OpenMP Implementation of Strassen's Algorithm

Abdullah Albayrak
Computer Engineering Department
Izmir Institute of Technology
İzmir, Türkiye
abdullahalbayrak@std.iyte.edu.tr

Mahsum Arık
Computer Engineering Department
Izmir Institute of Technology
İzmir, Türkiye
mahsumarik@std.iyte.edu.tr

***Abstract*—This project implements a parallelized version of Strassen's algorithm using OpenMP to improve the performance of matrix multiplication. By leveraging data and task parallelism, along with optimizations such as thresholding and dynamic task scheduling, the algorithm achieves efficient workload distribution across threads. Performance evaluation using Intel VTune Profiler highlights execution time improvements and identifies optimization opportunities. The results demonstrate significant speedup over naive matrix multiplication while providing insights for further refinement.**

I. Problem Definition

Matrix multiplication is a cornerstone operation in computational mathematics and computer science, frequently used in scientific computing, machine learning, and graphics. Despite its ubiquity, the naive matrix multiplication algorithm has

$$O(n^3) \tag{1}$$

computational complexity, which becomes a performance bottleneck for large matrices. Strassen's algorithm offers a faster alternative with

$$O(n^{2.8073}) \tag{2}$$

complexity by reducing the number of multiplications required through recursive decomposition of matrices.

The aim of this project is to design, implement, and evaluate a parallelized version of Strassen's algorithm using OpenMP. The algorithm divides input matrices into smaller submatrices, performs recursive multiplications and additions, and recombines results to produce the final product matrix. By leveraging OpenMP's parallel constructs, such as parallel loops and task parallelism, we aim to distribute the workload of the seven recursive multiplications and associated operations efficiently across multiple threads, thereby reducing execution time.

The implementation also integrates optimizations such as:

- Thresholding: For smaller matrix sizes, the recursion switches to a naive (1) matrix multiplication to reduce overhead from task management.
- Dynamic Task Scheduling: Tasks for recursive calls and matrix operations are dynamically managed to optimize thread utilization.

Performance evaluation is a critical component of this project. Using Intel VTune Profiler, hotspots in the code are identified to pinpoint sections where the most execution time is spent. This analysis is coupled with scalability tests on the UHEM environment to evaluate performance across varying problem sizes and thread counts. Metrics such as execution time, thread activity, and memory usage are used to assess the effectiveness of the parallelization techniques. The results will highlight computational bottlenecks and provide insights into how the OpenMP implementation of Strassen's algorithm performs compared to naive matrix multiplication.

This project ultimately demonstrates the applicability of parallel programming techniques in optimizing computationally intensive algorithms while providing a framework for analyzing and improving their performance using advanced profiling tools.

II. Methodology

This project implements a parallelized version of Strassen's matrix multiplication algorithm using OpenMP. The algorithm divides matrices into submatrices, recursively performs matrix multiplications, and combines results, achieving O(n 2.8073) complexity. OpenMP was used to parallelize matrix operations and recursive computations.

Key parallelization techniques include:

- Data Parallelism: Matrix addition and subtraction were parallelized using `#pragma omp parallel for`, distributing computation across threads to speed up element-wise operations.
- Task Parallelism: The seven recursive multiplications in Strassen's algorithm were executed concurrently using `#pragma omp task` to divide computation into independent tasks.
- Dynamic Task Scheduling: The pragma omp single directive ensured that only one thread initialized the recursive tasks, while other threads executed them concurrently.
- Optimization with Thresholding: A threshold size was set to switch to naive matrix multiplication for smaller matrices, reducing overhead from recursion and task management.

Additionally, a performance analysis was conducted using Intel VTune Profiler (formerly VTune Amplifier) to identify hotspots within the parallel implementation. The analysis focused on determining sections of code where most execution time was spent. The programs were run on the UHEM environment, allowing for the evaluation of performance metrics across different OpenMP configurations, including varying thread counts and scheduling strategies. This approach provided insights into computational bottlenecks and opportunities for optimization.

III. Experimental Work

A. Experimental setup

a) Hardware and Software Specifications:

- Local Computer:
  - Operating System: Windows 10 Home 22H2
  - CPU Name: AMD Ryzen 5 4600H with Radeon Graphics
  - CPU Frequency: 3.0 GHz
  - Logical CPU Count: 12
- UHEM:
  - Operating System: CentOS Linux 7 (Core)
  - Kernel Version: 3.10.0-327.36.1.el7.x86 64
  - CPU Name: Intel(R) Xeon(R) Processor (Broadwell)
  - CPU Frequency: 2.4 GHz
  - Logical CPU Count: 28

b) Input Dataset: The input dataset used is a matrix, as provided and described in the codes. The matrices are initialized with random values, and dimensions are typically powers of 2 to optimize performance.

B. Performance Metrics

TABLE I
Local Computer Execution Times

Matrix Size	Threads	Ex. Time (s)
1024	2	2.113
1024	4	1.143
1024	8	0.960
1024	16	0.904
2048	2	12.489
2048	4	8.011
2048	8	5.787
2048	16	5.012
4096	2	85.481
4096	4	53.590
4096	8	37.750
4096	16	31.853

TABLE II
UHEM System Execution Times with CPU Utilization

M. Size	Threads	Ex. Time (s)	CPU Time (s)	CPU Ut. (%)
1024	2	4.415	4.739	3.8
1024	4	1.551	4.980	11.2
1024	8	1.199	5.950	16.8
1024	16	1.068	6.800	21.2
2048	2	12.489	32.638	6.1
2048	4	10.874	37.120	11.6
2048	8	7.969	44.060	17.7
2048	16	5.539	47.250	27.8
4096	2	129.084	226.229	6.2
4096	4	73.563	247.294	11.5
4096	8	46.003	288.250	20.3
4096	16	41.113	357.790	26.4

C. Identification and Explanation of Performance Bottlenecks

TABLE III
UHEM System Hotspot Analysis for Matrix Size 4096

Threads	Top Hotspot Funct	% of CPU Time
2	strassen_multiply_internal	35.2
4	strassen_multiply_internal	34.1
8	strassen_multiply_internal	30
16	calloc	32.7

The table highlights key performance bottlenecks identified in the parallel implementation of Strassen's algorithm. These issues are categorized and explained in detail:

1. Memory Allocation Overheads

- The ‘calloc‘ function accounts for **32.7% of the total CPU time** (Table III) when executing with 16 threads and a matrix size of $n = 4096$.
- Frequent dynamic memory allocation for submatrices during recursive calls contributes significantly to this overhead.
- The recursive nature of Strassen's algorithm amplifies these inefficiencies, particularly for larger matrix sizes.

2. Imbalanced Workload Distribution

- As shown in Table II, CPU utilization remains **suboptimal across all thread counts**, even with increased parallelism.
- For instance, at $n = 4096$ and 16 threads, CPU utilization is **only 26.4%**, indicating poor task distribution and thread idling.
- Uneven workloads among threads lead to reduced overall efficiency.

3. Computational Hotspot in strassen_multiply_internal

- The `strassen_multiply_internal` function dominates the execution time, contributing to **30–35.2% of total CPU usage** across thread configurations (Table III).
- This function involves recursive multiplications and extensive memory access, making it a critical computational bottleneck.

4. Limited Scalability for Larger Matrices

- For larger matrix sizes (e.g., $n = 4096$), execution time decreases with increased thread counts but exhibits **non-linear scaling**.
- Synchronization overheads, such as those introduced by `#pragma omp taskwait`, and thread communication delays contribute to this bottleneck.

D. Scalability Analysis

UHEM System

The table below illustrates the execution time (*Ex. Time*), CPU time (*CPU Time*), and CPU utilization (*CPU Ut.*) for the UHEM system across varying matrix sizes and thread counts. Key observations include:

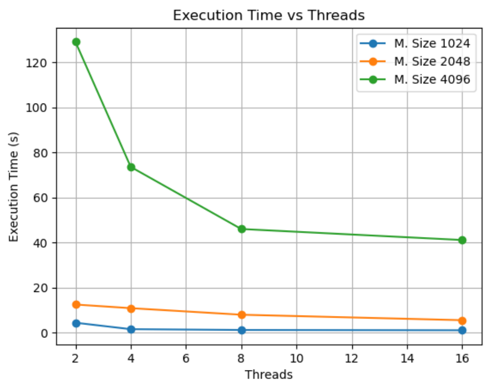

Fig. 1. Execution Time vs Threads for UHEM.

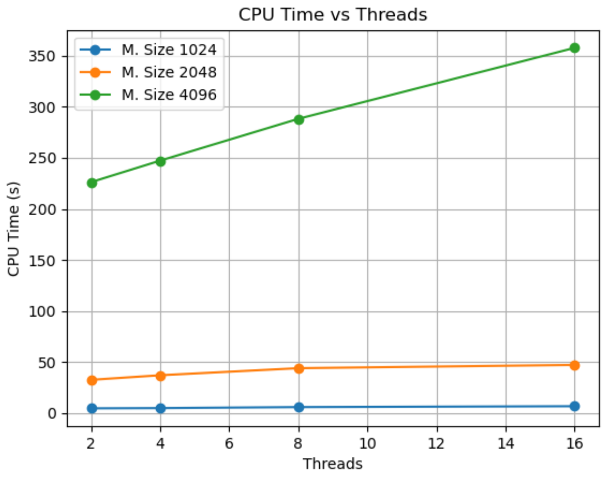

Fig. 2. CPU Time vs Threads for UHEM.

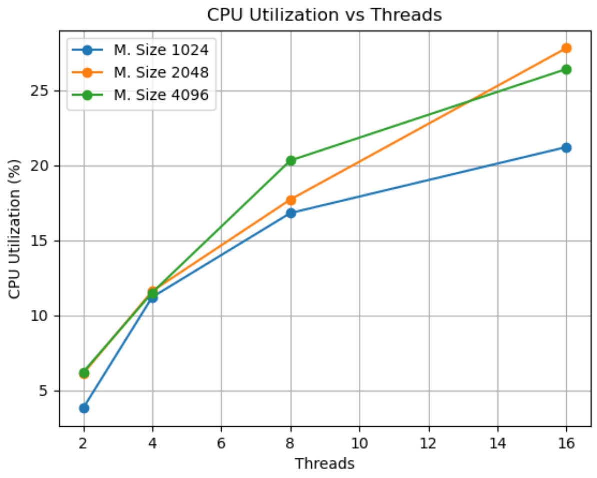

Fig. 3. CPU Utilization vs Threads for UHEM.

Execution Time vs. Matrix Size

- Execution time increases significantly with matrix size for a fixed thread count. For example:
  - At 2 threads, the execution time increases from **4.415 seconds** ($n = 1024$) to **129.084 seconds** ($n = 4096$).
- The recursive nature of Strassen's algorithm and increased computational complexity contribute to this scaling behavior.

Thread Scaling Efficiency

- Execution time decreases as thread count increases for all matrix sizes, indicating effective utilization of parallelism:
  - For $n = 1024$, the execution time decreases from **4.415 seconds (2 threads)** to **1.068 seconds (16 threads)**.
  - For $n = 4096$, the execution time decreases from **129.084 seconds (2 threads)** to **41.113 seconds (16 threads)**.

CPU Time Trends

- CPU time increases with thread count, reflecting the added synchronization overhead in OpenMP task scheduling:
  - For $n = 1024$, CPU time rises from **4.739 seconds (2 threads)** to **6.800 seconds (16 threads)**.
  - For $n = 4096$, CPU time rises significantly from **226.229 seconds (2 threads)** to **357.790 seconds (16 threads)**.

CPU Utilization

- Utilization increases with thread count but remains suboptimal, particularly for larger matrix sizes. For instance:
  - For $n = 4096$, 16 threads, CPU utilization is **26.4%**, reflecting inefficiencies in thread distribution and task granularity.

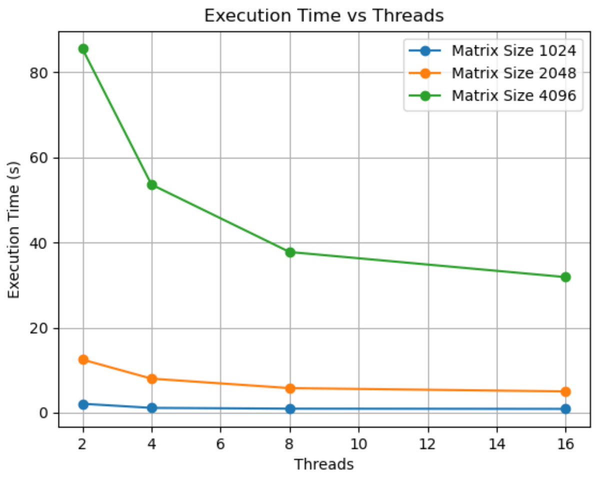

Fig. 4. Execution Time vs Threads for Local Computer.

LOCAL COMPUTER

The table provides execution times for the local computer, showing performance trends for varying matrix sizes and thread counts.

Execution Time vs. Matrix Size

- Execution time increases with matrix size, as expected, due to the cubic complexity of matrix multiplication:
  - For 2 threads, execution time rises from **2.113 seconds** ($n = 1024$) to **85.481 seconds** ($n = 4096$).
- This growth is consistent with the recursive calls and increased data size handled by the algorithm.

Thread Scaling Efficiency

- Increasing thread count reduces execution time across all matrix sizes, demonstrating effective parallelism:
  - For $n = 1024$, execution time decreases from **2.113 seconds (2 threads)** to **0.904 seconds (16 threads)**.
  - For $n = 4096$, execution time decreases from **85.481 seconds (2 threads)** to **31.853 seconds (16 threads)**.
- The scaling efficiency is more pronounced on the local computer due to a smaller number of logical cores, leading to better thread utilization per core.

E. Discussion/Comments About the Results

TABLE IV
EXECUTION TIME FOR SERIAL STRASSEN MULTIPLY

Matrix Dimension (n)	Time (seconds)
1024	2.594569
2048	18.263502
4096	128.844702

The performance results of the OpenMP implementation of Strassen's algorithm demonstrate clear benefits of parallelism for matrix multiplication, especially for larger matrix sizes. However, several key observations highlight areas of strength and opportunities for optimization:

Execution Time Reduction

- On both local and UHEM systems, increasing the thread count significantly reduces execution time for all matrix sizes tested.
- For example, the execution time for a 4096×4096 matrix on UHEM decreased from **129.084 seconds (2 threads)** to **41.113 seconds (16 threads)**.
- This trend confirms the effectiveness of task parallelism and data parallelism in distributing workloads across threads.

CPU Utilization Challenges

- Despite improvements in execution time, CPU utilization remains suboptimal, particularly for large matrices.
- At **16 threads** and matrix size 4096×4096, CPU utilization on UHEM was only **26.4%**.
- This inefficiency stems from:
  - Imbalanced workload distribution, where certain threads finish tasks earlier and remain idle.
  - Overheads from task synchronization.

Memory Allocation Overhead

- Memory allocation, particularly through the `calloc` function, is a major bottleneck.
- For a 4096×4096 matrix using 16 threads, `calloc` accounted for **32.7%** of CPU time.
- This issue is amplified by the recursive nature of the algorithm, which frequently allocates and deallocates submatrices.

Scalability Analysis

- While the implementation scales well with increasing thread counts, the scaling efficiency diminishes for larger matrices.
- This non-linear scaling is influenced by:
  - Synchronization overhead.
  - Thread communication delays.
- The local computer exhibited better scaling efficiency compared to UHEM, likely due to its smaller core count and reduced inter-thread communication overhead.

Comparison with Serial Implementation

- The serial implementation's execution time (e.g., **128.8447 seconds for a** 4096×4096 **matrix**) highlights the significant speedup achieved with parallelization.
- However, the recursive and memory-intensive nature of Strassen's algorithm poses inherent limitations, even with parallel optimization.

IV. FUTURE WORK

The implementation of the OpenMP-based Strassen's algorithm can be further extended and refined in the following ways:

Enhanced Memory Management

- Replace dynamic memory allocation with pre-allocated memory pools to reduce overhead caused by frequent allocations and deallocations during recursive calls.

Load Balancing Improvements

- Develop more sophisticated thread scheduling algorithms to ensure even distribution of workloads across threads, thereby enhancing CPU utilization.

Hybrid Parallelization Approaches

- Integrate OpenMP with MPI for distributed memory systems, enabling the algorithm to scale efficiently across clusters and supercomputers.

Algorithmic Refinements

- Investigate alternative recursive strategies or divide-and-conquer methods that could reduce computational overhead and improve scaling efficiency.

Hardware-Specific Optimizations

- Optimize the implementation for specific hardware, such as GPUs or many-core processors, leveraging libraries like CUDA or OpenCL to further accelerate computations.

Extended Profiling and Analysis

- Utilize advanced profiling tools to identify additional bottlenecks and optimize task synchronization, memory access patterns, and cache usage.

Real-World Applications

- Apply the optimized implementation to real-world problems, such as machine learning workloads or scientific simulations, to assess its practical performance and versatility.

Acknowledgment

This work was supported by UHeM under Project Number 4020502024i. We also thank our instructor, Dr. Işıl Öz, for her guidance and support throughout this study.

References

[1] Wikipedia contributors, "Strassen algorithm," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/Strassen_algorithm.

[2] TheAlgorithms, "Strassen matrix multiplication," GitHub Repository, [Online]. Available: https://github.com/TheAlgorithms/C-Plus-Plus/blob/master/divide_and_conquer/strassen_matrix_multiplication.cpp

[3] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming*, 1st ed., San Francisco, CA: Morgan Kaufmann, 2012.