Standard Matrix Exponentiation in C and Python

Matrix Multiplication in Series

Andrew Struthers
April 12, 2024

CS 530: High Performance Computing
Dr. Szilard VAJDA

Department of Computer Science
Central Washington University
Ellensburg, WA, United States of America
Spring Quarter, 2024

Contents

0.1 Problem Statement

The goal of this lab is to implement and assess the performance of calculating the Nth power of a square matrix of dimension M using different programming paradigms: C/C++ with dynamic arrays, and a scripting language, which in this case is Python. The primary objectives include timing algorithms for matrix exponentiation, evaluating their computational complexity, and visualizing the running times for varying matrix sizes and powers.

Matrix multiplication can be further improved using various methods of parallelization, however the focus of this lab is to evaluate the performance of expensive algorithms. Alongside future labs, this will provide an understanding of the efficiency gained by imploring high performance computing methodologies in future labs.

0.2 Introduction

Matrix exponentiation is a fundamental operation in computational mathematics with applications in various fields such as physics, engineering, and computer science. The direct computation of large matrix powers can be computationally expensive due to its polynomial complexity. In this lab, students explored different approaches to compute the Nth power of a matrix A and analyze their time complexity as a function of matrix size and power.

0.3 Methods

There were multiple methods used in this lab. Each method implemented the same algorithm and measured performance utilizing the same hardware. The methods used in this lab are as follows:

- C with Static Arrays: This implementation utilizes static arrays for matrix storage, but is limited by the size of the matrix. Static allocation works for small sizes of data, however is limited when data sizes increase due to the limitations of stack memory.
- C with 2D Arrays: This implementation utilizes dynamic memory allocation for matrix storage and employs standard C/C++ arrays for matrix operations.
- Python Scripting: The Python implementation provides a high-level approach using native list structures to represent and compute matrix operations.

Due to the similarities in the static array implementation and the dynamic array implementation within C, this report focuses on comparing dynamic arrays in C with the Python approach. The algorithms in C and Python are functionally identical, save for language-based differences.

0.3.1 Steps

1. Memory Allocation:
   - Allocate space for matrix A by using a singly linked list structure, where one dimension is an array of pointers to each row of the matrix.
   - Allocate the same amount of space for two placeholder matrices, identical in size to matrix A. These placeholders will be used for intermittent calculation.
2. Matrix Initialization:
   - Generate a square matrix A of dimension M filled with random values.
   - Alternatively, initialize A as an identity matrix for specific tests.
3. Matrix Exponentiation:
   - Compute the Nth power of matrix A iteratively, making use of the placeholder matrices to store current and previous iterations matrices while leaving matrix A untouched.
4. Complexity Analysis:

- Measure and compare the time complexity of each approach using multiple timing functions (e.g., gettimeofday(), time(), and clock() in C, time() and datetime() in Python).

5. Memory Deallocation:
   - Iteratively deallocate each row of the three matrices.
   - Free the linked list dimension of the dynamic arrays after each row has been freed as to not create unaccessable memory.

0.4 Results

0.4.1 Setup (Hardware)

The experiments were conducted on a desktop in Samuelson 140. The specifications of the desktop is as follows: 12th Gen Intel Core 19-12900 CPU clocked at 2.4GHz with 64GB of RAM clocked at 4800 MHz. The operating system was Linux Ubuntu and the C code was compiled using GCC. The timing was performed using the gettimeofday() function in C, and the time() function in Python. These timing libraries were used as opposed to others because they provided a more accurate timings. The clock() function in Linux has less precision and sums user and actual time together, which leads to slight variations. The time() function has a 1 second precision, making it useless for sub second timings. The gettimeofday() function has microsecond precision on systems where the system clock supports that level of granularity, making it the most accurate out of the three clock functions investigated. Likewise, the time() function was used in Python because time() provides monotonic, nano- or microsecond precision depending on how the computer tracks time, whereas datetime.now() is not as precice and measures wall time.

0.4.2 Data

Thirty independent experiments were run in both C and Python. Experimentation consisted of varying the matrix dimension while keeping a fixed power number, as well as a different experiment consisting of having a fixed dimension while varying the matrix power. Experiments were run with the following inputs:

Language	Dimension	Language	Power
C, Python	10	C, Python	100
C, Python	20	C, Python	200
C, Python	30	C, Python	300
C, Python	40	C, Python	400
C, Python	50	C, Python	500
C	60	C, Python	600
C	70	C, Python	700
C	80	C, Python	800
C	90	C, Python	900
C, Python	100	C, Python	1000
C, Python	200	C, Python	2000
C	300	C, Python	3000
C	400	C, Python	4000
C	500	C, Python	5000
		C, Python	6000
		C, Python	7000
		C, Python	8000
		C, Python	9000
		C, Python	10000

Table 1: Showcase of the various experimentations run. When varying dimensions, the power was fixed at 1000. When varying power, the dimension was fixed at 50.

0.4.3 Results

Time measurements were recorded for each implementation and plotted against varying matrix sizes and powers using Excel for graph visualization. Trendlines were added to the graphed data. On the graphs where dimensionality was varied, the trendlines were third order polynomials. When varying power, the trendline used was linear. Correlation constants were added to each of the trendlines, showcasing the claims made via algorithmic analysis. The algorithm for matrix multiplication has the asymptotic behavior of $O(n^3)$ where n is the dimension of the matrix. The algorithm for raising a matrix to a power scales linearly, with asymptotic behavior of $O(n)$ where n in this case is the power. Both of these trends are shown with correlation constants of $R^2 = 1.0$ for all data.

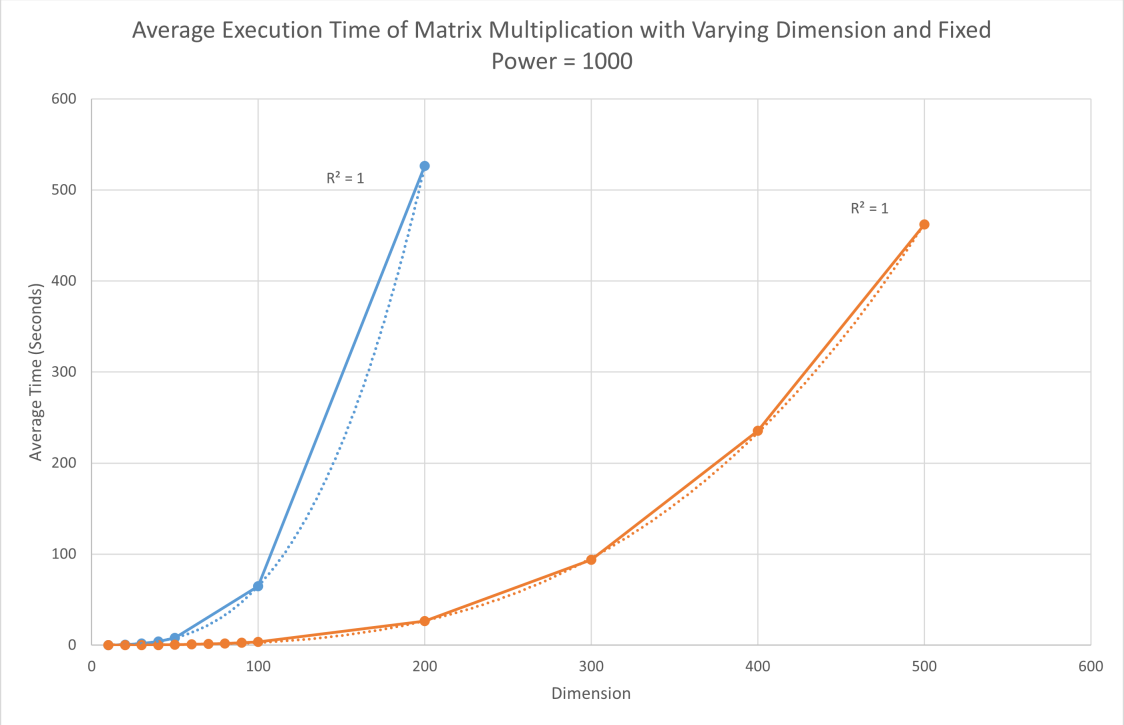

Figure 1: Graph of average execution time while varying dimension of matrix A. Python graphed in blue and C graphed in orange. Line of best fit is a third order polynomial.

In Figure 2, the correlation constant for the C execution time is not 1.0. This is due to matrices, despite being initialized with values between $[-1.0, 1.0]$, would be filled with $-\inf, +\inf, -NaN$, or $+NaN$. The matrix multiplication code had an early return condition if this was the case. As shown in the figure, the matrix multiplication time plateaued at A^{8000}, which is when the $NaN-$like early stopping conditions started happening frequently.

Figure 2: Graph of average execution time while varying power of computation. Python graphed in blue and C graphed in orange.

In Figure 3, the erroneous data from the C execution was removed, which returns the correlation constant back to 1.0. Both Figure 2 and Figure 3 provide strong evidence for the variation of the exponent to result in linear scaling of runtime.

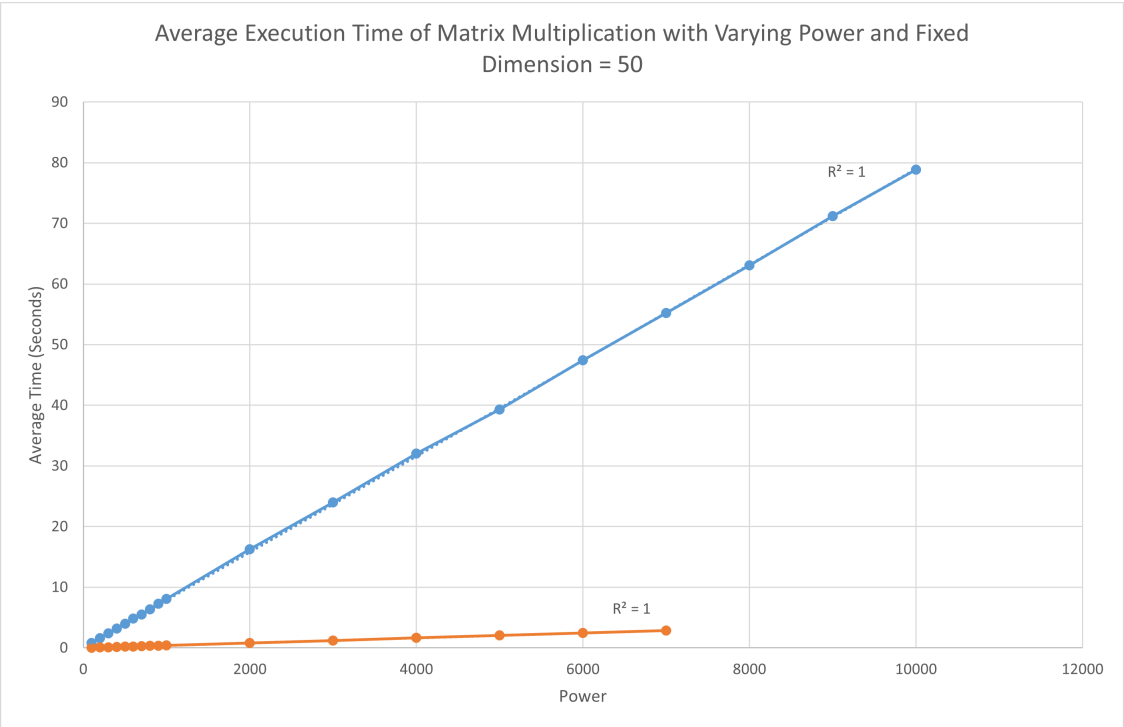

Figure 3: Graph showing the linear scaling nature of varying the power of matrix computation with erroneous data removed.

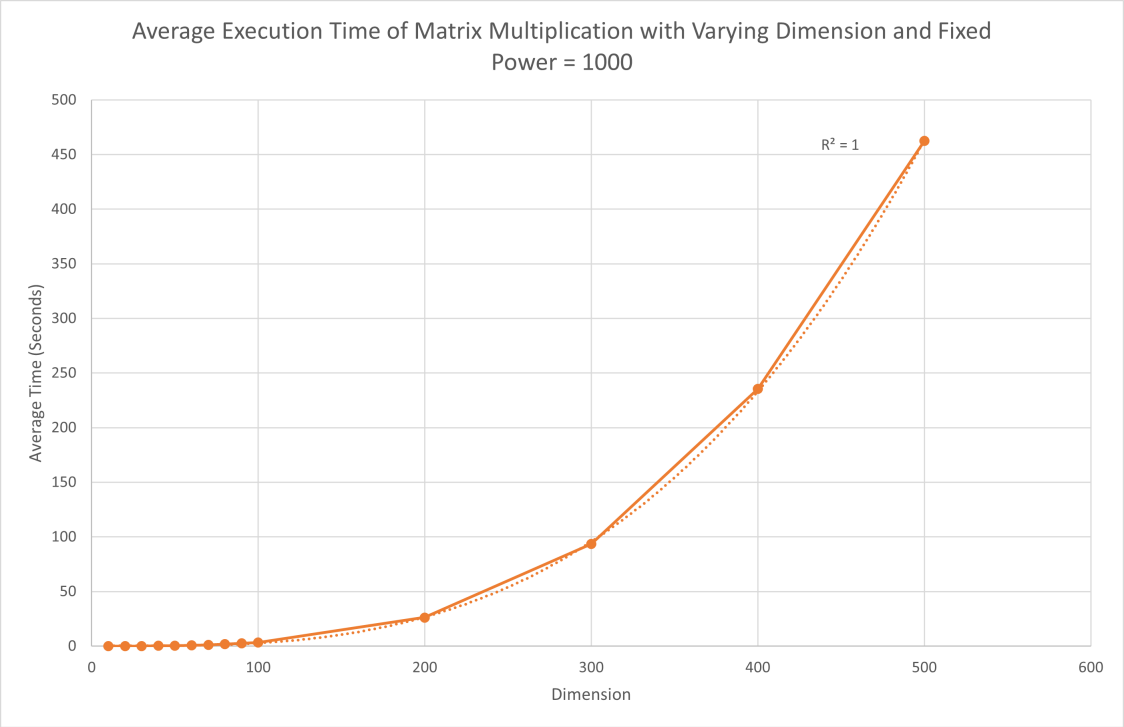

Figure 4: Average execution time as a result of varying the matrix dimension, performed using the C implementation.

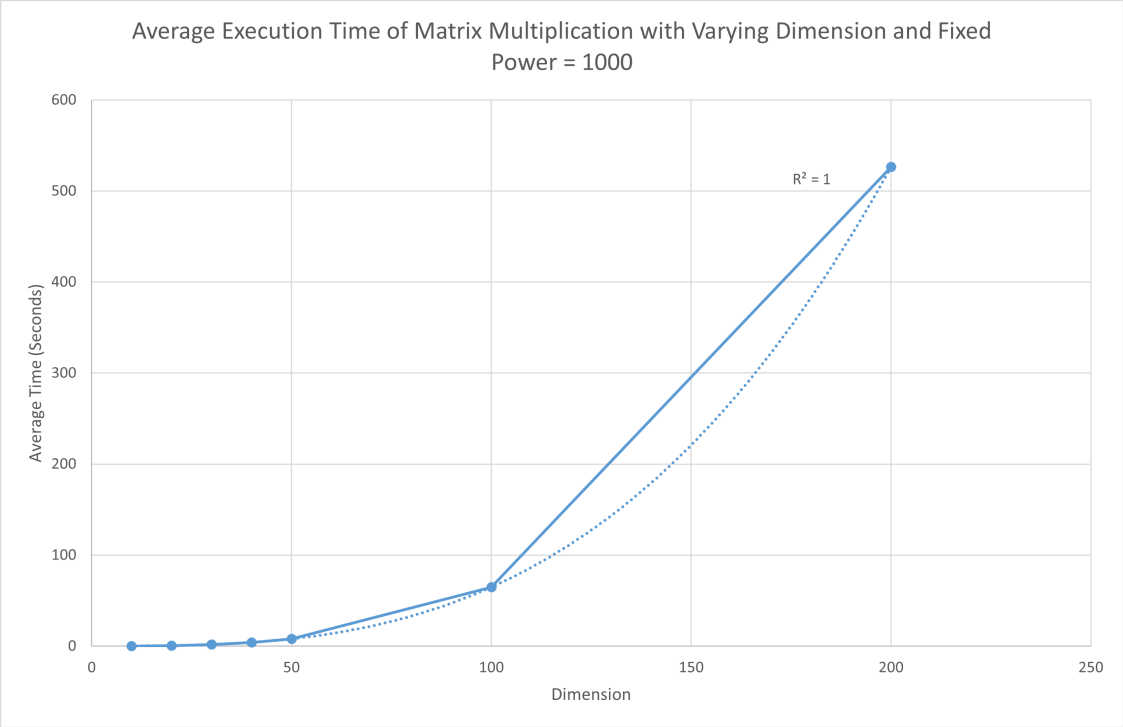

Figure 5: Average execution time as a result of varying the matrix dimension, performed using the Python implementation.

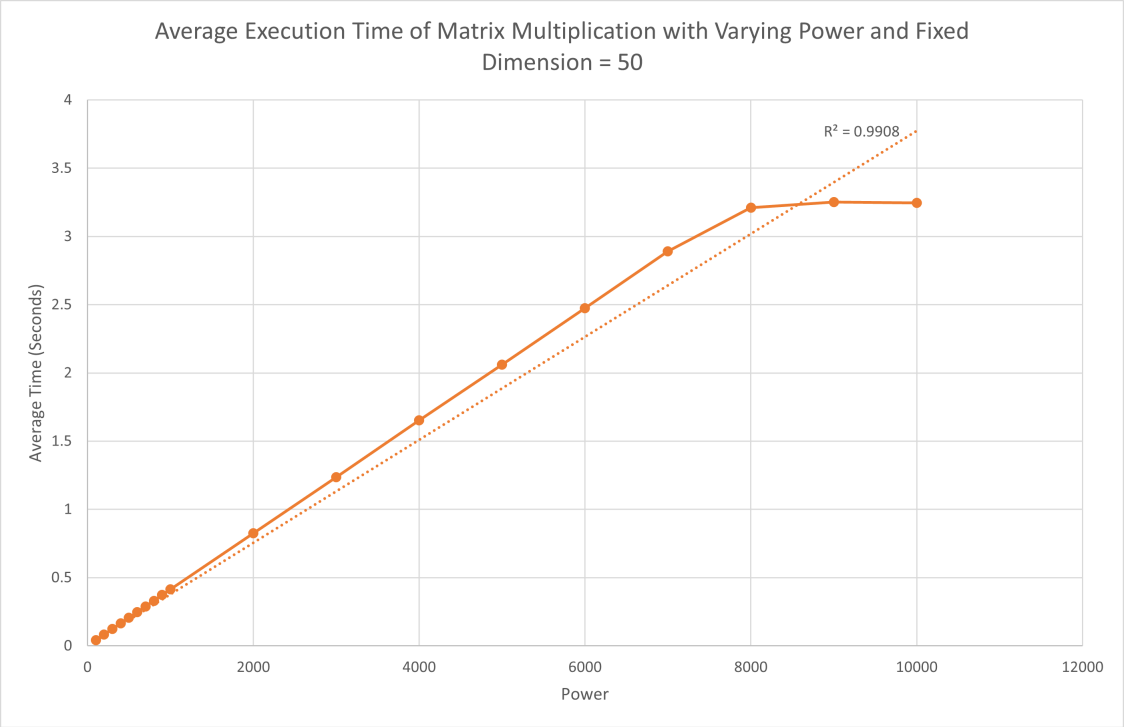

Figure 6: Average execution time as a result of varying the matrix power, performed using the C implementation. Erroneous data is left in, showing the results of early termination when the matrix became too large to represent.

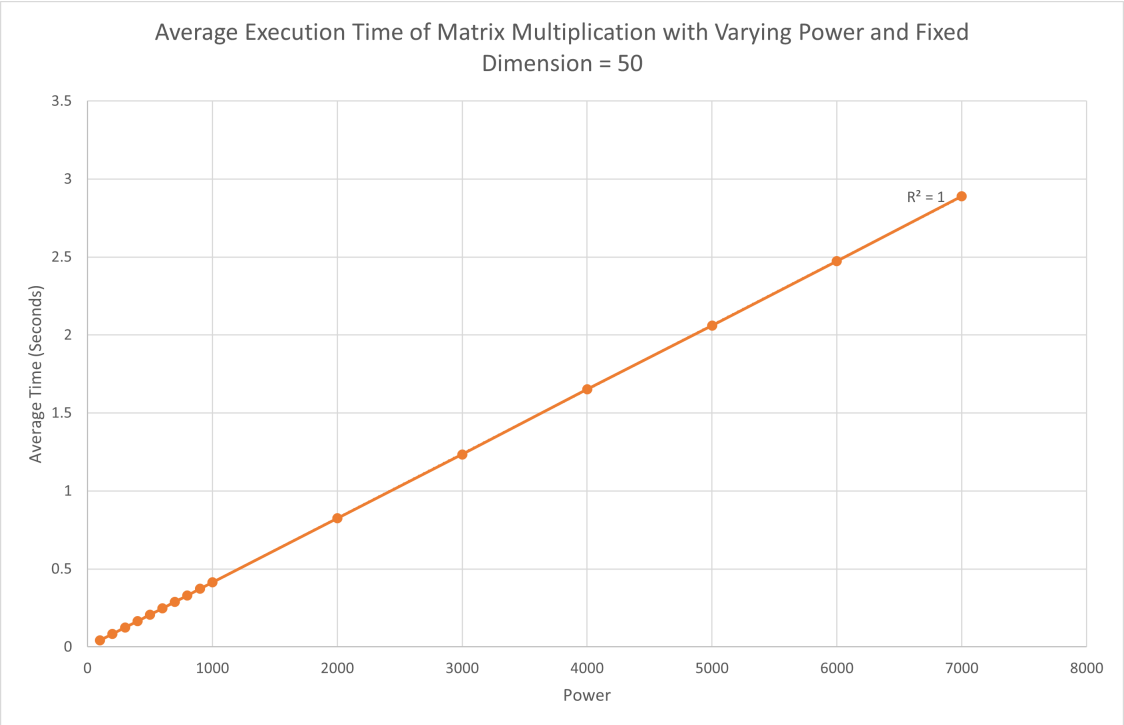

Figure 7: Average execution time as a result of varying the matrix power, performed using the C implementation. Erroneous data is removed to show perfect linear scaling.

Figure 8: Average execution time as a result of varying the matrix power, performed using the Python implementation.

0.5 Conclusion

In conclusion, matrix exponentiation is a computationally demanding task, especially for large matrices and exponents. Matrix multiplication scales with the asymptotic behavior of $O(n^3)$ where n is the size of the matrix. This is clearly shown in the figures and correlation values associated with the polynomial trendlines. Likewise, raising a matrix to a power scales asymptotically with the behevaior of $O(n)$ where n is the power. The choice of programming paradigm, between dynamic 2D arrays vs. scripting, significantly impacts the efficiency and scalability of the solution. From the experiments conducted, it is evident that C with dynamic 2D linked arrays outperforms the Python implementation. The scripting language exhibits higher execution times due to its interpreted nature, however the asymptotic behavior is the same regardless of the language used.