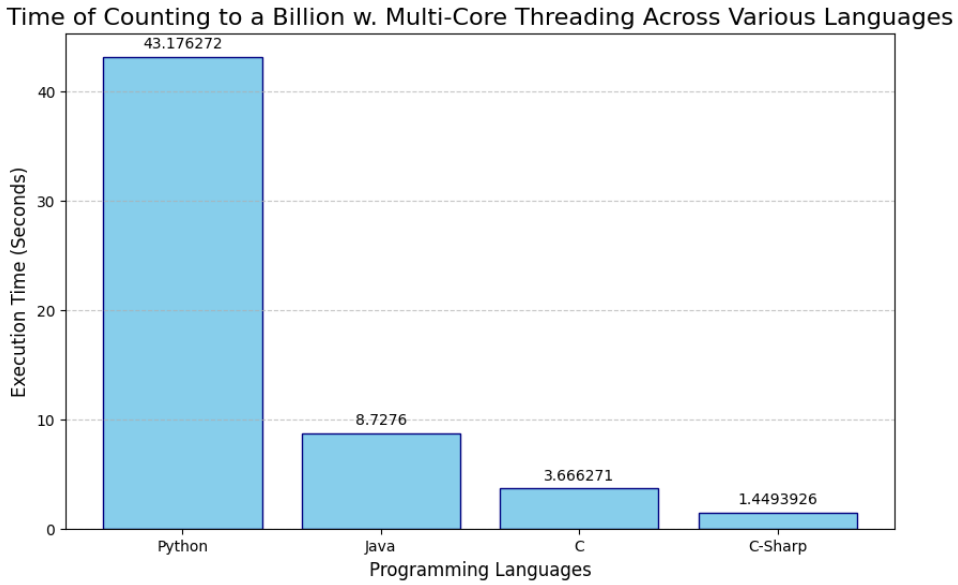

Figure 2: Execution time comparison of different programming languages running asynchronously with multi-threading when counting from 1 to 1-billion.

The asynchronous execution results reveal a great deal about the efficiency of the different

programming when using multiple cores. Python was able to achieve a time of 43 seconds on average which amounts to a little more than twice the performance increase. Java was able to achieve an astounding 8.7 seconds, a little less than 4 times the performance increase.

9 Discussion

Include thoughts about other programming languages.

10 Conclusion

Add content if applicable.

11 References

References

Class thread - java platform se 8. (n.d.). https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

General python faq — python 3.13.3 documentation. (n.d.). https://docs.python.org/3/faq/general.html

Global interpreter lock - python wiki. (n.d.). https://wiki.python.org/moin/GlobalInterpreterLock

Harbour, M. G. (2003). Real-time posix: An overview. *University of Cantabria*.

Johnson, O., & Dinyo, O. (2015). Comparative analysis of single-core and multi-core systems. *International Journal of Computer Science and Information Technology*, *7*, 117–130. https://doi.org/10.5121/ijcsit.2015.7610

Mattson, P. (2014). *Why haven't cpu clock speeds increased in the last few years? — comsol blog*. https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years

Moura, A. L. D., & Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, *31*. https://doi.org/10.1145/1462166.1462167

Multiprocessing — process-based parallelism — python 3.13.3 documentation. (n.d.). https://docs.python.org/3/library/multiprocessing.html

Posix (the gnu c library). (n.d.). https://www.gnu.org/software/libc/manual/html_node/POSIX.html

Posix documentation. (2025, January). https://pubs.opengroup.org/onlinepubs/9799919799/

Processor affinity property - processthread.processoraffinity - system.diagnostics. (n.d.). https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processthread.processoraffinity?view=net-9.0

Pthread.h - threads. (n.d.). https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

Pyeatt, L. D., & Ughetta, W. (2020). Structured programming. *ARM 64-Bit Assembly Language*, 113–153. https://doi.org/10.1016/B978-0-12-819221-4.00012-2

Rajsbaum, S., & Raynal, M. (2020). Years of mastering concurrent computing through sequential thinking. *ACM SIGACT News*, *2020*, 59–88. https://doi.org/10.1145/3406678.3406690

Rauber, T., & Rünger, G. (2023). *Parallel programming for multicore and cluster systems* (Third). Springer.

Smith, E. (2023, January). Updated amd epyc and intel xeon core counts over time. https://www.servethehome.com/updated-amd-epyc-and-intel-xeon-core-counts-over-time

Stevens, W. R., & Rago, S. A. (2005). *Advanced programming in the unix environment*. Addison-Wesley.

Thomas, J. (2025). *Best cpu 2025: Pick the right processor for your gaming pc.* https://www.ign.com/articles/the-best-cpus-for-gaming

Thread class - system.threading. (n.d.). https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-9.0

Threading — thread-based parallelism — python 3.13.3 documentation. (n.d.). https://docs.python.org/3/library/threading.html

Wagner, B., Poojari, P., Warren, G., & Lee, D. (2022). Using threads and threading - .net — microsoft learn. *Microsoft.* https://learn.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading

A Asynchronous Run-time Snippets

A.1 C Code Execution Times

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 4: Counted to 50000000
Thread 1: Counted to 50000000
Thread 2: Counted to 50000000
Thread 3: Counted to 50000000
Thread 4: Counted to 100000000
Thread 1: Counted to 100000000
Thread 2: Counted to 100000000
Thread 3: Counted to 100000000
Thread 4: Counted to 150000000
Thread 1: Counted to 150000000
Thread 2: Counted to 150000000
Thread 3: Counted to 150000000
Thread 1: Counted to 200000000
Thread 4: Counted to 200000000
Thread 2: Counted to 200000000
Thread 3: Counted to 200000000
Thread 1: Counted to 250000000
Thread 1: Time taken to count from 1 to 250000000: 3.444871 seconds
Thread 4: Counted to 250000000
Thread 4: Time taken to count from 1 to 250000000: 3.451413 seconds
Thread 2: Counted to 250000000
Thread 2: Time taken to count from 1 to 250000000: 3.552061 seconds
Thread 3: Counted to 250000000
Thread 3: Time taken to count from 1 to 250000000: 3.690577 seconds
All threads have finished counting.
Total time: 3.692103 seconds
```

Figure 3: First run of asynchronous C code execution.

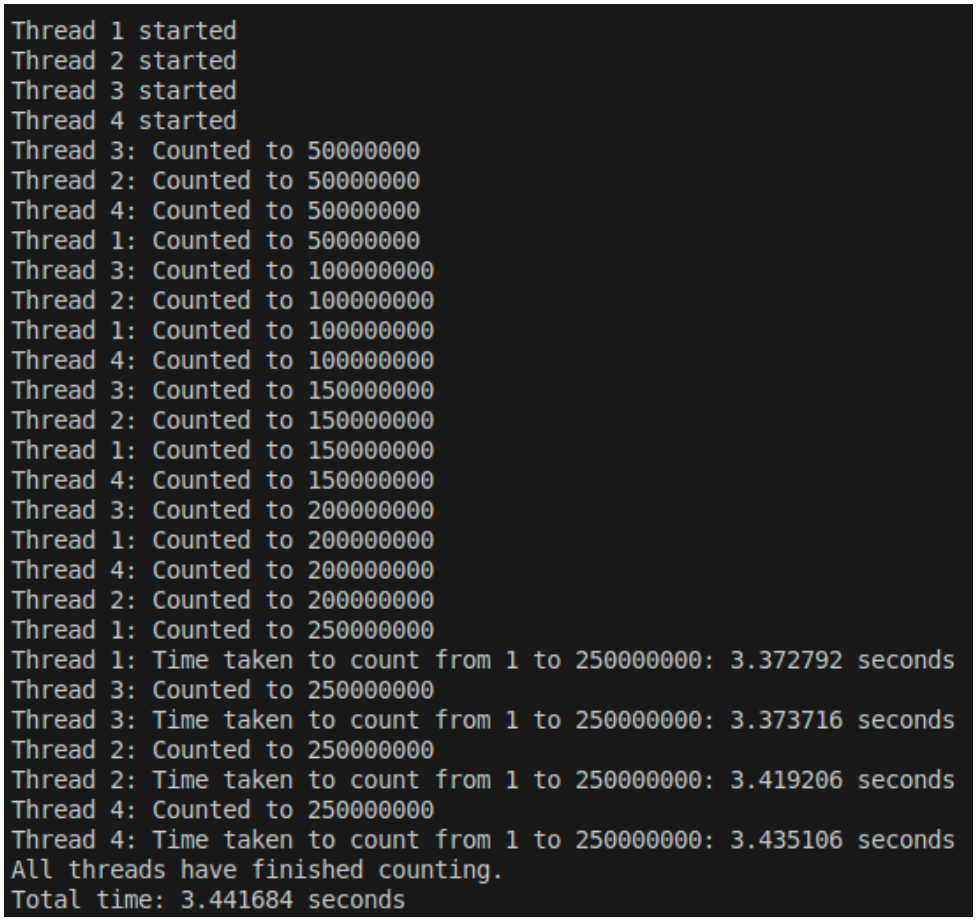

Figure 4: Second run of asynchronous C code execution.

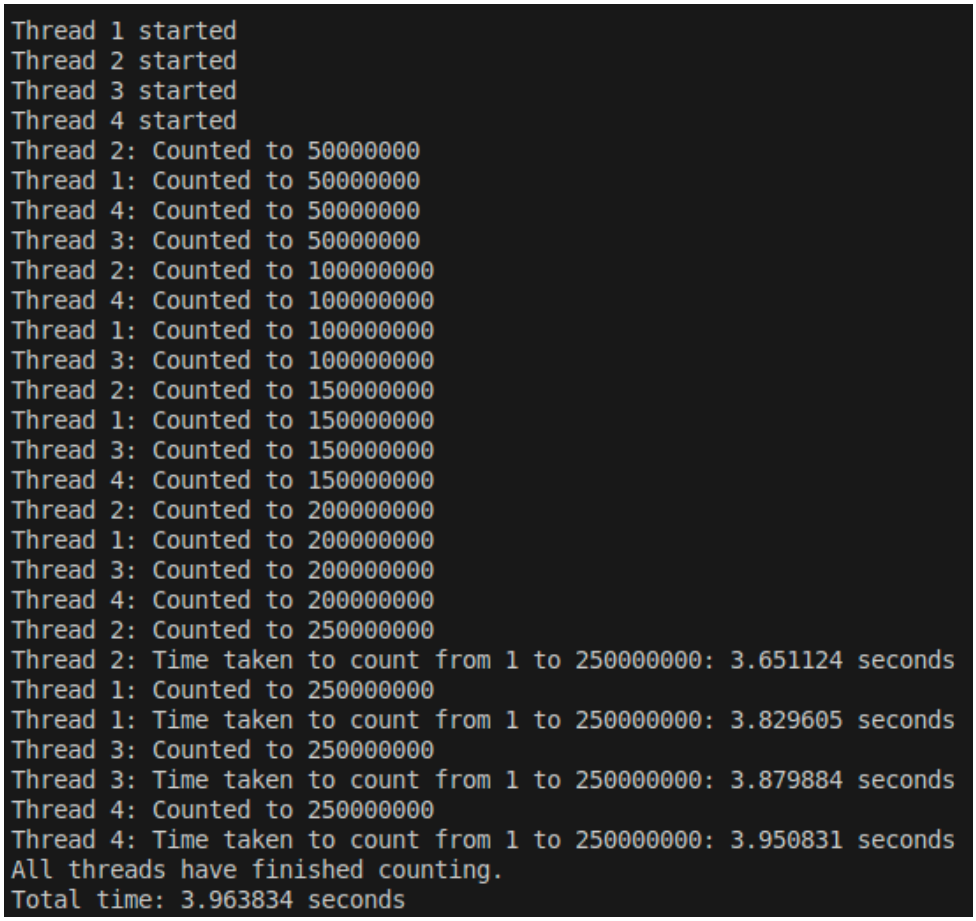

Figure 5: Third run of asynchronous C code execution.

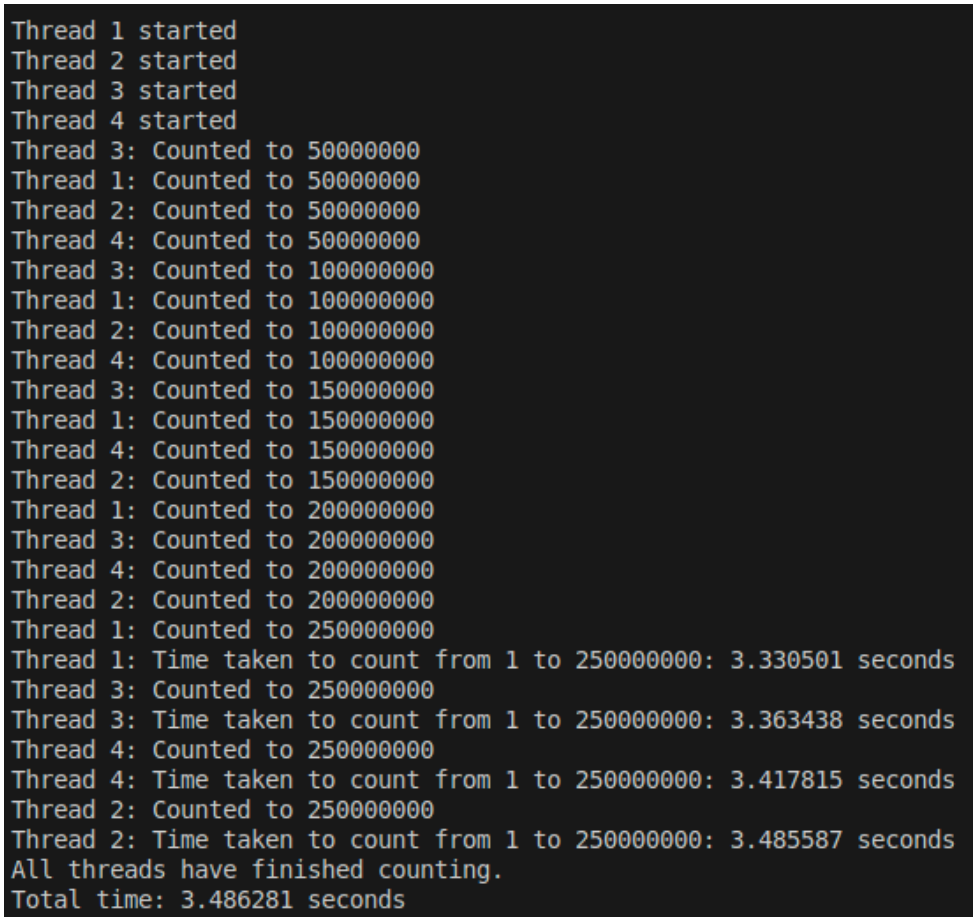

Figure 6: Fourth run of asynchronous C code execution.

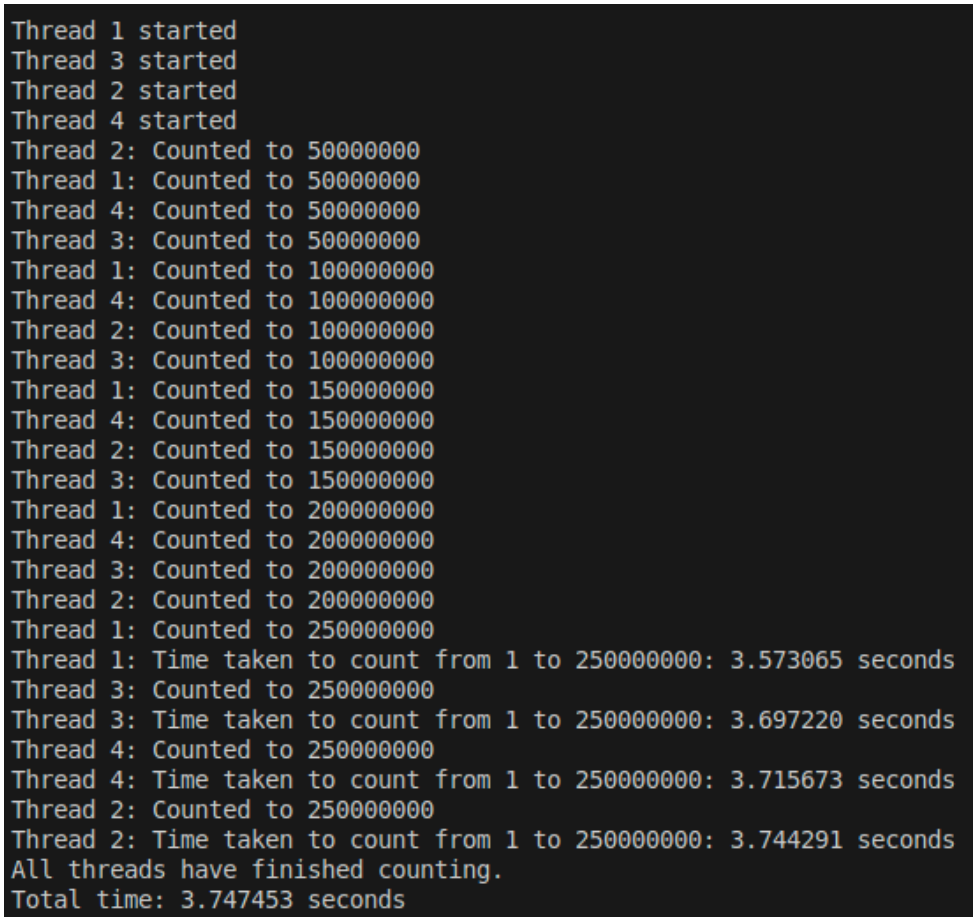

Figure 7: Fifth run of asynchronous C code execution.

A.2 C-Sharp Code Execution Times

```
Beginning Count
4 Count reached: 0
4 Count reached: 25000000
3 Count reached: 50000000
Count reached: 75000000
Count reached: 50000000
2 Count reached: 75000000
Count reached: 100000000
Count reached: 75000000
2 Count reached: 100000000
Count reached: 125000000
Count reached: 100000000
Count reached: 125000000
Count reached: 150000000
2 Count reached: 125000000
Count reached: 150000000
Count reached: 175000000
2 Count reached: 150000000
Count reached: 175000000
Count reached: 200000000
2 Count reached: 175000000
Count reached: 200000000
Count reached: 225000000
2 Count reached: 200000000
2 Count reached: 225000000
Count reached: 250000000
Counting complete!
```

```
Beginning Count
4 Count reached: 0
4 Count reached: 25000000
4 Count reached: 50000000
4 Count reached: 75000000
4 Count reached: 100000000
3 Count reached: 125000000
Count reached: 150000000
Count reached: 125000000
2 Count reached: 150000000
Count reached: 175000000
Count reached: 150000000
2 Count reached: 175000000
2 Count reached: 200000000
Count reached: 175000000
Count reached: 200000000
Count reached: 225000000
Count reached: 200000000
2 Count reached: 225000000
Count reached: 250000000
Counting complete!
Count reached: 225000000
Time elapsed: 00:00:01.4276984
Count reached: 250000000
Counting complete!
Time elapsed: 00:00:01.4566220
Count reached: 250000000
```

```
Beginning Count
4 Count reached: 0
4 Count reached: 25000000
3 Count reached: 50000000
3 Count reached: 75000000
Count reached: 50000000
3 Count reached: 100000000
Count reached: 75000000
3 Count reached: 125000000
Count reached: 100000000
3 Count reached: 150000000
Count reached: 125000000
3 Count reached: 175000000
Count reached: 150000000
3 Count reached: 200000000
Count reached: 175000000
3 Count reached: 225000000
Count reached: 200000000
Count reached: 250000000
Counting complete!
Time elapsed: 00:00:01.3299419
Count reached: 250000000
Counting complete!
Time elapsed: 00:00:01.3475107
Count reached: 225000000
Count reached: 250000000
Counting complete!
```

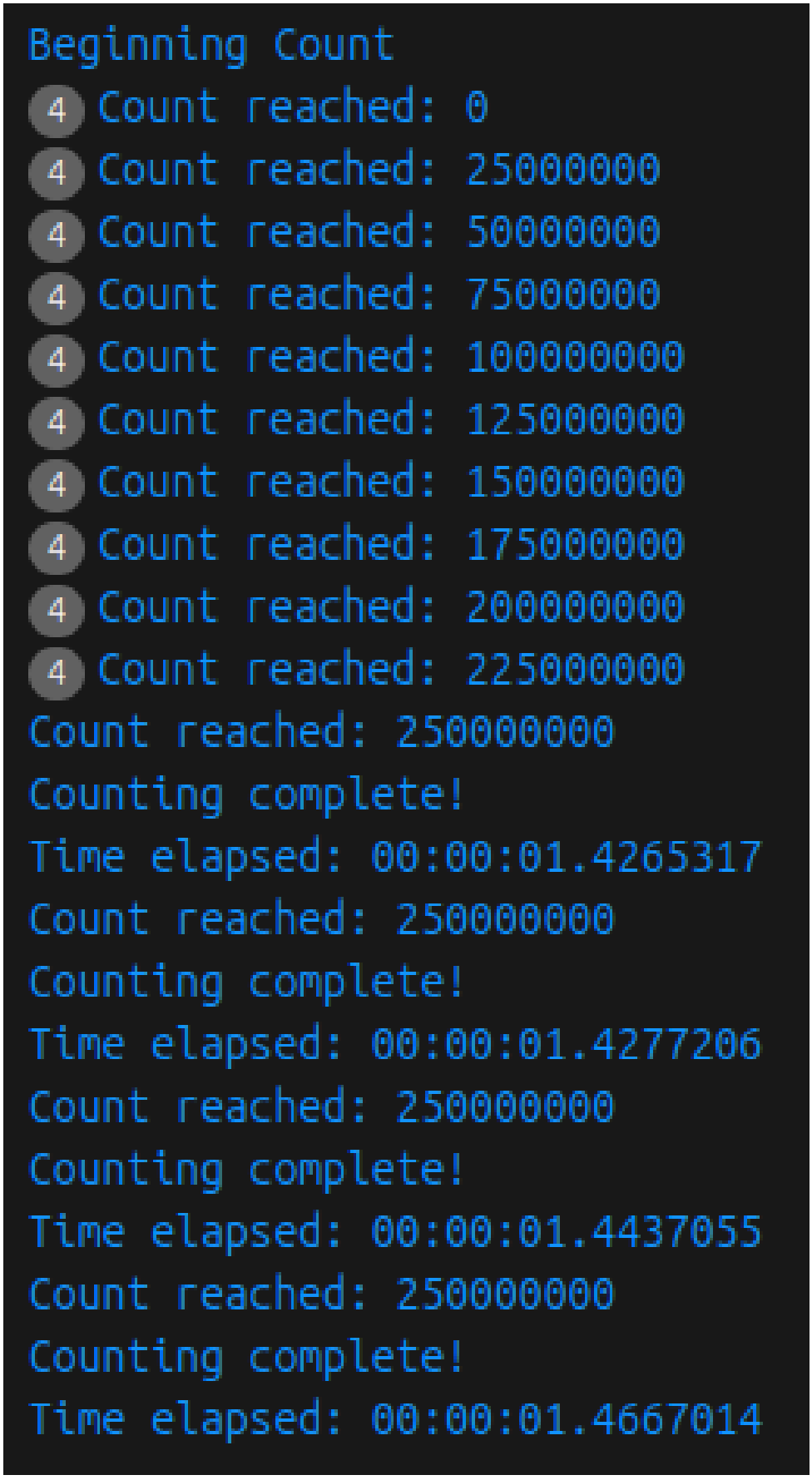

Figure 11: Fourth run of asynchronous C-Sharp code execution.

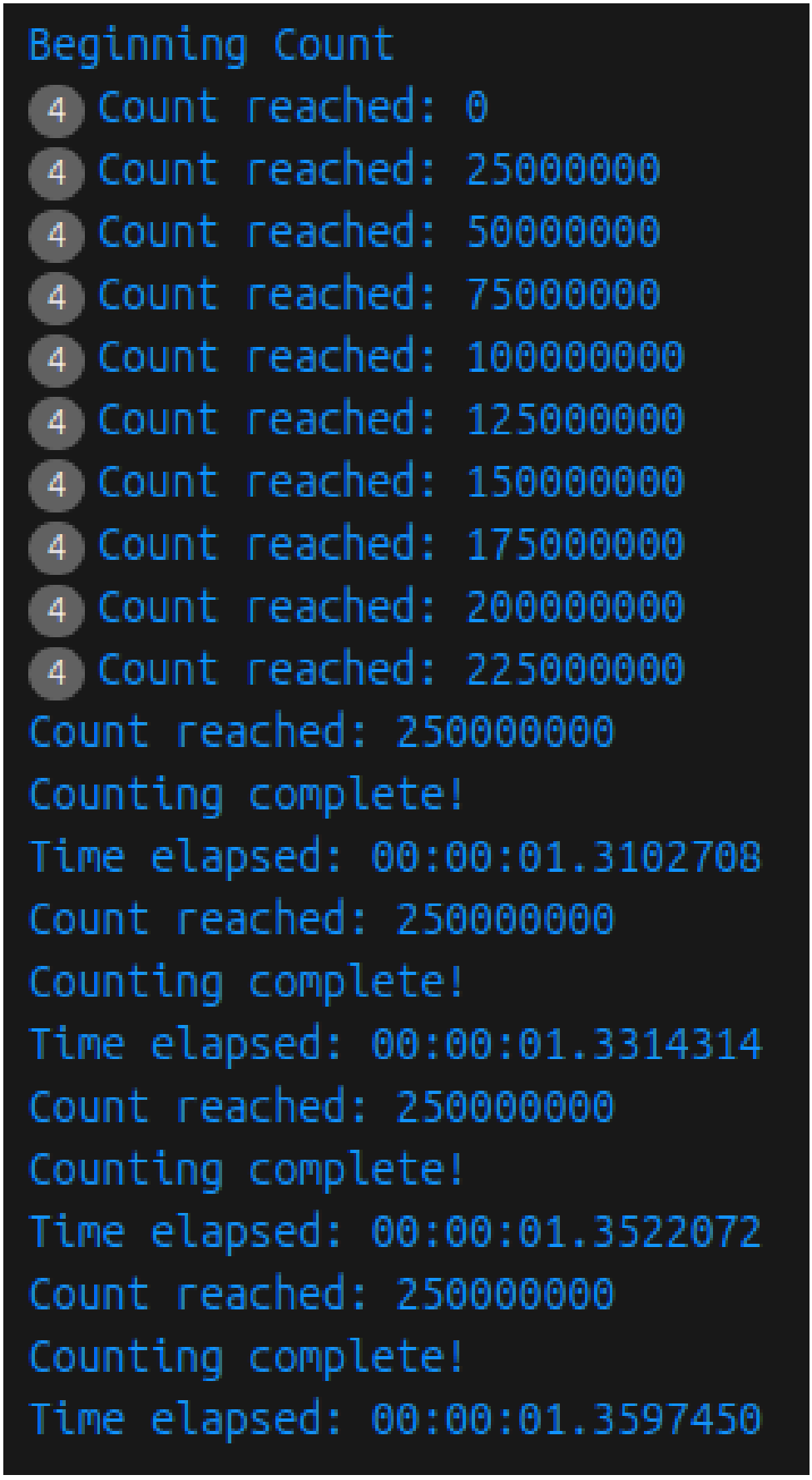

Figure 12: Fifth run of asynchronous C-Sharp code execution.

A.3 Java Execution Times

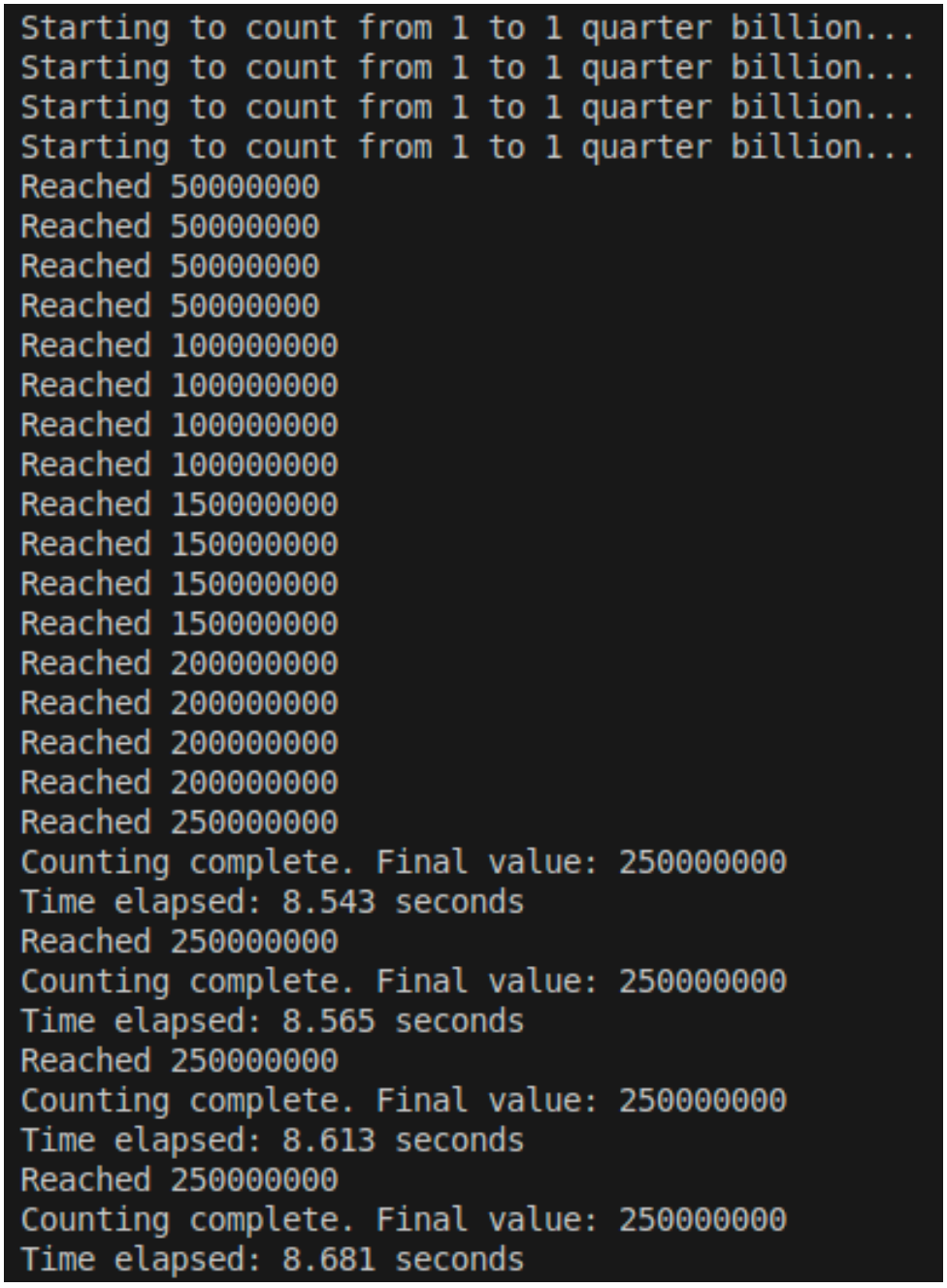

Figure 13: First run of asynchronous Java code execution.

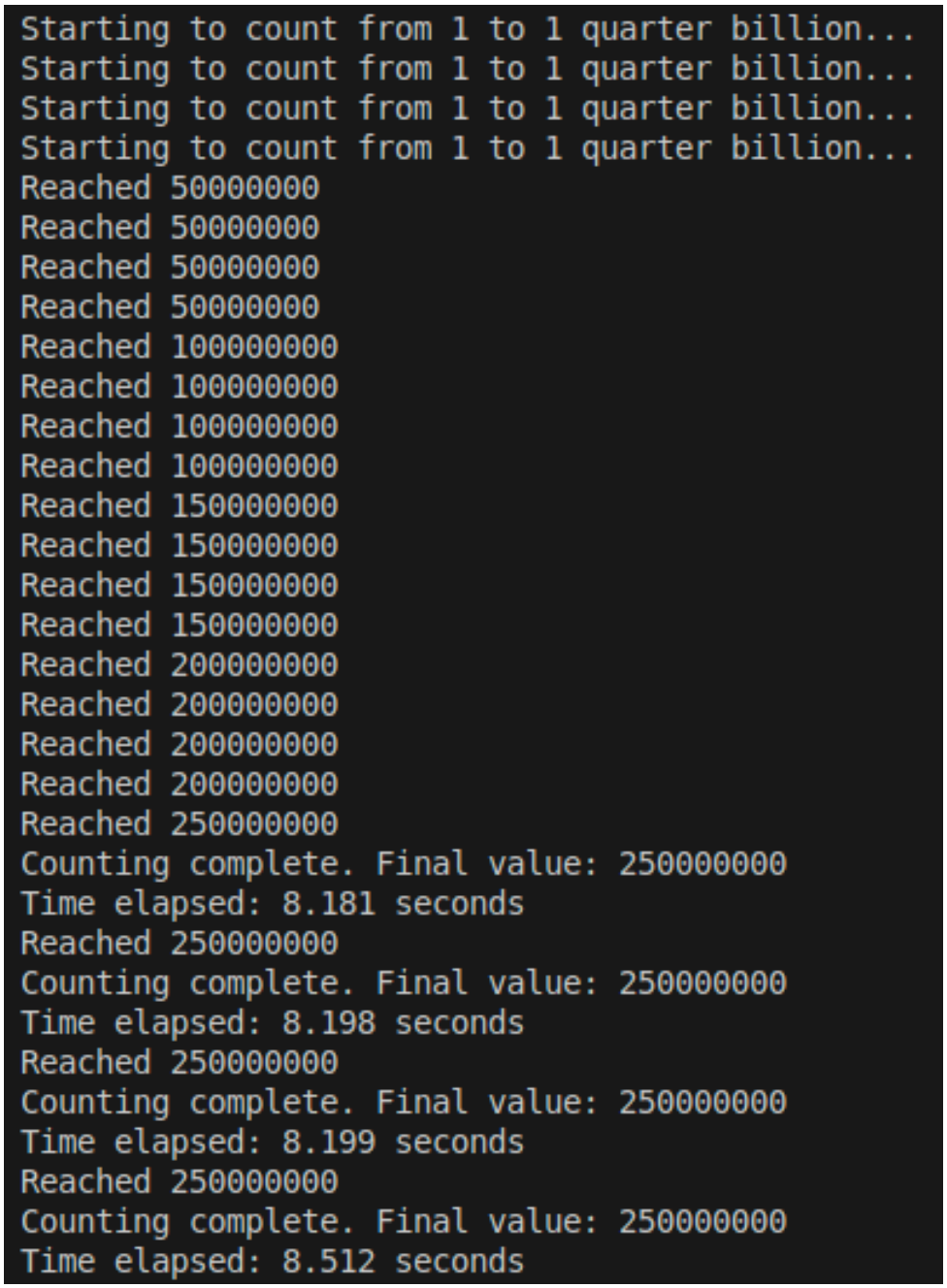

Figure 14: Second run of asynchronous Java code execution.

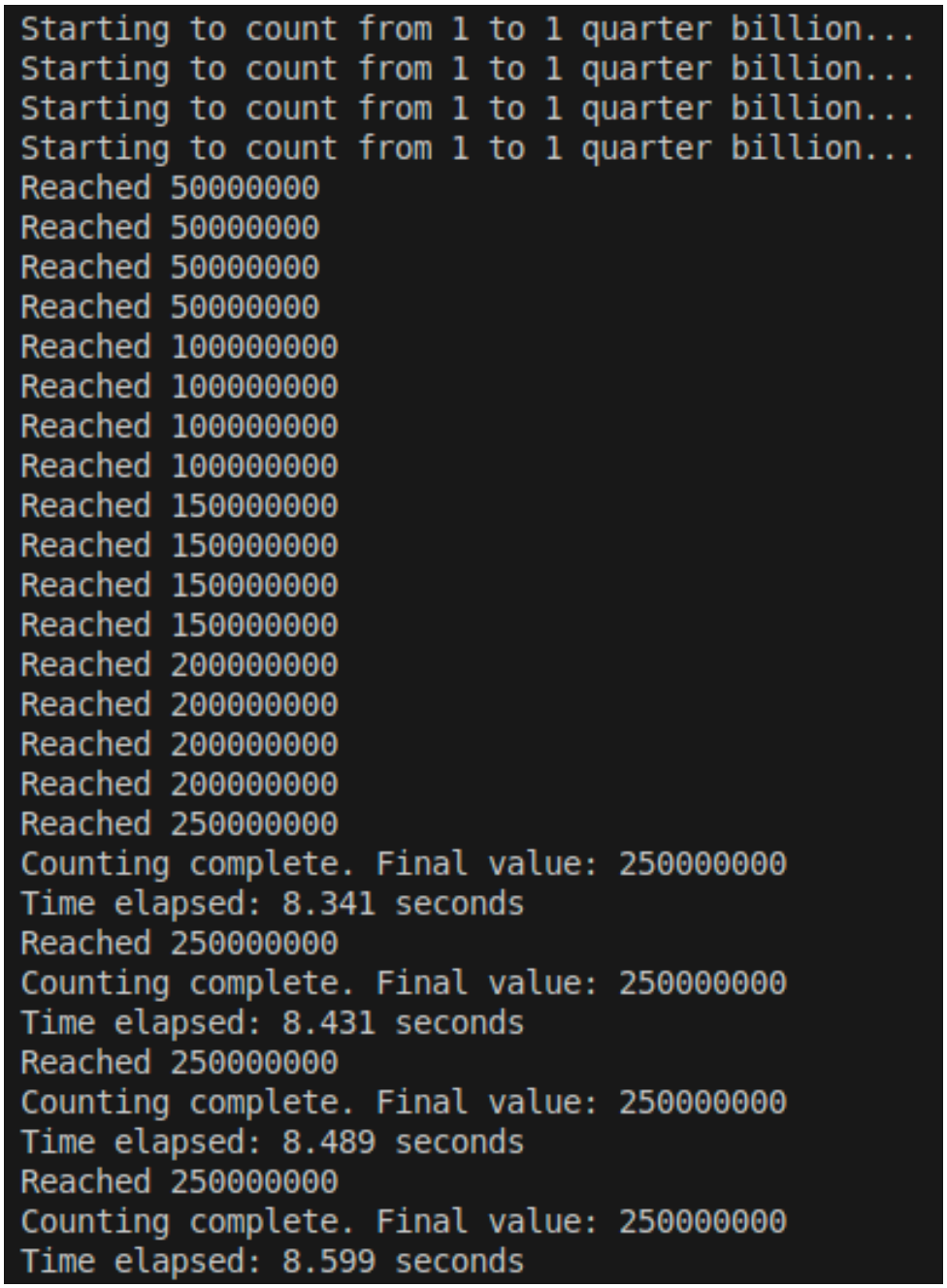

Figure 15: Third run of asynchronous Java code execution.

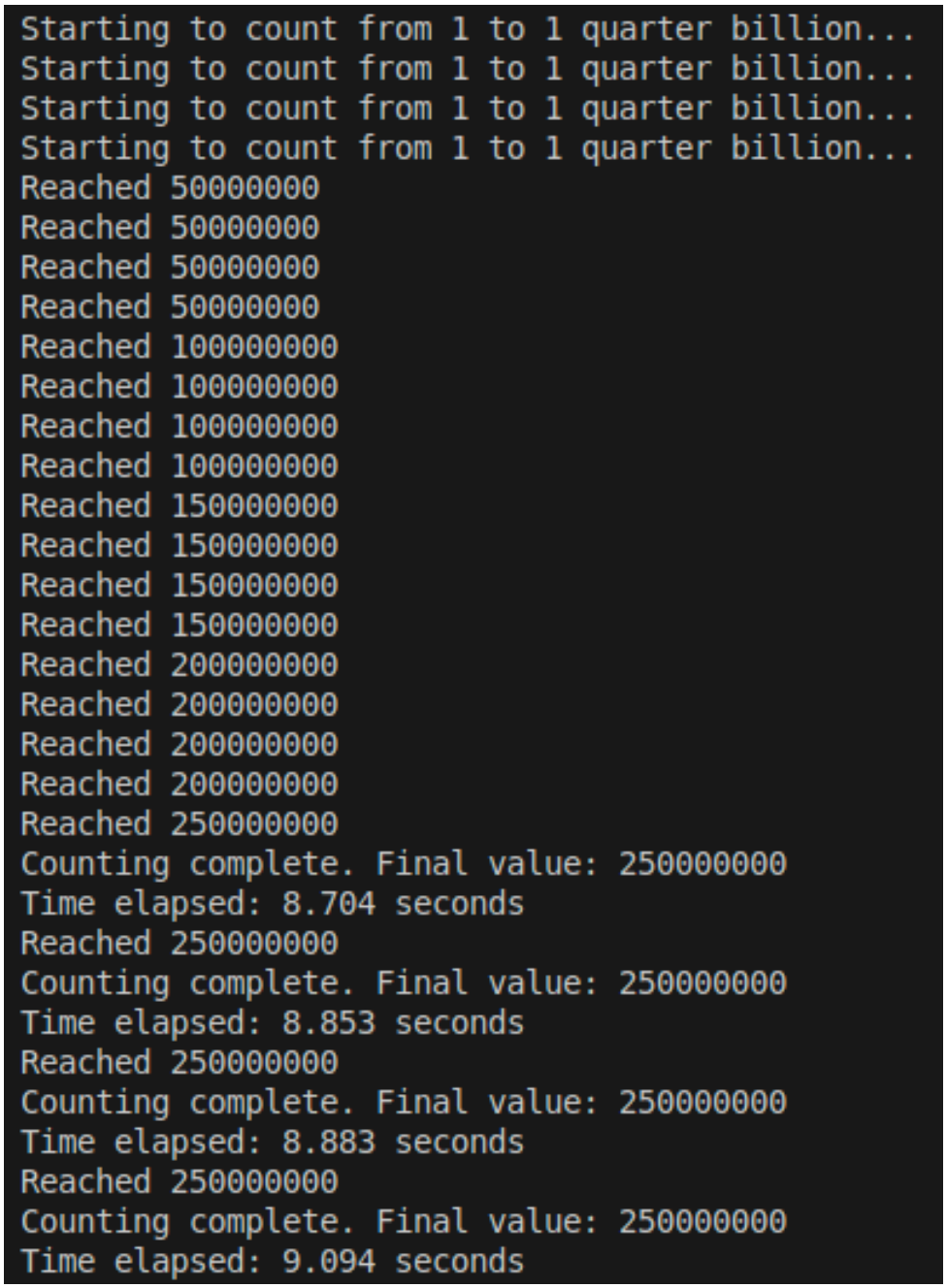

Figure 16: Fourth run of asynchronous Java code execution.

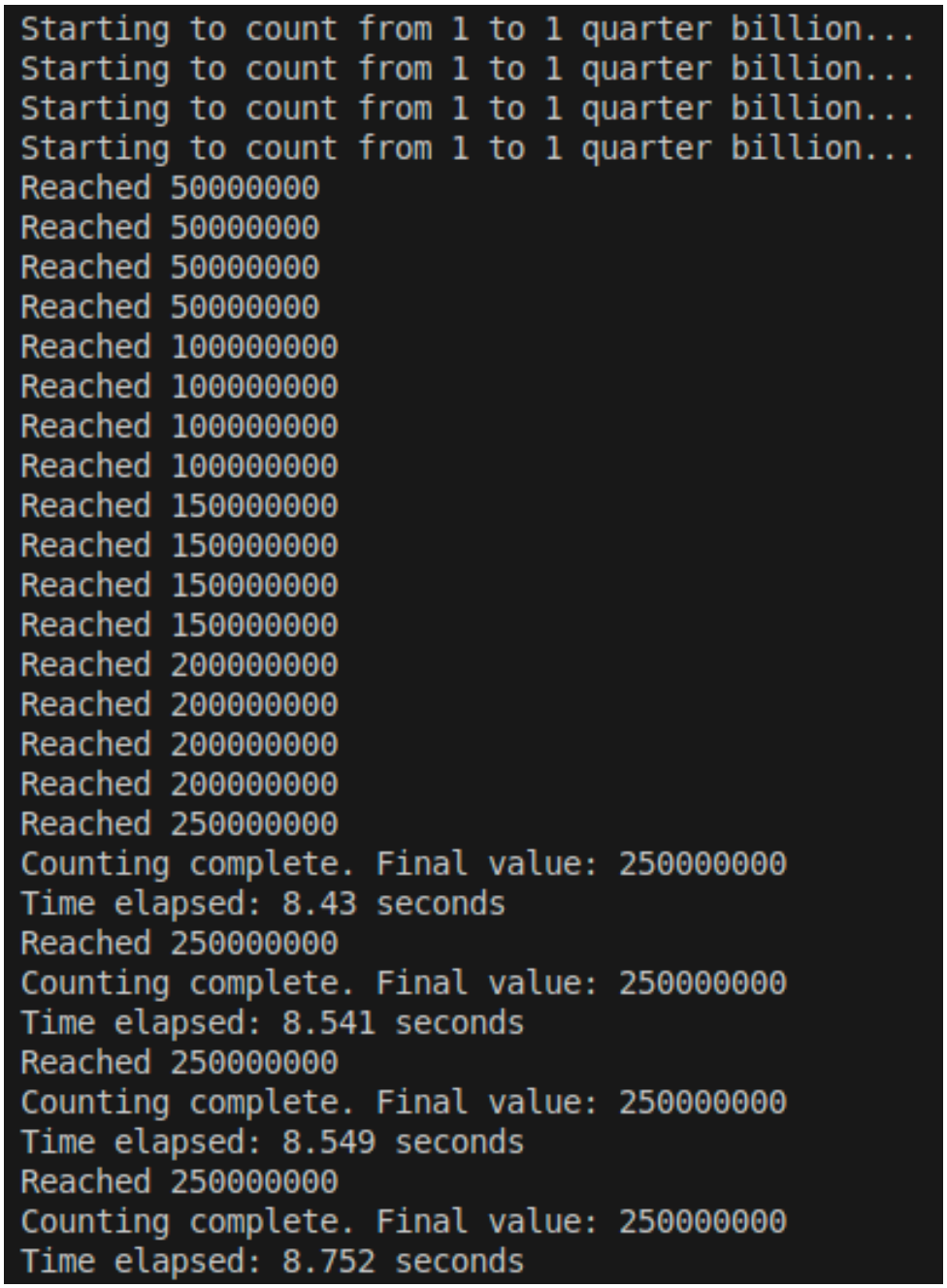

Figure 17: Fifth run of asynchronous Java code execution.

A.4 Python Execution Times

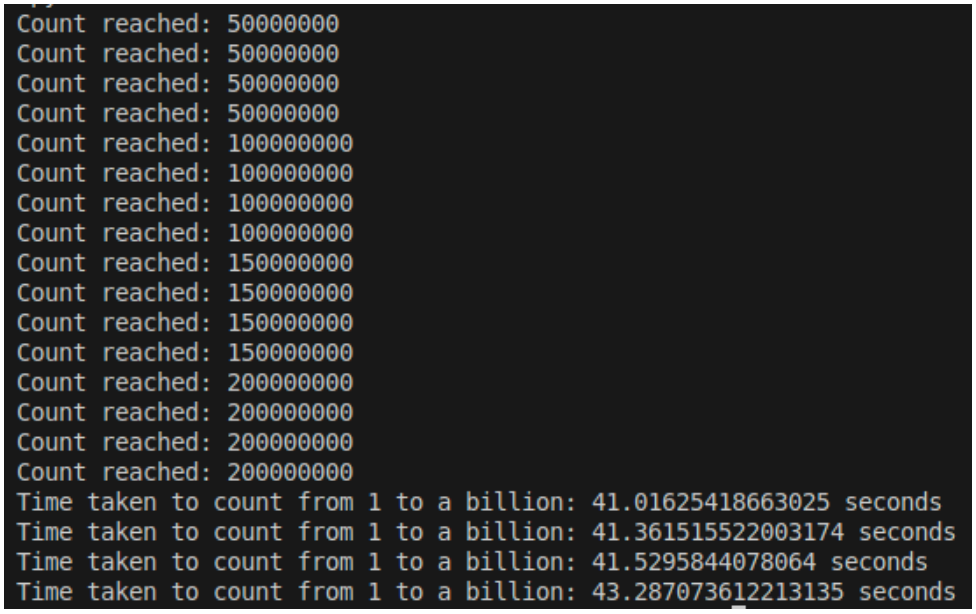

Figure 18: First run of asynchronous Python code execution.

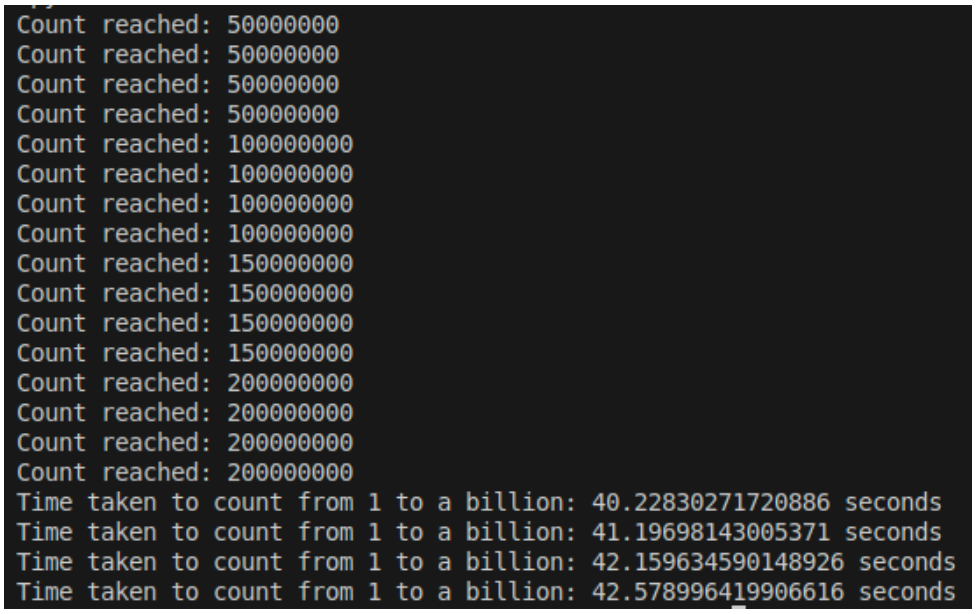

Figure 19: Second run of asynchronous Python code execution.

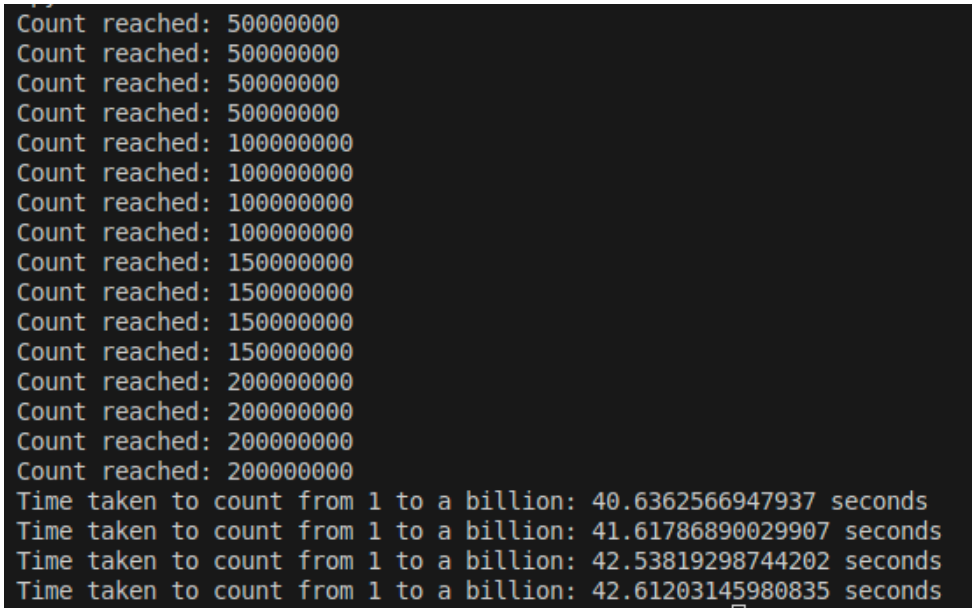

Figure 20: Third run of asynchronous Python code execution.

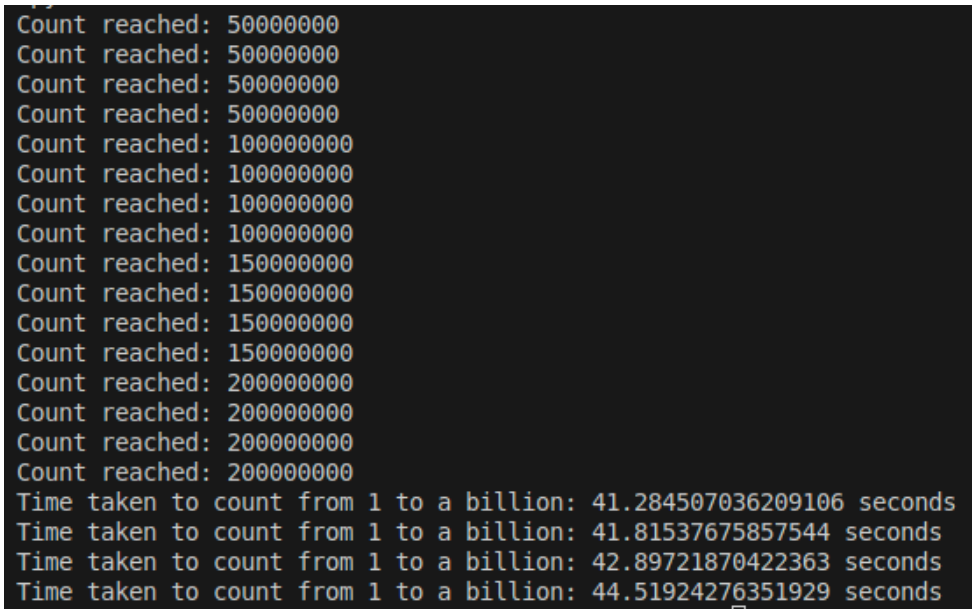

Figure 21: Fourth run of asynchronous Python code execution.

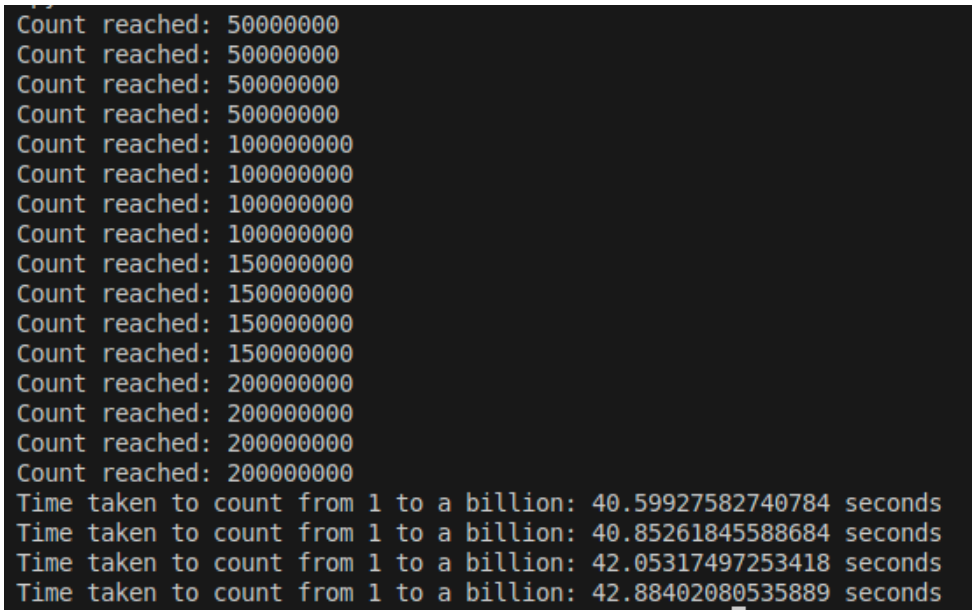

Figure 22: Fifth run of asynchronous Python code execution.

B Synchronous Run-time Snippets

B.1 C Code Execution Times

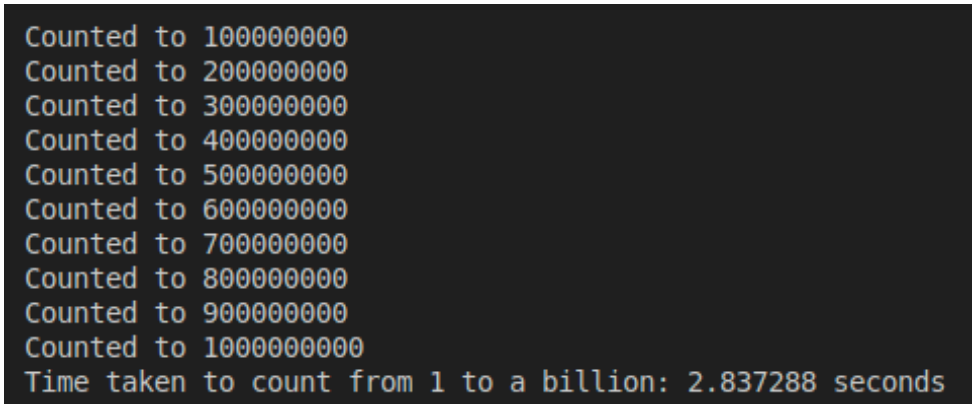

Figure 23: First run of synchronous C code execution.

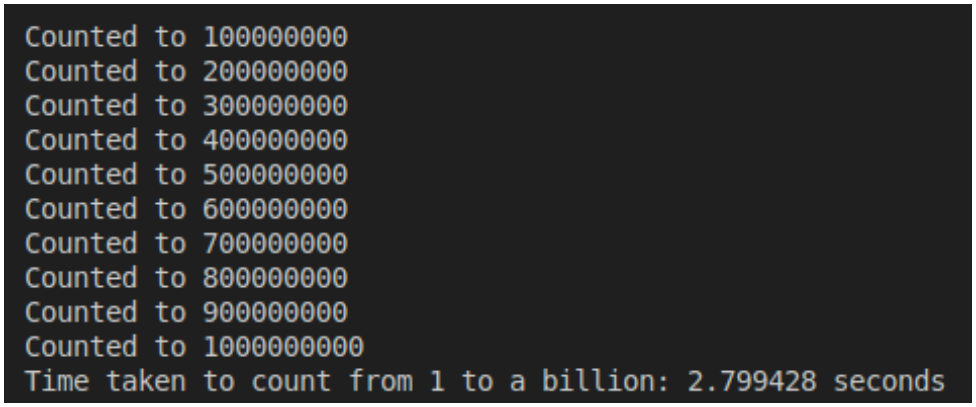

Figure 24: Second run of synchronous C code execution.

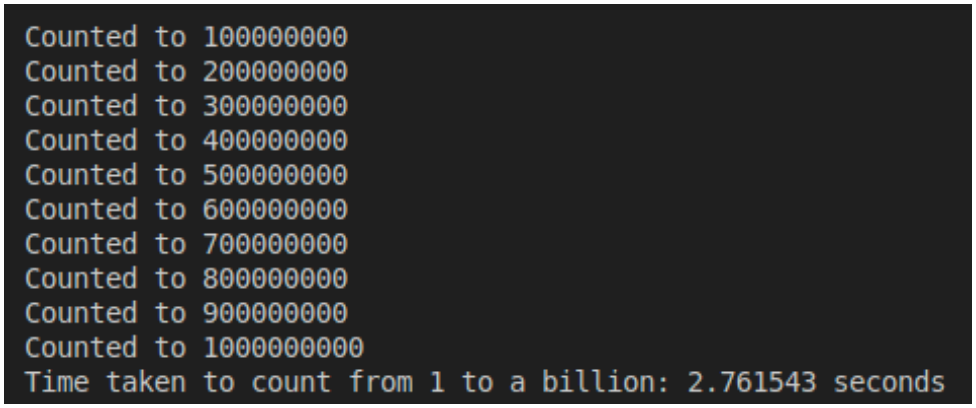

Figure 25: Third run of synchronous C code execution.

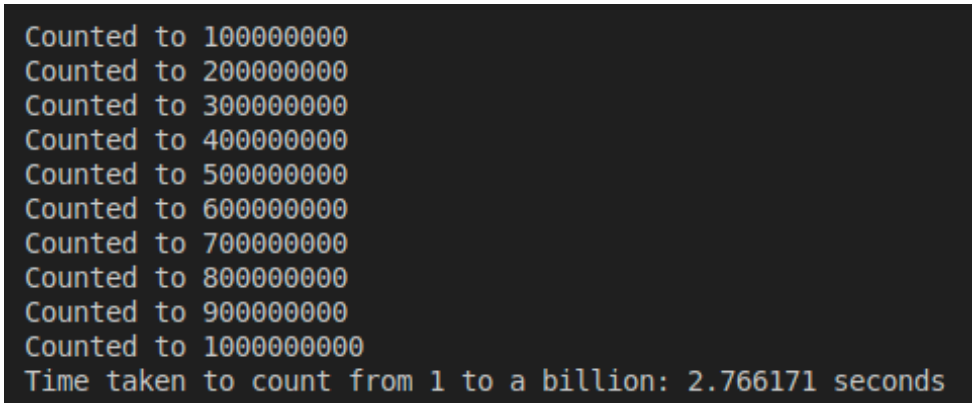

Figure 26: Fourth run of synchronous C code execution.

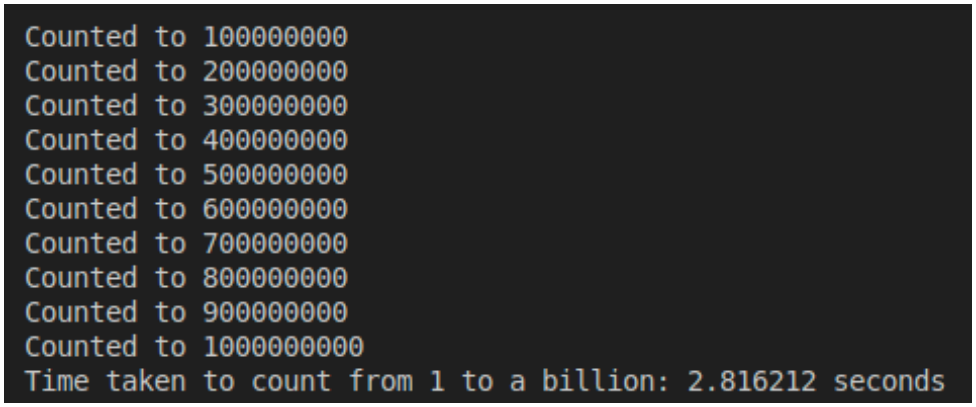

Figure 27: Fifth run of synchronous C code execution.

B.2 C-Sharp Code Execution Times

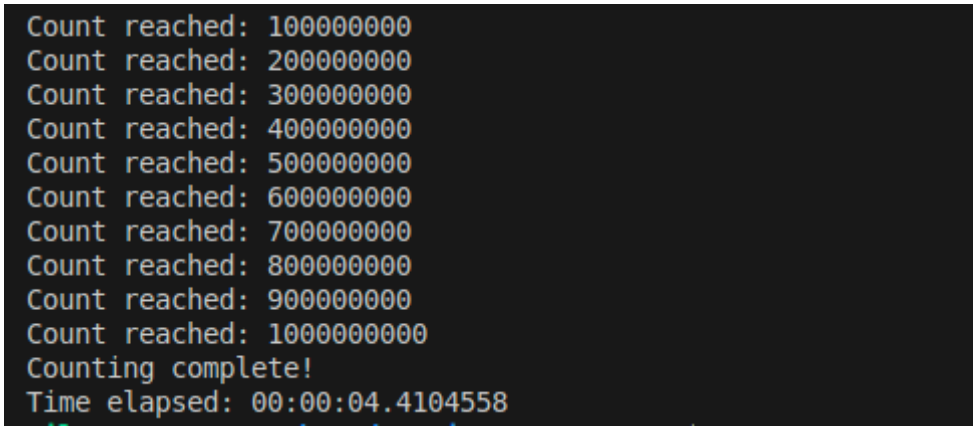

Figure 28: First run of synchronous C-Sharp code execution.

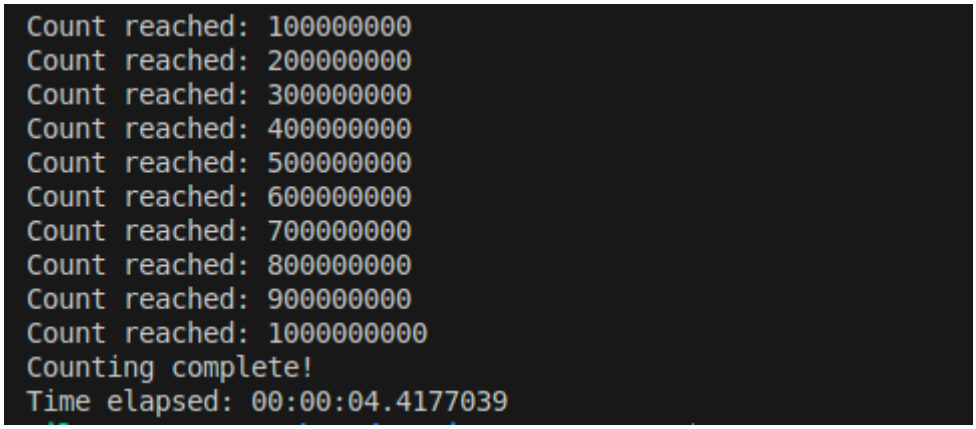

Figure 29: Second run of synchronous C-Sharp code execution.

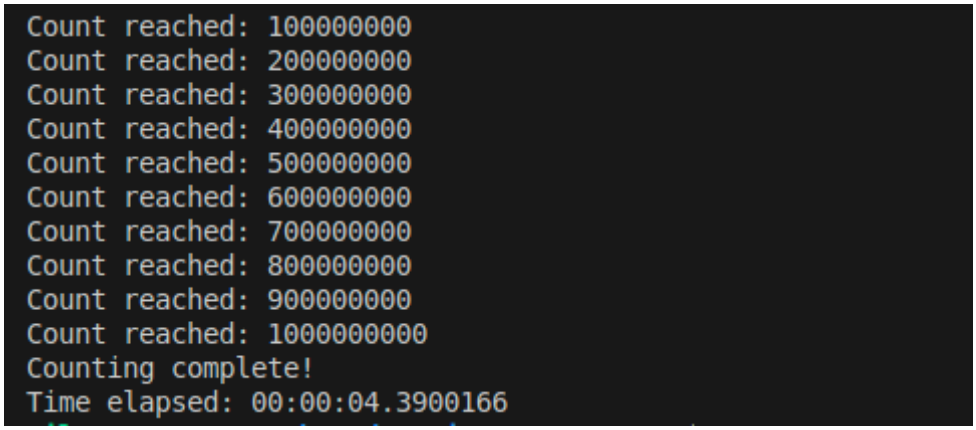

Figure 30: Third run of synchronous C-Sharp code execution.

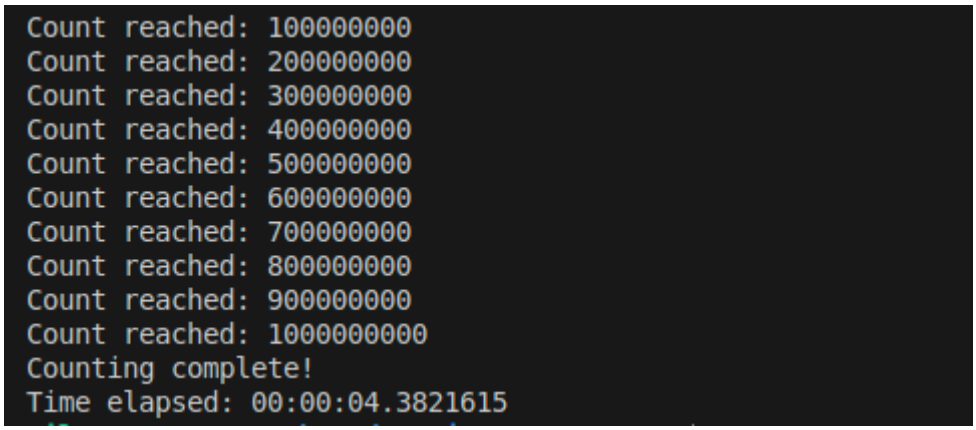

Figure 31: Fourth run of synchronous C-Sharp code execution.

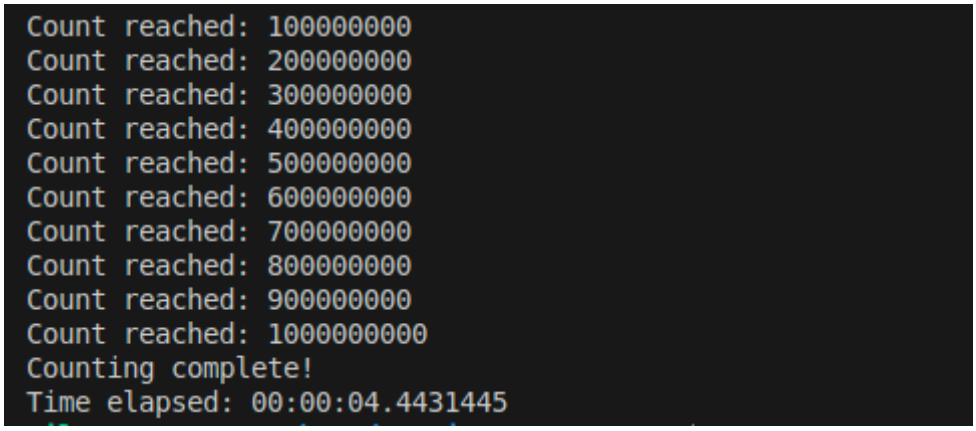

Figure 32: Fifth run of synchronous C-Sharp code execution.

B.3 Java Execution Times

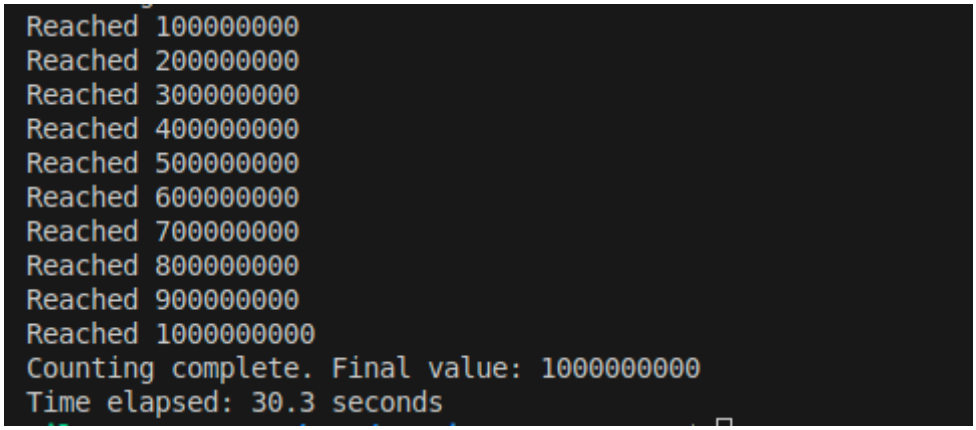

Figure 33: First run of synchronous Java code execution.

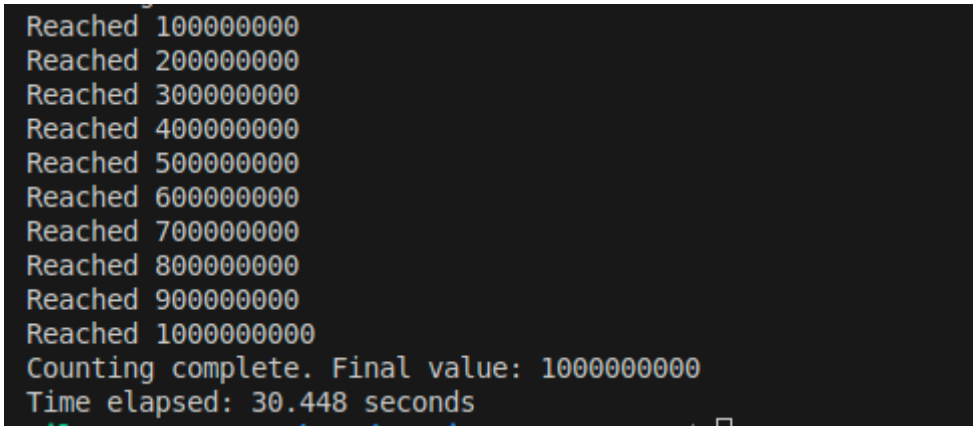

Figure 34: Second run of synchronous Java code execution.

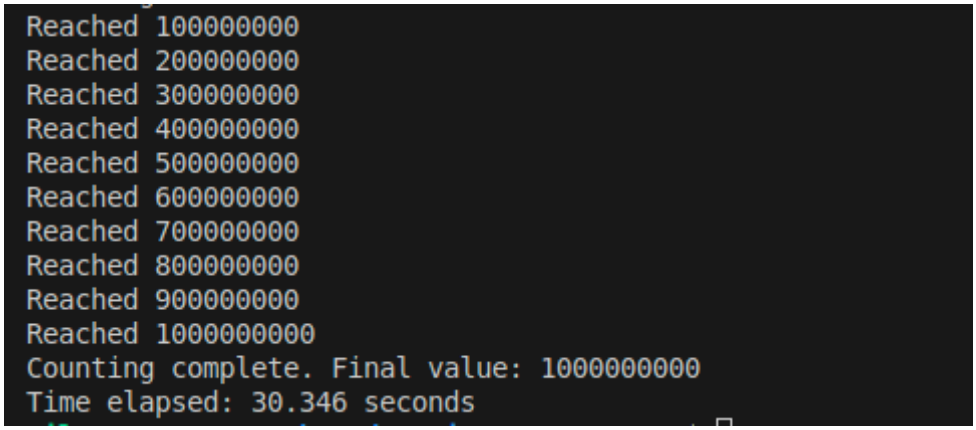

Figure 35: Third run of synchronous Java code execution.

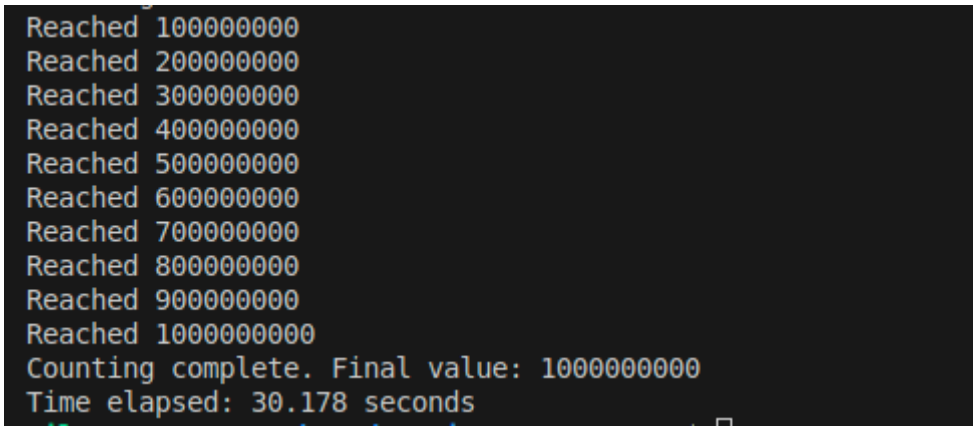

Figure 36: Fourth run of synchronous Java code execution.

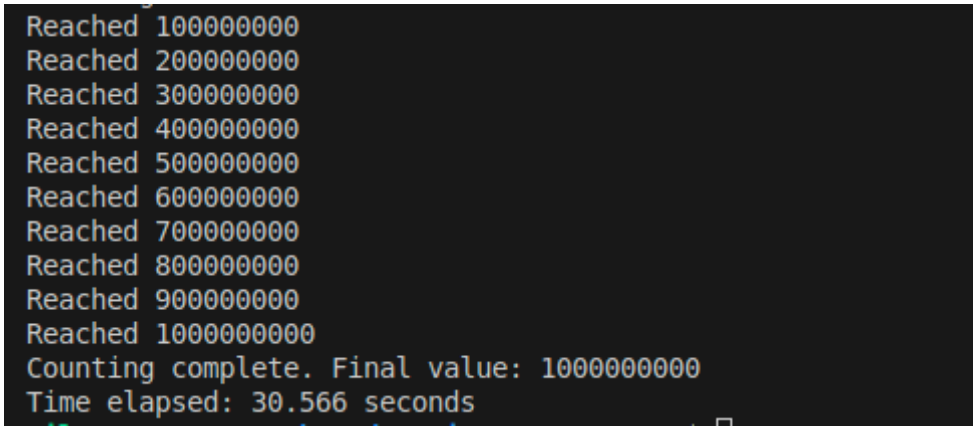

Figure 37: Fifth run of synchronous Java code execution.

B.4 Python Execution Times

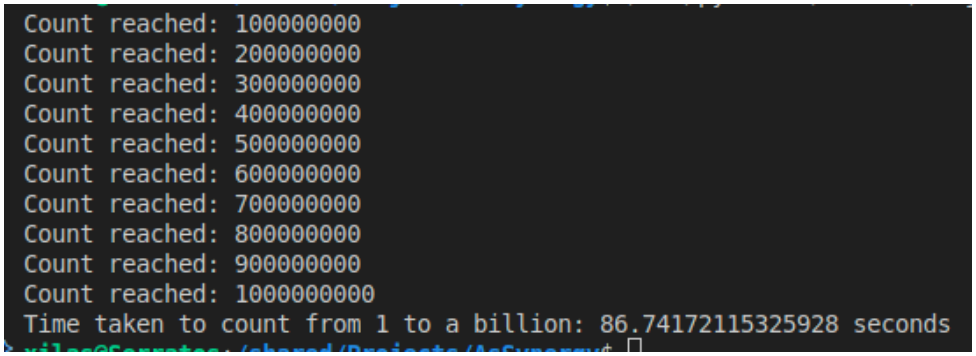

Figure 38: First run of synchronous Python code execution.

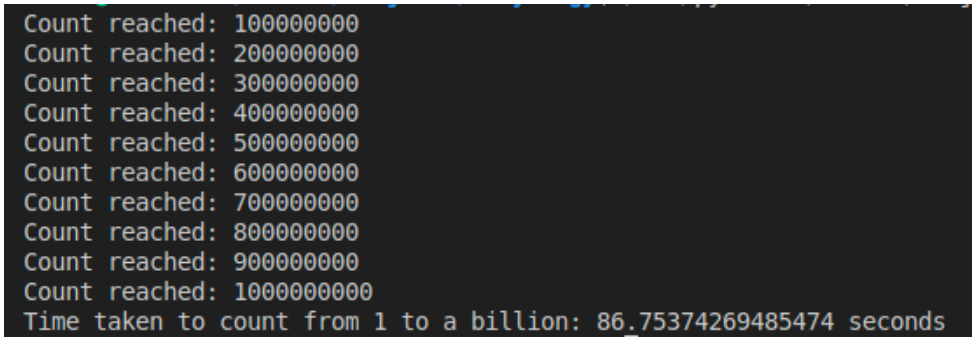

Figure 39: Second run of synchronous Python code execution.

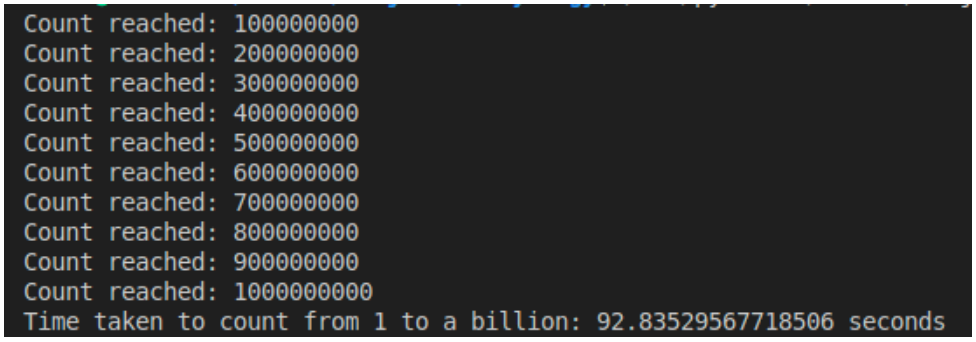

Figure 40: Third run of synchronous Python code execution.

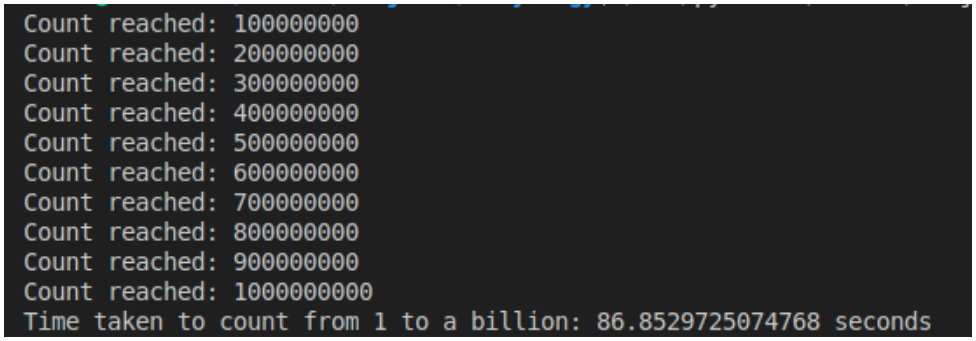

Figure 41: Fourth run of synchronous Python code execution.

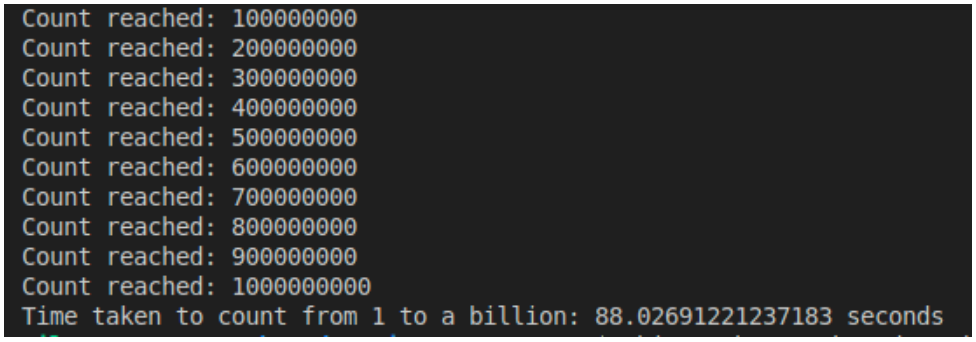

Figure 42: Fifth run of synchronous Python code execution.