INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Análisis y diseño de algoritmos avanzados

Gpo 604

Profesor:

Nezih Nieto Gutiérrez

Module 5 Activity

Sergio Gómez Guerrero A01571538

December 3rd, 2025

Problem 1: Parallel π Approximation (Multithreading Case Study)

The goal of the implemented code is to approximate the value of Pi using numerical integration via the Riemann sum method. It compares the performance of a sequential baseline against three parallel implementations: OpenMP, C++ std::thread, and POSIX Threads (pthreads). The script is designed to benchmark runtime, speedup, and numerical accuracy across increasing numbers of integration intervals and thread counts.

The problem is modeled as calculating the area under the curve f(x) = 4/(1+x^2) from 0 to 1. The integration interval is divided into N sub-intervals. The parallel implementations utilize domain decomposition, where the iteration space is partitioned among threads. Each thread maintains a private accumulator to sum its assigned slice, preventing data races, before reducing the partial sums into a global result.

The quality of the solution is determined by the absolute error between the computed approximation and the reference value of Pi. The results show that while all implementations achieve the same numerical accuracy for a given N, the parallel variants significantly reduce runtime. OpenMP provides the most concise implementation with competitive performance, while std::thread and pthreads offer lower-level control but require more verbose management of thread lifecycles and data partitioning.

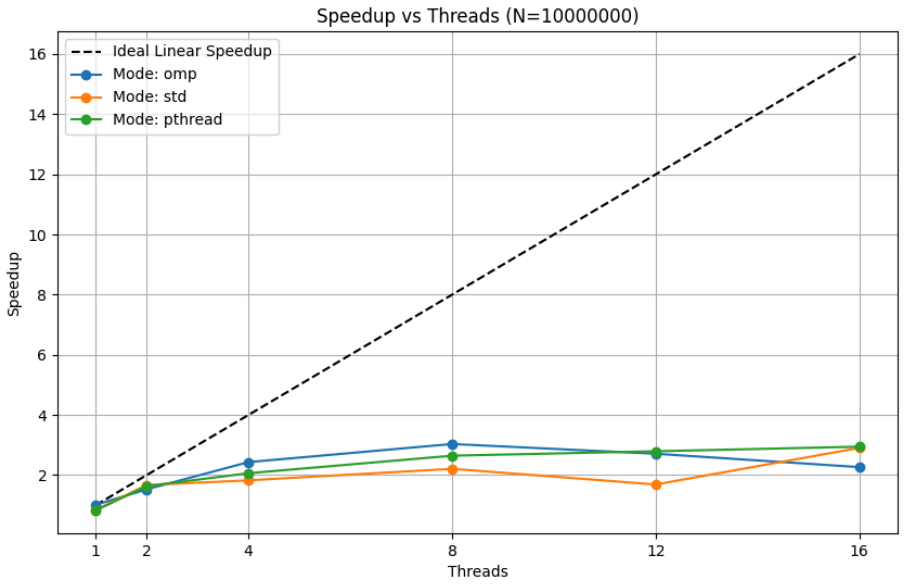

Fig 1.1: Speedup for N=10^7.

Mode	N	Threads	Time	PI_Approx	Error	Speedup
std	1000000	1	0.004535400	3.141592653589764	2.886579864e-14	1.1010
std	1000000	2	0.002844600	3.141592653589899	1.061373212e-13	1.7554
std	1000000	4	0.002377000	3.141592653589876	8.260059303e-14	2.1007
std	1000000	8	0.002182000	3.141592653589871	7.815970093e-14	2.2885
std	1000000	12	0.001885600	3.141592653589876	8.260059303e-14	2.6482
std	1000000	16	0.003561300	3.141592653589872	7.904787935e-14	1.4021
std	10000000	1	0.047369400	3.141592653589731	6.217248938e-14	0.8456
std	10000000	2	0.019460900	3.141592653589923	1.296740493e-13	2.0582
std	10000000	4	0.017122600	3.141592653589670	1.234568003e-13	2.3393
std	10000000	8	0.014220000	3.141592653589803	1.021405183e-14	2.8168
std	10000000	12	0.013318800	3.141592653589811	1.776356839e-14	3.0074
std	10000000	16	0.013318800	3.141592653589803	1.021405183e-14	3.0074
std	100000000	1	0.378671500	3.141592653590426	6.332712132e-13	0.9831
std	100000000	2	0.218503800	3.141592653589810	1.167954622e-13	1.7038
std	100000000	4	0.118037400	3.141592653589683	1.105782133e-13	3.1540
std	100000000	8	0.107916700	3.141592653589815	2.176037128e-14	3.4498
std	100000000	12	0.107350500	3.141592653589829	3.552713679e-14	3.4680
std	100000000	16	0.106988200	3.141592653589882	8.926193118e-14	3.4797
seq	1000000	1	0.004993400	3.141592653589764	2.886579864e-14	1.0000
seq	10000000	1	0.040055300	3.141592653589731	6.217248938e-14	1.0000
seq	100000000	1	0.372286700	3.141592653590426	6.332712132e-13	1.0000
pthread	1000000	1	0.005536300	3.141592653589764	2.886579864e-14	0.9019
pthread	1000000	2	0.003111000	3.141592653589899	1.061373212e-13	1.6051
pthread	1000000	4	0.001789800	3.141592653589876	8.260059303e-14	2.7899
pthread	1000000	8	0.002631000	3.141592653589871	7.815970093e-14	1.8979
pthread	1000000	12	0.002786900	3.141592653589876	8.260059303e-14	1.7917
pthread	1000000	16	0.002288600	3.141592653589872	7.904787935e-14	2.1819
pthread	10000000	1	0.042806600	3.141592653589731	6.217248938e-14	0.9357
pthread	10000000	2	0.024114300	3.141592653589923	1.296740493e-13	1.6611
pthread	10000000	4	0.014387900	3.141592653589670	1.234568003e-13	2.7840
pthread	10000000	8	0.014278800	3.141592653589803	1.021405183e-14	2.8052
pthread	10000000	12	0.017045000	3.141592653589811	1.776356839e-14	2.3500
pthread	10000000	16	0.014785900	3.141592653589803	1.021405183e-14	2.7090
pthread	100000000	1	0.352516900	3.141592653590426	6.332712132e-13	1.0561
pthread	100000000	2	0.206577500	3.141592653589910	1.167954622e-13	1.8022
pthread	100000000	4	0.135195800	3.141592653589683	1.105782133e-13	2.7537
pthread	100000000	8	0.103013700	3.141592653589815	2.176037128e-14	3.6140
pthread	100000000	12	0.115559900	3.141592653589829	3.552713679e-14	3.2216
pthread	100000000	16	0.112375400	3.141592653589882	8.926193118e-14	3.3129
omp	1000000	1	0.004558600	3.141592653589764	2.886579864e-14	1.0954
omp	1000000	2	0.004128100	3.141592653589764	2.886579864e-14	1.2096
omp	1000000	4	0.004407500	3.141592653589764	2.886579864e-14	1.1329
omp	1000000	8	0.004297700	3.141592653589764	2.886579864e-14	1.1619
omp	1000000	12	0.005163500	3.141592653589764	2.886579864e-14	0.9671
omp	1000000	16	0.006497100	3.141592653589764	2.886579864e-14	0.7686
omp	10000000	1	0.039358500	3.141592653589731	6.217248938e-14	1.0177
omp	10000000	2	0.037118900	3.141592653589731	6.217248938e-14	1.0791
omp	10000000	4	0.040424800	3.141592653589731	6.217248938e-14	0.9909
omp	10000000	8	0.038355500	3.141592653589731	6.217248938e-14	1.0443
omp	10000000	12	0.043404800	3.141592653589731	6.217248938e-14	0.9228
omp	10000000	16	0.052482200	3.141592653589731	6.217248938e-14	0.7632
omp	100000000	1	0.457240000	3.141592653590426	6.332712132e-13	0.8142
omp	100000000	2	0.363951700	3.141592653590426	6.332712132e-13	1.0229
omp	100000000	4	0.434112200	3.141592653590426	6.332712132e-13	0.8576
omp	100000000	8	0.492880600	3.141592653590426	6.332712132e-13	0.7553
omp	100000000	12	0.398579200	3.141592653590426	6.332712132e-13	0.9340
omp	100000000	16	0.378321700	3.141592653590426	6.332712132e-13	0.9840

Fig 1.2: Table of results.

The sequential algorithm has a time complexity of O(N) and space complexity of O(1). The parallel algorithms theoretically achieve a time complexity of O(N/P), where 'P' is the number of processors, though overhead and serial portions (Amdahl's Law) limit the actual speedup. Space complexity remains low, O(P), primarily for thread stack overhead.

Benchmarking was performed by measuring execution time and calculating speedup for N ranging from 10^6 to 10^8 and thread counts from 1 to 16. The results confirm that parallelization yields near-linear speedup for large N, but efficiency diminishes for small N due to thread creation overhead. Amdahl's Law analysis estimates the serial fraction of the code, highlighting the theoretical limits of parallel scaling.

Problem 2: Vectorized Matrix Calculator (SIMD Case Study)

The goal of the implemented code is to optimize element-wise matrix operations (addition, subtraction, multiplication, division) using hardware acceleration techniques. It compares a standard scalar C++ implementation against a vectorized version using AVX intrinsics (SIMD) and a hybrid version combining AVX with OpenMP multithreading. The benchmark evaluates the runtime performance improvements gained by exploiting both instruction-level parallelism and thread-level parallelism.

The matrices are represented as flattened 1D arrays to ensure memory contiguity, which is critical for cache performance and SIMD loading. The scalar version processes elements one by one. The SIMD implementation uses AVX registers (__m256) to process 8 floating-point numbers simultaneously per instruction. The hybrid SIMD+OpenMP implementation further distributes these vectorized chunks across multiple CPU cores.

The quality of the solution is measured by the execution time required to complete the operations. The results demonstrate that SIMD alone provides a significant speedup over the scalar baseline. The combination of SIMD and OpenMP also has high performance, although in this case SIMD alone seems to consistently perform better.

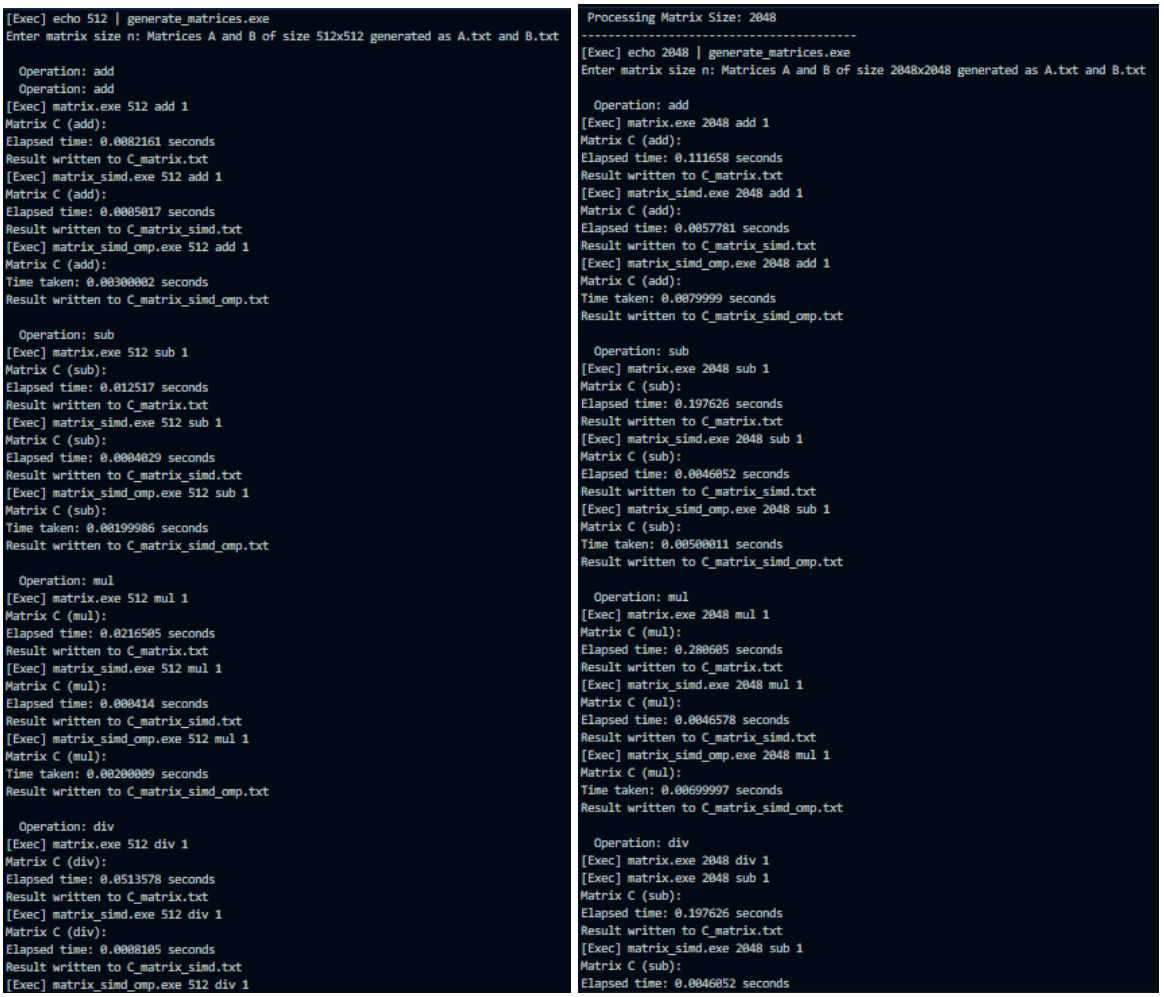

Fig 2.1 & 2.2: Results for Matrices of sizes 512 and 2048.

The scalar matrix operations have a time complexity of O(N^2), where 'N' is the matrix dimension. The SIMD implementation reduces the constant factor by the vector width (e.g., O(N^2/8)). The hybrid SIMD+OpenMP implementation further divides the work, approaching O(N^2 / (8 * P)). Space complexity is O(N^2) for storing the input and output matrices.

Benchmarking was performed on matrix sizes of 512, 1024, and 2048. The results highlight the massive performance gap between naive scalar code and optimized code. For large matrices, the hybrid approach theoretically dominates, although in testing the SIMD was the dominant approach. Furthermore, for smaller matrices the overhead of managing threads in OpenMP consistently makes the pure SIMD approach preferable.

Problem 3: CUDA Parallelization of a Graph Algorithm

The goal of the implemented code is to traverse a graph using Breadth-First Search (BFS) to visit nodes layer by layer, calculating the shortest path distance from a source node. It compares a sequential CPU implementation against a parallel GPU implementation using CUDA. The benchmark evaluates the trade-offs between CPU and GPU execution for graph algorithms, specifically focusing on runtime and speedup across varying graph sizes.

The graph is modeled using an Adjacency List for the CPU implementation and a Compressed Sparse Row (CSR) format for the GPU to optimize memory access patterns. The CPU version uses a standard queue-based approach. The GPU version employs a level-synchronous approach where a kernel is launched for each level of the BFS. Threads process nodes in the current frontier in parallel, atomically adding unvisited neighbors to the next frontier.

The results show that for the tested graph sizes (1,000 to 200,000 nodes), the CPU implementation consistently outperforms the GPU version (Speedup < 1.0x). This is like;y attributed to the significant overhead of memory transfers between host and device, kernel launch latency, and the irregular memory access patterns in the graph traversals, which lead to thread divergence and uncoalesced memory reads on the GPU.

```
Running BFS Tests and writing to bfs_results.csv...
V       CPU(ms) GPU(ms) Speedup
1000    0.283   1.712   0.17x
5000    1.41    2.86    0.49x
10000   2.98    4.60    0.65x
50000   15.23   22.40   0.68x
100000  33.78   50.22   0.67x
200000  85.47   104.08  0.82x
```

Fig 3.1: Results for BFS with CPU and GPU.

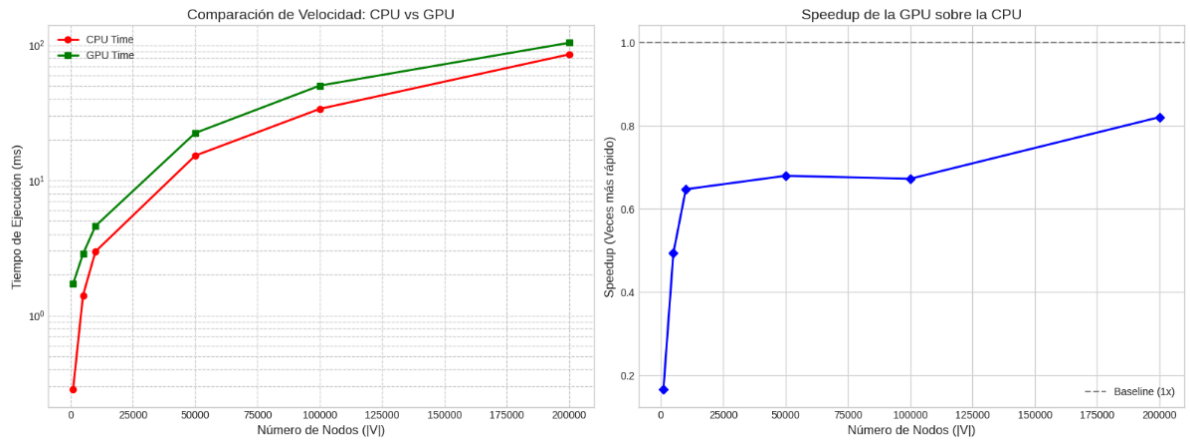

Fig 3.1: Graphed Results for BFS with CPU and GPU.

The BFS algorithm has a time complexity of O(V + E), where 'V' is the number of vertices and 'E' is the number of edges. The parallel GPU version theoretically reduces the time complexity by processing the frontier in parallel, but synchronization costs and memory latency often dominate for smaller or sparse graphs. Space complexity is O(V + E) to store the graph structure and auxiliary arrays (distance, frontiers).

Benchmarking was performed on random graphs with sizes ranging from 1,000 to 200,000 nodes. The results indicate that the GPU implementation suffers from high overhead for these graph sizes, achieving only 0.17x to 0.82x the speed of the CPU. However, the trend shows that as the graph size increases, the GPU's relative performance improves, suggesting that it may eventually surpass the CPU for massive graphs (millions of nodes) where the massive parallelism can outweigh the overhead.

Collaborative Discussion Members

Sergio Gómez Guerrero | A01571538
Angel Alexandro Angulo Prudencio | A00840313
Rutilo Alberto de la Pena Rodriguez | A01384647
Hector Aranda Garcia | A01178284