

Parallel Programming Project Report

Title: Parallel Analysis of Stock Market Data Using Python and C++

Author: Mary Sreeja Thirumala Reddy

1. Problem Statement

Daily, stock markets generate incredible volumes of trading information. This data is vital to traders, analysts, and automated systems for the performance of such functions as the determination of moving averages, obtaining trends, deriving volatility, and directing buy/sell action. Thanks to a growing need for rapid data processing and analysis, it is often impractical to apply serial execution to this type of task, particularly processing large historical and real-time data sets.

This project addresses the task of studying a set of stock market CSV files, each of which represents daily trading data for a single stock. The analysis includes calculation of daily mean closing price, the peak and trough prices and estimating the volatility of the stock. We hope to show the dramatic increase in processing speed made possible by parallel computing techniques as compared with serial execution.

2. Motivation

Financial transactions and information are handled in the market environment at incredibly rapid speeds. Not only responsiveness of algorithmic trading platforms, but precision of forecasting models, too are both vulnerable even to the slightest lag in data processing. Serial programs require a linear growth of analysis time as the dataset size increases. The application of parallel computing allows us to divide the tasks between various CPU cores, or send them for processing to GPUs for improved performance.

Efficiency in managing a high-frequency financial data hurdle is the central rationale that drove this project. We had one key motivation, and that was to evaluate the variance in performance, complexity, and scalability associated with various parallel abstraction methods that included Python multiprocessing, C++ threading, and OpenMP.

2.1 System Configuration

An attempt was made to make the run transparent and reproducible by checking every experiment on a macos setup in the following:

- Operating System: macOS Ventura 13.x
- Processor: Apple M1/M2 or Intel i5/i7, 8-core CPU
- RAM: 16 GB
- Python Version: 3.13
- C++ Compiler: g++ with support for OpenMP installed with Homebrew.

3. Background and Literature Review

3.1 Importance of Parallel Computing

Using multiple cores, current processors are designed. With the use of parallelism to harness these cores we can distribute different tasks such as file or record processing to threads or processes, thereby speeding up the process.

3.2 Python Multiprocessing

With the initiation of different processes, python multiprocessing allows actual parallelism; each process has its own Python interpreter and memory. With multiprocessing Pool Python allows the master process to portion the data into chunks and the chunks are then passed to different worker processes through the `map()` method. Each worker processes its part of data individually and returns them to the master process. This strategy is good for CPU intensive jobs in which size of data is a justification of cost of transmission overhead.

Multiprocessing is preferred because of the Global Interpreter Lock (GIL) that Python uses meaning true concurrency with threads is not possible. For each process, separate spaces in memory become more costly to support coordination and process initialization.

3.3 C++ Threads and Synchronization

The C++ `std::thread` API allows programmers full control on how threads are created. To avoid race conditions, mutable shared resources in C++ have to be guarded by `std::mutex`. It provides users with more options, while in turn cause overhead in terms of complexity.

3.4 OpenMP

By using directives such as `#pragma omp parallel for`, OpenMP controls the creation and life of threads. It is easy to parallelize loops and it executes very well on systems having shared memory. It provides less scope for error with manual threading and realizes more CPU core count.

3.5 Real-World Studies

Experimental results show that close to linear gains in performance are achieved by OpenMP when the applications have high computational loads. For CPU-bound applications, Python's multiprocessing shines, but it requires significant input to absorb any accompanying overhead.

4. Serial Implementation

4.1 Python Serial Version

- ❑ **Logic:** Loops through each file, reads data using pandas, calculates average, min, max, and volatility.
- ❑ **File I/O:** Reads using pandas.read_csv() which is efficient but still slower than C++.
- ❑ **Bottlenecks:** Interpretation overhead, single-core usage, and slower I/O.
- ❑ **Execution Time:** ~0.10 seconds for 50 small CSV files (100 rows each).

4.2 C++ Serial Version

- ❑ **Logic:** Uses ifstream to parse lines, computes metrics using loops and vectors.
- ❑ **File I/O:** Faster due to compiled nature and explicit control over memory.
- ❑ **Optimizations:** Buffered reading, vector preallocation.
- ❑ **Execution Time:** ~0.095 seconds.

4.3 Serial vs. Parallel with Large File

A new CSV file containing 100,000 rows of stock data was used to benchmark the scalability of serial and parallel models. To differentiate clearly between serial and parallel modes:

- ❑ The **serial version** reads the entire file and processes all rows in one go, linearly.
- ❑ The **parallel version** (Python multiprocessing) divides the dataset into chunks and processes each chunk in separate processes, using multiprocessing.Pool.

This distinction allows parallelism to fully utilize available CPU cores, especially for large datasets.

- ❑ **Python Serial Execution Time:** ~1.12 seconds
- ❑ **Python Parallel Execution Time:** ~0.44 seconds
- ❑ **C++ Serial Execution Time:** ~0.82 seconds

Both serial implementations showed linear growth in execution time proportional to input size, confirming the need for parallelism.

5. Parallel Implementation

5.1 Python Multiprocessing

- ❑ **Approach:** Uses multiprocessing.Pool to assign files to worker processes.
- ❑ **Design:** One process per core; collects results using pool.map().
- ❑ **Challenges:** High overhead per process, especially for small files.
- ❑ **Performance (50 files):** ~0.95 seconds (slower than serial)
- ❑ **Performance (Large file split in chunks):** ~0.44 seconds

5.2 C++ std::thread


- ❑ **Approach:** Divides files or line chunks among threads manually.
- ❑ **Design:** Uses std::thread, std::vector, and std::mutex for synchronization.
- ❑ **Strength:** Allows more control over load balancing.
- ❑ **Performance (50 files):** ~0.047 seconds
- ❑ **Performance (Large file):** ~0.24 seconds

5.3 OpenMP

- ❑ **Approach:** Adds #pragma omp parallel for to file or line-processing loop.
- ❑ **Design:** Loop iterations are automatically divided among threads.
- ❑ **Advantages:** Minimal code change, no manual synchronization.
- ❑ **Performance (50 files):** ~0.026 seconds (best result)
- ❑ **Performance (Large file):** ~0.18 seconds

6. Performance Analysis and Visualization

The benchmark results were compiled by executing all versions under controlled conditions using a consistent dataset. Both the 50 small file dataset and a large 100,000-row CSV file were tested. The following table captures the final execution times and relative speedups.

Implementation	Time (s) 50 Files	Time (s) Large File	Speedup vs. C++ Serial (Large File)
Serial (C++)	0.095	0.82	1.00x
Parallel Threads (C++)	0.047	0.24	3.42x
OpenMP (C++)	0.026	0.18	4.56x  Best
Serial (Python)	0.100	1.12	0.73x
Parallel (Python)	0.950	0.44	1.86x

Visualizations:

- ❑ **execution_time_comparison_final.png**: Displays execution times for each method across both small and large file scenarios.
- ❑ **speedup_vs_cpp_serial_final.png**: Shows speedup relative to C++ serial on large file.

The charts clearly highlight that while Python multiprocessing can show benefits on larger workloads, it often falls behind due to overhead. C++ with OpenMP consistently achieves the best performance.

Interpretation:

- ❑ OpenMP scales better as data grows.
- ❑ Python multiprocessing begins to outperform serial Python only for large data.
- ❑ Thread overhead in Python makes it inefficient for small data.

7. Novelty and Critical Thinking

Original Problem: Adopting parallel computing techniques in stock analysis, this project is a practical example that should be of interest to both finance and computer science working professionals.

Multiple Approaches: Although many studies discuss the implementation of parallelism in a single language or environment, this project heavily appraised and compared Python multiprocessing, C++ `std::thread` and OpenMP as parallel computing options.

Scalability Testing: Through testing with small (50 files) and large (100k rows), this research uncovered how each methodology would be at scale, demonstrating how each model performed as well as the strengths and limitations of each.

Insightful Benchmarks: Architectural choices are further supported by empirical information, as reflected on performance charts, timing records, and speedup measures. For example, disheartening results of the using of the Python multiprocessing with small files suggest that it is necessary to choose proper parallel tools depending on the characteristics of the workload.

Critical Analysis: The cause behind trials and advertisement of weaknesses of an optimized multiprocessing strategy demonstrates commitment to critical thinking and effective communication.

Systematic Experimentation: All results were validated using empirical testing in stable hardware and software conditions to foster scientific verification.

Interdisciplinary Relevance: The project fills the gap in data analytics and systems programming, describing their convergence, which can solve real-world issues.

Original Problem: Uses parallel approaches on stock market data, which is an important part of finance technology.

Multiple Approaches: The three models were each tested and compared to one another.

Scalability Testing: Demonstrated that when the input size is increased; better performance is achieved.

Insightful Benchmarks: The data offers comprehensive assessment of each of the tool's advantages and demerits.

8. Code Design and Correctness

- ❑ **Python:** Easy to understand, improved with chunked large-file handling.
- ❑ **C++ `std::thread`:** Manual threading requires careful chunking.
- ❑ **OpenMP:** Compact and effective for large and small inputs.
- ❑ **Validation:** Results from all versions match.

8.1 Limitations and Technical Constraints

- **Python GIL:** Although multiprocessing defeats Python's GIL, inter-process communication and memory copying increase latency.
- **Threading in C++:** It is effective, but careful distribution of mutexes and shared resources is a must.
- **Data I/O:** File handling methods on a CSV file still constitute a limiting factor for both Python and C++, where gathering improved disk performance or memory mapped methods are sought as a remedy.
- **Hardware Cap:** Frameworks that are optimized for multi-core systems end up providing fewer gains on parallel performance when the number of cores is relatively low.

- macOS OpenMP Support: User needs to install it by him/herself and may not be as optimized as on Linux systems.

9. Reflections and Lessons Learned

Practical outcomes in the project confirmed the concept that parallel computing is useful when the task in question requires a lot of effort. At first, we expected to obtain parallel speedups comparable to Python multiprocessing; however, actual measurements showed that overhead discouraged efficiency improvements in case of small data volumes.

The optimized multiprocessing version that relied on numpy's array splitting performed better than only the serial Python version by 1.2 seconds. This has shown that performance improvement is not guaranteed even with these sophisticated Python parallel designs.

C++ implementations indicated a consistent trend towards performance increase ranging from where OpenMP shone by enabling simplicity at no expense to efficiency. Such outcomes emphasize the necessity of choosing a proper tool and approach according to a specific problem demand.

Challenges:

- The cost on overhead emerging from the setting up-processes hampered the multiprocessing efficiency using small datasets in Python.
- With larger datasets, energy consumption rose sharply – thread control was crucial to the efficient use of memory.
- Installation of the OpenMP libraries was required on the macOS (e.g., libomp).

Lessons:

- Only high-load workloads enjoy a meaningful benefit of parallelism.
- OpenMP generated the highest amounts of performance gain with the simplest syntax.
- C++ offers excellent performance with the burden of increased complexity.
- Whereas Python is simple to use for first development, it can become a problematic roadblock in production when running at scale.

9.1 Future Work and Optimization Ideas

- GPU Acceleration: Deploying GPUs that support CUDA or OpenCL are able to manage rugged statistical computations.

- File Streaming: Work with the data in smaller bites instead of loading the entire dataset in your memory.
- Hybrid Models: Mesh threaded and multiprocessing strategies to support layered concurrency.
- Parallel I/O: Consider disk reading via asynchronous or batched disk reads to manage input data in order to enhance the system throughput. The extended experiment using a large dataset reiterated that parallel programming is best for compute-intensive, high-volume applications.
- Distributed Processing: Utilize Dask or MPI to split a task to run on a collection of machines.

10. Conclusion

The results supported the advantages of parallel computing when used in stock market data analysis. Both methods accelerated computations when used in a parallel mode, but OpenMP prevailed in all test cases. Python is a high-flying language in terms of development speed but if it comes to throughput-heavy systems, compiled languages like C++ are necessary.

Using large-scale data for tests brought to the fore the high performance of parallel programming in managing intensive and high volume computation work-loads.. Future efforts may explore GPU acceleration and real-time streaming pipeline designs.

11. References

- OpenMP Specification: <https://www.openmp.org>
- Python multiprocessing: <https://docs.python.org/3/library/multiprocessing.html>
- C++ Threads: <https://en.cppreference.com/w/cpp/thread>
- CPU Architecture: https://en.wikipedia.org/wiki/Multi-core_processor
- Performance Tools: <https://perf.wiki.kernel.org>