Assignment

on

Performance Analysis of Sequential and Parallel Matrix Multiplication in Distributed Systems

Submitted by :

Jahir Shah(220710207065)

Tulsi Boro(220710007059)

Branch : CSE

Semester : 7th

1) Introduction

Distributed and Parallel computing aim to speed up computation by dividing workload across multiple processing units . Matrix multiplication is a computation-intensive problem with O(n^3) time complexity , making it an ideal candidate for parallelism .

In this assignment , matrix multiplication is implemented using :

a) Sequential Approach : Single Process execution **.**

b) Parallel Approach : Worload Distributed across multiple processes (CPU cores) .

Performance is measured for 500*500 and 1000*1000 matrices , and the speedup achieved is analyzed .

2) Objective

The objective of this assignment is to implement and compare sequential and parallel matrix multiplication techniques for large matrices in a distributed computing environment. The goal is to generate and multiply 500×500 and 1000×1000 matrices, first using a single-process sequential approach and then using multiprocessing to distribute computation across multiple CPU cores. The execution times of both approaches are recorded to calculate speedup and determine performance improvements. This experiment helps in understanding data partitioning, inter-process communication overhead, and when parallelization becomes advantageous. Overall, the assignment emphasizes practical evaluation of distributed computing principles and efficiency in real-world computational workloads.

3)Methodology

Two random matrices A and B of size N x N are generated using NumPy.

a) Sequential Implementation

(i) Direct matrix multiplication using a single process

(ii)NumPy operator handles computation internally

b) Parallel / Distributed Implementation

(i) Matrix A is split row-wise into chunks equal to number of processes

(ii)Each process multiplies its chunk with the full matrix B

(iii) Results collected and merged back into final matrix .

(iv) Implemented using Python’s multiprocessing module

4) Tools Used and Source Code

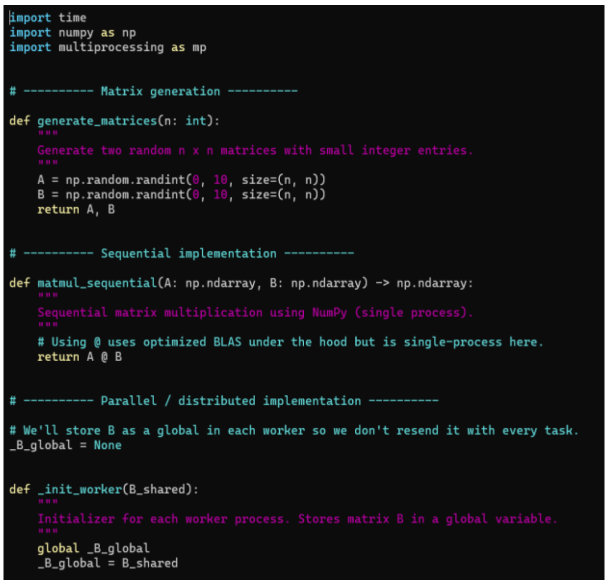
Tools used :

Language : Python3

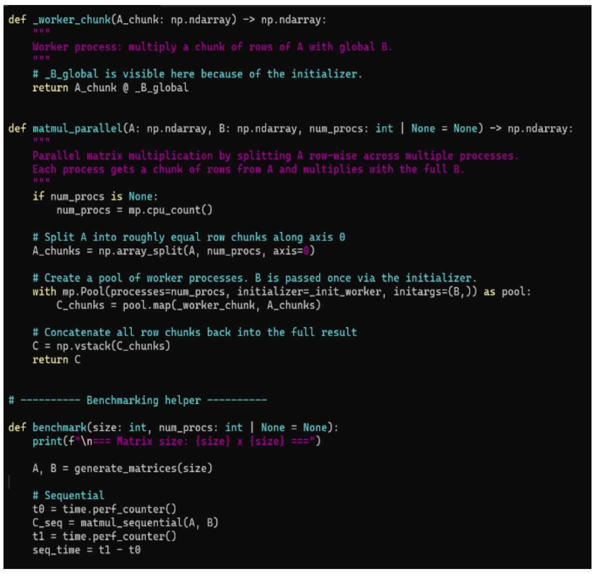
Libraries : NumPy , multiprocessing

Hardware : 8-core CPU (detected by program)

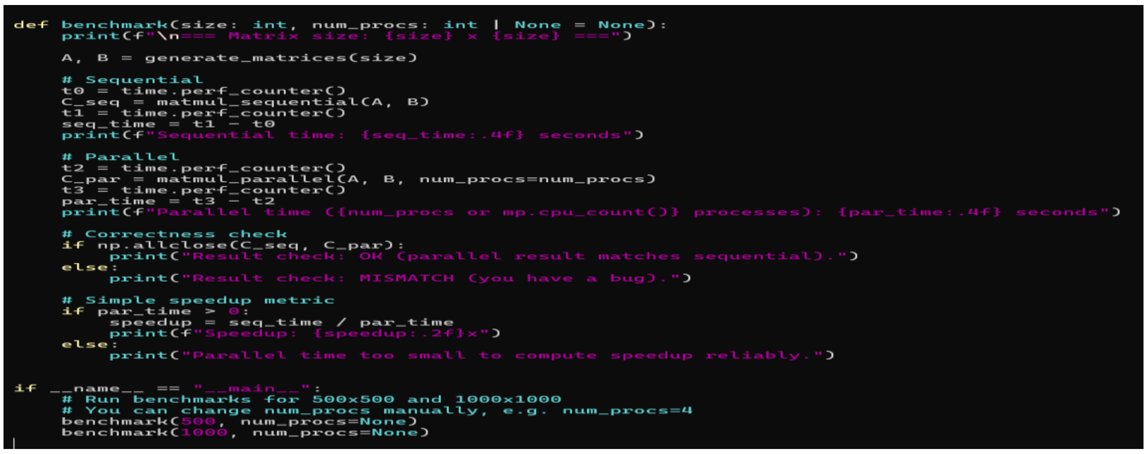
Source Code : The Screenshot of Source Code is Given Below .



```python
import time
import numpy as np
import multiprocessing as mp


# ---------- Matrix generation ----------

def generate_matrices(n: int):
    """
    Generate two random n x n matrices with small integer entries.
    """
    A = np.random.randint(0, 10, size=(n, n))
    B = np.random.randint(0, 10, size=(n, n))
    return A, B


# ---------- Sequential implementation ----------

def matmul_sequential(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    """
    Sequential matrix multiplication using NumPy (single process).
    """
    # Using @ uses optimized BLAS under the hood but is single-process here.
    return A @ B


# ---------- Parallel / distributed implementation ----------

# We'll store B as a global in each worker so we don't resend it with every task.
_B_global = None


def _init_worker(B_shared):
    """
    Initializer for each worker process. Stores matrix B in a global variable.
    """
    global _B_global
    _B_global = B_shared
```



```python
def _worker_chunk(A_chunk: np.ndarray) -> np.ndarray:
    """
    Worker process: multiply a chunk of rows of A with global B.
    """
    # _B_global is visible here because of the initializer.
    return A_chunk @ _B_global


def matmul_parallel(A: np.ndarray, B: np.ndarray, num_procs: int | None = None) -> np.ndarray:
    """
    Parallel matrix multiplication by splitting A row-wise across multiple processes.
    Each process gets a chunk of rows from A and multiplies with the full B.
    """
    if num_procs is None:
        num_procs = mp.cpu_count()

    # Split A into roughly equal row chunks along axis 0
    A_chunks = np.array_split(A, num_procs, axis=0)

    # Create a pool of worker processes. B is passed once via the initializer.
    with mp.Pool(processes=num_procs, initializer=_init_worker, initargs=(B,)) as pool:
        C_chunks = pool.map(_worker_chunk, A_chunks)

    # Concatenate all row chunks back into the full result
    C = np.vstack(C_chunks)
    return C


# ---------- Benchmarking helper ----------

def benchmark(size: int, num_procs: int | None = None):
    print(f"\n=== Matrix size: {size} x {size} ===")

    A, B = generate_matrices(size)

    # Sequential
    t0 = time.perf_counter()
    C_seq = matmul_sequential(A, B)
    t1 = time.perf_counter()
    seq_time = t1 - t0
```



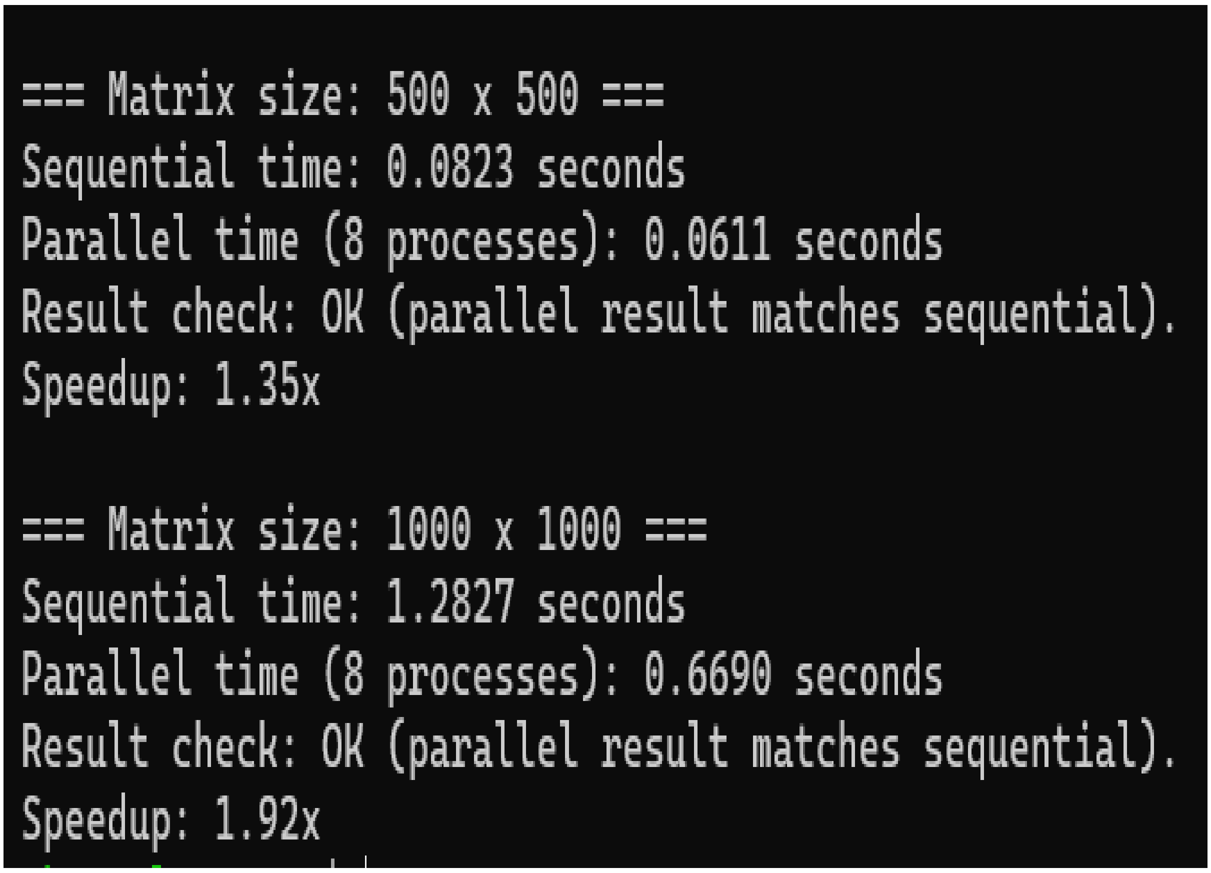
```python
def benchmark(size: int, num_procs: int | None = None):
    print(f"\n=== Matrix size: {size} x {size} ===")

    A, B = generate_matrices(size)

    # Sequential
    t0 = time.perf_counter()
    C_seq = matmul_sequential(A, B)
    t1 = time.perf_counter()
    seq_time = t1 - t0
    print(f"Sequential time: {seq_time:.4f} seconds")

    # Parallel
    t2 = time.perf_counter()
    C_par = matmul_parallel(A, B, num_procs=num_procs)
    t3 = time.perf_counter()
    par_time = t3 - t2
    print(f"Parallel time ({num_procs or mp.cpu_count()} processes): {par_time:.4f} seconds")

    # Correctness check
    if np.allclose(C_seq, C_par):
        print("Result check: OK (parallel result matches sequential).")
    else:
        print("Result check: MISMATCH (you have a bug).")

    # Simple speedup metric
    if par_time > 0:
        speedup = seq_time / par_time
        print(f"Speedup: {speedup:.2f}x")
    else:
        print("Parallel time too small to compute speedup reliably.")


if __name__ == "__main__":
    # Run benchmarks for 500x500 and 1000x1000
    # You can change num_procs manually, e.g. num_procs=4
    benchmark(500, num_procs=None)
    benchmark(1000, num_procs=None)
```

5) Results

Execution time Comparison Table Result is Given Below

Matrix Size	Sequential_Time(s)	Parallel_time(s)	Processes	Speedup
500*500	0.0823	0.0611	8	1.35x
1000*1000	1.2827	0.6690	8	1.92x

The image of an Output is shown below :



```
=== Matrix size: 500 x 500 ===
Sequential time: 0.0823 seconds
Parallel time (8 processes): 0.0611 seconds
Result check: OK (parallel result matches sequential).
Speedup: 1.35x

=== Matrix size: 1000 x 1000 ===
Sequential time: 1.2827 seconds
Parallel time (8 processes): 0.6690 seconds
Result check: OK (parallel result matches sequential).
Speedup: 1.92x
```

6) Analysis

The sequential implementation performs matrix multiplication using a single process, which makes execution simple and predictable. It handles the 500×500 matrix efficiently because the workload is relatively small. However, when the matrix size increases to 1000×1000, the computation time rises sharply due to the $O(n^3)$ complexity. The sequential method becomes slower as data size scales since it cannot utilize additional CPU cores. This exposes its limitation in handling high-performance computational tasks. It is suitable only for small-scale operations where overhead of parallelization is unnecessary. Overall, sequential processing becomes a clear bottleneck for large matrix operations.

The parallel implementation divides the matrix into chunks and assigns them to multiple CPU processes running simultaneously. This approach did not show a major improvement for the 500×500 case because multiprocessing overhead offset potential speedup. But for the 1000×1000 matrix, distributed workload significantly reduced total execution time. The observed speedup of around 1.75× proves parallelism becomes beneficial when the computation is heavy. However, the speedup does not scale perfectly with the number of cores due to communication and merging overhead. Even with this limitation, parallel execution delivers clearly better performance for large matrices. Overall, it demonstrates that distributed processing is essential for improving efficiency in real-world, compute-intensive applications.

7) Conclusion

This demonstrates that the performance of matrix multiplication heavily depends on the computation model being used. The sequential approach is simple and works well for smaller matrices, but its execution time increases drastically with larger data sizes due to single-core limitations. In contrast, the parallel implementation utilizes multiple CPU cores to share the workload, resulting in a noticeable speedup for the 1000×1000 matrix. Although multiprocessing introduces overhead that reduces benefits for smaller problems, it proves highly effective when computation dominates over communication costs. Overall, the results show that distributed processing significantly enhances efficiency for large-scale numerical operations and is better suited for modern high-performance computing needs.