Xinlei Huang
CS421 Programming Languages
August 1, 2023

Experiments derived from Concurrent Object-Oriented Programming

Overview

This report focuses on implementing two patterns of concurrency programming mentioned in *Concurrent Object-Oriented Programming (Gul Agha Communications of the ACM, Vol 33(9), 1990)*. Unit tests and performance tests are executed to verify the logic integrity and execution efficiency improvement. Theoretically, algorithms written in multi-processing pattern should be more efficient compared to their sequential counterparties.

Implementation

The paper mentions three concurrency programming patterns, pipeline, divide and conquer, and cooperative problem-solving, and illustrates concrete examples for first two. If not too oversimplified, concurrency is beneficial because a task can be distributed to multiple workers for almost parallel execution, however, the distribution process generates overheads when managing child processes and

sharable memories. Multi-processing algorithms may not out perform vanilla single processor on tasks requiring light computations. Sieve of Eratosthenes and Merge Sort are implemented sequentially and concurrently to evaluate their running time, given the conditions

- Maintain same time complexity
- Algorithms are consistent on functionality

Merge Sort

As described on *Concurrency Object-Oriented Programming, page 17,* one classic sorting algorithm is dividing the list into two with roughly half the size, then sorting two half-sized lists separately and merging them. Easiest implementation is recursion,

```
Func merge_sort:
        left <- merge_sort left_half
        right <- merge_sort right_half
        merge left right
```

However, recursion is not the most suitable algorithm for our experiment because the goal is to evaluate concurrency performance. Recursion will ask a spawned child process to spawn its grandchildren which is an anti-pattern from python multiprocessing perspective. A bottom-up iterative method can avoid the nested daemon issue.

```
Func merge_sort_bottom_up nums:
        buckets <- [[x] for x in nums]
        while length(buckets) > 1:
            for I = 1; I < length(buckets); I++:
                merge_inplace buckets[I] buckets[I+1]
```

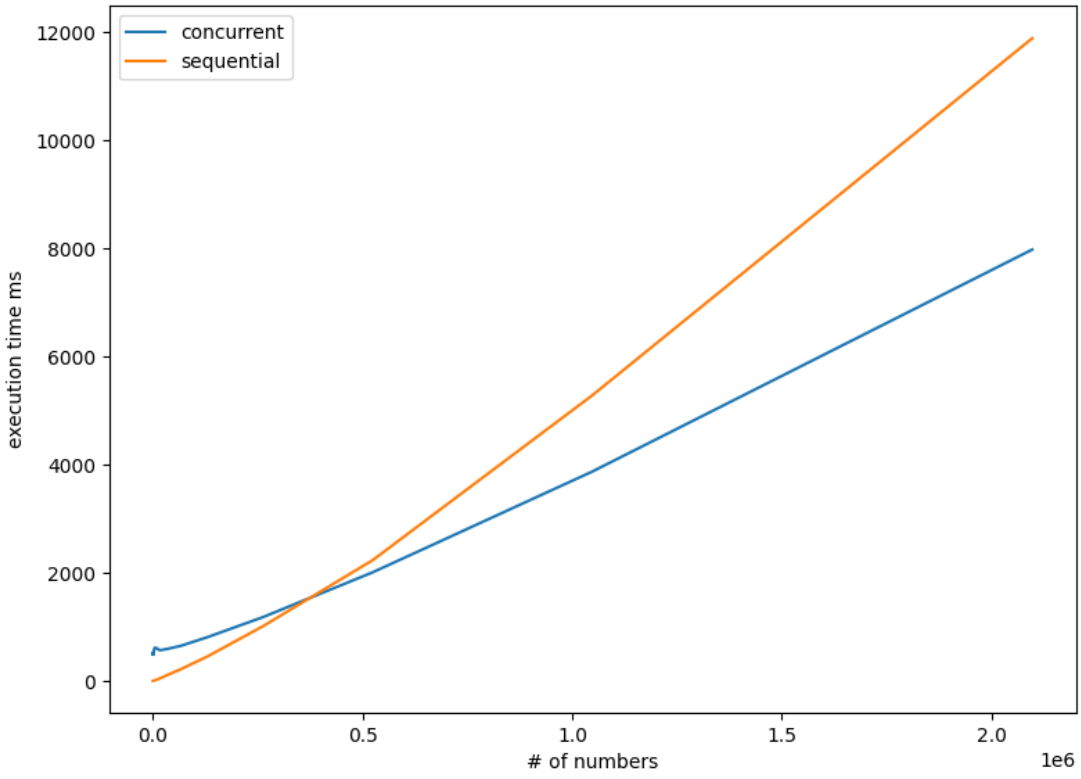
Sequential

Sequential algorithm simply implements the pseudo code above in python.

Concurrent

Concurrent algorithm distributes merging tasks into multiple processors so that they can be executed simultaneously.

Execution Time Comparison



The chart demonstrates that concurrent implementation won't outperform sequential one on small lists. As the size of list increases, sequential implementation consumes more time to sort the list, and the advantage of multi-processing becomes evident incrementally. Observation here testifies our hypothesis. Overheads from

coordinating workers slows down the multi-processor algorithm, but time saved by running tasks in parallel pushes concurrency implementation back to the lead.

Code

- Sequential algorithm: https://github.com/xinleihuang/cs421-final-project/blob/main/src/merge_sort/sequential.py
- Concurrency algorithm: https://github.com/xinleihuang/cs421-final-project/blob/main/src/merge_sort/divide_conquer.py

Sieve of Eratosthenes

The problem itself is less intimidatory than its name, *find all primes not greater than a upper limit*. One common solution is to generate composites from a prime, then filter out all composites. What left are prime numbers.

Sequential

Solve the problem with two for loops. The task is executed in a single processor, thus memory is sharable across iterations. It will mitigate redundant work to improve efficiency.

Concurrent

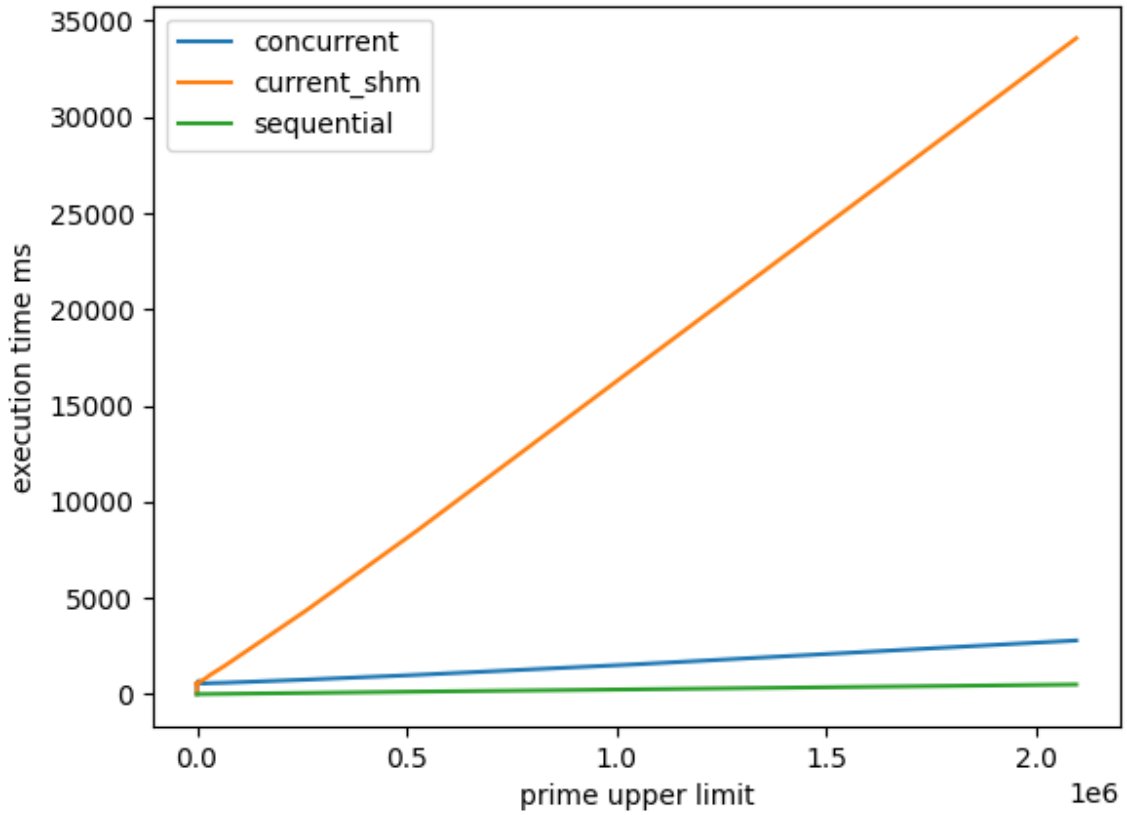
There are two concurrent implementations. Both of them stick to the pipeline concurrency pattern, which splits the task into multiple stages and executes independent tasks simultaneously at the same stage. The difference is whether workers share memory at runtime. Two stages for solving the prime sieve,

1. Find the first n primes, where n is the number of workers available
2. For each prime, derive composites from it and save composites in a list

A. No shareable memory: each worker creates a new list to store the exploration results

B. With shareable memory: all workers modify the same list

Execution Time Comparison



Sequential algorithm is significantly more efficient than concurrent ones, which rejects the hypothesis that multiprocessing algorithm is less time consuming. One potential reason is workers at Stage 2 are doing duplicated computations because running tasks in parallel loses the capacity of deriving composites from prime seeds in

ascending order. Additionally sharable memory slows down the concurrency because of extra burden on worker coordinations to avoid memory collison.

An alternative Stage 2 is

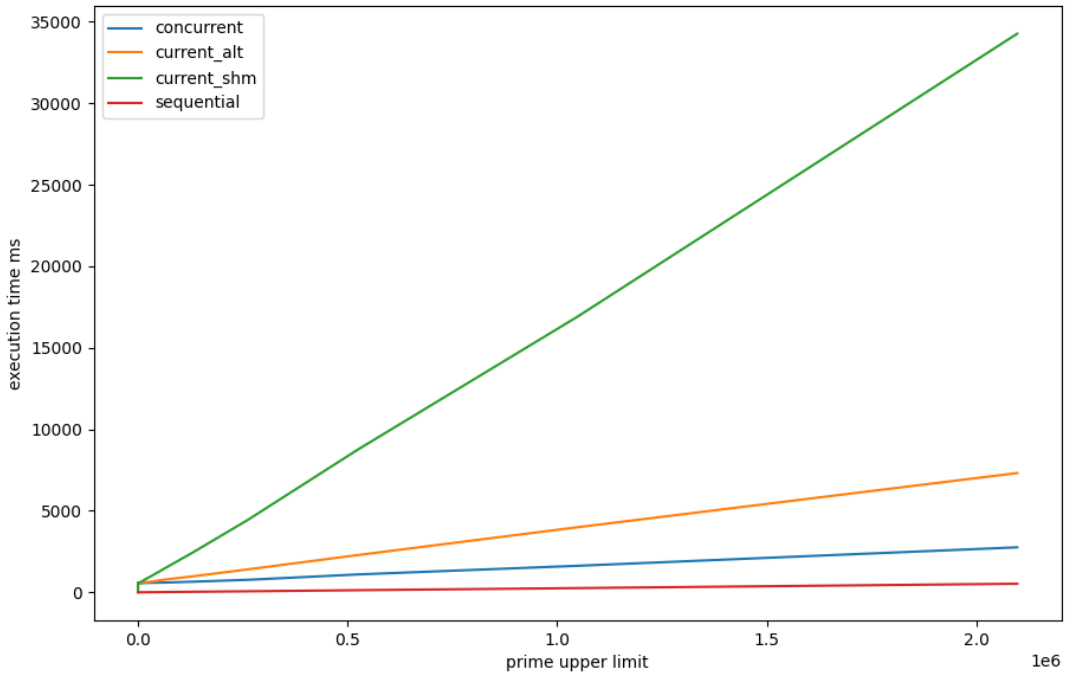
Func stage_2:

for seed in prime_seeds:

segments <- split [seed*seed, upper_limit] to m segments

parallel(derive_composites for segment in segments)

Pervious concurrency algorithm computes composites from multiple prime seeds in parallel, while the alternative version moves the parallelism one level down. It processes prime seeds in ascending order, but splits the interval [prime_seed*prime_seed, upper_limit] into multiple segments which can be executed simultaneously to mitigate redundant computations.



The alternative solution gets rid of shared memory, and mitigate duplicated computations to match the time complexity of the sequential algorithm, but its performance is continuously less optimistic than the original concurrent version. Reasons are uncertain up to this point, some further investigations could be carried out,

- Switch to a different programming language such as C++ and golang
  - Python is a single threaded language which may has negative impacts on concurrency algorithms
- Try a different algorithm such as Euler's Sieve in which each composite number is eliminated exactly once. *(A linear sieve algorithm for finding prime numbers, Gries & Misra, 1978)*

Code

- *Sequential:* https://github.com/xinleihuang/cs421-final-project/blob/main/src/prime_sieve/sequential.py
- *Pipeline:* https://github.com/xinleihuang/cs421-final-project/blob/main/src/prime_sieve/pipeline.py
- *Pipeline with shared memory:* https://github.com/xinleihuang/cs421-final-project/blob/main/src/prime_sieve/pipeline_shm.py
- *Alternative Pipeline without shared memory:* https://github.com/xinleihuang/cs421-final-project/blob/main/src/prime_sieve/pipeline_alten.py

Tests

Unit tests are written to verify concurrency and sequential have the same functionalities.

- https://github.com/xinleihuang/cs421-final-project/tree/main/src/merge_sort/__tests__
- https://github.com/xinleihuang/cs421-final-project/tree/main/src/prime_sieve/__tests__

Performance tests are written to collect running time of different algorithms for analysis

- https://github.com/xinleihuang/cs421-final-project/blob/main/src/performance/execution_time.py

Appendix

All code and a copy of this report can be found in GitHub. There are demonstrations about how to set up the code and run algorithms locally in Readme.