Parallel and Vectorized Matrix Multiplication in Python

Miguel Cabeza Lantigua

University of Las Palmas de Gran Canaria
School of Computer Engineering
Degree in Data Science and Engineering
Big Data

December 5, 2025

GitHub Repository: `https://github.com/migcablan/IndividualTask3`

Contents

Abstract

This work evaluates the performance of different implementations of matrix multiplication in Python: a basic sequential version, a process–based parallel version using the `multiprocessing` module, and a vectorized implementation using NumPy. The study analyzes execution time, speedup, and parallel efficiency for square matrices up to 2048×2048.

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data analytics. This report focuses on parallel and vectorized implementations in Python, following the assignment requirements to implement a parallel version, an optional vectorized version, and to compare both with the basic algorithm in terms of speedup and efficiency.

2 Methodology

2.1 Tools

All experiments were carried out on Windows 11 using Visual Studio Code as the development environment. The main tools and libraries were:

- **Python 3.x** (CPython distribution).
- `time.perf_counter()` for high–resolution timing.
- `multiprocessing` to spawn multiple worker processes.
- `NumPy` to implement the vectorized matrix multiplication.
- `csv` and `pandas` (in an external notebook) to store and analyze the experimental results from `mm_results.csv`.
- `matplotlib` to generate the plots included in the Results section.

All measurements were executed on the same machine, in an otherwise idle system, to reduce noise and obtain comparable timings.

2.2 Implementations

The project consists of several Python modules: `sequential.py`, `parallel.py`, `vectorized_numpy.py`, `resources.py`, and `compare_metrics.py`. The last one orchestrates the benchmarks and writes all metrics to the CSV file `mm_results.csv`.

Basic sequential multiplication The baseline implementation operates on standard Python lists using the classic triple nested loop scheme:

```
def matmul_basic(A, B):
    n = len(A)
    m = len(A[0])
    p = len(B[0])
    C = [[0.0 for _ in range(p)] for _ in range(n)]
    for i in range(n):
        for k in range(m):
            aik = A[i][k]
            for j in range(p):
                C[i][j] += aik * B[k][j]
    return C
```

This version is used as the reference to compute speedup and to evaluate the benefits of parallelization and vectorization.

Parallel multiplication with `multiprocessing` The parallel version splits the rows of matrix A into chunks and assigns each chunk to a worker process. Each worker computes a partial result, which is then concatenated:

```
def matmul_worker(args):
    A_chunk, B = args
    m = len(A_chunk[0])
    p = len(B[0])
    C_chunk = []
    for i in range(len(A_chunk)):
        row = [0.0] * p
        for k in range(m):
            aik = A_chunk[i][k]
            for j in range(p):
                row[j] += aik * B[k][j]
        C_chunk.append(row)
    return C_chunk

def matmul_parallel(A, B, n_procs=None):
    if n_procs is None:
        n_procs = mp.cpu_count()
    n = len(A)
    chunk_size = (n + n_procs - 1) // n_procs
    chunks = [A[i:i + chunk_size] for i in range(0, n,
       chunk_size)]
    with mp.Pool(processes=n_procs) as pool:
        C_chunks = pool.map(matmul_worker, [(chunk, B) for
           chunk in chunks])
    C = []
    for chunk in C_chunks:
        C.extend(chunk)
    return C
```

The number of processes is varied between 2, 4 and the total logical cores reported by `mp.cpu_count()`.

Vectorized multiplication with NumPy The vectorized implementation relies on NumPy arrays and the `@` operator, which internally uses optimized BLAS routines with SIMD and multithreading when available:

```
def benchmark_numpy(size):
    A = np.random.rand(size, size)
    B = np.random.rand(size, size)
    start = time.perf_counter()
    C = A @ B
    end = time.perf_counter()
    return end - start
```

This implementation represents the "vectorized" approach requested in the assignment.

2.3 Benchmark procedure

The script `compare_metrics.py` generates random square matrices of sizes

$$n \in \{32, 64, 128, 256, 512, 1024, 2048\},$$

runs the three implementations (basic, parallel, NumPy) and records:

- Execution time in seconds.
- Speedup with respect to the basic implementation.
- Parallel efficiency (speedup/processes) for the parallel version.
- Total number of logical cores on the machine.

All records are stored in the CSV file `mm_results.csv`, which is later imported in a notebook to generate the plots shown below.

3 Results

3.1 Execution time vs. matrix size

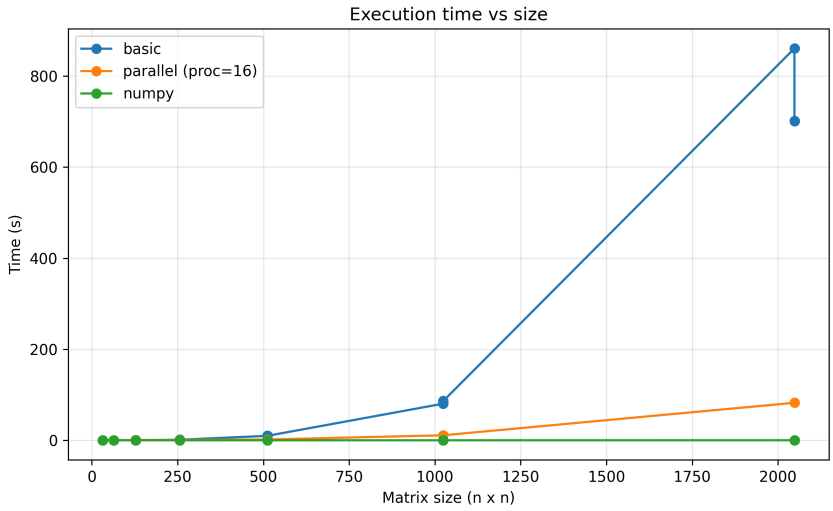

Figure 1: Execution time of the basic, parallel and NumPy implementations as a function of matrix size, obtained from `mm_results.csv`. The basic implementation exhibits cubic growth, while the parallel version reduces runtime for large sizes when enough processes are used. NumPy is consistently the fastest due to optimized native code.

3.2 Speedup and parallel efficiency

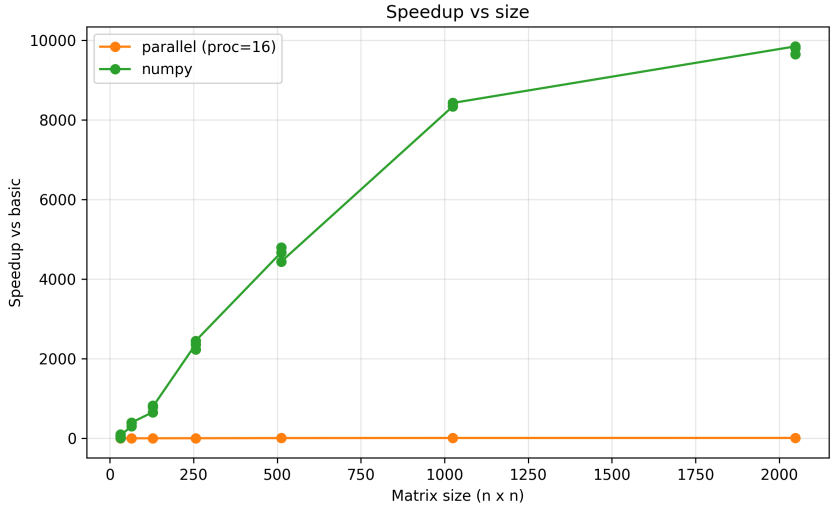

Figure 2: Speedup of the parallel and NumPy implementations with respect to the basic version. For small matrices the overhead of process creation limits the speedup, but for sizes above 512×512 the parallel version reaches meaningful acceleration. NumPy shows the highest speedup across all tested sizes.

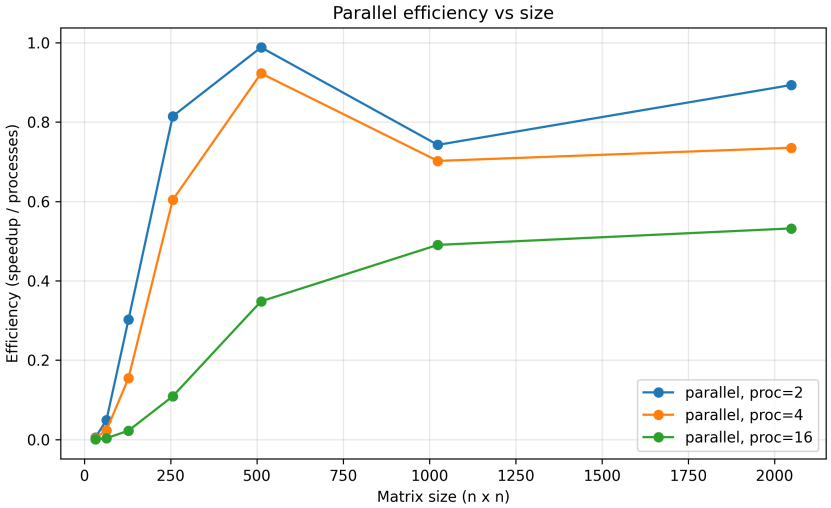

Figure 3: Parallel efficiency (speedup/processes) for different process counts. Efficiency decreases as more processes are used, illustrating the impact of synchronization and data sharing overheads, especially for smaller matrices.

4 Discussion

The experimental data in `mm_results.csv` confirm the expected cubic time complexity of the naive algorithm. For small matrix sizes the overhead associated with spawning multiple processes dominates, and the parallel version is not significantly faster than the sequential baseline. As the matrix size grows, the parallel implementation achieves noticeable speedups, although efficiency decreases when the number of processes approaches the number of logical cores, due to contention and communication overhead.

The NumPy implementation consistently outperforms both Python–level versions. Its high performance is explained by optimized BLAS kernels implemented in low–level languages and by effective use of SIMD instructions and multithreading inside the numerical libraries.

Overall, the experiments highlight the trade–off between implementation complexity and performance: pure Python parallelism can provide benefits for sufficiently large problems, but the best results are obtained when delegating the heavy computation to optimized numerical libraries.

5 Conclusion

This study has implemented and evaluated three approaches to matrix multiplication in Python: a basic sequential algorithm, a parallel version based on the `multiprocessing` module, and a vectorized implementation using NumPy. The main conclusions are:

- The basic algorithm is easy to implement but scales poorly in time.

- Process–based parallelization improves performance for large matrices, at the cost of higher overhead and reduced efficiency when many processes are used.
- NumPy provides the best performance and should be the default choice when available, thanks to highly optimized native routines.
- Recording results in a structured CSV file simplifies the analysis of speedup and efficiency, and makes it easy to produce clear plots and tables.