Final Project Report

QR FACTORIZATION OF LARGE-LEAST SQUARE NUMBERS WITH HADOOP

TEAM MEMBERS:
Himansh Jain(15BCB0016)
Aaditya Vaidya(15BCB0032)
Dev Kumar(15BCB0060)
Yash Shah(15BCB0123)
GUIDED BY:
Prof. Manoov R

VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)

Table of Contents

PROJECT LINK

1)Abstract

2)Introduction

3)Scope

4)Cost

5)Communication

6)Algorithm

7)Pseudo code

8)Code

9)Output

10)Theorem used

11)Conclusion

PROJECT LINK

https://drive.google.com/file/d/1GWawRDTFOtj2-FwZrNEhaPZ3PECKg-1 /view

Final Project Summary

Abstract

A new algorithm is presented to provide an efficient solution to most of the square problems, in which the linear mechanism's multiplication of matrix is two small dimensions Matroso's CraneCare product. The solution algorithm is based on the QR terminology of the small dimension matrix. Full load equilibrium is achieved by absorbing the property of 'Crookcrake' property, and the communication needs can be reduced by using the binary exchange algorithm for matrix transposition. Parallel algorithms are presented, and timing results are shown on test runs on the Intel i860 computer.

Introduction

QR Factorization is used to solve linear equations ,least squares problems
And constrained least squares problems
In this project we will try to implement a parallel algorithm to solve the least square problem (full order)
(AQB) x = t, Where AE Mora, Page and B E MN, Q, Sequence (A) = P, Sequence (B) = q, and xE ~ RM, TE ~
According to the pre-existing research a new method for halting based on A and B curare disintegration. Here we describe an improved algorithm which needs to be compiled in one of the R-matrices and discusses its implementation on the Intel i860 computer. Improved algorithm uses Crankker's product 'Comutability' property. A pair of high triangular systems is produced, from which at least squares can be solved by backside

of the two systems in parallel. It has significantly less communication overhead than the version of algorithm given in a better computational load equilibration and.

Scope

We are going to use Kronecker product large-least squares algorithm for parallelization to improvise the QR-factorization problem in Hadoop.

Costs

Hardware costs include the PC to interface, and other peripheral hardware required.

All additional costs have been development, software and programming time.

Communications Strategy

The team utilized several communication methods to collaborate with each other; including github ,email, instant web meeting technology, voice calls and voicemail. The team also used both email as well as Microsoft OneNote to keep notes on meetings, and other conversations.

Communications

All members of the team have responded, contributed, and interacted well with each other throughout the project. Assignments have been completed in a timely manner with few grammatical errors, using correct terminology, and in an organized easy to read fashion in the correct APA format. Specifically, the weekly deliverables demanded enough work that teammates would have to quickly divide the work, complete tasks, and submit the work back to the group for review. Submitting tasks back to the group gave teammates an opportunity review the work and provide appropriate feedback.

During the meetings and through email, there were times that teammates would challenge each other on the direction or selected method for completing a particular task. These ‘challenges’ often would provide unique insight to the problem and demonstrated the synthesis and understanding of the topic leading to an enhanced solution to the problem at hand.

Algorithm

1. Compute the vector sr according to Sr = (QIIt)~ ~ 13)tr;
2. Compute the vector hr according to hr = (I~ ® QI2)')s7;
3. Block back substitute to obtain the solution vector zr to the system (I] ® RC2))Zr = hr;
4. Form the vector z from the vector ZT, i.e., perform a matrix transpose operation on zT;
5. Block back substitute to obtain the solution vector y to the system (12 ® RC~))y = z;
6. Repermute the vector yr according to .qr= (1~ ® P2)y7;
7. Compute the vector dr according to d7.--(I~ ® A),qT.;
8. Form the vector d from the vector dr, i.e., perform a matrix transpose operation on DT;
9. Repermute the vector d according to f = (13 ® Pj)d;
10. Compute the residual vector r according to r = t - (13 ® B)f.

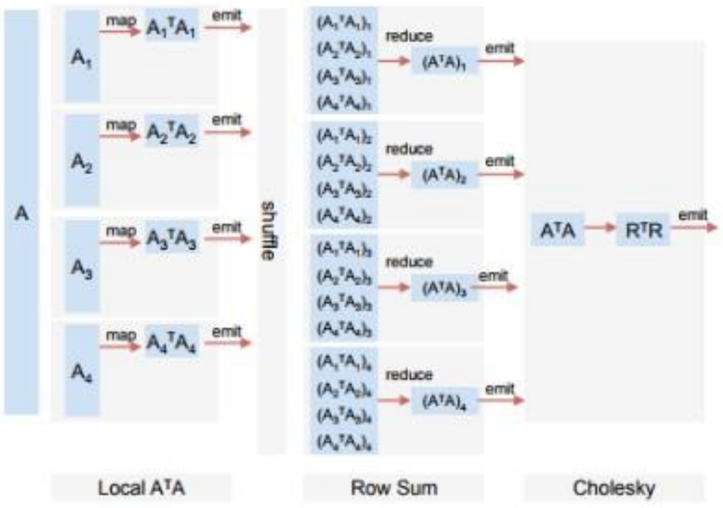


Figure 1. MapReduce Cholesky QR computation for a matrix A with 4 columns.

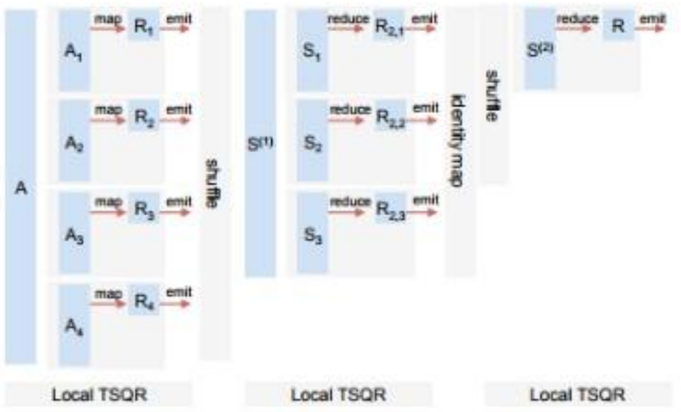


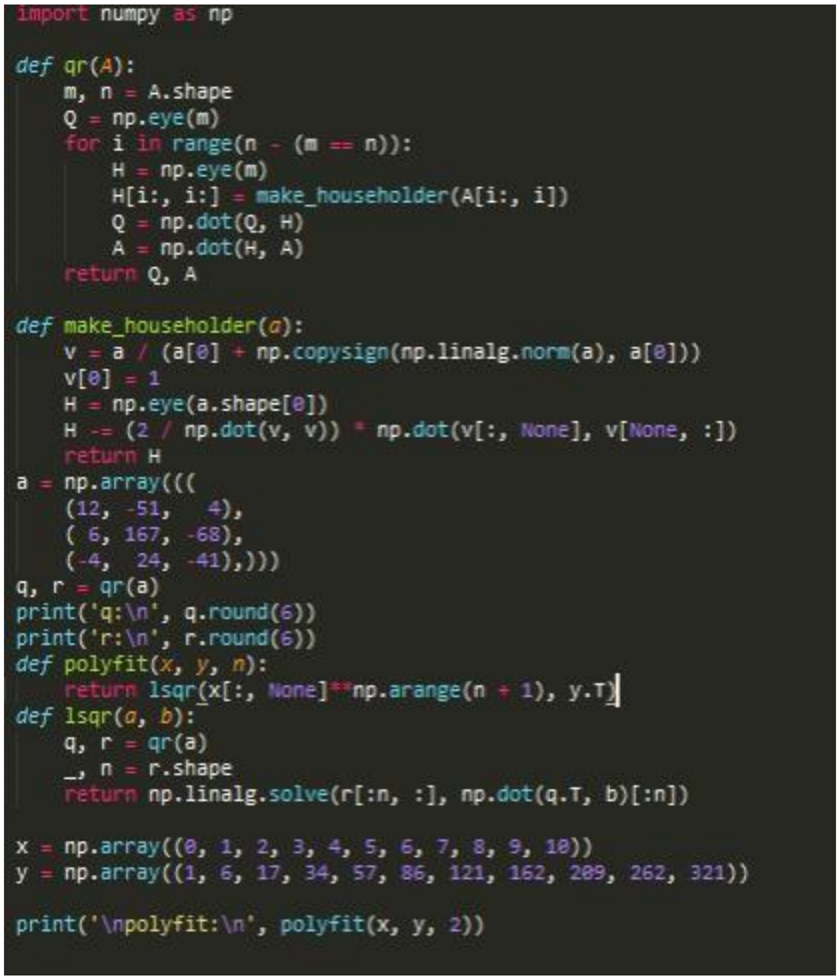
Figure 2. MapReduce TSQR computation. $S^{(1)}$ is the matrix consisting of the rows of the R_i factors stacked on top of each other,

FIG: TSQR MapReduce

Psedo Code Snippet

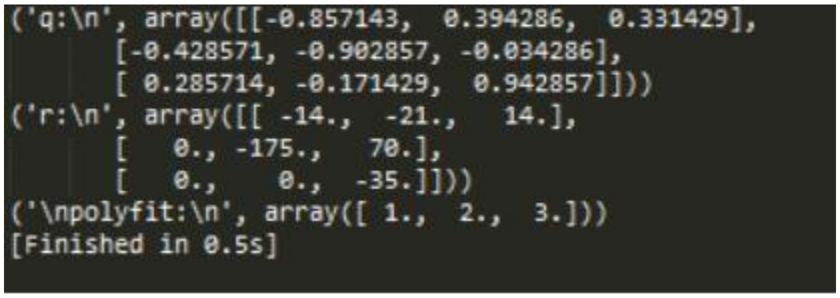
```
import random, numpy, hadoopy
class SerialTSQR:
  def __init__(self,blocksize,isreducer):
    self.bsize=blocksize
    self.data = []
    if isreducer: self.__call__ = self.reducer
    else: self.__call__ = self.mapper

  def compress(self):
    R = numpy.linalg.qr(numpy.array(self.data),'r')
    # reset data and re-initialize to R
    self.data = []
    for row in R:
      self.data.append([float(v) for v in row])

  def collect(self,key,value):
    self.data.append(value)
    if len(self.data)>self.bsize*len(self.data[0]):
      self.compress()

  def close(self):
    self.compress()
    for row in self.data:
      key = random.randint(0,2000000000)
      yield key, row

  def mapper(self,key,value):
    self.collect(key,value)

  def reducer(self,key,values):
    for value in values: self.mapper(key,value)

if __name__=='__main__':
  mapper = SerialTSQR(blocksize=3,isreducer=False)
  reducer = SerialTSQR(blocksize=3,isreducer=True)
  hadoopy.run(mapper, reducer)
```

Code *(python implementation for testing data sets)*



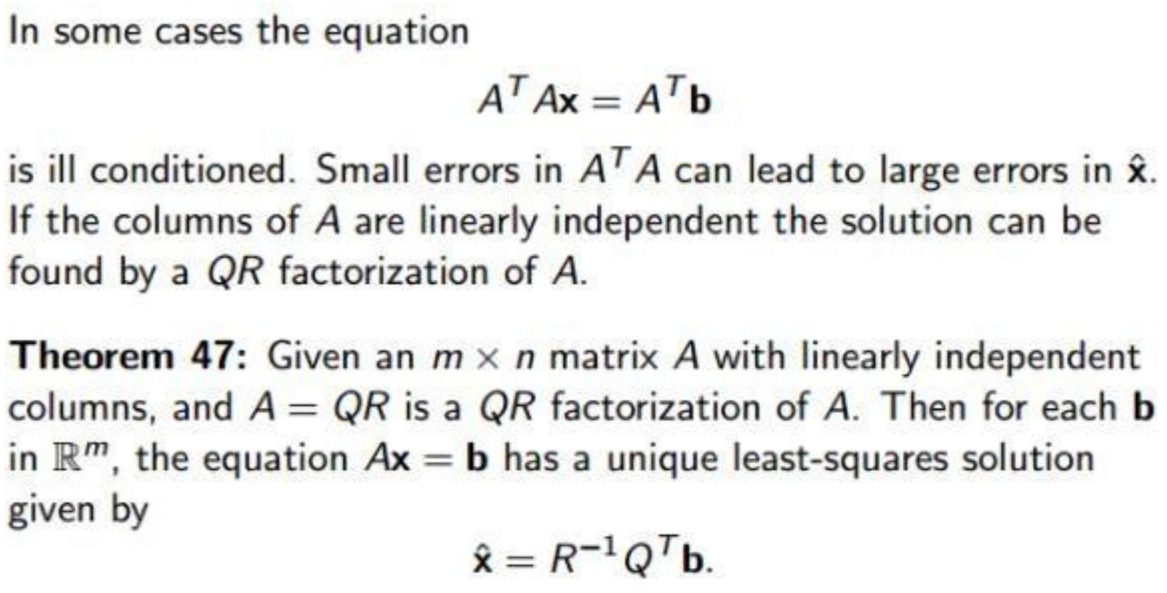
```
import numpy as np

def qr(A):
    m, n = A.shape
    Q = np.eye(m)
    for i in range(n - (m == n)):
        H = np.eye(m)
        H[i:, i:] = make_householder(A[i:, i])
        Q = np.dot(Q, H)
        A = np.dot(H, A)
    return Q, A

def make_householder(a):
    v = a / (a[0] + np.copysign(np.linalg.norm(a), a[0]))
    v[0] = 1
    H = np.eye(a.shape[0])
    H -= (2 / np.dot(v, v)) * np.dot(v[:, None], v[None, :])
    return H
a = np.array(((
    (12, -51,   4),
    ( 6, 167, -68),
    (-4,  24, -41),)))
q, r = qr(a)
print('q:\n', q.round(6))
print('r:\n', r.round(6))
def polyfit(x, y, n):
    return lsqr(x[:, None]**np.arange(n + 1), y.T)
def lsqr(a, b):
    q, r = qr(a)
    _, n = r.shape
    return np.linalg.solve(r[:n, :], np.dot(q.T, b)[:n])

x = np.array((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
y = np.array((1, 6, 17, 34, 57, 86, 121, 162, 209, 262, 321))

print('\npolyfit:\n', polyfit(x, y, 2))
```

Output



```
('q:\n', array([[-0.857143,  0.394286,  0.331429],
       [-0.428571, -0.902857, -0.034286],
       [ 0.285714, -0.171429,  0.942857]]))
('r:\n', array([[ -14.,  -21.,   14.],
       [   0., -175.,   70.],
       [   0.,    0.,  -35.]]))
('\npolyfit:\n', array([ 1.,  2.,  3.]))
[Finished in 0.5s]
```

Theorem *(For simplification used in code.)*



In some cases the equation

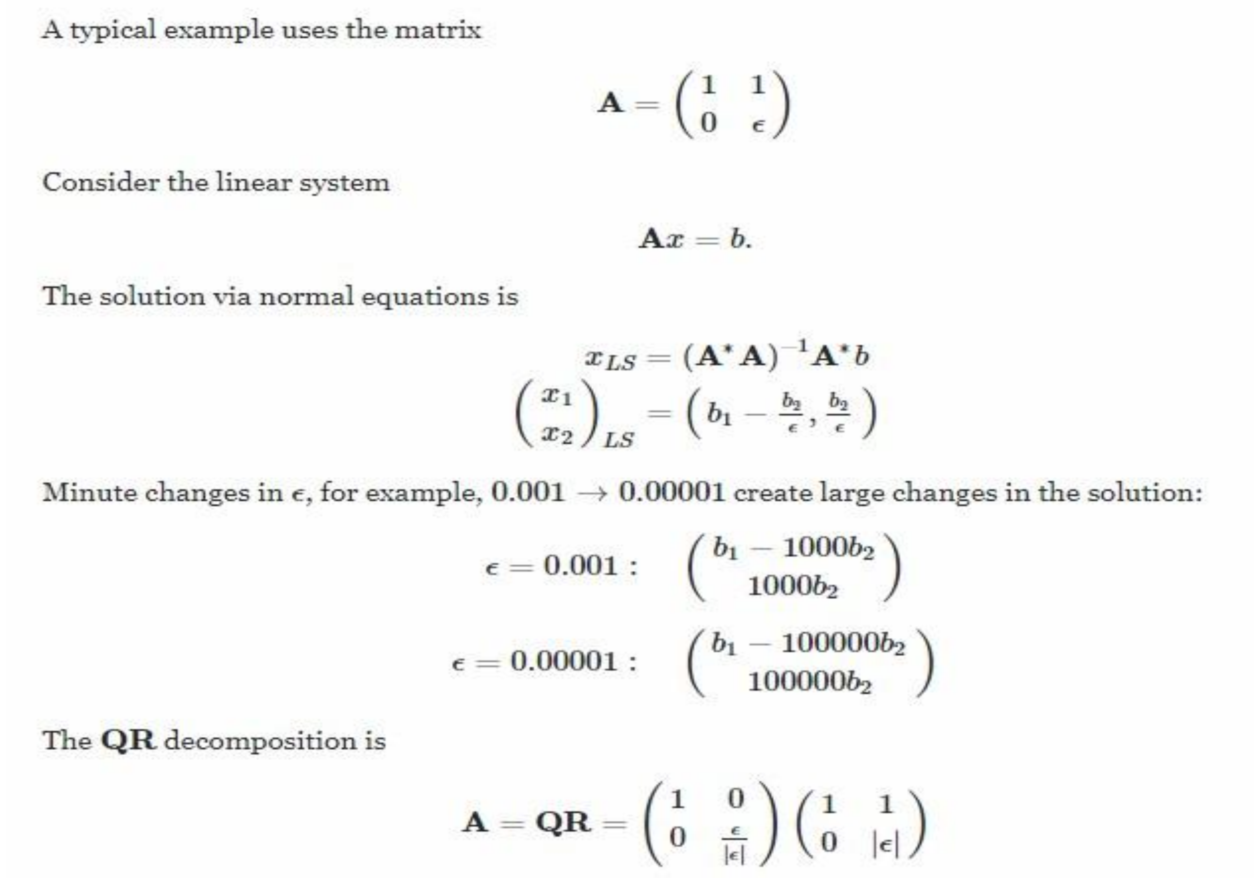
$$A^T A\mathbf{x} = A^T\mathbf{b}$$

is ill conditioned. Small errors in $A^T A$ can lead to large errors in $\hat{\mathbf{x}}$. If the columns of A are linearly independent the solution can be found by a QR factorization of A.

Theorem 47: Given an $m \times n$ matrix A with linearly independent columns, and $A = QR$ is a QR factorization of A. Then for each $\mathbf{b}$ in $\mathbb{R}^m$, the equation $A\mathbf{x} = \mathbf{b}$ has a unique least-squares solution given by

$$\hat{\mathbf{x}} = R^{-1}Q^T\mathbf{b}.$$

Example --*for the theorm*--



A typical example uses the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & \epsilon \end{pmatrix}$$

Consider the linear system

$$\mathbf{A}x = b.$$

The solution via normal equations is

$$x_{LS} = (\mathbf{A}^*\mathbf{A})^{-1}\mathbf{A}^* b$$
$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_{LS} = \left(b_1 - \frac{b_2}{\epsilon}, \frac{b_2}{\epsilon}\right)$$

Minute changes in ϵ, for example, $0.001 \to 0.00001$ create large changes in the solution:

$$\epsilon = 0.001: \quad \begin{pmatrix} b_1 - 1000b_2 \\ 1000b_2 \end{pmatrix}$$

$$\epsilon = 0.00001: \quad \begin{pmatrix} b_1 - 100000b_2 \\ 100000b_2 \end{pmatrix}$$

The $\mathbf{QR}$ decomposition is

$$\mathbf{A} = \mathbf{QR} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{\epsilon}{|\epsilon|} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & |\epsilon| \end{pmatrix}$$

CONCLUSION

In this paper the theoretical analysis of specific prerequisite low-squares solvors was done. Solvars uses a prendander, which is related to the coefficient matrix's lower-order vicissitudes. Confusion can be done as a result of updating (following the precursor calculation or original coefficient of matriculation factor), leaving intimate rows, or trying to make the preconditioner better, while multiplication matrix is conditioned or defective, we note that rare QR More research is needed to put rows effectively in the term; Here we have just given evidence that this idea can be effective, but we have not provided a single row-leaving algorithm. Certain methods related to the QR factor of misfortune or rank deficiency matrix by paper were proposed.
Our theoretical analysis uses a novel approach: we calculate the number of common Eigenvalues that are away from the object cluster (sometimes only containing the value 1), because the contrast. This allows us to stop the number of repetitions in the LASQR, such as the least square solvars, which are implicit versions of the corresponding components on common equations. This approach adjusts the most common method of matching repeats in the best Krylov-subspace solvers, which are based on the condition number of the prerendering system.
We have also provided limited experimental results, which explain the use of techniques rather than installing their efficacy or efficiency. We are planning to design and implement the QR factorial code of SPAR incorporating these technologies, but this is not beyond this paper. Once we have been implementing for the SPAR case, we want to do extensive testing of this technology which is the theoretical analysis of this paper.