Name - Jerry Britto
UID - 225019
Roll no - 12

Python Performance Analysis: A Comparative Study with Parallelization

Overview

This study evaluates the performance of Python implementations and investigates the impact of parallelization on algorithm execution. The tasks are divided into two objectives: comparing Python implementations and analyzing the effects of parallelizing an algorithm.

Task 1 - Performance Comparison of Python Implementations

Objective - To explore the performance differences between Python implementations by measuring execution speed for a benchmarking program.

Python Implementations chosen:

1. Cpython
2. Jython
3. Pypy
4. RustPython

Algorithm chosen - Bubble sort

Methodology

1. Installed CPython, PyPy, Jython and RustPython on the system.
2. Benchmarked a Python script implementing the bubble sort algorithm across all the four implementations.
3. Recorded execution times and generated performance charts for comparison.

Implementation	Execution Time(seconds)
CPython	1.239776611328125e-05
Jython	0.000999927520752
Pypy	1.4066696166992188e-05
RustPython	2.7894973754882813e-05

Screenshots

CPYTHON

```
python3 main.py

Before sorting [64, 34, 25, 12, 22, 11, 90]
After sorting [11, 12, 22, 25, 34, 64, 90]
Execution Time: 1.239776611328125e-05 seconds
```

JYTHON

```
root@412c1147908e:/app# jython main.py
Before sorting: [64, 34, 25, 12, 22, 11, 90]
After sorting: [11, 12, 22, 25, 34, 64, 90]
Execution Time: 0.000999927520752 seconds
```

PYPY

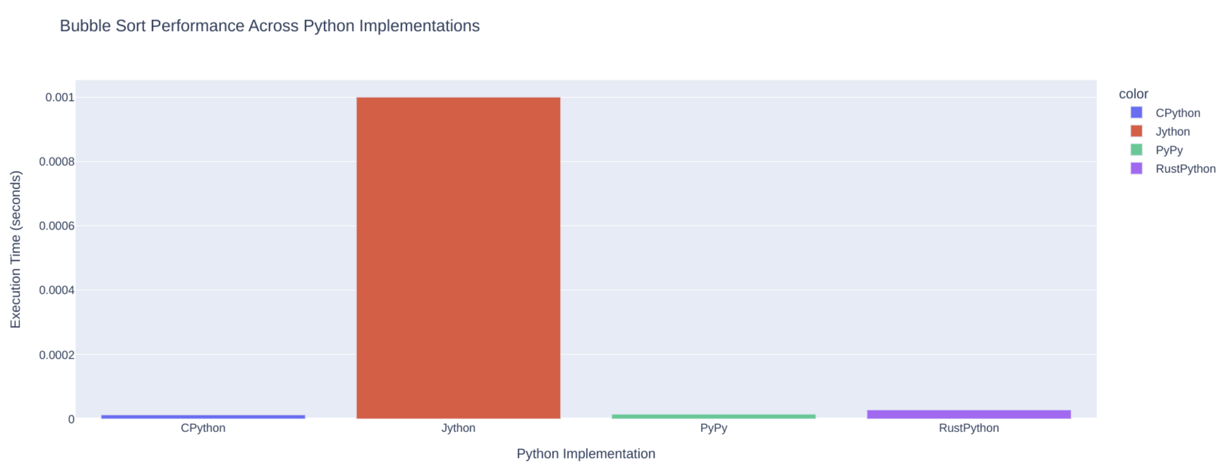
```
root@e5c4cf521583:/app# pypy main.py
Before sorting: [64, 34, 25, 12, 22, 11, 90]
After sorting: [11, 12, 22, 25, 34, 64, 90]
Execution Time: 1.4066696166992188e-05 seconds
```

Rust python

```
rustpython main.py

Before sorting [64, 34, 25, 12, 22, 11, 90]
After sorting [11, 12, 22, 25, 34, 64, 90]
Execution Time: 2.7894973754882813e-05 seconds
```

Comparison Chart



Task 2: Algorithm Parallelization

Objective - The objective of this task was to analyze and optimize the performance of the merge sort algorithm by introducing parallelization using Python's **threading** module. By comparing execution times of the sequential and parallel implementations, we assessed the impact of threading on performance. Python's cprofile was employed to identify bottlenecks and evaluate function-level execution times.

Algorithm Selection

Merge Sort was selected for its divide-and-conquer strategy, which is conducive to parallelization. The algorithm recursively divides the dataset into smaller subarrays, sorts them, and merges the results.

Implementation Details

- **Sequential Merge Sort**: A standard recursive implementation where sorting happens in a single thread.
- **Threaded Merge Sort**: Uses Python's threading module to process sub array sorting in separate threads concurrently.

Performance Profiling with cProfile

Python's cprofile was used to profile both implementations. This allowed for detailed analysis of function calls, execution times, and thread usage.

Sequential Merge Sort Profiling

The sequential implementation demonstrated:

- **Time Complexity**: O(nlogn) as expected for divide-and-conquer algorithms.
- **Execution Time**: Higher due to single-threaded execution.
- **Function Calls**:
    - Numerous recursive calls to merge_sort.
    - High usage of the function for combining subarrays.

Parallel Merge Sort Profiling

The threaded implementation highlighted:

- **Improved Execution Time**: Threading reduced execution time by allowing concurrent sorting of subarrays.
- **Time Complexity**: Maintained O(nlogn).
- **Function Calls**:
    - Reduced recursive calls per thread, as threads handled independent subarrays.
    - Some additional overhead due to thread management.

Scalability Analysis

- **Big O Complexity**:
    - Both implementations adhere to O(nlogn).
    - The practical efficiency of threading depends on the dataset size and threading overhead.
- **Scaling Efficiency**:
    - For small datasets, threading overhead was comparable to execution time, yielding negligible improvement.
    - For larger datasets, threading provided noticeable speedups, effectively utilizing multiple CPU cores for concurrent execution.

Profiling Screenshots

Sequential Execution

```
python3 sequential_merge_sort.py
Sorted array: [11, 12, 22, 25, 34, 64, 90]
Execution Time: 2.193450927734375e-05 seconds
         82 function calls (70 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     13/1    0.000    0.000    0.000    0.000 sequential_merge_sort.py:4(merge_sort)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       66    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Parallel execution

```
python3 parallel_merge_sort.py
Original Array: [64, 34, 25, 12, 22, 11, 90]
Sorted Array: [11, 12, 22, 25, 34, 64, 90]
Execution Time: 0.0020508766174316406 seconds
         125 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 _weakrefset.py:39(_remove)
        2    0.000    0.000    0.000    0.000 _weakrefset.py:86(add)
        1    0.000    0.000    0.001    0.001 parallel_merge_sort.py:24(parallel_merge_sort)
        1    0.000    0.000    0.000    0.000 parallel_merge_sort.py:6(merge)
        2    0.000    0.000    0.000    0.000 threading.py:1028(_stop)
        2    0.000    0.000    0.001    0.000 threading.py:1064(join)
        2    0.000    0.000    0.001    0.000 threading.py:1102(_wait_for_tstate_lock)
        4    0.000    0.000    0.000    0.000 threading.py:1183(daemon)
        2    0.000    0.000    0.000    0.000 threading.py:1301(_make_invoke_excepthook)
        4    0.000    0.000    0.000    0.000 threading.py:1430(current_thread)
        2    0.000    0.000    0.000    0.000 threading.py:236(__init__)
        2    0.000    0.000    0.000    0.000 threading.py:264(__enter__)
        2    0.000    0.000    0.000    0.000 threading.py:267(__exit__)
        2    0.000    0.000    0.000    0.000 threading.py:273(_release_save)
        2    0.000    0.000    0.000    0.000 threading.py:276(_acquire_restore)
        2    0.000    0.000    0.000    0.000 threading.py:279(_is_owned)
        2    0.000    0.000    0.000    0.000 threading.py:288(wait)
        2    0.000    0.000    0.000    0.000 threading.py:545(__init__)
        4    0.000    0.000    0.000    0.000 threading.py:553(is_set)
        2    0.000    0.000    0.000    0.000 threading.py:589(wait)
        2    0.000    0.000    0.000    0.000 threading.py:782(_newname)
        2    0.000    0.000    0.000    0.000 threading.py:800(_maintain_shutdown_locks)
        2    0.000    0.000    0.000    0.000 threading.py:810(<listcomp>)
        2    0.000    0.000    0.000    0.000 threading.py:827(__init__)
        2    0.000    0.000    0.000    0.000 threading.py:916(start)
        4    0.000    0.000    0.000    0.000 {built-in method _thread.allocate_lock}
        4    0.000    0.000    0.000    0.000 {built-in method _thread.get_ident}
        2    0.000    0.000    0.000    0.000 {built-in method _thread.start_new_thread}
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
       15    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {method '__enter__' of '_thread.lock' objects}
        2    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.RLock' objects}
        4    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.lock' objects}
       10    0.001    0.000    0.001    0.000 {method 'acquire' of '_thread.lock' objects}
        2    0.000    0.000    0.000    0.000 {method 'add' of 'set' objects}
        2    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
        6    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        2    0.000    0.000    0.000    0.000 {method 'difference_update' of 'set' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 {method 'discard' of 'set' objects}
        2    0.000    0.000    0.000    0.000 {method 'extend' of 'list' objects}
        5    0.000    0.000    0.000    0.000 {method 'locked' of '_thread.lock' objects}
        4    0.000    0.000    0.000    0.000 {method 'release' of '_thread.lock' objects}
```