CS 530: High-Performance Computing
Benchmarking Matrix Multiplication: Sequential vs Multithreaded

Nathan Chapman

Department of Computer Science
Central Washington University

May 14, 2024

Contents

1 Introduction

- Many things can benefit from parallel execution
- Matrix multiplication is “embarissingly parallel”
- Makes it a good benchmark to compare sequential execution to parallel

2 Methods

While there is support for the manual creation and management of threads in C with `pthread_create` and `pthread_join`, the Julia programming language natively supports creating thread pools when starting the Julia runtime process with `julia -t n`, where `n` is the number of threads to created in the thread pool. Additionally, Julia can automatically assign tasks in a `for` loop to its different threads with the use of OpenMP-pragma-like macros such as `Threads.@threads`. For example, a matrix multiplication parallelized over the rows of the product matrix can be seen in algorithm 1. Further detail and examples can be found in the source code bundled with this report.

Algorithm 1: Row-parallelized matrix multiplication using Julia’s `Threads.@threads`

```
Input: $N \times L$ Matrix A, $L \times M$ Matrix B
Output: Matrix C of zeros
Threads.@threads for $i \in \{1, 2, \dots, N\}$ do
    for $j \in \{1, 2, \dots, M\}$ do
        for $k \in \{1, 2, \dots, L\}$ do
            C[i,j] ← C[i,j] + A[i, k] × B[k, j]
        end
    end
end
```

In the representation shown in algorithm 1 it can be seen that the runtime complexity of this implementation is reduced from the traditional implementation of matrix multiplication as $O(N^3)$ to $O(N^2)$. This simplication comes from the fact that, while there are still N column calculations that need to be considered, each with N terms needing to be added together, these N^2 calculations are done at the same time, effectively transforming the outer-most N operations in a factor of simply 1. Table 1 shows the associated runtime complexities for other forms of parallelized matrix multiplication.

Parallelism	Rows	Columns	Summands	Complexity	Matrix Power Complexity
Sequential	N	N	N	$O(N^3)$	$O(pN^3)$
Row	1	N	N	$O(N^2)$	$O(pN^2)$
Column	N	1	N	$O(N^2)$	$O(pN^2)$
Element	1	1	N	$O(N)$	$O(pN)$

Table 1: Different runtime complexities for different levels of paralleism in matrix multiplication.

For any form of parallelism (none, row, column, or element), when considering the *power* p of a matrix, the runtime complexity is simply linearly scaled by p. This additional linear factor arises from each `for` loop needing to be evaluated p times, essentially creating another `for` loop in p. Algorithm 2 shows how algorithm 1 can be modified to yield the power of matrix, while also highlighting how the runtime complexity changes.

Algorithm 2: Row-parallelized matrix power using Julia's `Threads.@threads`

```
Input: $N \times N$ Matrix A, Power $P$
Output: Matrix C of zeros
for $p \in \{1, 2, \ldots, P\}$ do
    Threads.@threads for $i \in \{1, 2, \ldots, N\}$ do
        for $j \in \{1, 2, \ldots, N\}$ do
            for $k \in \{1, 2, \ldots, N\}$ do
                C[i,j] ← C[i,j] + A[i, k] × B[k, j]
            end
        end
    end
end
```

3 Results

Figures (1, 2, 3) show the timings of matrix powers for different regions of the dimension-power space. Figure (1) shows the comparison between the sequential and paralleized matrix powers for different forms such as element-wise parallelism, row-wise, and column-wise. Additionally, the relative timings, as compared to the sequential execution, are also displayed for comparison. The other figures follow the same scheme.

Figure (4) shows how the execution time of the paralleized matrix power depends on the power to which the matrix is raised.

The results in figures (2, 4, 1) were gathered on a machine with the specifications shown in table 2, while the results in figure (3) were gathered on a machine with specifications as show in table (3).

4 Discussion

While each figure could not more clearly suggest that using multiple threads to execute matrix multiplications is preferred when it comes to execution time, there are discrepancies between them. Figure (1) shows the parallel methods executing in only about 50% of the time compared to the sequential version. Compared to the over-50% for large-dimensions (figure (3)) or even 80% (wow!) reduction for medium-dimensions (figure 2)

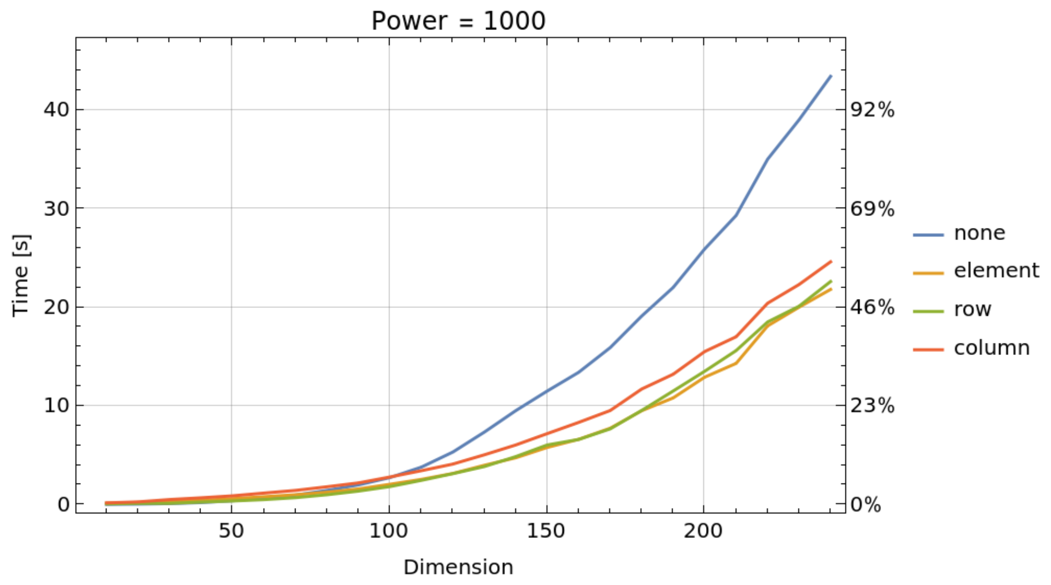

Figure 1: Benchmarking for different forms of parallelism over matrix dimension, and a power of 1000.

OS	Manjaro Linux
Kernel	Linux 6.6.30-2-MANJARO
Shell	bash 5.2.26
CPU	13th Gen Intel i7-13700K (24) @ 5.3GHz
GPU	NVIDIA GeForce RTX 3060 Ti Lite Hash Rate
Memory	31.18GiB
GPU Driver	NVIDIA 550.78

Table 2: Different runtime complexities for different levels of paralleism in matrix multiplication.

5 Conclusion

- a lot of applications take a long time because a lot of things need to be done
- these applications can benefit from parallelism via multithreading
- multithreading can be hard to implement
- sometimes it can be easier when the problem is embarissingly parallel
- new languages make parallelism trivial
- parallel implementation can save potentially massive amounts of time
- further parallelism and performance can be achieved with GPU programming like CUDA

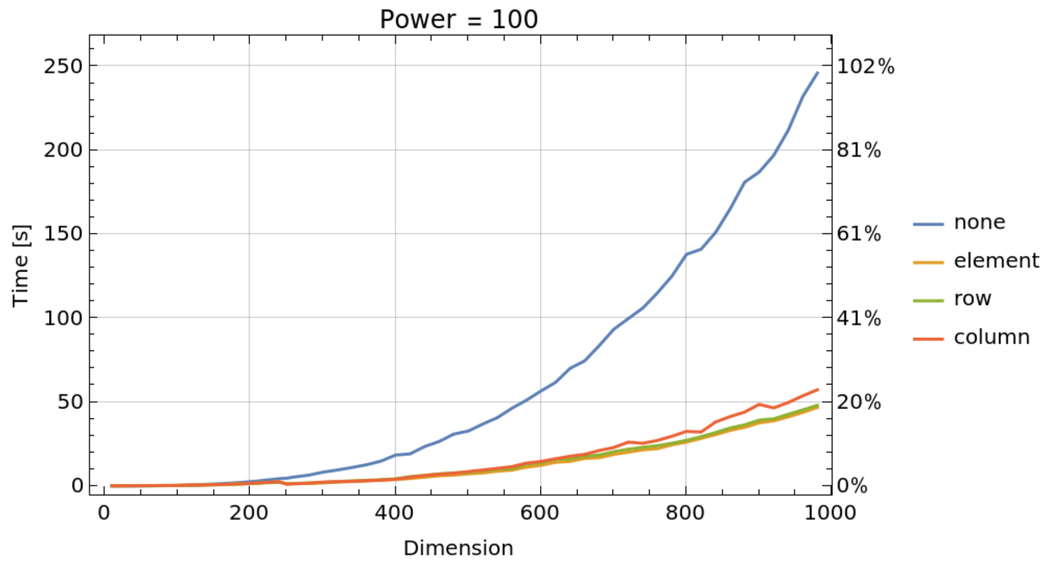

Figure 2: Benchmarking for different forms of parallelism over matrix dimension, and a power of 100

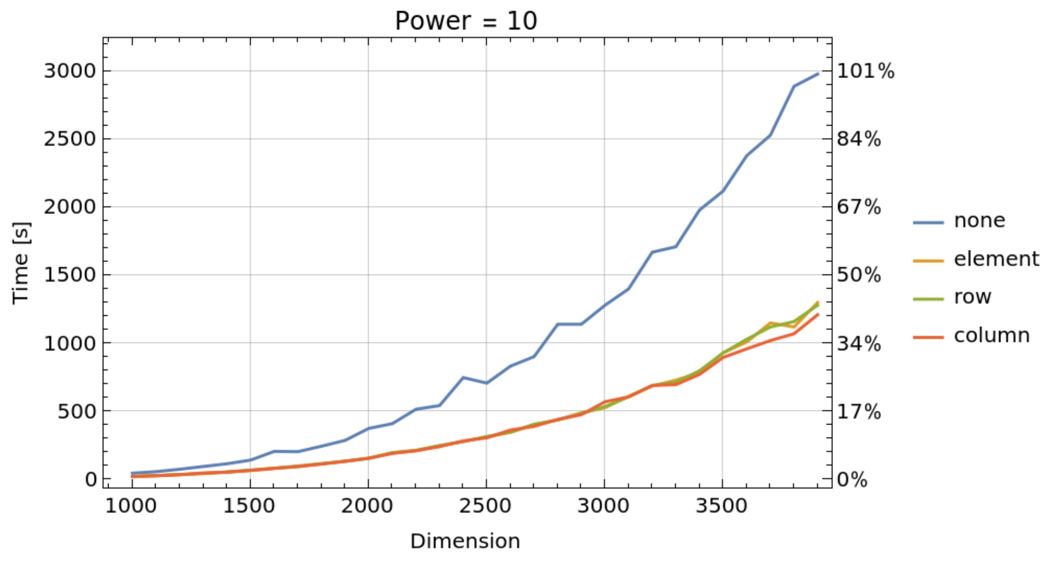

Figure 3: Benchmarking for different forms of parallelism over large matrix dimensions, and a power of 10

OS	Arch Linux
Kernel	Linux 6.6.30-2
Shell	bash 5.2.26
CPU	6th Gen Intel i5-6500 (4) @ 3.2GHz
GPU	NVIDIA GeForce GTX 1660 Super
Memory	16GiB

Table 3: Different runtime complexities for different levels of paralleism in matrix multiplication.

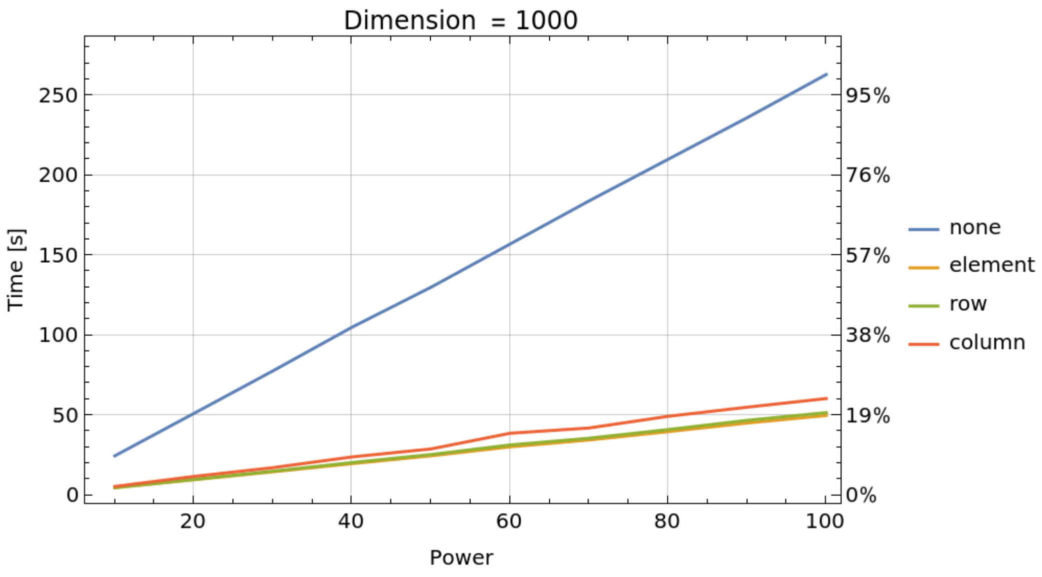

Figure 4: Benchmarking for different forms of parallelism over large matrix dimensions, and a power of 10