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 2 selected algorithms.

Here is a brief introduction to each flavour / implementation:

1. **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** 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** 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** 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.

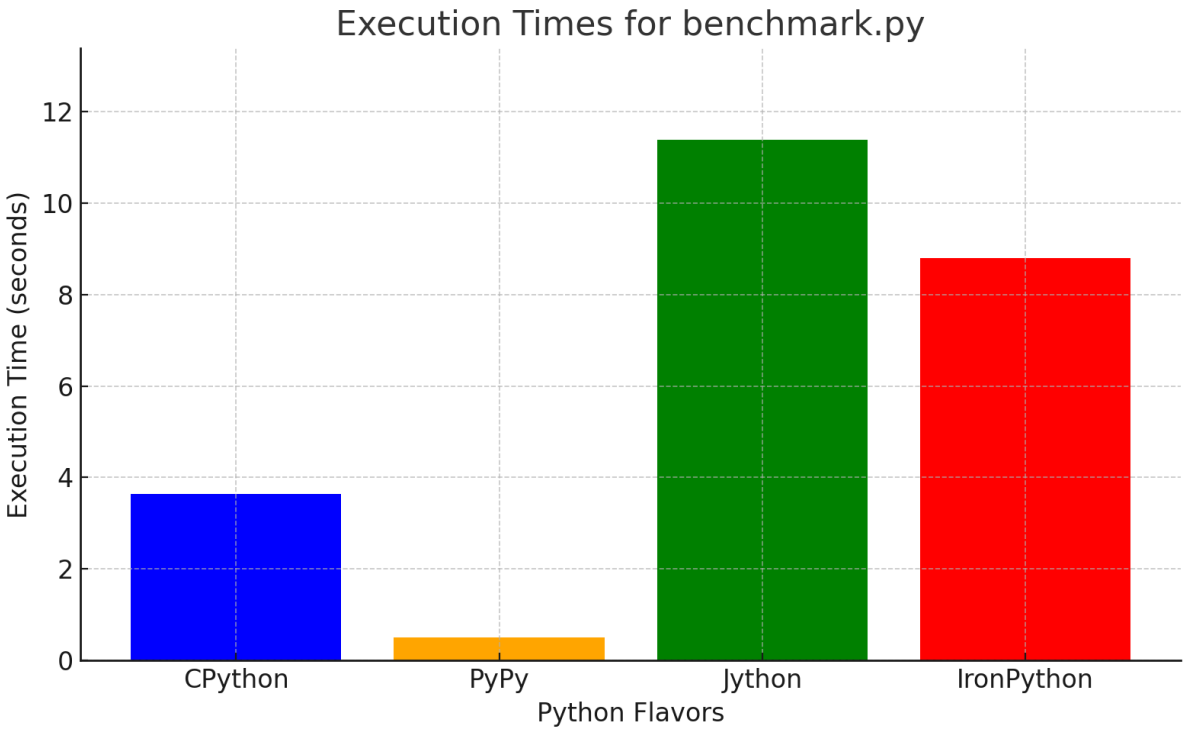
Benchmarking Programs and Observations:

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.

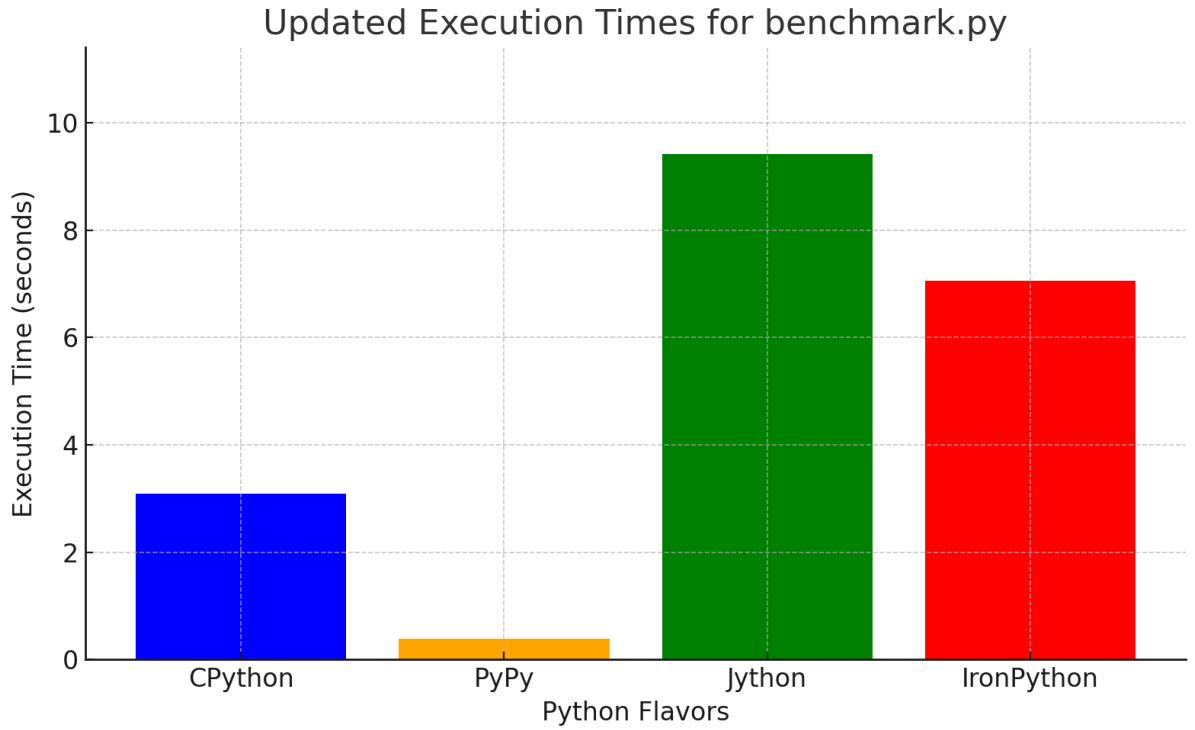
Execution Time for a Single Run:

```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> jython benchmark.py
11.3919999599
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> ipy benchmark.py
8.79663085938
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> pypy benchmark.py
0.5012481212615967
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> python benchmark.py
3.646057367324829
```



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

```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> python benchmark.py
3.0876346588134767
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> pypy benchmark.py
0.3894967079162598
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> jython benchmark.py
9.41399998665
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> ipy benchmark.py
7.05006866455
```



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

```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> pypy benchmark.py
0.37494421005249023
```

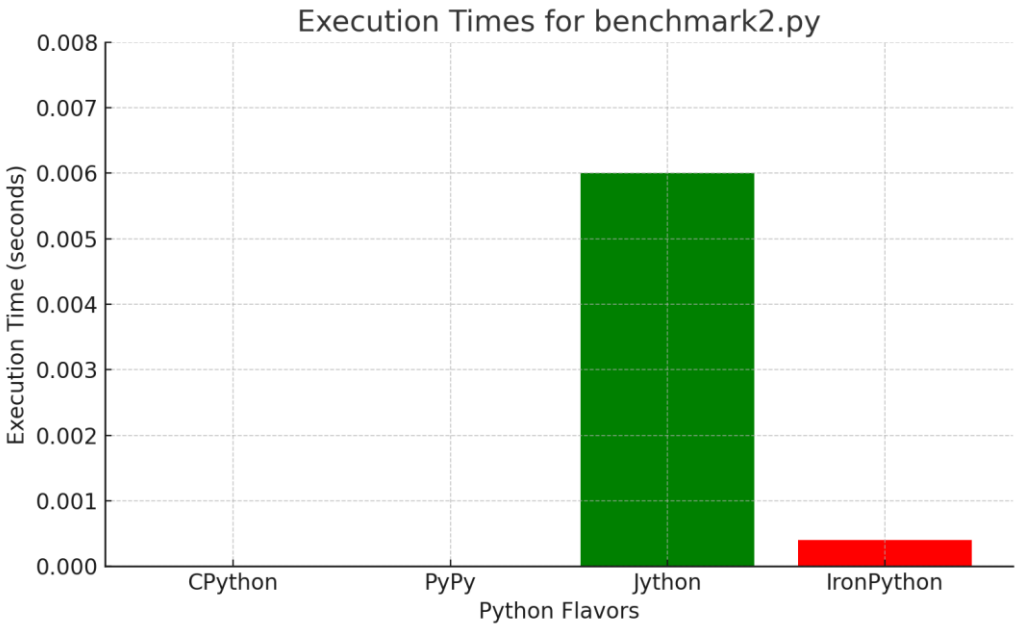
A Just-In-Time (JIT) compiler, like the one used in PyPy, needs warm-up iterations to optimize performance. During these iterations, the JIT collects runtime information about frequently executed code paths (hot spots) and compiles them into highly optimized machine code. This process involves analyzing 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.

2. **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:

```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> python benchmark2.py
0.0
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> pypy benchmark2.py
0.0
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> jython benchmark2.py
0.000600004196167
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1> ipy benchmark2.py
0.000401306152344
```



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: Merge – Sort Algorithm

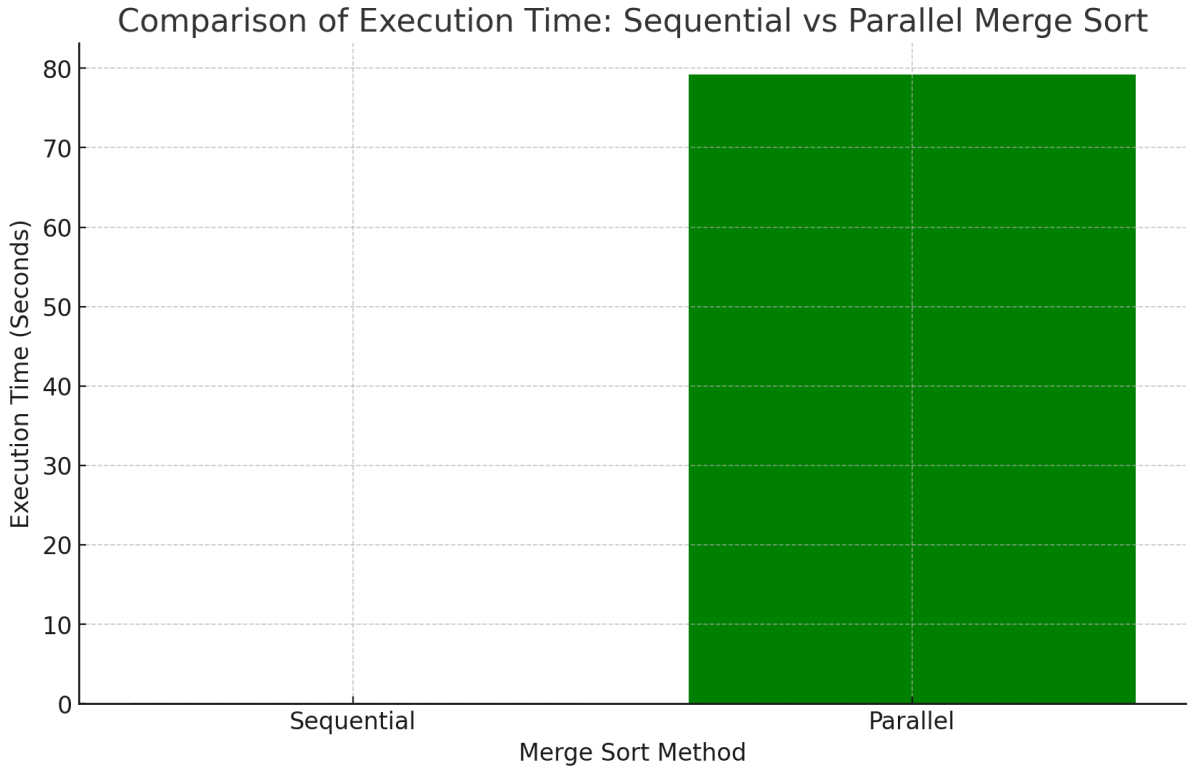
Merge Sort is a **divide-and-conquer** algorithm that works by recursively dividing the array into two halves, sorting each half, and then merging the sorted halves to produce the final sorted array. This process involves repeatedly splitting the array until each subarray contains a single element, and then merging them back in sorted order. The time complexity of Merge Sort is **O(n log n)**, making it an efficient algorithm for large datasets.

Why Merge Sort?

Merge Sort is particularly well-suited for parallelization due to its recursive nature. The sorting of the two halves can be done independently, making it an ideal candidate for multi-threading. By parallelizing the recursive calls to sort the left and right halves of the array, we can significantly reduce the execution time for large datasets.

Here, we aim to demonstrate the impact of parallelization using **threading** to improve the performance of the Merge Sort algorithm, especially as the input size grows.

Profiling and Time Analysis (Sorting an array of 10,000 elements):



Sequential Merge Sort

```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1\bdcc-assignment> python mergesort_sequential.py
         282558 function calls (262560 primitive calls) in 0.078 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.078    0.078 <string>:1(<module>)
  19999/1    0.052    0.000    0.078    0.078 mergesort_sequential.py:5(merge_sort)
        1    0.000    0.000    0.078    0.078 {built-in method builtins.exec}
   262556    0.026    0.000    0.026    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

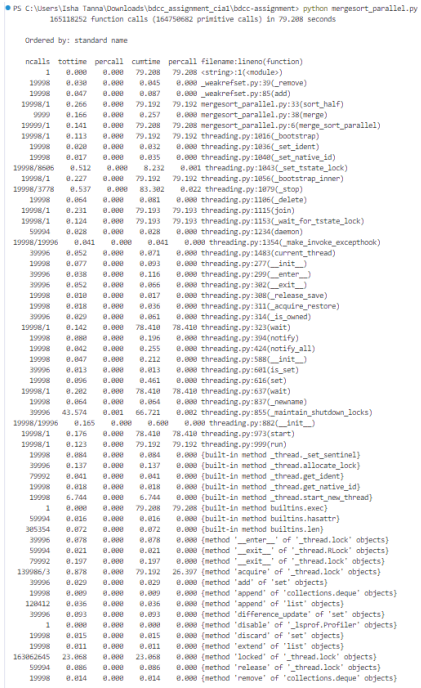
Execution Time: 0.078 seconds.

Total Function Calls: 282,558 function calls (262,560 primitive calls).

Key Insights:

- The main function (merge_sort) is called approximately 20,000 times, and each call performs basic operations like merging and sorting the array.
- A significant portion of the time is spent on simple operations like calling the len() function (which is called 262,556 times).
- The function call stack is fairly shallow, with minimal overhead from other functions like exec and builtins.

Parallelized Merge Sort (Threading)



```
PS C:\Users\Isha Tanna\Downloads\bdcc_assignment_cia1\bdcc-assignment> python mergesort_parallel.py
         165118252 function calls (164750682 primitive calls) in 79.208 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   79.208   79.208 <string>:1(<module>)
    19998    0.030    0.000    0.045    0.000 _weakrefset.py:39(_remove)
    19998    0.047    0.000    0.087    0.000 _weakrefset.py:85(add)
  19998/1    0.266    0.000   79.192   79.192 mergesort_parallel.py:33(sort_half)
     9999    0.166    0.000    0.257    0.000 mergesort_parallel.py:38(merge)
  19999/1    0.141    0.000   79.208   79.208 mergesort_parallel.py:6(merge_sort_parallel)
  19998/1    0.113    0.000   79.192   79.192 threading.py:1016(_bootstrap)
    19998    0.020    0.000    0.032    0.000 threading.py:1036(_set_ident)
    19998    0.017    0.000    0.035    0.000 threading.py:1040(_set_native_id)
19998/8686    0.512    0.000    8.232    0.001 threading.py:1043(_set_tstate_lock)
  19998/1    0.227    0.000   79.192   79.192 threading.py:1056(_bootstrap_inner)
19998/3778    0.537    0.000   83.302    0.022 threading.py:1079(_stop)
    19998    0.064    0.000    0.081    0.000 threading.py:1106(_delete)
  19998/1    0.231    0.000   79.193   79.193 threading.py:1115(join)
  19998/1    0.124    0.000   79.193   79.193 threading.py:1153(_wait_for_tstate_lock)
    59994    0.028    0.000    0.028    0.000 threading.py:1234(daemon)
19998/19996    0.041    0.000    0.041    0.000 threading.py:1354(_make_invoke_excepthook)
    39996    0.052    0.000    0.071    0.000 threading.py:1483(current_thread)
    19998    0.077    0.000    0.093    0.000 threading.py:277(__init__)
    39996    0.038    0.000    0.116    0.000 threading.py:299(__enter__)
    39996    0.052    0.000    0.066    0.000 threading.py:302(__exit__)
    19998    0.010    0.000    0.017    0.000 threading.py:306(_release_save)
    19998    0.018    0.000    0.036    0.000 threading.py:311(_acquire_restore)
    39996    0.029    0.000    0.061    0.000 threading.py:314(_is_owned)
  19998/1    0.142    0.000   78.410   78.410 threading.py:323(wait)
    19998    0.080    0.000    0.196    0.000 threading.py:394(notify)
    19998    0.042    0.000    0.255    0.000 threading.py:424(notify_all)
    19998    0.047    0.000    0.212    0.000 threading.py:588(__init__)
    39996    0.013    0.000    0.013    0.000 threading.py:601(is_set)
    19998    0.056    0.000    0.461    0.000 threading.py:616(set)
  19998/1    0.202    0.000   78.410   78.410 threading.py:637(wait)
    19998    0.064    0.000    0.064    0.000 threading.py:837(_newname)
    39996   43.574    0.001   66.721    0.002 threading.py:855(_maintain_shutdown_locks)
19998/19996    0.165    0.000    0.600    0.000 threading.py:862(__init__)
  19998/1    0.176    0.000   78.410   78.410 threading.py:973(start)
  19998/1    0.123    0.000   79.192   79.192 threading.py:999(run)
    19998    0.084    0.000    0.084    0.000 {built-in method _thread._set_sentinel}
    39996    0.137    0.000    0.137    0.000 {built-in method _thread.allocate_lock}
    79992    0.041    0.000    0.041    0.000 {built-in method _thread.get_ident}
    19998    0.018    0.000    0.018    0.000 {built-in method _thread.get_native_id}
    19998    6.744    0.000    6.744    0.000 {built-in method _thread.start_new_thread}
        1    0.000    0.000   79.208   79.208 {built-in method builtins.exec}
    59994    0.016    0.000    0.016    0.000 {built-in method builtins.hasattr}
   305354    0.072    0.000    0.072    0.000 {built-in method builtins.len}
    39996    0.078    0.000    0.078    0.000 {method '__enter__' of '_thread.lock' objects}
    59994    0.021    0.000    0.021    0.000 {method '__exit__' of '_thread.RLock' objects}
    79992    0.197    0.000    0.197    0.000 {method '__exit__' of '_thread.lock' objects}
 139986/3    0.878    0.000   79.192   26.397 {method 'acquire' of '_thread.lock' objects}
    39996    0.029    0.000    0.029    0.000 {method 'add' of 'set' objects}
    19998    0.009    0.000    0.009    0.000 {method 'append' of 'collections.deque' objects}
   120412    0.036    0.000    0.036    0.000 {method 'append' of 'list' objects}
    39996    0.093    0.000    0.093    0.000 {method 'difference_update' of 'set' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
    19998    0.015    0.000    0.015    0.000 {method 'discard' of 'set' objects}
    19998    0.011    0.000    0.011    0.000 {method 'extend' of 'list' objects}
163852645   23.058    0.000   23.058    0.000 {method 'locked' of '_thread.lock' objects}
    59994    0.086    0.000    0.086    0.000 {method 'release' of '_thread.lock' objects}
    19998    0.014    0.000    0.014    0.000 {method 'remove' of 'collections.deque' objects}
```

Execution Time: 79.208 seconds (much higher than the sequential execution).

Total Function Calls: 165,118,252 function calls (164,750,682 primitive calls).

Key Insights:

- The merge_sort_parallel function is invoked approximately 20,000 times, similar to the sequential version.
- There is a significant amount of time spent in threading-related functions like _bootstrap_inner, join, start_new_thread, and lock methods (acquire, release), with a large number of calls to functions such as threading.py:323(wait).
- The major overhead in the parallel version comes from the thread management system, especially with functions like _set_tstate_lock, _stop, and acquire/release of locks.
- Even though there are many threads running concurrently, the overhead of thread creation, synchronization (locks), and thread joining adds up significantly. This leads to a **higher execution time** than expected, which is a typical challenge when using threads for computationally intensive tasks without proper optimization (e.g., excessive context switching or thread contention).

Analysis

- **Thread Management Overhead:** The parallel execution shows a large number of calls to thread management functions (e.g., start_new_thread, acquire, release, wait). These overheads can sometimes outweigh the benefits of parallelization, especially for smaller arrays or when there are many threads that need to be managed.
- **Lock Contention:** A large portion of time is spent on acquiring and releasing locks, which can become a bottleneck. This issue arises when multiple threads compete for the same resources or data, leading to delays.

While parallel merge sort is theoretically faster for large datasets due to its ability to sort multiple subarrays simultaneously, in practice, the overhead of thread management and synchronization can slow down the execution for smaller datasets or insufficiently optimized parallelization.

Scalability Analysis

Sequential Merge Sort

Time Complexity (Big O Notation):
The time complexity of the **sequential merge sort** algorithm is **O(n log n)**, where:

- n is the number of elements in the array.
- Merge sort divides the array in half repeatedly, and the merging process takes linear time (O(n)).
- The logarithmic term (log n) comes from the repeated division of the array into subarrays.

Scalability Efficiency:

- As n (the array size) increases, the time complexity of merge sort remains **O(n log n)**.
- The sequential implementation is quite efficient for a moderate number of elements (10,000), and the function call stack remains shallow with minimal overhead.

Parallel Merge Sort

Time Complexity (Big O Notation):
The time complexity of parallel merge sort depends on the number of processors (or threads) available:

- **Ideal Case:** If we assume an ideal parallelization with no overhead, the time complexity could approach **O(n log n / p)**, where p is the number of threads.
- **Worst Case:** In practical scenarios, the time complexity might be affected by the overhead of thread management, which includes synchronization, thread creation, and data sharing. In this case, the time complexity could still be O(n log n), but the constant factors would increase due to these overheads.

Scalability Efficiency:

- The overhead of thread management, context switching, and synchronization among threads significantly impacts the efficiency of parallel merge sort, especially for smaller datasets (like the 10,000-element array).

- While parallel merge sort theoretically scales well for larger datasets, the benefits of parallelism are overshadowed by the overhead for smaller arrays.
- For smaller datasets, the overhead from managing threads (thread creation, synchronization, and destruction) may be larger than the actual benefit of parallelizing the sorting algorithm. This is reflected in the profiling data, where the parallel merge sort takes significantly longer than the sequential version for an array of 10,000 elements.

Conclusions

- **For Small Arrays (like 10,000 elements, i.e., the current scenario):** Sequential merge sort is more efficient because the overhead of parallelization (thread management, synchronization) is significant, and the performance gains from parallelization are minimal.
- **For Large Arrays:** As the size of the input array increases, the parallel version of merge sort would become more beneficial. For very large datasets, the ability to divide the problem into smaller tasks and use multiple threads would lead to a significant reduction in execution time, making parallel merge sort more efficient.
- **Threshold for Parallel Efficiency:**

  The parallel version would become faster than the sequential version once the array size is large enough to absorb the overhead from parallelization. This threshold depends on factors like the number of available CPU cores, the specific implementation of the parallel merge sort, and the system architecture. For small datasets, the sequential approach remains the better choice.