CS 301
High-Performance Computing

QR Decomposition of a Matrix

Harsh Makwana (202001264)
Vivek Godhasara (202001451)

April 10, 2023

Contents

1 Introduction

In this lab, we have done both serial and parallel versions of QR decomposition, a classical linear algebra problem, using Gram-Schmidt process. QR Decomposition is a standard algorithm which is used in many applications such as calculating eigenvalues, solving system of linear equations and linear least squares problem. It can be also useful in image processing and machine learning applications such as automatic removal of an object from an image.

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 QR_DECOMPOSITION

3.1 Problem Statement

QR Decomposition is a standard algorithm which is used in many applications such as calculating eigenvalues, solving system of linear equations and linear least squares problem. It can be also useful in image processing and machine learning applications such as automatic removal of an object from an image. In this assignment we will use the Gram-Schmidt process to compute the decomposition.

3.2 Implementation Details

3.2.1 Description about the Serial implementation

The following is a basic outline of a serial algorithm for Gram-Schmidt for QR Decomposition:
1. We start with a nn 2-D matrix with real entries, whose columns represent a vector of dimension n.
2. In the i^{th} iteration of the Gram-Schmidt algorithm, i^{th} vector is normalized to unit length, and the subsequent vectors ($i+1$ to n) are updated in that iteration by subtracting the multiple of the i^{th} vector, so as to obtain the orthogonal matrix.
3. The entries of the upper triangular matrix, r_{ij} is the projection of the j^{th} vector on i^{th} vector.

$$r_{ij} = \frac{a_i^T a_j}{\sqrt{a_i^T a_i}}, \;\; j = 1...n$$

$$a_j \;\; a_j - r_{ii} a_i, \;\; j = i+1...n$$

3.2.2 Description about the Parallel implementation

The following is a basic outline of a parallel algorithm for block matrix multiplication using OpenMP:
1. Divide the matrices A, B, and C into smaller sub-matrices or blocks.
2. Distribute the blocks of matrix C among the available threads using OpenMP parallelism.
3. For each block in the assigned portion of matrix C, calculate the corresponding block using the sub-matrices from A and B.
4. Multiply the blocks from A and B using a nested loop over the subscripts of the sub-matrices.
5. Add the resulting sub-matrix to the corresponding block in the assigned portion of matrix C.
6. Synchronize the threads to ensure that all blocks in matrix C are properly calculated.

3.3 Complexity

3.3.1 Complexity of serial code

There are 3 loops each running over N. These are nested in each other so the total complexity is $O(N^3)$.

3.3.2 Complexity of parallel code

The work related to the outer loop is divided equally among all the threads which can run concurrently so the time taken would be $O(\frac{N}{p})O(N^2) = O(\frac{1}{p}N^3)$ since the inner loops take $O(N^2)$.

3.3.3 Cost of Parallel algorithm

Since the entire code is parallelizable, the total parallel cost would be $pO(\frac{N^3}{p}) = O(N^3)$

3.3.4 Theoretical Speed Up

Since the entire code is parallelizable, the theoretical speedup would be $\frac{O(N^3)}{O(\frac{N^3}{p})} = p$.

3.4 Graphs:

3.4.1 LAB207 Computer perfomance

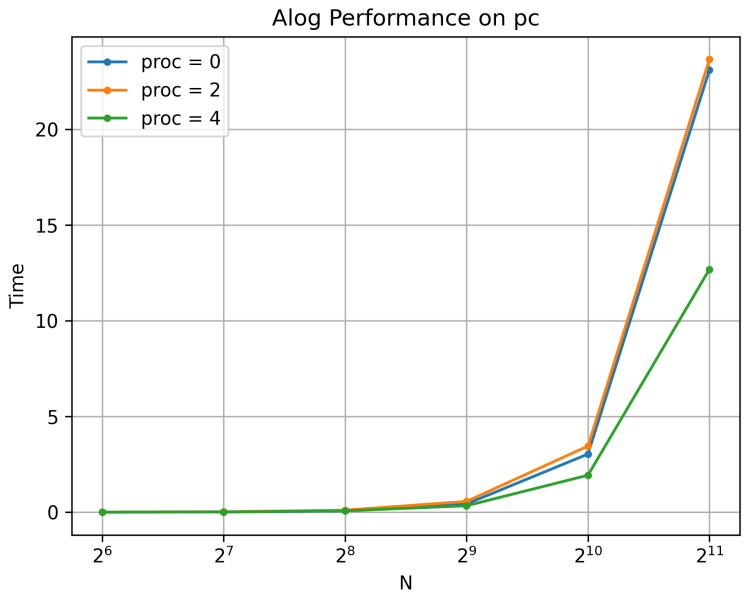

Figure 1: Algorithm Time vs Problem Size

3.4.2 LAB207 Computer speedup

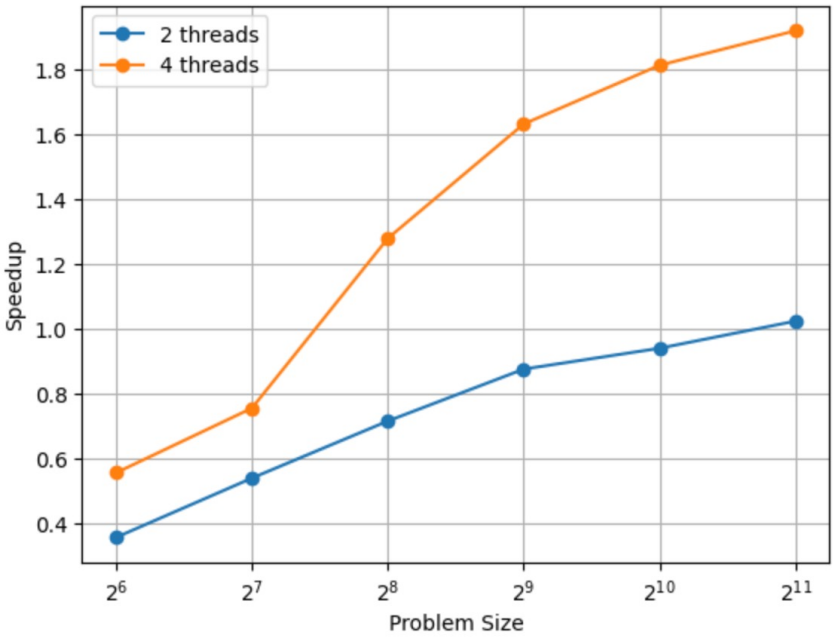

Figure 2: Algorithm Time vs Problem Size

3.4.3 Cluster perfomance

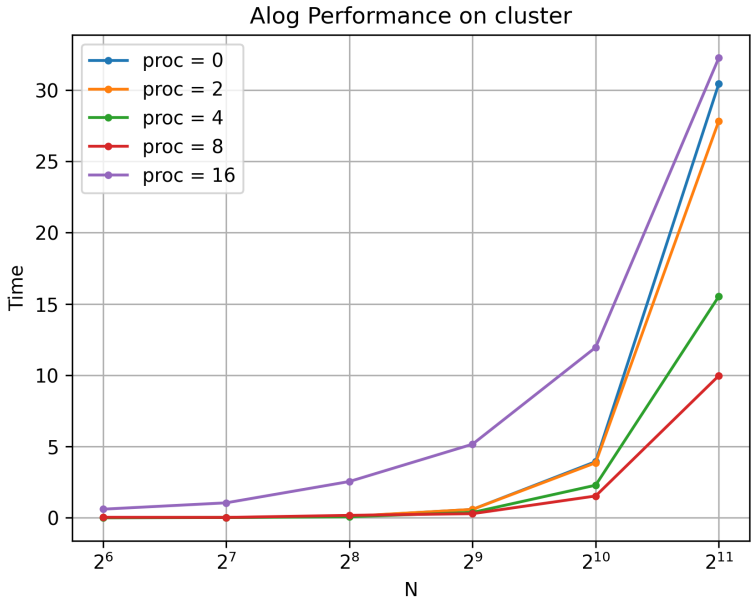

Figure 3: Algorithm Time vs Problem Size

3.4.4 Cluster speedup

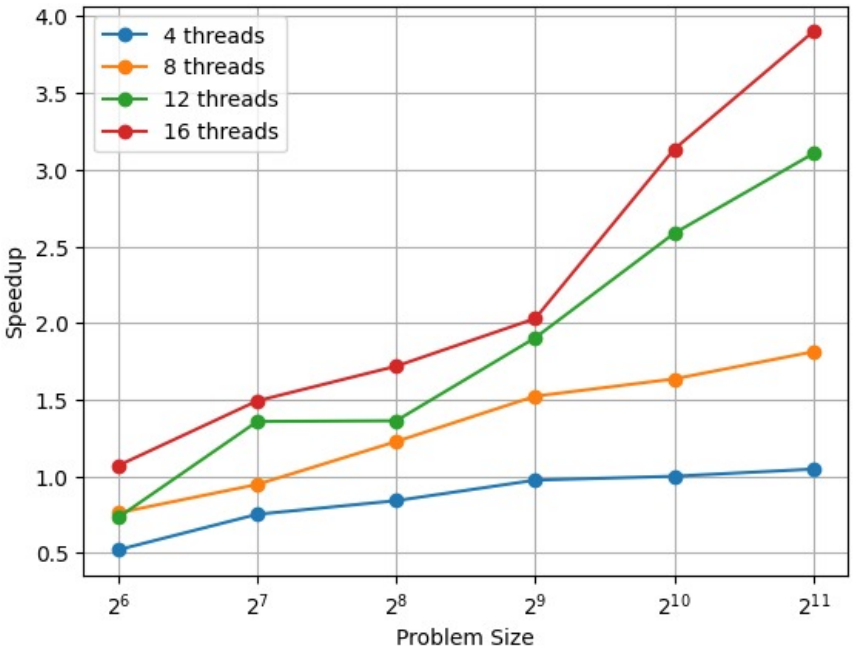

Figure 4: Algorithm Time vs Problem Size