

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

J. Wolffrom

May 14, 2025

Abstract

Contents

1	Disclosure	5
2	Introduction	5
2.1	The Problem Area	6
2.1.1	Research Question	6
2.1.2	Limitations	6
3	Asynchronous Systems	7
3.1	Subroutines and Coroutines	8
4	Threading	9
4.1	Threading in different languages	9
4.1.1	Threading in C	9
4.1.2	Threading in C-Sharp	9
4.1.3	Threading in Python	10
4.1.4	Threading in Java	10
5	Posix	10
6	Parallelism	11
6.1	Degree of Parallelism	11
6.2	Thread Level Parallelism (TLP)	11
6.3	Schedulers	11
6.3.1	OS Schedulers	11
6.3.2	.NET	11
6.3.3	JVM	12
6.4	Parallel Execution Time (PET)	12
6.5	Load Balancing	12
6.6	Idle Time	12
7	Memory	13
7.1	Shared Memory	13
7.2	Local Caching	13
7.3	Memory Access Time (MAT)	13
8	TRIN Model	13
9	Further Reading	13
10	Data Collection	13
10.1	Experiment Setup	13
10.2	Results	14
11	Discussion	16
12	Conclusion	17
13	References	18

A	Asynchronous Run-time Snippets	19
A.1	C Code Execution Times	19
A.2	C-Sharp Code Execution Times	24
A.3	Java Execution Times	30
A.4	Python Execution Times	35
B	Synchronous Run-time Snippets	38
B.1	C Code Execution Times	38
B.2	C-Sharp Code Execution Times	40
B.3	Java Execution Times	42
B.4	Python Execution Times	44

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.

2 Introduction

Quoted by Rauber and R  nger 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 R  nger posits that the number of transistors on a microchip used to double roughly 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 R  nger 2023, p. 11) which is a far cry from the 50% annual increases we were seeing in the early 2000's (Rauber and R  nger 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 simultaneously, 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 laymans terms are just functions, a piece of code that can be called from anywhere in the program. This is a common approach in programming, 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.

Coroutines carry an additional layer of complexity, as they allow for the suspension and resumption of execution at specific points. 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.

Subroutines represent the conventional approach to task execution in programming. They are typically implemented as functions that execute sequentially, completing their entire operation before returning control to the calling code. Once a subroutine begins execution, it runs to completion without interruption, following a strict linear execution path.

Coroutines, by contrast, extend the functionality of subroutines with the critical ability to suspend their execution at defined points and later resume from those same points. This suspension and resumption capability is often expressed through 'yield' and 'resume' operations. This fundamental difference enables 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' functionality. 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

The reason why subroutines and coroutines are important to understand is because they relate quite heavily to how threads operate.

Threading is a way of executing multiple different tasks at the same time in a computer operating system. Every program on a computer intrinsically define at least one thread of operations they want the operating system to execute. This is often referred to as the main thread, and is the thread that is executed first when a program is started, which can then spawn sub-threads to execute other tasks. The OS then handles what order threads are executed in using what is known as a "Scheduler", this scheduler is then also responsible for 'Suspending' threads to make it seem like all programs are executed concurrently.

This process of suspending threads is what allows for the illusion of concurrency, allowing the operating system to seemingly execute multiple threads at the same time by switching back and forth rapidly between tasks. This is known as "context switching", and is a key feature of modern operating systems, making even a single core processing system capable of multi-tasking.

This is not to be confused with multi-core processing, which is the ability to execute multiple threads at the same time with different cores. This is where the real performance improvements come from, as we can execute multiple tasks at the same time without having to switch as often.

4.1 Threading in different languages

4.1.1 Threading in C

C, unlike the previous mentions, is a low-level programming language. It is a compiled language that is designed to be fast and efficient, making it a popular choice for system programming and embedded systems. C is often used in applications where performance is critical, such as operating systems, device drivers, and high-performance computing, as well as game engines. It is therefore a prime candidate for multi-core processing optimization, as wherever C is used in practice it is often done so with performance in mind.

C provides a library for multi-core threading called Posix, labeling their way of threading as 'pthreads' or 'POSIX threads'. This library is a standard for multi-core threading in C, and comes pre-installed with unix systems such as Linux in the GCC compiler used for this study.

4.1.2 Threading in C-Sharp

C# is a compiled language that is part of the C language family like Python, however where it differs is its striking resemblance to Java. It is a high-level language that is designed to be easy to use and understand, gathering a lot of attention due to its usage

in the Unity game engine as well as developers transitioning from other languages. It is therefore a prime candidate for multi-core processing optimization as Java developers come potentially pre-equipped with a solid foundation of threading, as well as new game developers looking for performance improvements.

The way C# handles multi-core threading is not the same as Java, as it provides a library to assign tasks to threads, which in turn are managed by the operating system's scheduler. This means that C# is able to utilize multiple cores in a program, but it is not as easy to implement as it is in Java. However, C# does have some advantages over Java when it comes to performance, as it is a compiled language and therefore faster than Java. In addition, C# has a more efficient memory management system than Java, which can lead to performance improvements in large programs.

4.1.3 Threading in Python

As it stands, Python is one of the most popular programming languages in the field of computer science. It is widely used in a variety of applications due to its simplicity and ease of use. However, it is an interpreted language, which means that it is not as fast as compiled languages like C or C++. In fact it is written in C, which is a compiled language. This means that in places it keeps the speed of C by pre-compiling certain commands, but in other places it is slower than C, with up to 10 times the execution in difference.

This is important due to the way Python handles multi-core threading. By default, Python comes pre-installed with threading and asynchronous libraries, which allow for the use of multiple threads in a program, however the presence of the Global Interpreter Lock (GIL) means that only one core will be in use at any time. This means that the performance improvements of multi-core threading are not easily accessed, but by using libraries such as 'multiprocessing' or 'numpy', we can bypass the GIL using some of the same utilities in C. This allows us to use multiple cores in a program, but it is not as easy to implement as it is in other languages.

4.1.4 Threading in Java

Java comes with a built-in threading library, which allows for the use of multiple threads in a program, managed by the Java Virtual Machine (JVM). Usually this is accomplished by inheritance, designating a class as a thread, allowing it to use common threading methods such as sleep (suspension) and join (synchronization). This means that Java is able to utilize multiple cores in a program by use of their own scheduler, and the performance improvements are easily accessible. However, this is not without its drawbacks, as the JVM is an interpreted language, which means that it is not as fast as compiled languages like C or C++. In addition, the JVM is not as efficient as other languages when it comes to memory management, which can lead to performance issues in large programs.

5 Posix

Posix is a standard for multi-core threading in C, and comes pre-installed with unix systems such as Linux in the GCC compiler used for this study. It is a library that provides a set of functions for creating and managing threads, as well as synchronizing

access to shared resources. This means that Posix is able to utilize multiple cores in a program, but it is not as easy to implement as it is in other languages.

6 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.

6.1 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.

6.2 Thread Level Parallelism (TLP)

6.3 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.

6.3.1 OS Schedulers

6.3.2 .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.

6.3.3 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.

6.4 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.

6.5 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.

6.6 Idle Time

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

7 Memory

7.1 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".

7.2 Local Caching

7.3 Memory Access Time (MAT)

Add content if applicable.

8 TRIN Model

Add content if applicable.

9 Further Reading

efficiency cores vs performance cores

10 Data Collection

10.1 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.

10.2 Results

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

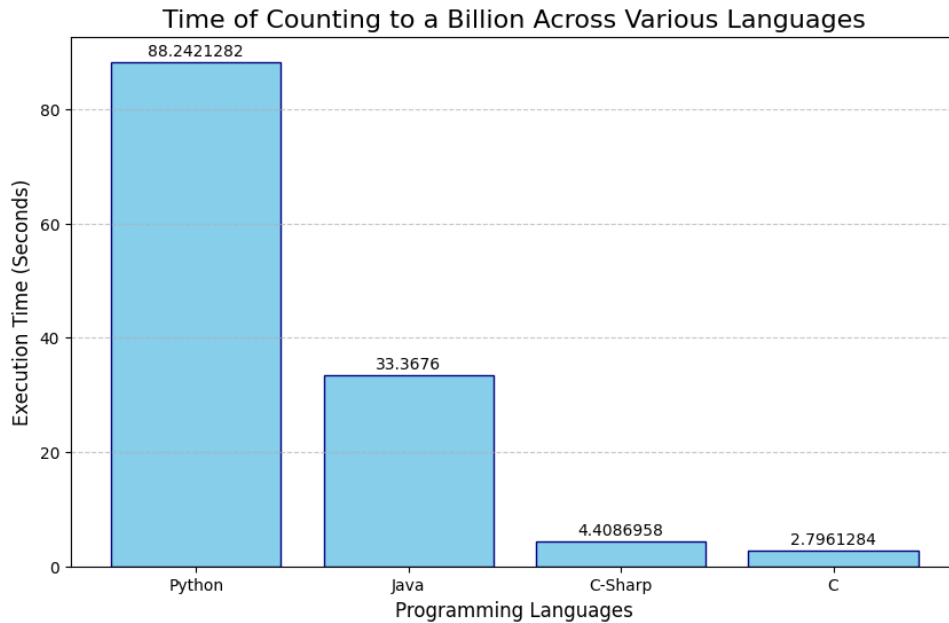


Figure 1: Execution time comparison of different programming languages running synchronously when counting from 1 to 1-billion.

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.

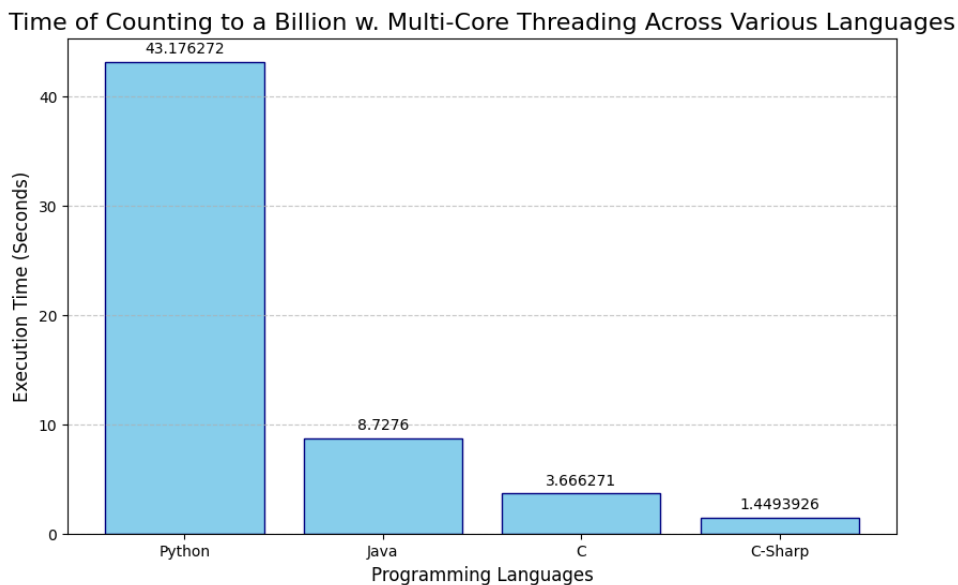


Figure 2: Execution time comparison of different programming languages running asynchronously with multi-threading when counting from 1 to 1-billion.

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.

11 Discussion

Include thoughts about other programming languages.

12 Conclusion

Add content if applicable.

13 References

References

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

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.

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 3: Counted to 50000000
Thread 2: Counted to 50000000
Thread 4: Counted to 50000000
Thread 1: Counted to 50000000
Thread 3: Counted to 100000000
Thread 2: Counted to 100000000
Thread 1: Counted to 100000000
Thread 4: Counted to 100000000
Thread 3: Counted to 150000000
Thread 2: Counted to 150000000
Thread 1: Counted to 150000000
Thread 4: Counted to 150000000
Thread 3: Counted to 200000000
Thread 1: Counted to 200000000
Thread 4: Counted to 200000000
Thread 2: Counted to 200000000
Thread 1: Counted to 250000000
Thread 1: Time taken to count from 1 to 250000000: 3.372792 seconds
Thread 3: Counted to 250000000
Thread 3: Time taken to count from 1 to 250000000: 3.373716 seconds
Thread 2: Counted to 250000000
Thread 2: Time taken to count from 1 to 250000000: 3.419206 seconds
Thread 4: Counted to 250000000
Thread 4: Time taken to count from 1 to 250000000: 3.435106 seconds
All threads have finished counting.
Total time: 3.441684 seconds
```

Figure 4: Second run of asynchronous C code execution.

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 2: Counted to 50000000
Thread 1: Counted to 50000000
Thread 4: Counted to 50000000
Thread 3: Counted to 50000000
Thread 2: Counted to 100000000
Thread 4: Counted to 100000000
Thread 1: Counted to 100000000
Thread 3: Counted to 100000000
Thread 2: Counted to 150000000
Thread 1: Counted to 150000000
Thread 3: Counted to 150000000
Thread 4: Counted to 150000000
Thread 2: Counted to 200000000
Thread 1: Counted to 200000000
Thread 3: Counted to 200000000
Thread 4: Counted to 200000000
Thread 2: Counted to 250000000
Thread 2: Time taken to count from 1 to 250000000: 3.651124 seconds
Thread 1: Counted to 250000000
Thread 1: Time taken to count from 1 to 250000000: 3.829605 seconds
Thread 3: Counted to 250000000
Thread 3: Time taken to count from 1 to 250000000: 3.879884 seconds
Thread 4: Counted to 250000000
Thread 4: Time taken to count from 1 to 250000000: 3.950831 seconds
All threads have finished counting.
Total time: 3.963834 seconds
```

Figure 5: Third run of asynchronous C code execution.

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 3: Counted to 50000000
Thread 1: Counted to 50000000
Thread 2: Counted to 50000000
Thread 4: Counted to 50000000
Thread 3: Counted to 100000000
Thread 1: Counted to 100000000
Thread 2: Counted to 100000000
Thread 4: Counted to 100000000
Thread 3: Counted to 150000000
Thread 1: Counted to 150000000
Thread 4: Counted to 150000000
Thread 2: Counted to 150000000
Thread 1: Counted to 200000000
Thread 3: Counted to 200000000
Thread 4: Counted to 200000000
Thread 2: Counted to 200000000
Thread 1: Counted to 250000000
Thread 1: Time taken to count from 1 to 250000000: 3.330501 seconds
Thread 3: Counted to 250000000
Thread 3: Time taken to count from 1 to 250000000: 3.363438 seconds
Thread 4: Counted to 250000000
Thread 4: Time taken to count from 1 to 250000000: 3.417815 seconds
Thread 2: Counted to 250000000
Thread 2: Time taken to count from 1 to 250000000: 3.485587 seconds
All threads have finished counting.
Total time: 3.486281 seconds
```

Figure 6: Fourth run of asynchronous C code execution.

```
Thread 1 started
Thread 3 started
Thread 2 started
Thread 4 started
Thread 2: Counted to 50000000
Thread 1: Counted to 50000000
Thread 4: Counted to 50000000
Thread 3: Counted to 50000000
Thread 1: Counted to 100000000
Thread 4: Counted to 100000000
Thread 2: Counted to 100000000
Thread 3: Counted to 100000000
Thread 1: Counted to 150000000
Thread 4: Counted to 150000000
Thread 2: Counted to 150000000
Thread 3: Counted to 150000000
Thread 1: Counted to 200000000
Thread 4: Counted to 200000000
Thread 3: Counted to 200000000
Thread 2: Counted to 200000000
Thread 1: Counted to 250000000
Thread 1: Time taken to count from 1 to 250000000: 3.573065 seconds
Thread 3: Counted to 250000000
Thread 3: Time taken to count from 1 to 250000000: 3.697220 seconds
Thread 4: Counted to 250000000
Thread 4: Time taken to count from 1 to 250000000: 3.715673 seconds
Thread 2: Counted to 250000000
Thread 2: Time taken to count from 1 to 250000000: 3.744291 seconds
All threads have finished counting.
Total time: 3.747453 seconds
```

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!

Time taken: 1 hour, 20 min, 04.4067165

Beginning Count

4 Count reached: 0
4 Count reached: 250000000
4 Count reached: 500000000
4 Count reached: 750000000
4 Count reached: 1000000000
3 Count reached: 1250000000

Count reached: 1500000000

Count reached: 1250000000

2 Count reached: 1500000000

Count reached: 1750000000

Count reached: 1500000000

2 Count reached: 1750000000

2 Count reached: 2000000000

Count reached: 1750000000

Count reached: 2000000000

Count reached: 2250000000

Count reached: 2000000000

2 Count reached: 2250000000

Count reached: 2500000000

Counting complete!

Count reached: 2250000000

Time elapsed: 00:00:01.4276984

Count reached: 2500000000

Counting complete!

Time elapsed: 00:00:01.4566220

Count reached: 2500000000

Beginning Count

4 Count reached: 0

4 Count reached: 250000000

3 Count reached: 500000000

3 Count reached: 750000000

Count reached: 500000000

3 Count reached: 1000000000

Count reached: 750000000

3 Count reached: 1250000000

Count reached: 1000000000

3 Count reached: 1500000000

Count reached: 1250000000

3 Count reached: 1750000000

Count reached: 1500000000

3 Count reached: 2000000000

Count reached: 1750000000

3 Count reached: 2250000000

Count reached: 2000000000

Count reached: 2500000000

Counting complete!

Time elapsed: 00:00:01.3299419

Count reached: 2500000000

Counting complete!

Time elapsed: 00:00:01.3475107

Count reached: 2250000000

Count reached: 2500000000

Counting complete!

```
Beginning Count
4 Count reached: 0
4 Count reached: 250000000
4 Count reached: 500000000
4 Count reached: 750000000
4 Count reached: 1000000000
4 Count reached: 1250000000
4 Count reached: 1500000000
4 Count reached: 1750000000
4 Count reached: 2000000000
4 Count reached: 2250000000
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.4265317
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.4277206
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.4437055
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.4667014
```

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

```
Beginning Count
4 Count reached: 0
4 Count reached: 250000000
4 Count reached: 500000000
4 Count reached: 750000000
4 Count reached: 1000000000
4 Count reached: 1250000000
4 Count reached: 1500000000
4 Count reached: 1750000000
4 Count reached: 2000000000
4 Count reached: 2250000000
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.3102708
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.3314314
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.3522072
Count reached: 2500000000
Counting complete!
Time elapsed: 00:00:01.3597450
```

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

A.3 Java Execution Times

```
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Reached 50000000
Reached 50000000
Reached 50000000
Reached 50000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.543 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.565 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.613 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.681 seconds
```

Figure 13: First run of asynchronous Java code execution.

```
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Reached 50000000
Reached 50000000
Reached 50000000
Reached 50000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.181 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.198 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.199 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.512 seconds
```

Figure 14: Second run of asynchronous Java code execution.

```
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Reached 50000000
Reached 50000000
Reached 50000000
Reached 50000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.341 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.431 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.489 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.599 seconds
```

Figure 15: Third run of asynchronous Java code execution.


```
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Reached 50000000
Reached 50000000
Reached 50000000
Reached 50000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.704 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.853 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.883 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 9.094 seconds
```

Figure 16: Fourth run of asynchronous Java code execution.

```
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Starting to count from 1 to 1 quarter billion...
Reached 50000000
Reached 50000000
Reached 50000000
Reached 50000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 100000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 150000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 200000000
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.43 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.541 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.549 seconds
Reached 250000000
Counting complete. Final value: 250000000
Time elapsed: 8.752 seconds
```

Figure 17: Fifth run of asynchronous Java code execution.

A.4 Python Execution Times

```
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Time taken to count from 1 to a billion: 41.01625418663025 seconds
Time taken to count from 1 to a billion: 41.361515522003174 seconds
Time taken to count from 1 to a billion: 41.5295844078064 seconds
Time taken to count from 1 to a billion: 43.287073612213135 seconds
```

Figure 18: First run of asynchronous Python code execution.

```
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Time taken to count from 1 to a billion: 40.22830271720886 seconds
Time taken to count from 1 to a billion: 41.19698143005371 seconds
Time taken to count from 1 to a billion: 42.159634590148926 seconds
Time taken to count from 1 to a billion: 42.578996419906616 seconds
```

Figure 19: Second run of asynchronous Python code execution.

```

Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Time taken to count from 1 to a billion: 40.6362566947937 seconds
Time taken to count from 1 to a billion: 41.61786890029907 seconds
Time taken to count from 1 to a billion: 42.53819298744202 seconds
Time taken to count from 1 to a billion: 42.61203145980835 seconds

```

Figure 20: Third run of asynchronous Python code execution.

```

Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Time taken to count from 1 to a billion: 41.284507036209106 seconds
Time taken to count from 1 to a billion: 41.81537675857544 seconds
Time taken to count from 1 to a billion: 42.89721870422363 seconds
Time taken to count from 1 to a billion: 44.51924276351929 seconds

```

Figure 21: Fourth run of asynchronous Python code execution.

```
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 50000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 100000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 150000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Count reached: 200000000
Time taken to count from 1 to a billion: 40.59927582740784 seconds
Time taken to count from 1 to a billion: 40.85261845588684 seconds
Time taken to count from 1 to a billion: 42.05317497253418 seconds
Time taken to count from 1 to a billion: 42.88402080535889 seconds
```

Figure 22: Fifth run of asynchronous Python code execution.

B Synchronous Run-time Snippets

B.1 C Code Execution Times

```
Counted to 100000000  
Counted to 200000000  
Counted to 300000000  
Counted to 400000000  
Counted to 500000000  
Counted to 600000000  
Counted to 700000000  
Counted to 800000000  
Counted to 900000000  
Counted to 1000000000  
Time taken to count from 1 to a billion: 2.837288 seconds
```

Figure 23: First run of synchronous C code execution.

```
Counted to 100000000  
Counted to 200000000  
Counted to 300000000  
Counted to 400000000  
Counted to 500000000  
Counted to 600000000  
Counted to 700000000  
Counted to 800000000  
Counted to 900000000  
Counted to 1000000000  
Time taken to count from 1 to a billion: 2.799428 seconds
```

Figure 24: Second run of synchronous C code execution.

```
Counted to 100000000
Counted to 200000000
Counted to 300000000
Counted to 400000000
Counted to 500000000
Counted to 600000000
Counted to 700000000
Counted to 800000000
Counted to 900000000
Counted to 1000000000
Time taken to count from 1 to a billion: 2.761543 seconds
```

Figure 25: Third run of synchronous C code execution.

```
Counted to 100000000
Counted to 200000000
Counted to 300000000
Counted to 400000000
Counted to 500000000
Counted to 600000000
Counted to 700000000
Counted to 800000000
Counted to 900000000
Counted to 1000000000
Time taken to count from 1 to a billion: 2.766171 seconds
```

Figure 26: Fourth run of synchronous C code execution.

```
Counted to 100000000
Counted to 200000000
Counted to 300000000
Counted to 400000000
Counted to 500000000
Counted to 600000000
Counted to 700000000
Counted to 800000000
Counted to 900000000
Counted to 1000000000
Time taken to count from 1 to a billion: 2.816212 seconds
```

Figure 27: Fifth run of synchronous C code execution.

B.2 C-Sharp Code Execution Times

```
Count reached: 100000000  
Count reached: 200000000  
Count reached: 300000000  
Count reached: 400000000  
Count reached: 500000000  
Count reached: 600000000  
Count reached: 700000000  
Count reached: 800000000  
Count reached: 900000000  
Count reached: 1000000000  
Counting complete!  
Time elapsed: 00:00:04.4104558
```

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

```
Count reached: 100000000  
Count reached: 200000000  
Count reached: 300000000  
Count reached: 400000000  
Count reached: 500000000  
Count reached: 600000000  
Count reached: 700000000  
Count reached: 800000000  
Count reached: 900000000  
Count reached: 1000000000  
Counting complete!  
Time elapsed: 00:00:04.4177039
```

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


```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Counting complete!
Time elapsed: 00:00:04.3900166
```

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

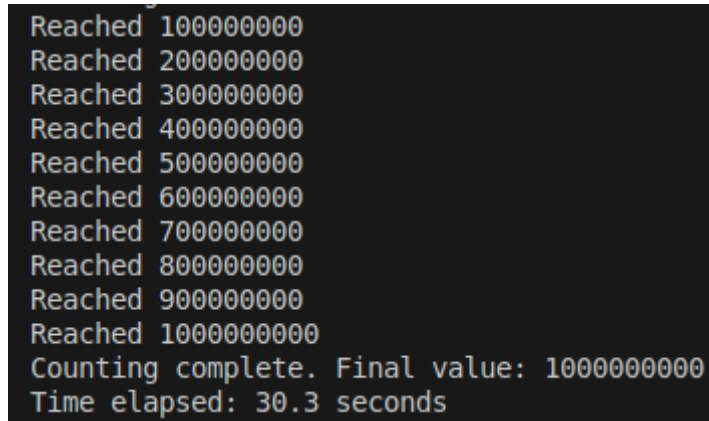
```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Counting complete!
Time elapsed: 00:00:04.3821615
```

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

```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Counting complete!
Time elapsed: 00:00:04.4431445
```

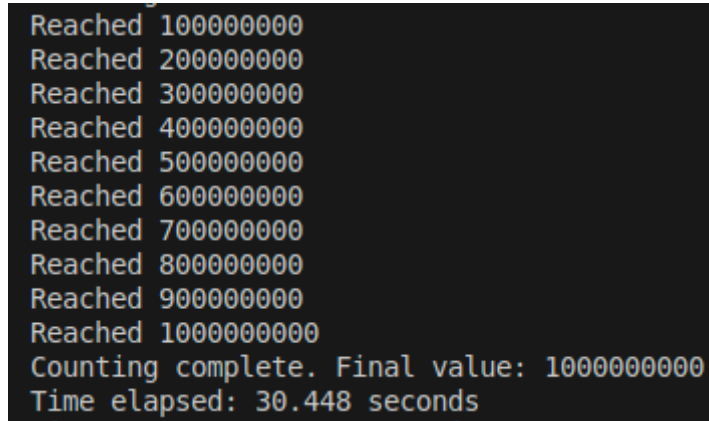
Figure 32: Fifth run of synchronous C-Sharp code execution.

B.3 Java Execution Times



```
Reached 100000000
Reached 200000000
Reached 300000000
Reached 400000000
Reached 500000000
Reached 600000000
Reached 700000000
Reached 800000000
Reached 900000000
Reached 1000000000
Counting complete. Final value: 1000000000
Time elapsed: 30.3 seconds
```

Figure 33: First run of synchronous Java code execution.



```
Reached 100000000
Reached 200000000
Reached 300000000
Reached 400000000
Reached 500000000
Reached 600000000
Reached 700000000
Reached 800000000
Reached 900000000
Reached 1000000000
Counting complete. Final value: 1000000000
Time elapsed: 30.448 seconds
```

Figure 34: Second run of synchronous Java code execution.

```
Reached 100000000
Reached 200000000
Reached 300000000
Reached 400000000
Reached 500000000
Reached 600000000
Reached 700000000
Reached 800000000
Reached 900000000
Reached 1000000000
Counting complete. Final value: 1000000000
Time elapsed: 30.346 seconds
```

Figure 35: Third run of synchronous Java code execution.

```
Reached 100000000
Reached 200000000
Reached 300000000
Reached 400000000
Reached 500000000
Reached 600000000
Reached 700000000
Reached 800000000
Reached 900000000
Reached 1000000000
Counting complete. Final value: 1000000000
Time elapsed: 30.178 seconds
```

Figure 36: Fourth run of synchronous Java code execution.

```
Reached 100000000
Reached 200000000
Reached 300000000
Reached 400000000
Reached 500000000
Reached 600000000
Reached 700000000
Reached 800000000
Reached 900000000
Reached 1000000000
Counting complete. Final value: 1000000000
Time elapsed: 30.566 seconds
```

Figure 37: Fifth run of synchronous Java code execution.

B.4 Python Execution Times

```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Time taken to count from 1 to a billion: 86.74172115325928 seconds
```

Figure 38: First run of synchronous Python code execution.

```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Time taken to count from 1 to a billion: 86.75374269485474 seconds
```

Figure 39: Second run of synchronous Python code execution.

```
Count reached: 100000000
Count reached: 200000000
Count reached: 300000000
Count reached: 400000000
Count reached: 500000000
Count reached: 600000000
Count reached: 700000000
Count reached: 800000000
Count reached: 900000000
Count reached: 1000000000
Time taken to count from 1 to a billion: 92.83529567718506 seconds
```

Figure 40: Third run of synchronous Python code execution.

```
Count reached: 100000000  
Count reached: 200000000  
Count reached: 300000000  
Count reached: 400000000  
Count reached: 500000000  
Count reached: 600000000  
Count reached: 700000000  
Count reached: 800000000  
Count reached: 900000000  
Count reached: 1000000000  
Time taken to count from 1 to a billion: 86.8529725074768 seconds
```

Figure 41: Fourth run of synchronous Python code execution.

```
Count reached: 100000000  
Count reached: 200000000  
Count reached: 300000000  
Count reached: 400000000  
Count reached: 500000000  
Count reached: 600000000  
Count reached: 700000000  
Count reached: 800000000  
Count reached: 900000000  
Count reached: 1000000000  
Time taken to count from 1 to a billion: 88.02691221237183 seconds
```

Figure 42: Fifth run of synchronous Python code execution.