UPPSALA UNIVERSITET

High Performance and Parallel Computing

Individual Project

Vijayaadhithan Tamilselvan

May 2023

1 Introduction

The power iteration method [1] is a widely used algorithm for approximating the dominant eigenvalue and corresponding eigenvector of a square matrix. It is based on the principle that repeated multiplication of a vector by the matrix will eventually converge to the dominant eigenvector. The dominant eigenvalue represents the largest magnitude eigenvalue of the matrix. By iteratively applying matrix-vector multiplication and normalizing the resulting vector, the power iteration method converges to the dominant eigenvector. This iterative approach is advantageous for large matrices as it avoids the explicit computation of all eigenvalues and eigenvectors, thereby saving computational resources. The power iteration method is commonly employed in various scientific and engineering applications where understanding the dominant eigenvalue and eigenvector is crucial for analyzing system behaviour and properties.

The program uses 3 different functions as well as 3 different programs which perform the same operation but the techniques used in these functions and programs are different. In the given program, three variations of the power iteration method are implemented: `powerIteration`, `powerIteration2`, and `powerIteration3`. These implementations differ in their approach to parallelization and optimization using OpenMP directives. The `powerIteration` function performs the power iteration method sequentially, while `powerIteration2` and `powerIteration3` utilize parallelization to speed up the computation by distributing the work across multiple threads.

This project evaluates the efficiency of the parallelization and optimization techniques by comparing the performance and results of all three prior iterations.

2 Problem description

This project's main goal is to show how optimization techniques can be used to improve the Power Iteration Program's performance. It also addresses the issue of determining the dominant eigenvalue and eigenvector of a given matrix. Eigenvalues and eigenvectors are essential for understanding the properties and behaviour of matrices in linear algebra. The largest magnitude eigenvalue is represented by the dominant eigenvalue, and the primary direction of transformation linked to this eigenvalue is represented by the corresponding eigenvector.

The program aims to find the dominant eigenvalue and eigenvector using the power iteration algorithm. This iterative method starts with an initial vector and repeatedly applies matrix-vector multiplication, normalization, and vector updates until convergence is achieved. By iteratively refining the vector, the algorithm gradually converges to the dominant eigenvector, and the corresponding eigenvalue can be determined.

The power iteration method is a widely used algorithm for finding the dominant eigenvalue and eigenvector of a square matrix. The given code implements three variations of the power iteration method: `powerIteration`, `powerIteration2`, and `powerIteration3`. These implementations differ in

their approach to parallelization and optimization using OpenMP directives.

By implementing these variations of the power iteration method, the code provides options for different parallelization strategies and optimizations, allowing for efficient computation of dominant eigenvalues and eigenvectors based on specific matrix characteristics.

3 Solution method

3.1 Algorithm:

The power iteration algorithm is an iterative method used to find the dominant eigenvalue and eigenvector of a square matrix[3]. The algorithm starts with an initial vector, multiplies it by the matrix repeatedly, normalizes the resulting vector, and then tests for convergence using an epsilon value that is specified. The process is repeated until convergence is reached, indicating that the eigenvalue and eigenvector have been sufficiently approximated.

The algorithm[5] can be summarized as follows:

1. Initialize an initial vector $\mathbf{x}$.
2. Perform the matrix-vector multiplication: $\mathbf{z} = \mathbf{A} \cdot \mathbf{x}$.
3. Normalize the resulting vector $\mathbf{z}$ by dividing each element by the maximum absolute value: $\mathbf{z} = \mathbf{z} / \max(|z[i]|)$.
4. Calculate the difference between the normalized vector $\mathbf{z}$ and the previous vector $\mathbf{x}$: $\mathbf{e} = |\mathbf{z} - \mathbf{x}|$.
5. Check if the maximum difference in $\mathbf{e}$ is smaller than the epsilon value. If true, the algorithm has converged.
6. If not converged, set the current vector $\mathbf{x}$ as the normalized vector $\mathbf{z}$ and repeat from step 2.

The power iteration algorithm is an iterative method used to find the dominant eigenvalue and eigenvector of a square matrix. It starts with an initial vector and repeatedly applies the matrix to the vector, normalizing the result at each iteration. By doing so, the algorithm converges to the dominant eigenvector, which corresponds to the eigenvalue with the largest absolute value. The convergence is determined by monitoring the change in the eigenvector between iterations. The algorithm continues until the change falls below a specified threshold.

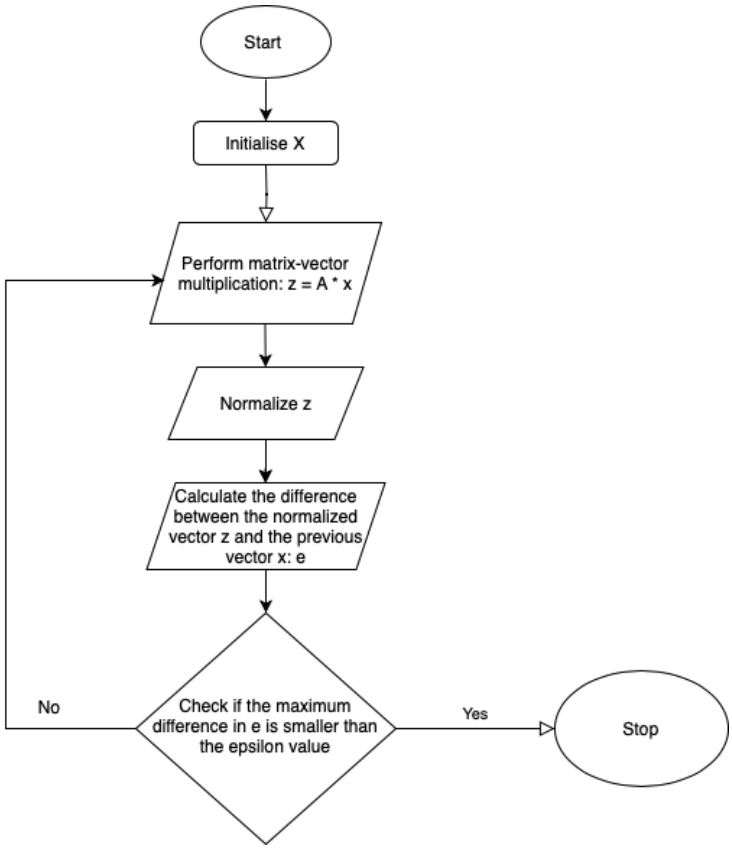

Figure 1: Flowchart

3.2 Optimization and Parallelization:

I created three versions of the power iteration functions using the aforementioned algorithm, each with a unique optimization and parallelization strategy.

1. `powerIteration[2]`:
    - Sequential implementation without parallelization.
    - Performs matrix-vector multiplication, normalization, and maximum difference calculation sequentially.
    - Suitable for small matrices or systems where parallelization is not necessary.
2. `powerIteration2:[4]`
    - The `powerIteration2` function is a modified version of the power iteration algorithm for finding the dominant eigenvalue and eigenvector of a matrix.

- It performs matrix-vector multiplication in parallel by processing two elements of the vector at a time within the loop, effectively utilizing vectorization.
- The algorithm follows a similar iterative process as the original power iteration but with optimized computations for improved efficiency.

3. `powerIteration3:[4]`

   - The `powerIteration3` is an optimized version of the power iteration algorithm that utilizes parallelization to improve performance.
   - It employs OpenMP directives to parallelize the matrix-vector multiplication, normalization, error calculation, and vector update loops, reducing the overall computation time.
   - By dividing the work among multiple threads, the algorithm can leverage the available processing power of a multi-core CPU, potentially achieving faster convergence.

4 Experiments

The various experiments were conducted to assess the program's performance under different scenarios and the outcomes of each experiment are presented with tables and graphs.

Table 1: Compilation Flag and Time Comparison - `powerIteration`

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.029937	0.029706	0.007350	0.022331	0.007350

Table 2: Compilation Flag and Time Comparison - `powerIteration2`

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.027408	0.027449	0.004770	0.0198756	0.004770

Table 3: Compilation Flag and Time Comparison - `powerIteration3` - 2

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.016240	0.004638	0.004528	0.008568	0.004528

Table 4: Compilation Flag and Time Comparison - `powerIteration3` - 3

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.010512	0.003157	0.003253	0.005640	0.003157

Table 5: Compilation Flag and Time Comparison - `powerIteration3` - 4

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.008453	0.002892	0.003020	0.004788	0.002892

Table 6: Compilation Flag and Time Comparison - `powerIteration3` - 5

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.006609	0.002095	0.002193	0.003632	0.002095

Table 7: Compilation Flag and Time Comparison - `powerIteration3` - 6

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.005545	0.001976	0.001997	0.003172	0.001976

Table 8: Compilation Flag and Time Comparison - `powerIteration3` - 8

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.004403	0.001689	0.001693	0.002595	0.001689

Table 9: Compilation Flag and Time Comparison - `powerIteration3` - 10

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
Time (s)	0.003727	0.001726	0.001738	0.002397	0.001726

Table 10: Compilation Flag and Time Comparison - `3 power iterations`

Compilation Flag	-O1	-O2	-O3	Avg time	Best time
poweriteration	0.029869	0.008874	0.007347	0.015363	0.007347
poweriteration2	0.006869	0.001366	0.001176	0.003137	0.001176
poweriteration3	0.002441	0.001133	0.001158	0.001577	0.001133

Based on the above we can plot the graph and analyse the outcome of the optimization and parallelisation of the power iteration method.

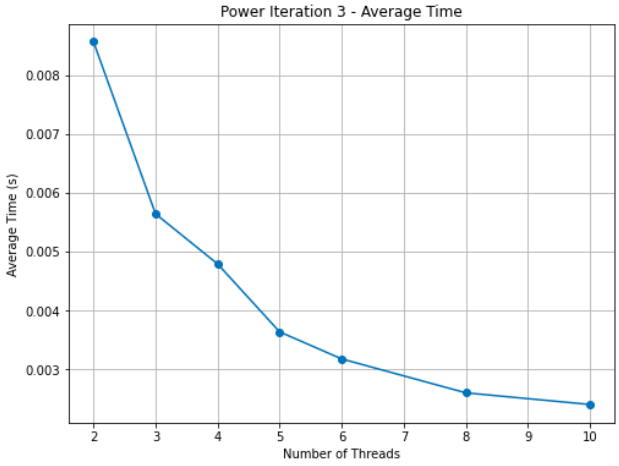

Figure 2: OpenMP-Avg

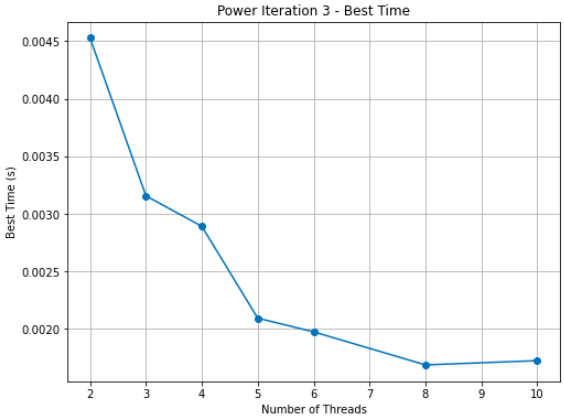

Figure 3: OpenMP-best

From the graph and the table for `powerIteration3` I observed different numbers of threads (2, 3, 4, 5, 6, 8, and 10).

1. As the number of threads increases (from 2 to 10), the execution time decreases consistently. This demonstrates the benefits of parallelization, where multiple threads can perform computations simultaneously, leading to faster execution times.

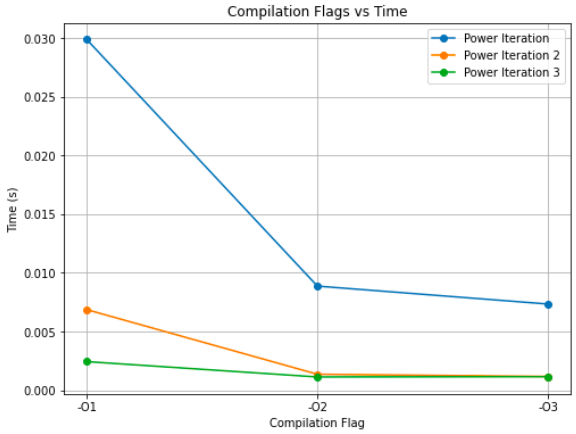

Figure 4: 3 power iterations

2. The average time represents the average execution time across multiple runs, while the best time represents the fastest execution time achieved. The best time is generally lower than the average time, indicating that there might be some variation in the execution times, and the best time represents the optimal performance.

3. The -O2 and -O3 optimization levels apply different optimization techniques to the code. -O2 focuses on general optimizations, while -O3 includes more aggressive optimizations that might not always yield better results for every codebase. Depending on the specific characteristics of the code being compiled, -O2 optimizations may be more effective than -O3 optimizations in certain scenarios.

From the graph and the table for `3 power iterations` I observed

1. As the optimization level increases from -O1 to -O3, the average and best times decrease for all three power iterations. This indicates that higher optimization levels can improve the performance of the programs.

2. From the average time and best time columns, we can see that "poweriteration2" consistently performs the best among the three programs across all optimization levels. It has the lowest average and best times, indicating that it is the most efficient implementation.

3. It should be noted that for this the output differs for all 3 iterations because the result of each iteration may depend on the state of the variables from previous iterations. This can introduce subtle differences in the calculations, leading to slightly different outputs

5 Conclusions

In conclusion, parallelization significantly improves performance In the case of `powerIteration3`, as the number of threads increases, the execution time consistently decreases. This highlights the benefits of parallelization, where multiple threads can perform computations simultaneously, leading to faster execution times. The speedup achieved with parallelization demonstrates the potential for efficient computation of dominant eigenvalues and eigenvectors, particularly for large matrices. Further, The experiment results indicate that increasing the optimization level from -O1 to -O3 generally leads to lower average and best execution times across all three power iterations. This suggests that higher optimization levels can improve the performance of the programs. However, it's important to note that the impact of optimization levels may vary depending on the specific characteristics of the code being compiled. In some cases, less aggressive optimizations (-O2) may yield better results. It should be acknowledged that the outputs of the three power iterations may differ slightly due to subtle differences in calculations. This is expected because the result of each iteration can depend on the state of variables from previous iterations. Therefore, slight variations in the calculations can lead to slightly different outputs.

References

[1] CodeWithC. *C Program for Power Method*. `https://www.codewithc.com/c-program-for-power-method/`. 2023.

[2] Gagandeep Kaur. "Performance Analysis of Power Iteration Method using OpenMP". In: *International Journal of Emerging Technologies in Engineering Research* 5.6 (2017), pp. 91–96.

[3] Kent State University. *Eigenvalue Methods*. `https://www.math.kent.edu/~reichel/courses/intr.num.comp.2/lecture21/evmeth.pdf`. 2023.

[4] Jarmo Rantakokko. *OpenMP Case Studies*. 2023.

[5] University of Granada. *Fortran Power Method*. `https://ergodic.ugr.es/cphys/lecciones/fortran/power_method.pdf`. 2023.