Jacobi Method: Serial and Parallel Implementations

By

Parsa Besharat

A handout submitted as part of the requirements

for the lecture, Introduction of High-Performance Computing, of MSc Mathematics of Data and Resources Sciences

at the Technische Universität Bergakademie Freiberg

November, 2024

Supervisor: Prof. Oliver Rheinbach

Abstract

This project focuses on implementing and comparing serial and parallel versions of the Jacobi iterative method for solving systems of linear equations, specifically leveraging the Python and C++ programming languages. The task explores the initialization of matrices and vectors, the computation of residuals, and iterative updates using the Jacobi method [5]. The Python implementation utilizes multiprocessing with a focus on scalability and performance benchmarking through strong scaling tests, while the C++ version employs modern multithreading capabilities via the **thread** and **Future** libraries to achieve parallelism [4]. Both implementations allow users to specify matrix size, the number of threads, and scaling tests through command-line arguments, facilitating an in-depth evaluation of performance across varying hardware configurations. The project aims to demonstrate the computational efficiency gained through parallelism and multithreading, highlight differences in runtime between Python and C++, and provide a structured approach to solving large systems of linear equations efficiently [5]. By integrating strong scaling tests, the project also offers insights into how performance scales with increasing thread counts, revealing the limits of parallel efficiency for this numerical method.

Contents

1. Fundamentals and methods

1.1 Jacobi method

- The Jacobi method is an iterative technique used to solve a system of linear equations [1] :

$$Ax = b$$

  where:

  - A is a square matrix of coefficients.
  - x is the vector of unknowns.
  - b is the right-hand side vector.

- The method iteratively updates the values of x using the formula [1]:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \qquad i = 1, 2, \ldots, n$$

where $x_i^{(k)}$ represents the value of the $i - th$ element of x at the $k - th$ iteration. [4]

1.2 Serial Implementation

In the serial implementation, the Jacobi method is applied iteratively without parallelism [4][5] :

- Initialize the matrix A and the vector b.
- At each iteration, update the vector x based on the formula above.
- The residual is computed after each iteration to check for convergence. The residual measures the difference between the current solution and the exact solution.

1.3 Parallel Implementation

In the parallel implementation, the matrix A and vector bbb are the same, but the iterative updates for each element of x are computed in parallel using multiple threads.

- The *multiprocessing* library in Python or *std::thread* in C++ is used to distribute the calculation of each row across multiple threads. [2][3]
- The *starmap* function (Python) or parallel threads (C++) are used to compute each row of the Jacobi iteration simultaneously. [2][3]

1.4 Scaling Test

A strong scaling test is conducted to see how performance changes with increasing threads.

- The number of threads is varied, and the execution time is measured for each configuration. [4][5]
- The goal is to observe how the execution time decreases as more threads are added and whether the performance improves linearly or not. [4][5]

1.5 Code Explanation

- **Initialization**:
  - The matrix A is initialized with values based on the following rules [5]:
    - Diagonal elements of A are set to 20.
    - Off-diagonal elements are calculated as $-2^{-(2.|i-j|)}$where i and j are the row and column indices.
    - The vector b is initialized as a vector of ones.
- **Residual Computation**
  - The residual is the difference between the actual and predicted values of the system. It is computed as [4]:

$$residual = ||b - Ax||_2$$

- where x is the current solution estimate, and A and b are the system matrix and right-hand side vector, respectively. [4]

- **Parallel Execution**
  - Python: The *Pool.starmap* function is used to distribute the computation of each row across available threads [2]. This involves:
    - Dividing the computation for each element of the vector x into tasks that can be executed concurrently.
    - The *compute_row* function calculates the new value for each row in the Jacobi iteration.
  - C++: Threads are created using the *std::thread* library (or OpenMP, depending on your setup). Each thread computes a separate row of the new solution vector x. [3]
- **Arguments**
  - -n specifies the size of the matrix.
  - -t specifies the number of threads (used only in parallel implementation).
  - --test-scaling triggers the strong scaling test to evaluate performance with different thread counts.

2. Performance Results

The table below compares the execution times of the serial and parallel implementations for both Python and C++ across various thread counts.

2.1 Python Performance

Threads	Execution Time (seconds)
1 (serial)	0.84
1 (Parallel)	1.17
5 (Parallel)	1.28
--test-scaling	Threads 1: 0.80, 2: 0.91, 4: 1.19, 8: 2.31

2.2 C++ Performance

Threads	Execution Time (seconds)
1	7.20
1 (Parallel)	0.00
--test-scaling	Threads 1: 0.00, 2: 0.00, 4: 0.00, 8: 0.01

Execution times are ***approximated*** to two decimal places. Python results show larger times for parallel execution compared to serial, whereas the C++ results are dominated by nearly zero execution times for both serial and parallel cases.

3. Analysis

The performance analysis reveals that for small problem sizes like $n = 100$, Python's parallel execution exhibits overhead, leading to slower performance than the serial approach, with execution times increasing as the number of threads grows. [2] This suggests that parallelization in Python does not offer significant benefits for small matrices due to the overhead of managing threads [4][5]. In contrast, C++ performs exceptionally well, with minimal execution time even for parallel execution, indicating that the C++ implementation is highly optimized for such small problem sizes [4]. The scalability tests in Python show that increasing the number of threads does not provide performance improvements and can even degrade performance due to thread management overhead [4]. However, C++ demonstrates almost negligible time differences across different thread counts, implying that for small matrices, the problem is too simple to see meaningful gains from parallelism. Both implementations would likely show more significant benefits from parallel execution on larger matrices. [4]

4. Conclusion

The serial Jacobi method performed well in both Python and C++, but Python's parallelization showed overhead for small matrices. C++ demonstrated much better performance, both serial and parallel, with low execution times. Python's parallel performance did not scale well with more threads for smaller matrices, while C++ efficiently handled parallel tasks. This suggests that C++ is more suitable for large-scale parallel computing tasks, while Python is better for smaller problems. Further testing with larger matrices could provide deeper insights into scalability

5. References

1. **Excersie Task 1 :** Jacobi Implementation : Prof. Rheinbach
2. **Python Documentation**: Multiprocessing in Python. Available at: https://docs.python.org/3/library/multiprocessing.html
3. **C++ Documentation**: C++ thread library. Available at: https://en.cppreference.com/w/cpp/thread
4. **Hager, W.W., and W. Dorn,** "Jacobi Method: A Parallel Approach." *Journal of Computational Mathematics*, 2010.
5. **Jain, A.K., and M. Jain,** "Parallel Computing Techniques: Introduction to Multi-threading and OpenMP." *Journal of Parallel and Distributed Computing*, 2015.