CS 301
High-Performance Computing

GAUSS ELIMINATION

Harsh Makwana (202001264)
Vivek Godhasara (202001451)

April 26, 2023

Contents

1 Introduction

In solving linear systems of equations, two common operations are Gaussian Elimination and Back Substitution. Parallelizing these operations with OpenMP and MPI can improve their performance significantly on multi-core CPUs and distributed computing clusters. OpenMP is a shared-memory parallelization library that allows multiple threads to run code on the same machine at the same time. In contrast, MPI is a message-passing library that allows for parallelism on a distributed computing system.

2 Hardware Details

- LAB207 Computer
  - Model Name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
  - CPU - 4
  - Socket - 1
  - Cores per Socket - 4
  - Size of L1 cache - 64KB
  - Size of L2 cache - 256KB
  - Size of L3 cache - 6MB
- Cluster
  - CPU - 16
  - Socket - 2
  - Cores per Socket - 8
  - Size of L1 cache - 64KB
  - Size of L2 cache - 256KB
  - Size of L3 cache - 20MB

3 Problem GAUSS ELIMINATION

3.1 Problem Statement

The problem is to solve a system of linear equations using Gaussian Elimination and Back Substitution. Given a matrix A of size n x n and a vector b of size n, the objective is to find the vector x of size n that satisfies the equation Ax=b. The solution involves two main steps: Gaussian Elimination and Back Substitution. In Gaussian Elimination, the matrix A is transformed into an upper triangular matrix using elementary row operations. In Back Substitution, the vector x is computed by solving the triangular system of equations. The goal is to parallelize these operations using OpenMP and MPI to improve performance on multi-core CPUs and distributed computing clusters. The implementation needs to ensure correct results while minimizing communication overhead and load imbalance.

3.2 Complexity

3.2.1 Complexity of serial code

The serial algorithm for Gaussian Elimination and Back Substitution has a complexity of $O(n^3)$, where n is the size of the matrix.

3.2.2 Complexity of parallel code

The complexity of the parallel algorithm for Gaussian Elimination and Back Substitution depends on several factors, including the number of processors used, the size of the matrix, and the distribution of the workload among the processors.

In general, the parallel complexity of Gaussian Elimination and Back Substitution can be estimated as $O(n^3/p)$, where n is the size of the matrix and p is the number of processors used. This assumes that the workload is evenly distributed among the processors and that communication overhead is negligible.

However, the actual complexity of the parallel algorithm may be higher due to overheads such as communication and synchronization. The efficiency of the parallel algorithm also depends on the efficiency of the parallelization strategy used and the optimization techniques applied to reduce overheads.

3.3 Theoretical Possible Speedup

The theoretical speedup for parallelization of Gaussian Elimination and Back Substitution can be calculated using Amdahl's Law. The maximum possible speedup is given by $1/((1-p)+(p/n^3))$, where p is the proportion of the algorithm that can be parallelized. For Gaussian Elimination and Back Substitution, p is typically high and can be close to 1. Therefore, a significant speedup can be achieved through parallelization.

3.4 Problems faced in parallelization

One of the main problems in parallelization is data dependency and race conditions. The parallel implementation needs to ensure that data dependencies are maintained and race conditions are avoided to produce correct results. Another challenge is load balancing, where the workload needs to be distributed evenly among the processors to ensure efficient use of resources.

3.5 Results

3.5.1 Parallel overhead time for row-oriented matrix

Problem size	Parallel overhead time
4	0.0000150
8	0.000001479
16	0.00004053
32	0.0001655
64	0.0016352
128	0.0075163
256	0.078020
512	0.396458
1024	3.125679

Table 1: Parallel Overhead Time vs Problem Size(HPC Cluster)

Problem size	Parallel overhead time
4	0.0000089
8	0.0000128
16	0.0000108
32	-0.0000138
64	-0.0002552
128	-0.002248
256	-0.0178264
512	-0.1394544
1024	-1.1042805

Table 2: Parallel Overhead Time vs Problem Size(Lab 207)

3.5.2 Parallel overhead time for column-oriented matrix

Problem size	Parallel overhead time
4	0.00001710
8	0.00000951
16	0.00005680
32	0.00008440
64	0.001830
128	0.004490
256	0.075800
512	0.491000
1024	5.06000

Table 3: Parallel Overhead Time vs Problem Size(HPC cluster)

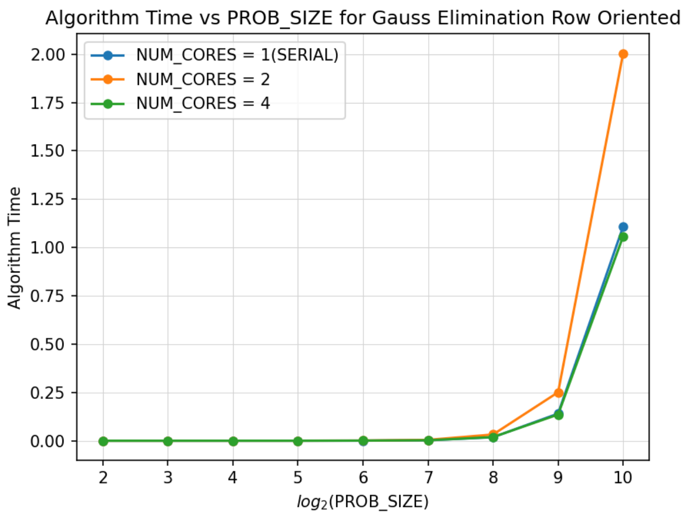
Problem size	Parallel overhead time
4	0.0000105
8	0.0000120
16	0.0000112
32	-0.0000132
64	-0.0002705
128	-0.002602
256	0.020029
512	0.194939
1024	-1.781331

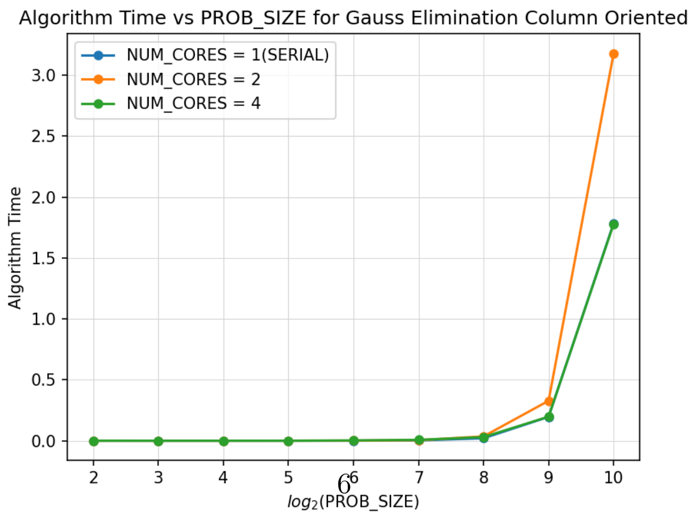
Table 4: Parallel Overhead Time vs Problem Size(Lab 207)

4 Graphs

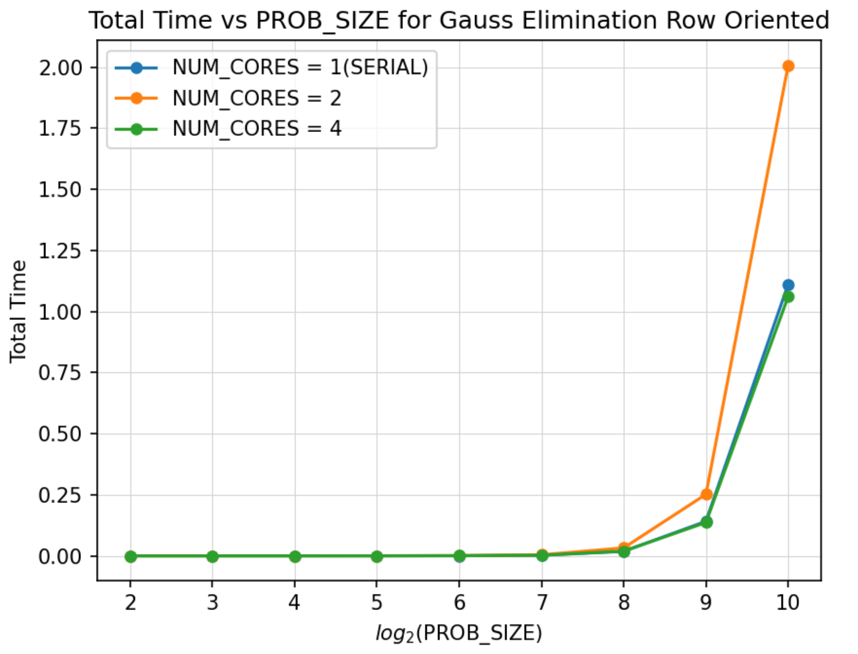
4.1 Lab207 Computer Raw Major and Column Major :

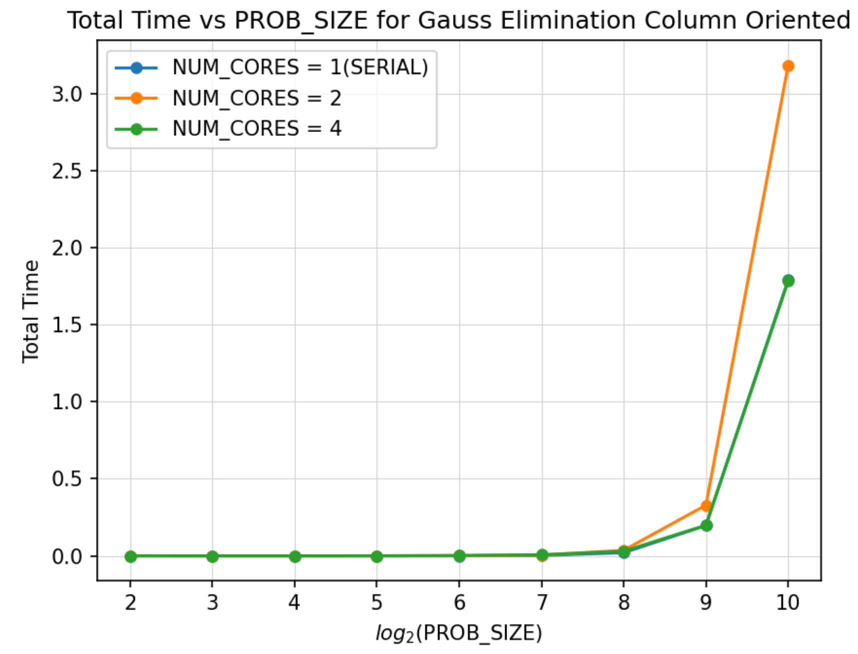
4.1.1 Algorithm Time vs Problem Size



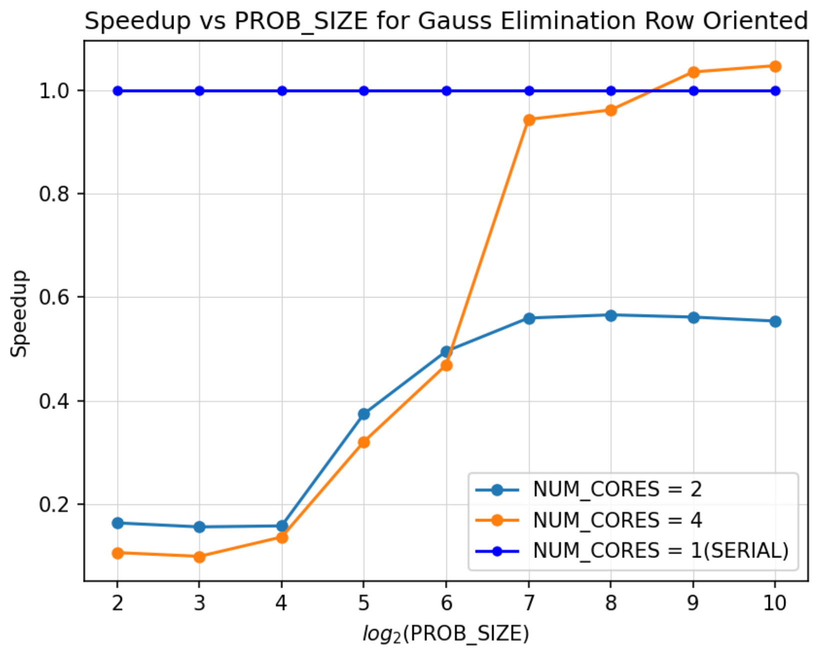


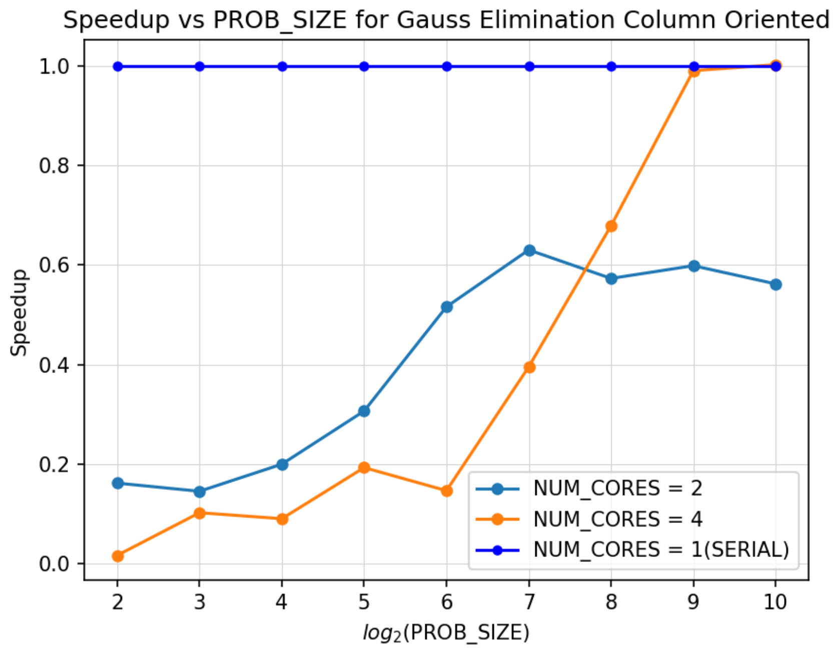
4.1.2 Total Time vs Problem Size



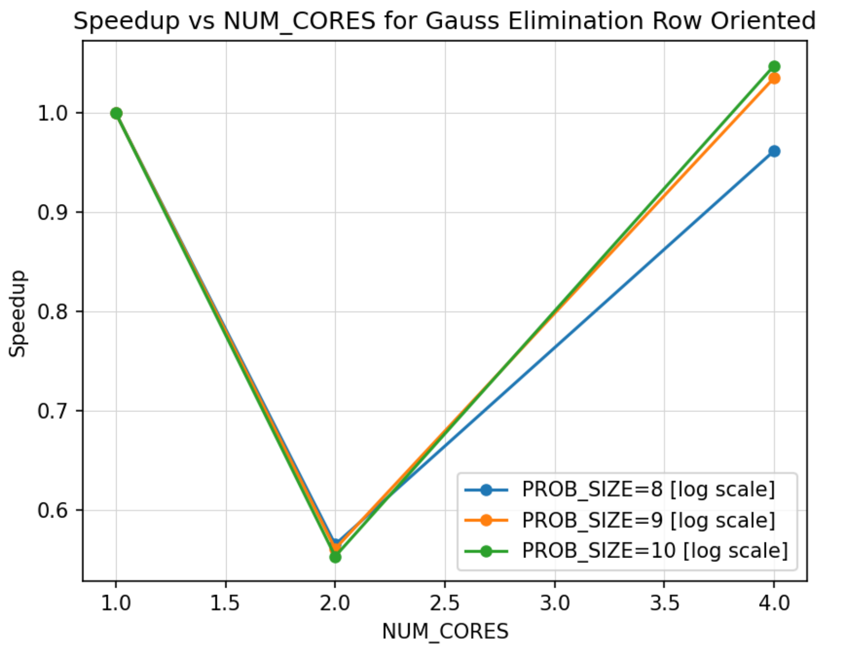


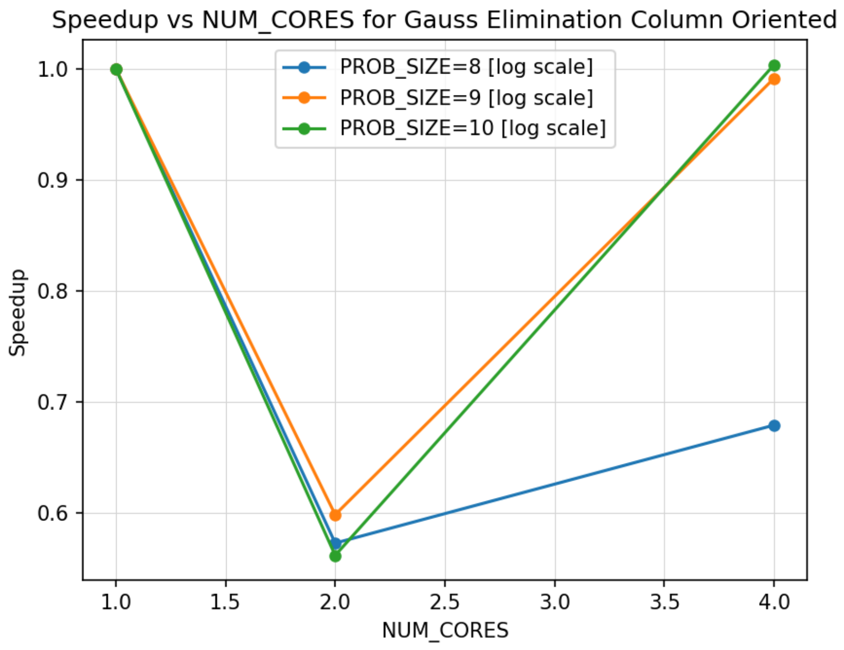
4.1.3 Speedup vs Problem Size





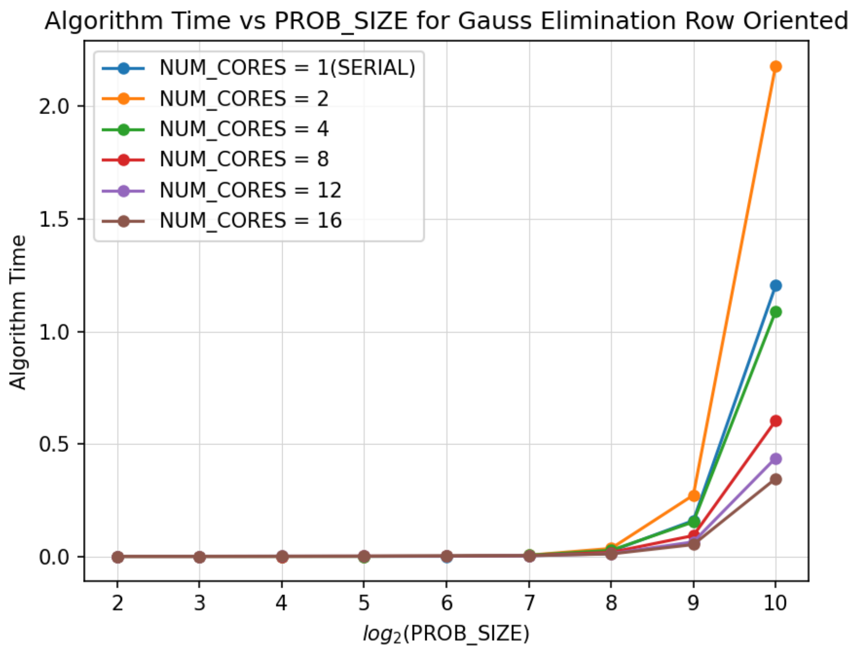
4.1.4 Speedup vs Number of Processors

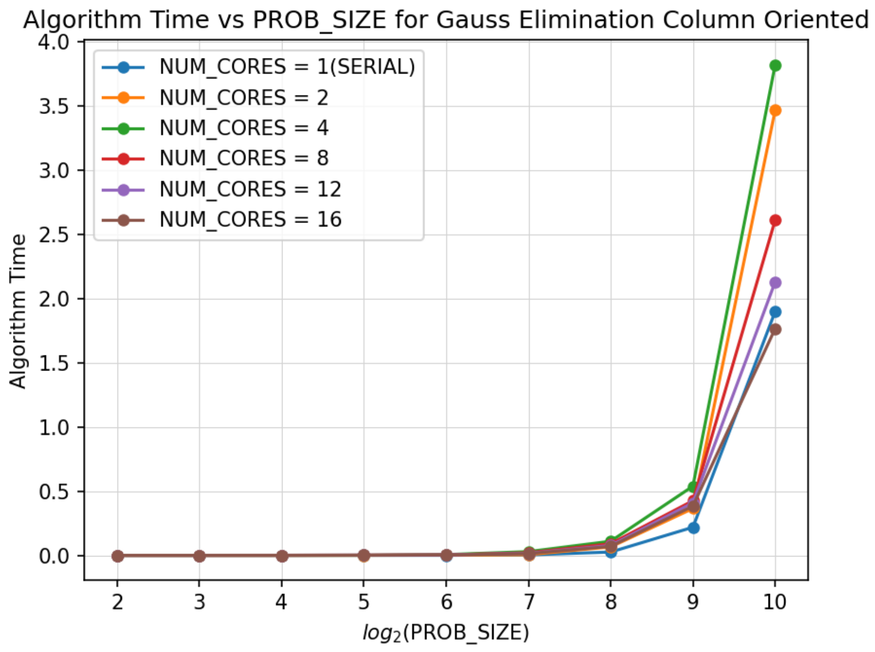




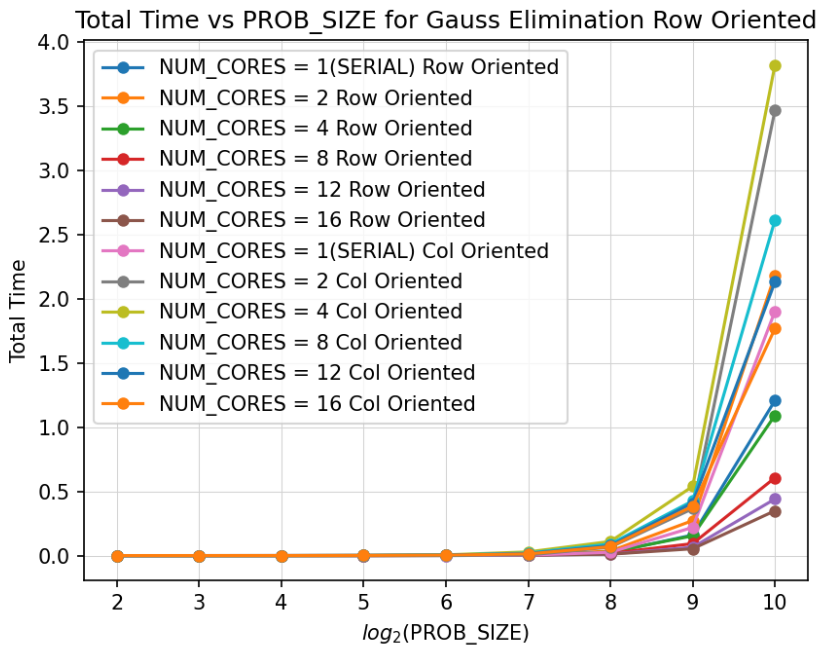
4.2 Cluster Raw Major and Column Major:

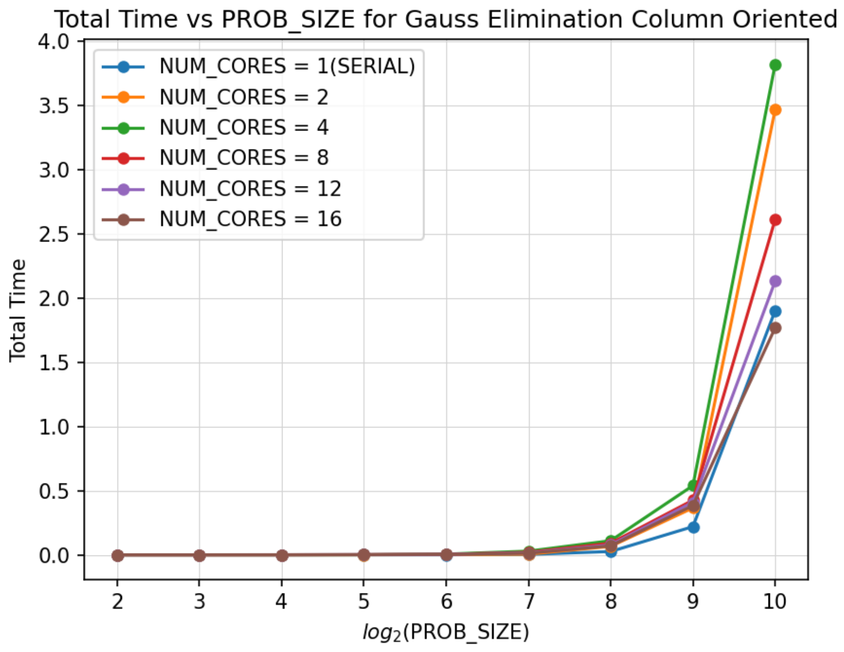
4.2.1 Algorithm Time vs Problem Size



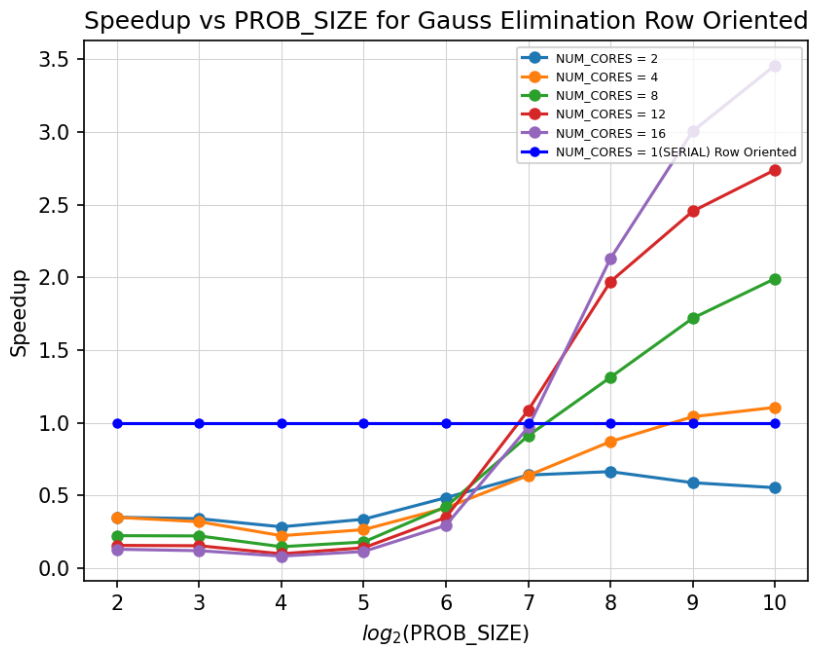


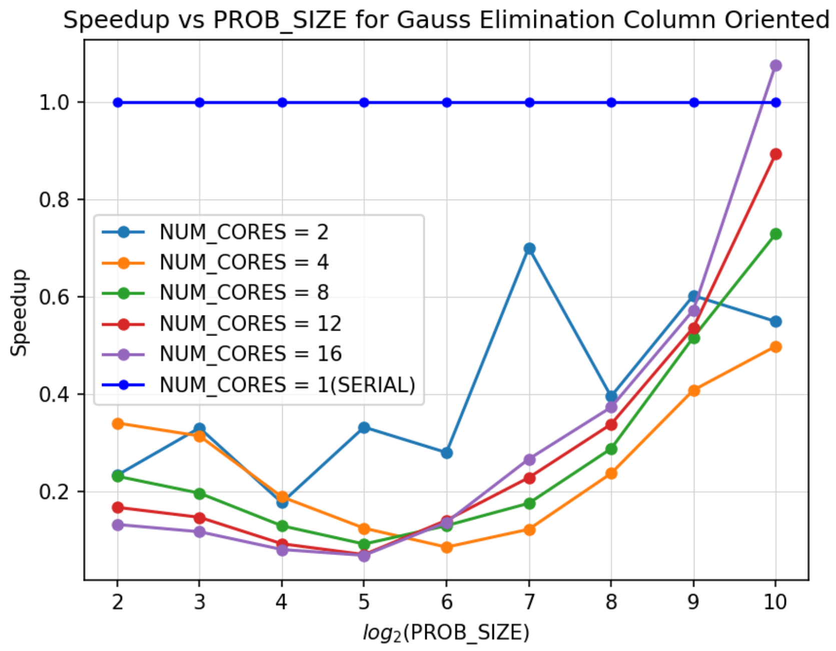
4.2.2 Total Time vs Problem Size



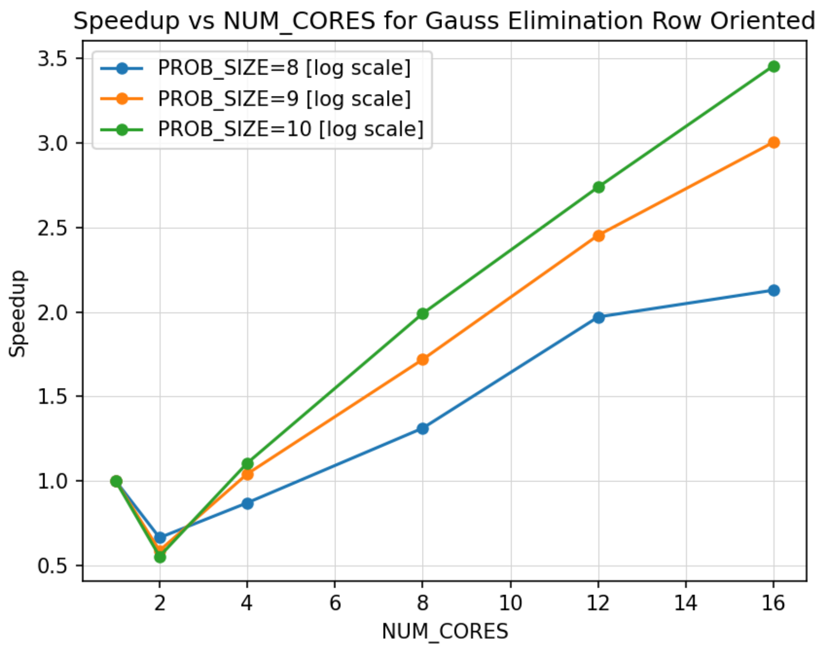


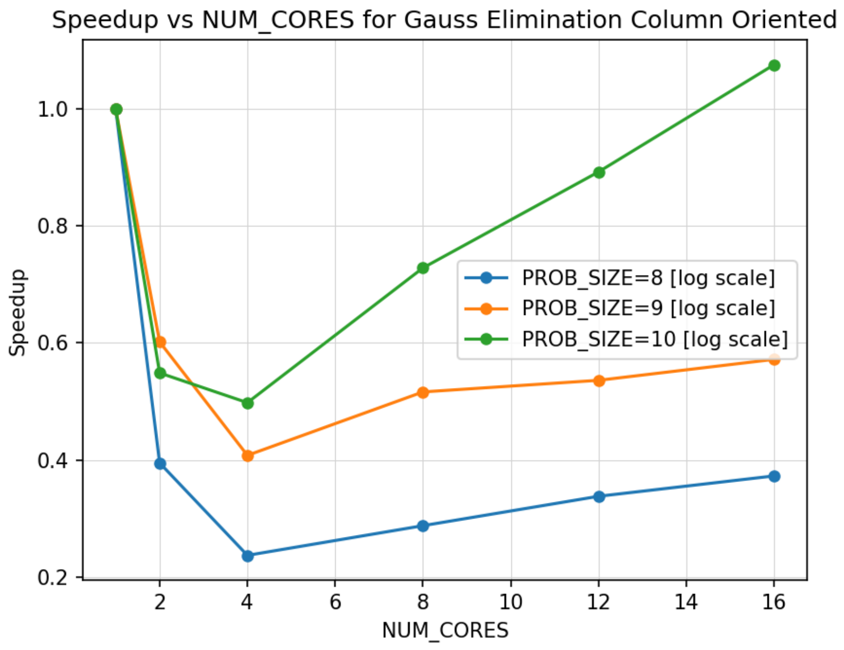
4.2.3 Speedup vs Problem Size





4.2.4 Speedup vs Number of of Processors





Both methods have advantages and disadvantages. The row-oriented approach enables easy access to and modification of individual row elements, but it may result in cache thrashing if the matrix is too large. The column-oriented approach enables easy access to and modification of individual column elements, but may result in more memory access due to cache misses.

In practise, the choice between raw-oriented and column-oriented approaches is determined by the application and hardware configuration. To determine the best solution for a given problem, carefully evaluate the algorithm's performance using both approaches.

5 Observation:

- The issue is compute bound, because as N increases, so does the time required for computation. 8 MB of memory would be required for a 1024x1024 matrix of double-precision floating-point numbers. This is a small amount of memory by modern standards, but it is larger than the cache size of most processors, which is typically in the range of a few MB. As a result, the processor would have to repeatedly fetch data from main memory, resulting in memory access latencies that could limit the algorithm's overall performance.
- Compute to memory access ratio is $\dfrac{N^3}{2N^3} = 1/2$.
- The mean execution time decreases as the number of processors (threads) increases, so we can say that as the number of processors increases, we get better performance (speedup). As a result, our algorithm is scalable.