A PROJECT REPORT ON

“Parallelization of Regression Analysis using OpenMP”

COURSE:

CS 301 : High Performance Computing

UNDER THE GUIDANCE OF
PROF. BHASKAR CHAUDHURY

Authors:

Ujaval Bhatt	201501403
Smit Thakkar	201501440

1) Introduction

This is the generation of data. We find data all around our self.We use these data to understand any system, find the underlying patterns and predict the future value based on it. We need to train our models using the data to predict the output.This training takes huge amount of time as size of data increases. So we are trying to reduce the time of training process by doing some sort of computation parallel using openMP. We are doing regression analysis, which is fundamental of machine learning and useful in many areas.We are parallelizing the linear regression code using openMp.

2) Problem Statement

Regression analysis contains linear regression and logistic regression. In linear regression we find the relationship between output variable and input variable.

- Suppose X is input variable which has n features.

  $X = [x_1, x_2, x_3, ..., x_n]$

- And Y is output variable containing singular value or vector.

  $Y = [y]$

- You need to find out relationship between Y and X, as Y is the function of X.

$$Y = f(x)$$

$$y = f(x_1, x_2, x_3 ...., x_n)$$

- In linear regression we y is linearly dependent on all the input variables,

So ,

$$y = \theta_1 * x_1 + \theta_2 * x_2 + \theta_3 * x_3 + ... + \theta_n * x_n$$

Where θ=[θ1,θ2,θ3,......θn], which represents the relationship between X and Y.

- **Example:**

Data is given as:

```
X=[1 3 4       Y =[0            m=3
   3 1 6          -2            n=3
   2 3 1]          4]
```

X1 = (1,3,4) , X2 = (3,1,6) ,X1 = (2,3,1)

And

Y = [y1,y2,y3] = [0,-2,4];

So from the data Y can be defined as,

$$y = 1 * x_1 + 1 * x_2 - 1 * x_3;$$

So , our θ would be θ=[1,1,-1];

- **Algorithm:**

Input:

Matrix X: input matrix

Dimensions: m x n

m = number of examples

n = number of features

Matrix Y: output matrix

Dimensions: m x 1

Output:

Matrix θ:

Dimensions : m x 1

θ would be find in a way such that

$$X*\theta \approx Y$$

Logic:

We are using extreme learning algorithm to find the value of θ.

We want to get X*θ nearly equals to Y so,

$$X*\theta = Y$$

$$\theta = X^{-1} * Y$$

X is $m\ x\ n$ matrix, which is rectangular matrix. Inverse doesn't exist for rectangular matrices.So we are finding **Moore–Penrose inverse (pseudo inverse) of matrix X.**

$$X^{+} = (X^{T} * X)^{-1} * X^{T}$$

Where X^{T} = Transpose of matrix X

$X^{T} * X$ is now square matrix with dimensions n * n.

We are finding inverse of this matrix using Gaussian elimination process.

So by doing this we get,

$$\theta = X^{+} * Y$$

$$\theta = (X^{T} * X)^{-1} * X^{T} * Y$$

So θ obtained by this process resembles the relationship between X and Y.

- **Scope of parallelism:**

Processes (step by step)	Complexity
1. Find X^T	m x n
2. Do A = $X^T * X$	n^2 x m
3. Find B= A^{-1}	n^3
4. Find C = $B * X^T$	n^2 x m
5. Find $C * Y$	m x n

Steps 2,3 and 4 are taking much amount of computation. So we can parallelize these 3 steps.

Steps 2 and 4 are matrix multiplication, Which can be parallelized using different clauses provided by OpenMP (like *#pragma omp for* and *#pragma omp schedule*)

Step 3 is matrix inversion by Gaussian Elimination process

Sequential Algorithm Gaussian Elimination Phase:

A = input matrix , I = identity matrix

```
For i = 1 to n, do
    a) If A[i,i] = 0 and A[m,i] = 0 for all m > i, conclude that $A^{-1}$ does not exist and halt the algorithm.
    b) If A[i,i] = 0 and A[m, i] ≠ 0 for some smallest m > i, interchange rows i and m in the array A and in the array I.
    c) Divide row i of A and row i of I by A[i, i]. That is, let scale = A[i, i] and then for j = 1 to n, replace A[i, j] by A[i, j]/scale.
    d) Now we have A[i, i] = 1. If i < n, then for r > i, subtract A[r, i] times row i from row r in both the arrays A and I.
```

Steps c and d contains 2 for loops running for n iterations and can be parallelized.

- **Parallelization strategy**

  Steps 2 and 4 can be parallelized in these many ways using clauses provided by openMp

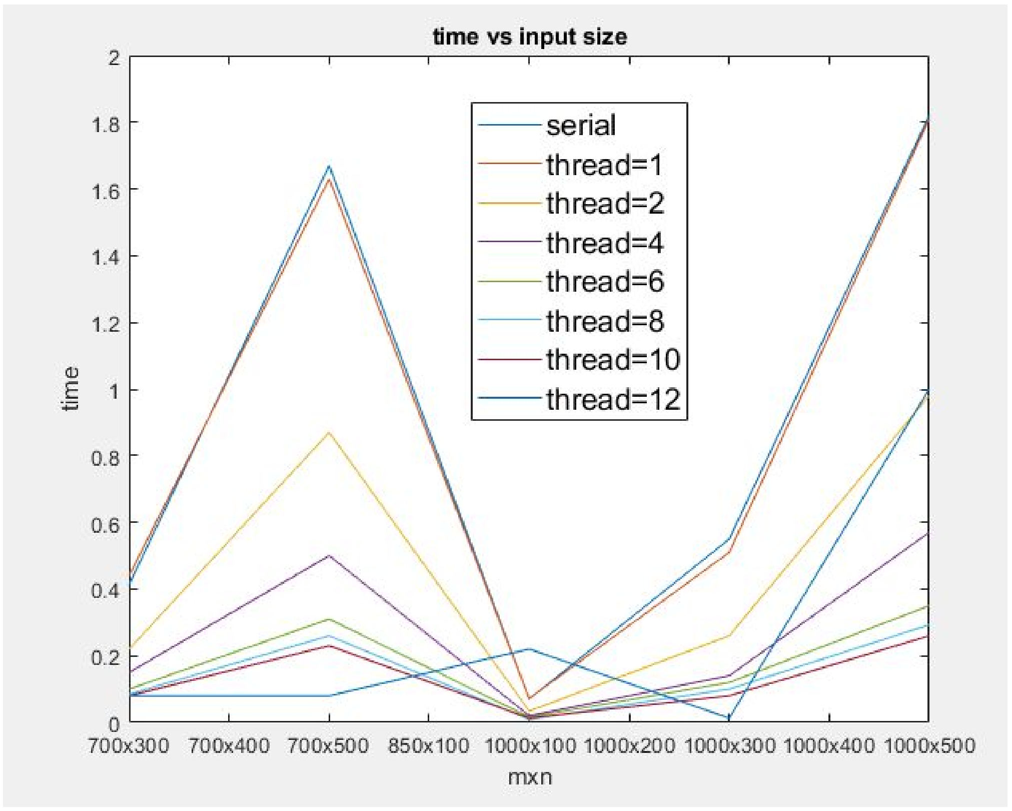
  - #pragma omp parallel for
  - #pragma omp parallel for schedule(static,chunksize)
  - #pragma omp parallel for schedule(dynamic)
  - #pragma omp parallel for schedule(guided)

  In step 3, c and d can be parallelized in 2 ways.

  1. Row wise parallelism
     a. #pragma omp parallel for
     b. #pragma omp parallel for schedule
  2. Column wise parallelism
     a. #pragma omp parallel for
     b. #pragma omp parallel for schedule

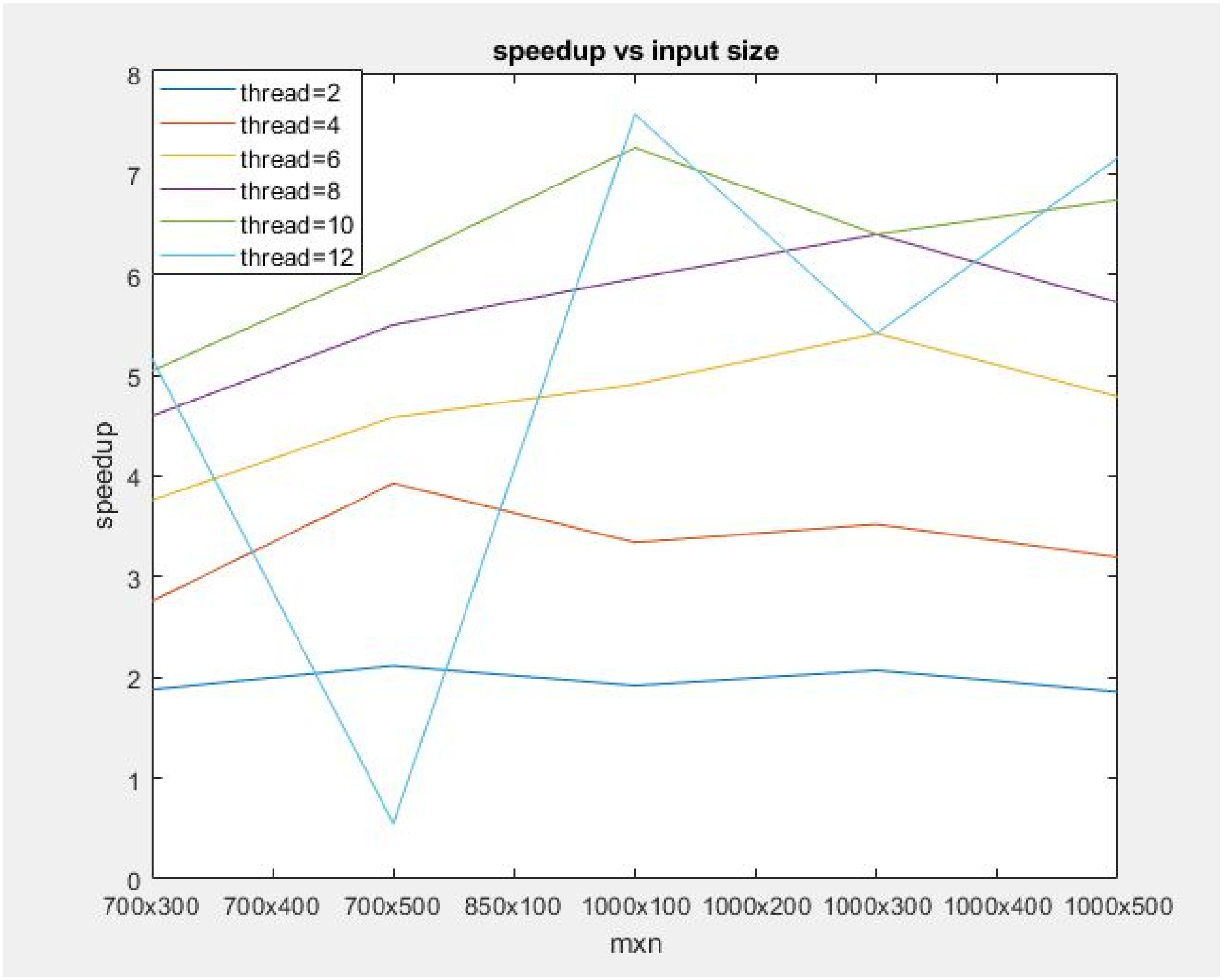
- **Graphs:-**

1. **Time Vs. Input size**.



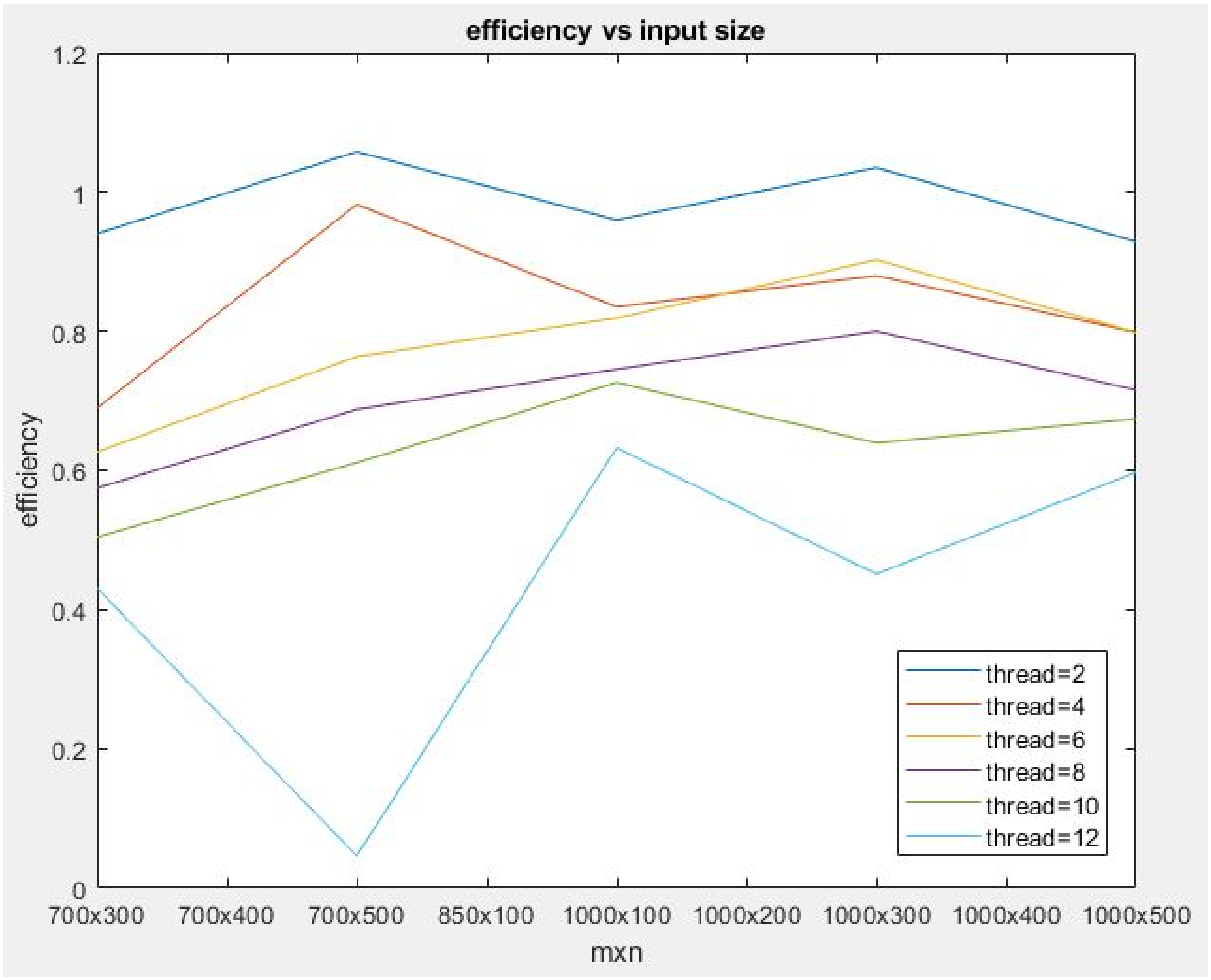
1. As the input size increases the number of computation increases hence the time increases.
2. The time of running is dependent on the dimension of the matrix .The m and n. The time is largely dependent on the value of the n because the main algorithmic calculation of the code is dependent on the value of the n.
3. But for some cases time depends upon both the value m and n, as shown in complexity section that matrix multiplication takes $m * n^2$ times so whenever m>>n then time majorly depend upon value of m

2) Speedup vs. Input size



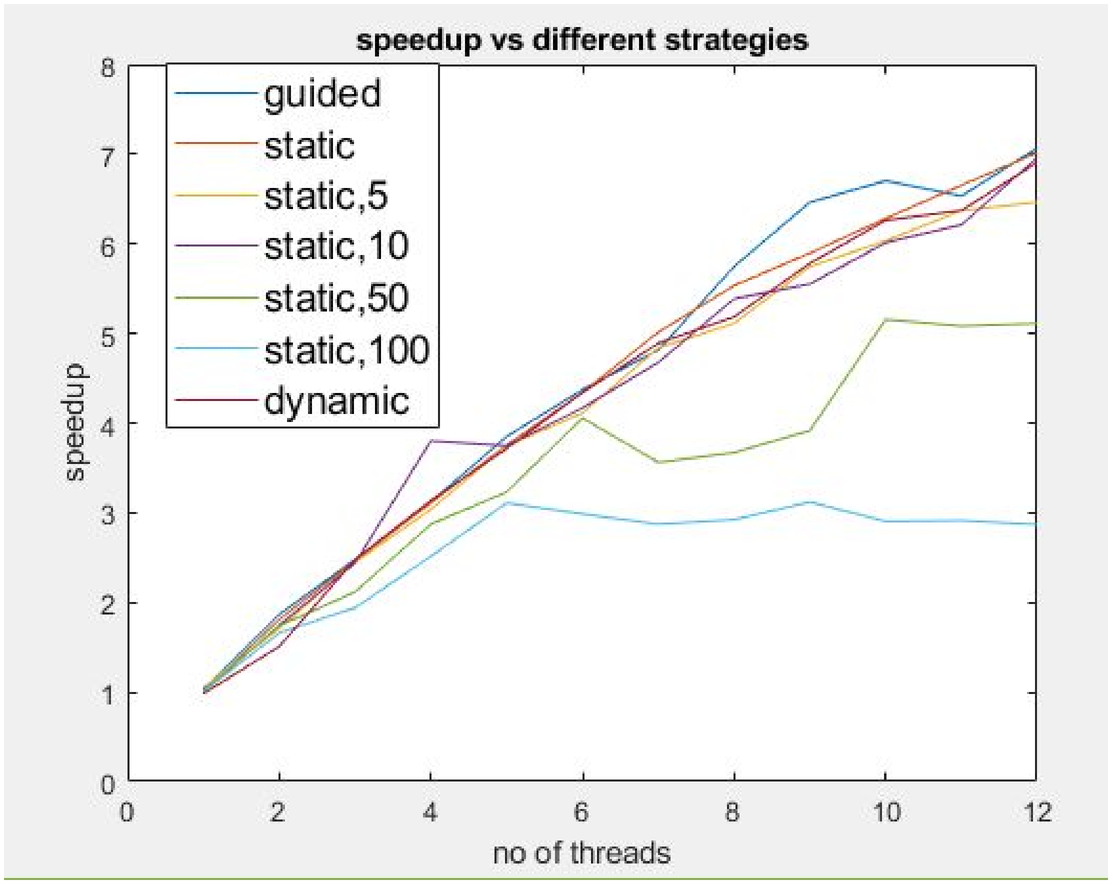
1. As the number of threads increases for the same problem size speedup increases.
2. We have not get the good speedup for the large value of number of threads because there is loop dependency in the algorithm So there is always amount of serial part in the code which we cannot parallelize so we can not get speedup near to number of cores.

3) Efficiency vs. Input size



1. For all the input size we have get the efficiency nearly equal to 1.
2. For more number of threads we can not get the efficiency nearly to 1 because there is is some amount of serial part in the code.

4) Comparison of different techniques:-



- We are getting equal speedup for schedule guided, static with default chunk size =1 , and dynamic.
- When we change the chunksize, speedup changes too. Higher Chunksize led to lower speedup.This basically happens because of value of n , which is no more than in order of 100.Lower chunk values allows threads to compute lesser and load will be balanced equally.