INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Introduction to Parallel Scientific Computing (CSE-504)

Dr. Pawan Kumar

Parallel Image Compression and Decompression using PCA.

***Team*(5)**

Vishal Bidwatka (2018201004)

Kshitij Paliwal (2018201063)

Sandeep Kumar Gupta (2018201076)

Contents

1 Introduction

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components. If there are n observations with p variables, then the number of distinct principal components is min(n-1,p). This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. The resulting vectors (each being a linear combination of the variables and containing n observations) are an uncorrelated orthogonal basis set. PCA is sensitive to the relative scaling of the original variables.

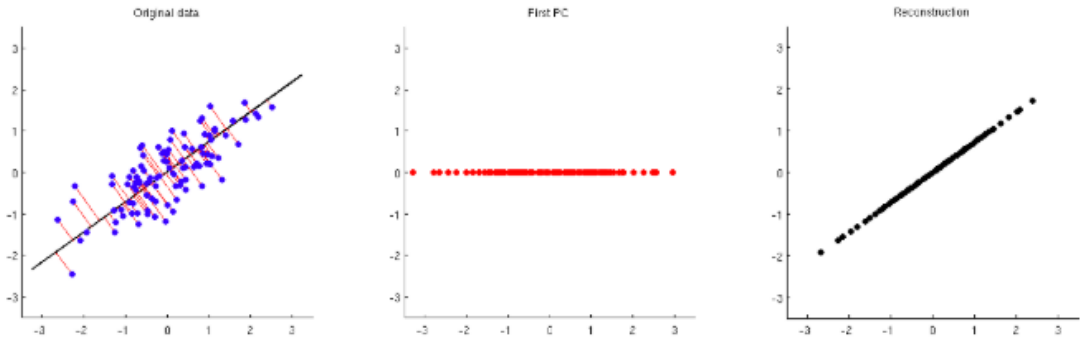

Figure 1: PCA Intuition

To compute the PCA, we have to maximize the variance. To do this, we require the maximum eigen values and vectors of the covariance matrix. This can be shown by minimizing the standard lagrangian formed for the above problem. So therefore, to compute the PCA, we are required to find the eigen values and vectors of the covariance matrix. This can be computed by finding the SVD of the input matrix

2 Jacobi Methods

Jacobi Methods are a highly parallelizable algorithm that can be used to solve the symmetric eigen value problem. This can be used to compute the SVD as it involves

computing the eigen values and vectors of covariance matrix. The algorithm works by applying 'Given's rotations' on the matrix and reduce the off-diagonal elements to zero. Since the rotators are orthogonal, the covariance matrix is similar to the diagonal matrix formed on convergence, and hence the diagonal matrix contains the eigen values of the covaiance matrix and the eigen vectors can be obtained by multplying by the rotators.

The idea behind the Jacobi method is to reduce the quantity,

$$off(A) = \sum_{i=1}^{n} \sum_{j=1, j!=i}^{n} a_{ij}^2$$

i,e, the norm of the non-diagonal elements. The tools for doing these are rotators of the form -

$$J(p,q,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ p \\ \\ q \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & p & & q & & \end{matrix}$$

which are called the Jacobi or the Given's rotators.

The basic step in the Jacobi method involves (1) choosing an index pair (p,q) that satisfies $1 \leq p \leq q \leq n$, (2) compute cosine, sine pair such that,

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

and (3), overwritting A whith $B = J^T AJ$. Its an impotant observation that B agrees with A except in rows and columns p and q. Moreover, Frobenious norm is preserved by orthogonal transformations and hence we find that,

$$\begin{aligned} \text{off}(B)^2 &= \| B \|_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= \| A \|_F^2 - \sum_{i=1}^{n} a_{ii}^2 + (a_{pp}^2 + a_{qq}^2 - b_{pp}^2 - b_{qq}^2) \\ &= \text{off}(A)^2 - 2a_{pq}^2 . \end{aligned}$$

It is in this sense that we move closer to diagonal form after each Jacobi step.

2.1 The 2-by-2 Symmetric Schur Decomposition

To say we diagonalize the abve form is to say that,

$$0 = b_{pq} = a_{pq}(c^2 - s^2) + (a_{pp} - a_{qq})cs.$$

Let us define,

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}} \quad \text{and} \quad t = s/c$$

and conclude that $t = tan(\theta)$ solves the quadratic,

$$t^2 + 2\tau t - 1 = 0\,.$$

Now it is easy to find s and c.

3 Parallel Jacobi Methods

An important observation that allows us to parallelize the method is that a (p,q) rotation affects only the p and q rows and columns of A.

To illustrate this, suppose n = 4, and group the six subproblems into three groups as follows :

$$\begin{aligned} rot.set(1) &= \{(1,2),(3,4)\} \\ rot.set(2) &= \{(1,3),(2,4)\} \\ rot.set(3) &= \{(1,4),(2,3)\} \end{aligned}$$

Now, all the rotations within each of the set is non-conflicting Hence, they can be carried out in parallel.

4 Dataset

For this project,we have used MNIST dataset, which is one of the standard datasets.

5 Results

5.1 Reconstructed Images

5.1.1 For Image of 8

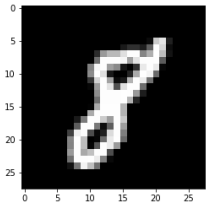

Figure 2: Original image

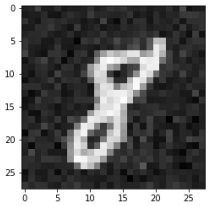

Figure 3: 10 % loss

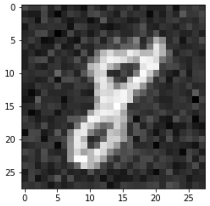

Figure 4: 20 % loss

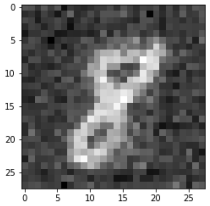

Figure 5: 40 % loss

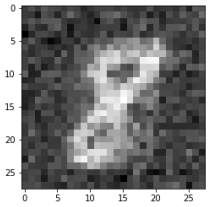

Figure 6: 60 % loss

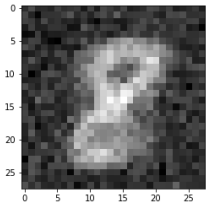

Figure 7: 80 % loss

5.1.2 For image of 0

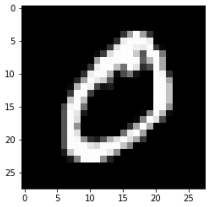

Figure 8: Original image

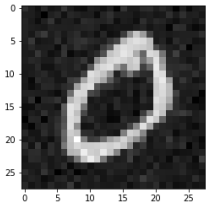

Figure 9: 10 % loss

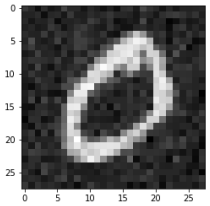

Figure 10: 20 % loss

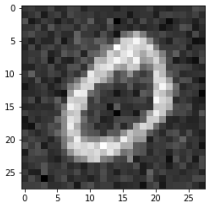

Figure 11: 40 % loss

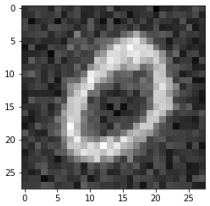

Figure 12: 60 % loss

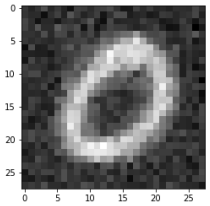

Figure 13: 80 % loss

5.2 Number of threads vs Runtime

5.2.1 Serial

- 326.42 seconds

5.2.2 Parallel

- 2 threads - 24.64 seconds
- 4 threads - 18.46 seconds
- 5 threads - 22.05 seconds

- 6 threads - 18.86 seconds
- 7 threads - 16.78 seconds
- 8 threads - 18.22 seconds
- 9 threads - 18.77 seconds

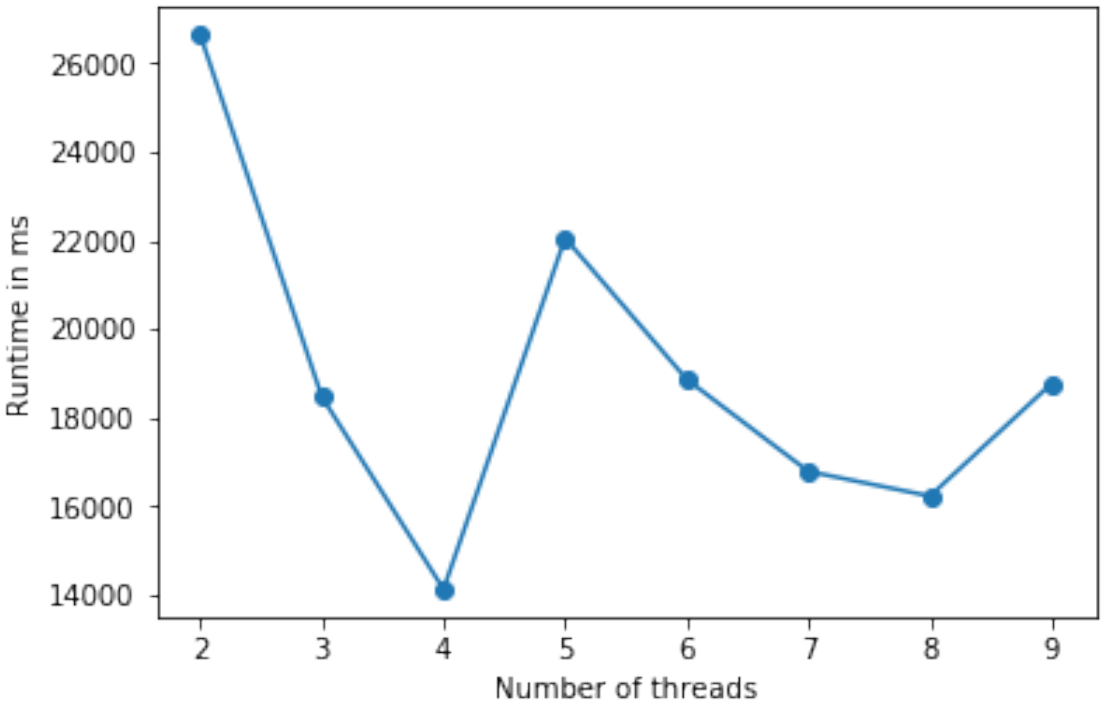

Figure 14: No: of threads vs time

Gain Ratio = 23.116

The best performance is obtained when number of threads was set to 4. This is because when number of threads are set greater than 4, a lot of time is consumed for scheduling these threads defeating the reason for parallelizing in the first place. If the number of threads are lesser than 4, then more threads can be added to increase parallelism. Thus 4 is the optimum number to perform parallelizm for out code.