**NAME** : ISHA TANNA

**UID** : 225037

**ROLL NO**. : 24

**CLASS** : TY BSC IT

**SUBJECT** : BIG DATA AND CLOUD COMPUTING (SITS0601)

**CIA 1 ASSIGNMENT** : PYTHON PERFORMANCE ANALYSIS – A COMPARATIVE STUDY WITH PARALLELIZATION

**[I] Performance Comparison of Python Implementations**

Flavors Chosen for Comparison: **CPython, PyPy, Jython and IronPython.**

Python is a versatile programming language, and its performance can vary depending on the implementation used. In this task, we aim to evaluate the performance of four Python implementations: **CPython**, **PyPy**, **Jython**, and **IronPython**, by benchmarking their execution times on selected algorithms. Below is a brief introduction to each implementation:

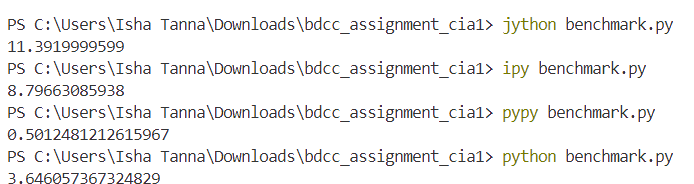
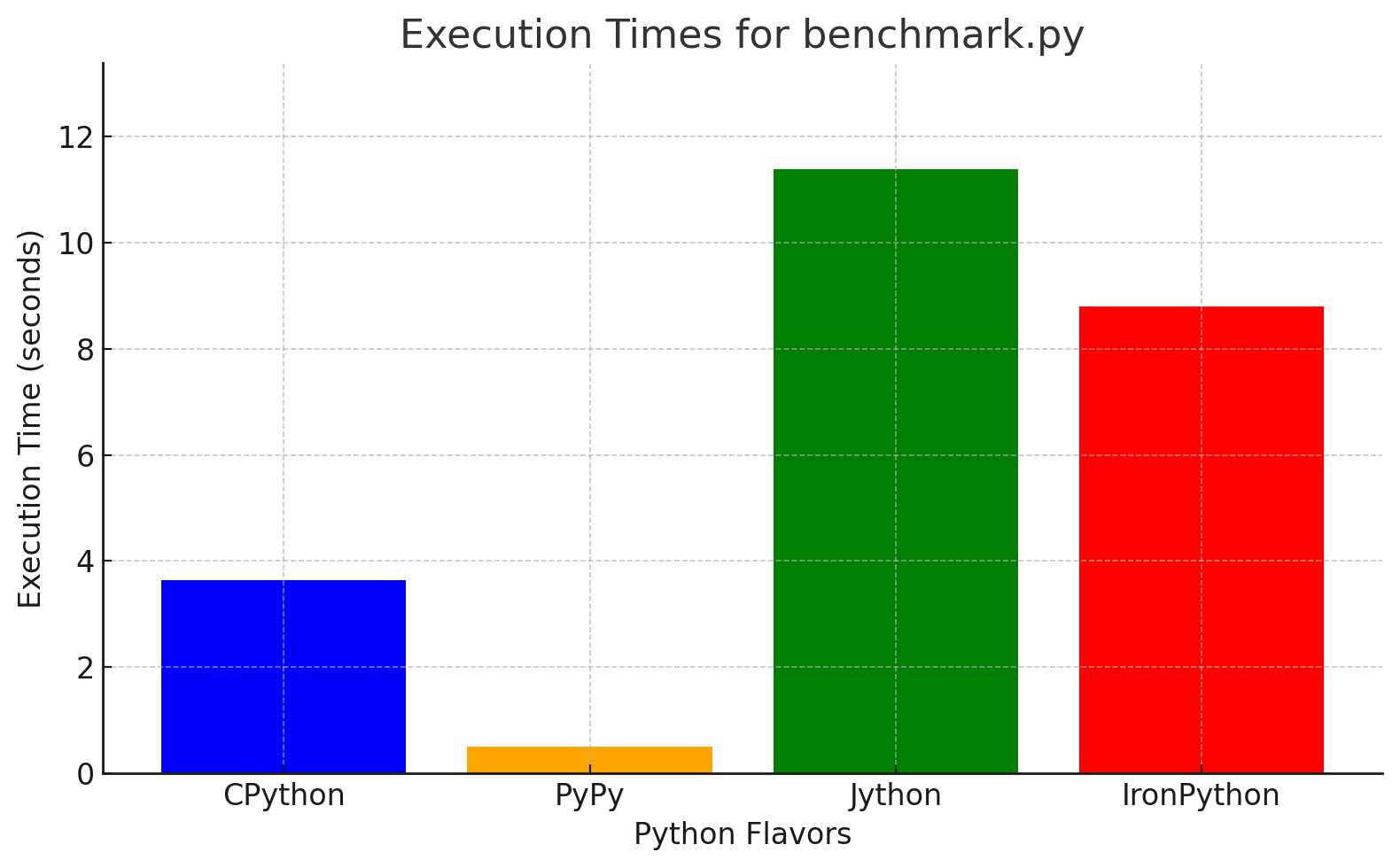
1. **CPython**:  
   CPython is the default and most widely used implementation of Python. It is written in C and is known for its ease of use and extensive library support. However, its performance may be limited in computationally intensive tasks due to the Global Interpreter Lock (GIL).
2. **PyPy**:  
   PyPy is an alternative implementation of Python that focuses on speed and efficiency. It includes a Just-In-Time (JIT) compiler, which dynamically translates Python code into machine code, significantly improving execution speed for long-running programs.
3. **Jython**:  
   Jython is a Python implementation written in Java. It runs on the Java Virtual Machine (JVM), allowing seamless integration with Java libraries. While it benefits from the JVM's optimizations, it does not support some Python features, such as C extensions.
4. **IronPython**:  
   IronPython is a Python implementation for the .NET framework. It is written in C# and integrates well with .NET libraries. However, its performance is often slower than CPython for some use cases due to differences in its underlying architecture.

Through this comparison, we aim to identify the strengths and weaknesses of each implementation and determine their suitability for specific types of workloads.

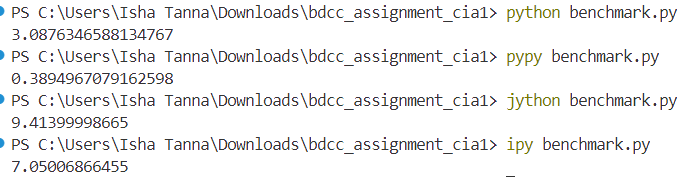
Benchmarking Programs and Observations:

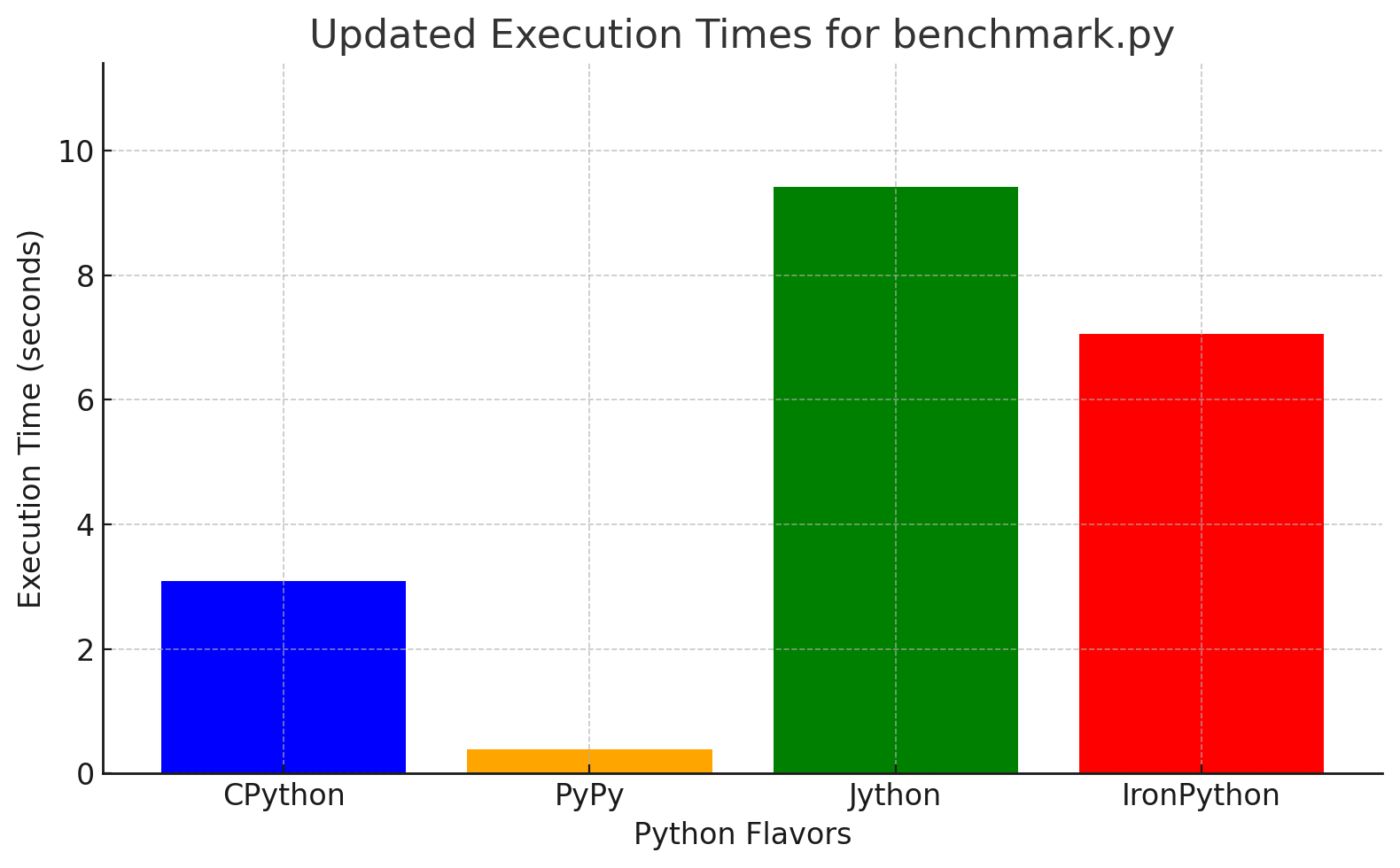
1. **Matrix Multiplication** is a computationally intensive operation commonly used in various fields such as scientific computing, machine learning, graphics processing, and data analysis. Benchmarking Python implementations using matrix multiplication is useful because it highlights their efficiency in handling numerical computations and memory management. It stresses the interpreter’s ability to handle iterative and nested operations, as well as its optimization for mathematical workloads. Since matrix multiplication scales with the size of matrices, it also demonstrates the interpreter’s performance in scenarios involving large datasets. This makes it an ideal candidate for comparing Python flavors, especially for applications that rely on numerical precision and speed.

Execution Time for a Single Run:

Average Execution Time after Multiple Runs (helps remove noise) :



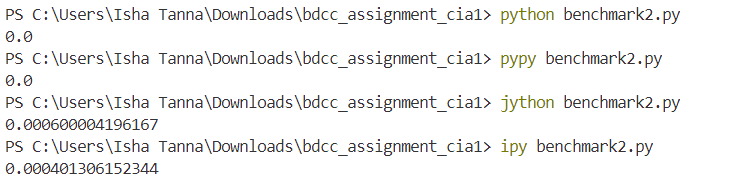


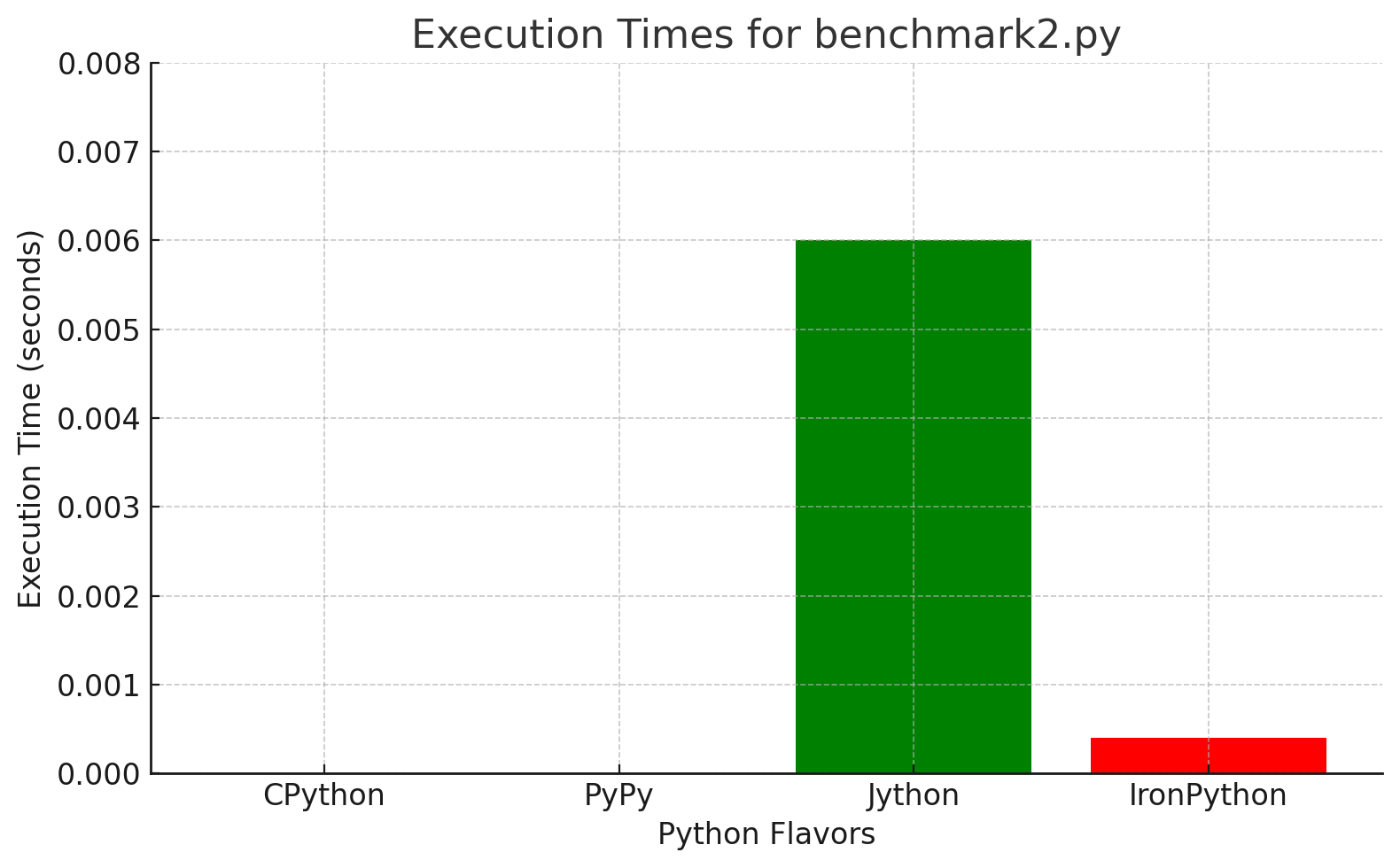
We also notice a reduction in the execution time for the PyPy compiler when we perform a few extra warm-up iterations :



A Just-In-Time (JIT) compiler, like the one used in PyPy, needs warm-up iterations to optimize performance. During these iterations, the JIT collects runtime information about frequently executed code paths (hot spots) and compiles them into highly optimized machine code. This process involves analyzing and adapting to the specific workload, which can initially result in slower execution as the compiler gathers data. However, once optimization is complete, the compiled code runs significantly faster than interpreted code, resulting in overall improved performance. This warm-up phase is why benchmarking with JIT compilers should include multiple iterations to capture stabilized performance.

1. Dijkstra's algorithm is widely used for finding the shortest paths in graphs, making it a cornerstone of network optimization, routing, and logistics. Benchmarking Python implementations with this algorithm showcases their efficiency in handling data structures like heaps and dictionaries, as well as recursive and iterative operations. It tests the interpreter’s ability to manage dynamic memory allocation and prioritize computational tasks effectively. Since the algorithm operates on graphs with varying sizes and edge weights, it highlights the interpreter’s performance in scenarios requiring real-time updates and complex data handling. This makes it a suitable choice for benchmarking in applications involving graph traversal and optimization.

Average Execution Time After Multiple Runs:  




Analysis:

**1. PyPy**

* **Observation**: PyPy is the fastest in the results, which is expected.
* **Reason**:
  + PyPy uses Just-In-Time (JIT) compilation to optimize runtime performance.
  + It is particularly well-suited for long-running and computationally intensive tasks like matrix multiplication.
* **Typical Behavior**: PyPy often outperforms CPython by a significant margin (3-10x or more), as seen here.

**2. CPython**

* **Observation**: CPython is slower than PyPy but faster than Jython and IronPython.
* **Reason**:
  + CPython is the standard Python implementation and uses an interpreter, not a JIT compiler.
  + It relies on well-optimized standard libraries but doesn't benefit from runtime optimizations like PyPy.
* **Typical Behavior**: CPython is considered the baseline for Python performance.

**3. IronPython**

* **Observation**: IronPython is slower than CPython and PyPy.
* **Reason**:
  + IronPython runs on the .NET framework, which can add overhead compared to native Python implementations.
  + It lacks support for many performance-critical Python libraries, and its performance is often hindered in computationally heavy tasks.
* **Typical Behavior**: IronPython performs well in .NET-specific tasks but is generally slower for Python-native operations.

**4. Jython**

* **Observation**: Jython is the slowest in the results.
* **Reason**:
  + Jython runs on the Java Virtual Machine (JVM), which introduces overhead for Python code.
  + It doesn't support Python's C extensions (e.g., NumPy), limiting performance optimization for computationally intensive tasks.
  + JVM optimizations are less effective for Python's dynamic typing and data structures.
* **Typical Behavior**: Jython's performance is usually slower than other Python flavors for computation-heavy tasks.

**[II] Algorithm Parallelization**

Algorithm Chosen : Quick-Sort

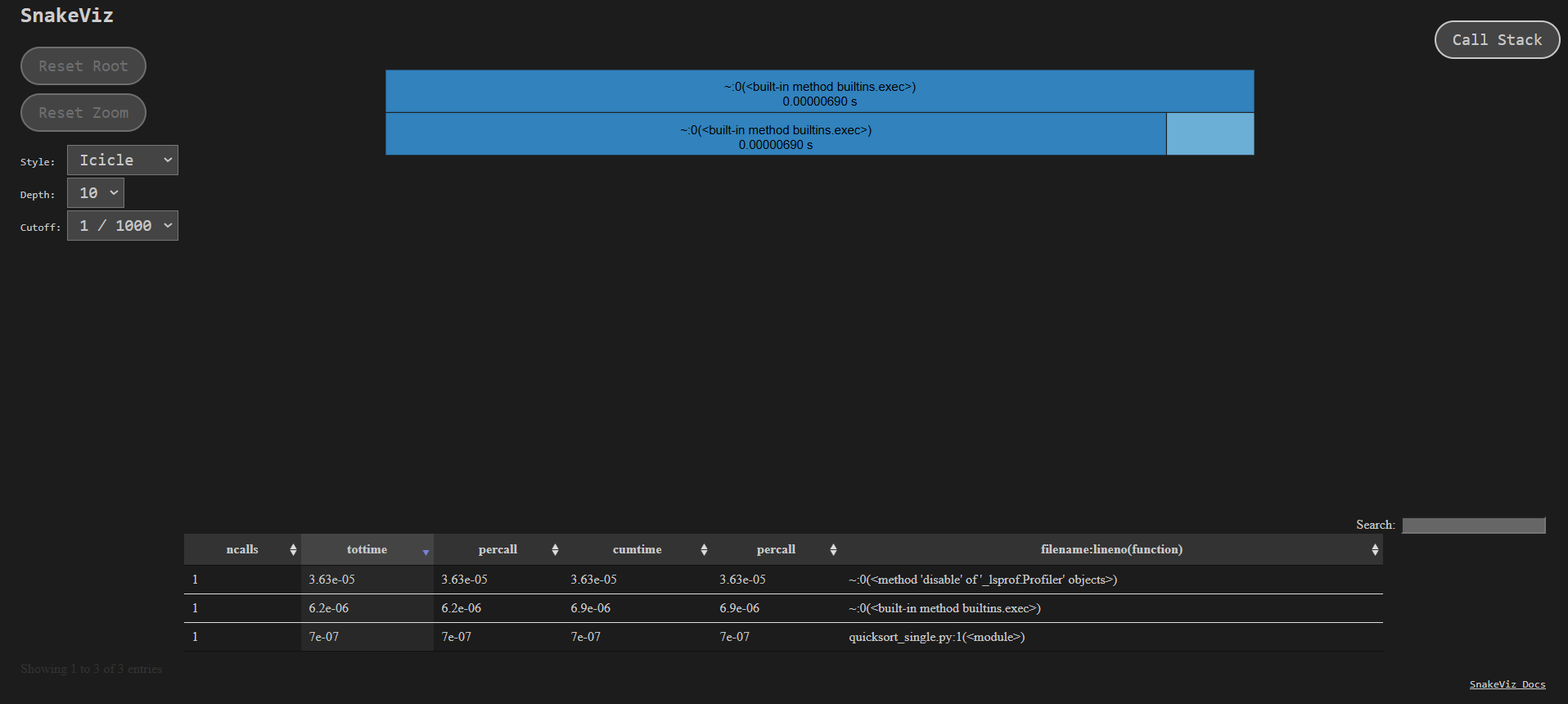
Quick Sort is a divide-and-conquer algorithm that recursively divides the input array into smaller subarrays, making it suitable for parallelization. Quick Sort divides the array using a pivot element, placing smaller elements to the left and larger elements to the right, and recursively sorting each part.

Time Complexity:

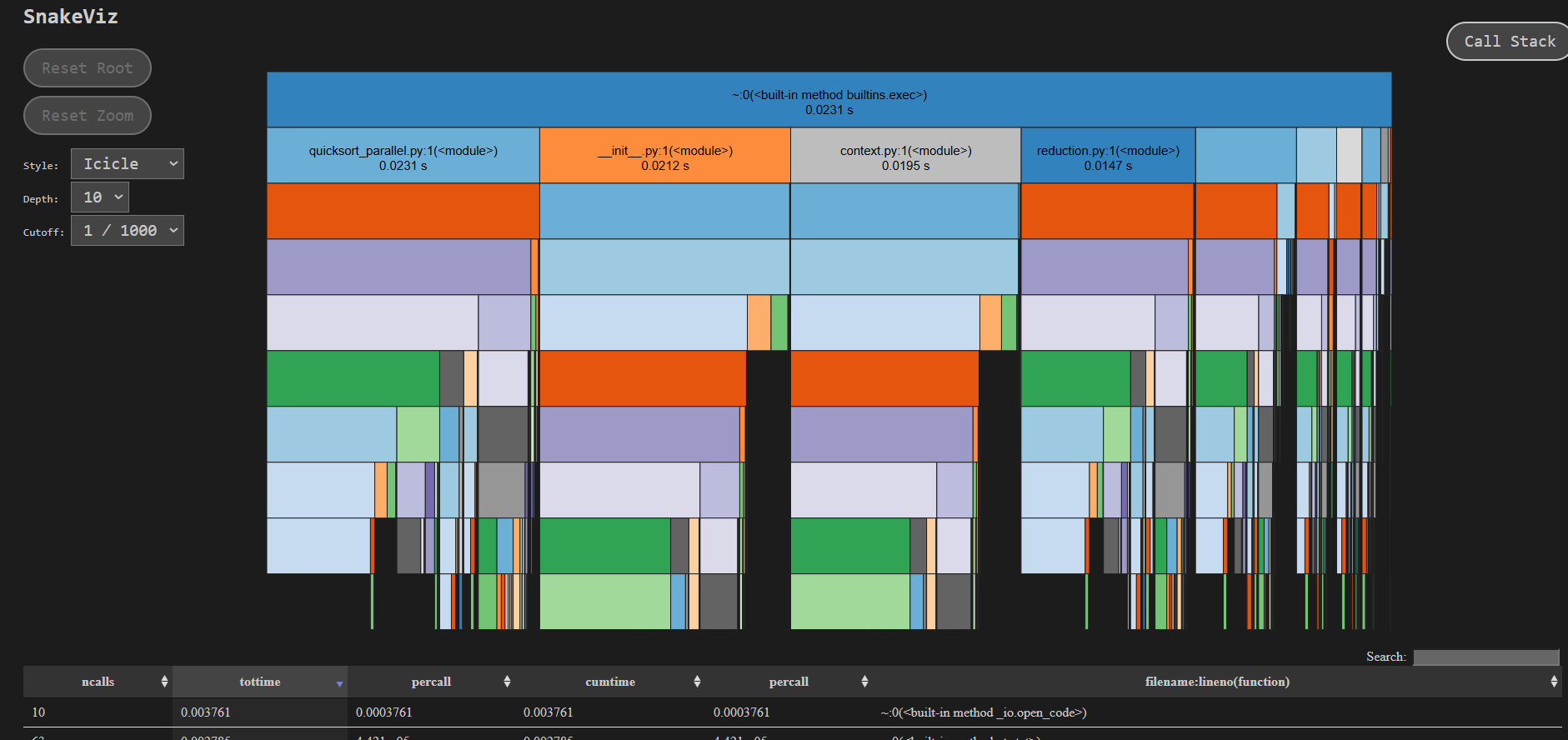
* Best/Average Case: O(n log n)
* Worst Case: O(n^2)

1. **Profiling and Time Analysis:**

(Single Threaded Quicksort cProfiling Result)



(Parallelized Quicksort cProfiling Result)

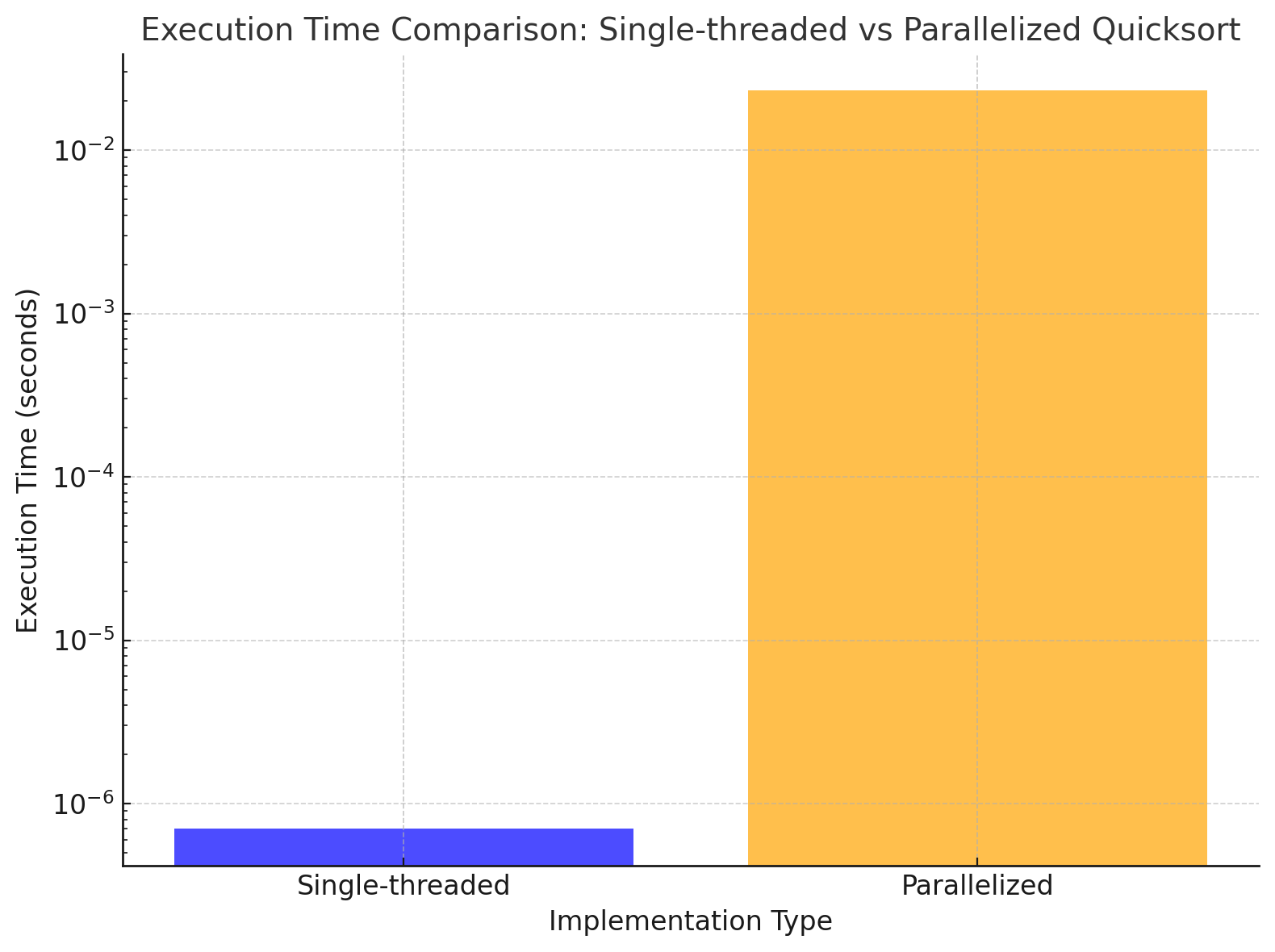


**Profiling Data:**

1. **Single-Threaded Quicksort**:
   * **Execution Time (tottime)**:
     + Key function (quicksort): ~7×10−77 \times 10^{-7}7×10−7 seconds.
   * **Number of Calls**:
     + Function invoked only once.
2. **Parallelized Quicksort**:
   * **Execution Time (tottime)**:
     + Key function (parallel\_quicksort): ~2.31×10−22.31 \times 10^{-2}2.31×10−2 seconds.
   * **Number of Calls**:
     + Significantly higher due to multiprocessing overhead.

**Observations:**

* The **parallelized version** demonstrates increased overhead due to inter-process communication and spawning of multiple processes.
* For smaller datasets, the overhead outweighs the benefits of parallelization.
* The single-threaded version is faster for small inputs but will scale poorly as input size grows.



The bar chart illustrates the execution times for the single-threaded and parallelized implementations of quicksort. It uses a logarithmic scale to emphasize the significant difference in performance for the provided dataset.

**Findings:**

1. **Single-threaded Quicksort**:
   * Very efficient for small input sizes.
   * Minimal overhead, resulting in faster execution for this dataset.
2. **Parallelized Quicksort**:
   * High overhead due to process creation and inter-process communication.
   * Execution time is significantly higher for smaller datasets, as shown in the chart.
3. **Scalability Analysis:**

* **Single-threaded quicksort**:
  + Time complexity: O(n log n) in the average case.
  + Sequential execution limits scalability on multi-core systems.
* **Parallelized quicksort**:
  + Parallel execution can reduce the depth of recursion by splitting the array into parts.
  + **Potential Speedup**: Approximately proportional to the number of CPU cores, minus overhead.
  + **Limitation**: Small arrays or tasks that are not compute-intensive can lead to inefficiencies.

**Implications:**

* **Small Inputs**:
  + Parallelization is inefficient due to its overhead.
  + Single-threaded execution is preferable.
* **Large Inputs**:
  + As input size increases, parallelized quicksort becomes more effective, leveraging multiple CPU cores.
  + Speedup is capped by Amdahl's Law, with diminishing returns as the number of cores increases.