QR Decomposition in a Multicore Environment

Omar Ahsan
University of Maryland-College Park
Advised by Professor Howard Elman
College Park, MD
oha@cs.umd.edu

ABSTRACT

In this study we examine performance benefits of implementing the QR decomposition in a way that takes advantage of multiple processes or threads. This is done by partitioning the matrix into blocks of a certain number of rows, which is called the blocksize. We examine this algorithm on "tall and skinny" matrices, which are matrices that have a very large number of rows, but comparatively fewer columns. These matrices are very important, as one of their most common uses is Linear Regression, which is a tool used in many different fields. We also compare this implementation to one which uses a MapReduce environment to compute the QR decomposition. We find that partitioning the matrix and using multiple processes to compute the QR decomposition in parallel provides for computing the decomposition much faster than computing the QR decomposition immediately on the original matrix.

1. INTRODUCTION

This paper studies the performance of several different ways of computing the QR decomposition of a matrix $\mathbf{A}$, specifically, the case where $\mathbf{A}$ has many more rows than columns. The standard method for computing the QR factorization of a matrix is not so efficient performance-wise when the matrix has many more rows than columns. Computing QR factorizations of those kinds of matrices is a common problem that arises in many real-world situations, such as regression analysis [5]. The algorithm in this paper is specifically designed for the case where $\mathbf{A}$ is *tall and skinny*. The paper looks at a few variations of this algorithm such as single-process vs multi-process and also examines an implementation of this algorithm using a Map-Reduce library [3].

1.1 The QR Decomposition

The QR decomposition of a matrix $\mathbf{A}$ is the factorization of $\mathbf{A}$ into two matrices $\mathbf{Q}$ and $\mathbf{R}$ where $\mathbf{Q}$ is *orthogonal* (or *semi-orthogonal*, if $\mathbf{A}$ is not square) and $\mathbf{R}$ is *upper triangular*. The matrix $\mathbf{Q}$ is *orthogonal* if $\mathbf{Q}^T = \mathbf{Q}^{-1}$, or $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. The matrix $\mathbf{R}$ is *upper triangular*, i.e., all entries below the main diagonal are 0.

The QR Decomposition

$$\underbrace{\begin{pmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ a_{31} & \dots & a_{3n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}}_{\mathbf{A} \in \mathbb{R}^{m \times n}} = \underbrace{\begin{pmatrix} q_{11} & \dots & q_{1n} \\ q_{21} & \dots & q_{2n} \\ q_{31} & \dots & q_{3n} \\ \vdots & \ddots & \vdots \\ q_{m1} & \dots & q_{mn} \end{pmatrix}}_{\mathbf{Q} \in \mathbb{R}^{m \times n}} \underbrace{\begin{pmatrix} r_{11} & \dots & r_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \dots & r_{nn} \end{pmatrix}}_{\mathbf{R} \in \mathbb{R}^{n \times n}}$$

One remark to make here is that the QR decomposition of a matrix $\mathbf{A}$ is not unique. If $\mathbf{A}$ is $m \times n$ where $m \neq n$, then there are both QR decompositions where $\mathbf{Q}$ is $m \times m$, $\mathbf{R}$ is $m \times n$ and ones where $\mathbf{Q}$ is $m \times n$, $\mathbf{R}$ is $n \times n$ (as in the figure above).

A standard method to compute the QR decomposition uses Householder Transformations. This is the method used by the Python library *numpy*. The Householder Transformation of a vector v is defined by

$$\mathbf{H} = \mathbf{I} - \beta \mathbf{v}\mathbf{v}^T$$

where

$$\beta = \frac{2}{\mathbf{v}^T\mathbf{v}}$$

Suppose we have a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$. Let $\mathbf{A}^{(0)} = \mathbf{A}$, and denote column i of $\mathbf{A}^{(0)}$ as $\mathbf{a}_i^{(0)}$.

To compute the QR decomposition via Householder reflections [4], the first step is the compute the first reflector vector, $\mathbf{v}_1$, by the formula:

$$\mathbf{v}_1 = \mathbf{a}_1^{(0)} - \text{sign}(a_{11}^{(0)})||\mathbf{a}_1^{(0)}||\mathbf{e}_1$$

The first Householder reflection matrix is given by

$$\mathbf{H}_1 = \mathbf{I} - 2\frac{\mathbf{v}_1\mathbf{v}_1^T}{\mathbf{v}_1^T\mathbf{v}_1}$$

$\mathbf{H}_1$ times $\mathbf{A}^{(0)}$ creates a new matrix which has all zeroes below the main diagonal in the first column

$$\mathbf{A}^{(1)} = \mathbf{H}_1\mathbf{A}^{(0)}$$

In theory this is the process for computing the QR decomposition with Householder Reflections, however, computing $\mathbf{H}_1$ is generally a very expensive computation. However, $\mathbf{H}_1$

can be represented implicitly through $\mathbf{v}_1$. The process now generalizes. For $1 < i < n$, we compute:

$$\mathbf{v}_i = \mathbf{a}_i^{(i-1)} + \text{sign}(\mathbf{A}^{(i-1)}(i,i))||\mathbf{a}_{i-1}^{(i-1)}||\mathbf{e_1}$$

$$\hat{\mathbf{H}}_i = \mathbf{I} - 2\frac{\hat{\mathbf{v}}_i\hat{\mathbf{v}}_i^T}{\hat{\mathbf{v}}_i^T\hat{\mathbf{v}}_i}$$

where matrices and vectors with hats are just the bottom $n-i$ rows of the original matrix or vector.

$$\mathbf{A}^{(i)} = \mathbf{H}_i\mathbf{A}^{(i-1)}$$

The majority of the computation in this matrix multiplication is actually taking place in the sub-blocks of $\mathbf{A}$ and $\mathbf{H}$ of size $(m-i)\times(m-i)$ and $(m-i)\times(n-i)$ At the final step we have

$$\mathbf{A}^{(n-1)} = \mathbf{H}_{n-1}\mathbf{A}^{(n-2)}$$

The $\mathbf{R}$ matrix from $\mathbf{A} = \mathbf{QR}$ is the nonzero rows from this final $\mathbf{A}^{(n-1)}$ matrix.

1.2 The TSQR Algorithm

The Tall and Skinny QR (TSQR) factorization algorithm produces the factorization $\mathbf{A} = \mathbf{QR}$ of a matrix $\mathbf{A}$ that is designed for the case where $\mathbf{A}$ is a *tall and skinny* matrix. A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is considered *tall and skinny* if $m \gg n$. Such matrices have many applications, one of which is solving a least squares problem $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{x},\ \mathbf{b} \in \mathbb{R}^n$. Consider the application of this to a linear regression. In these cases matrices would have rows representing observations and columns representing variables being used in the model. This matrix would be very tall and skinny as the number of observations (m) often exceeds several thousand whereas the number of variables in the model (n) is usually less than very small in comparison. n is often less than 10 in many regression models.

The algorithm works by taking the matrix $\mathbf{A}$ and splitting it into blocks of equal number of rows. In the sequential case with no multi-processing the algorithm is represented by the following pseudocode.

Algorithm 1. Single-process TSQR

```
B[] ← array s.t. B[i] is block i of A
M ← B[1]
for (i=2 to numBlocks)
    concatenate B[i] below M
    compute M = QR
    M ← R
end for
return M
```

The resulting $\mathbf{M}$ at end of this algorithm is $\mathbf{R}$ from the factorization $\mathbf{A} = \mathbf{QR}$. This algorithm only computes $\mathbf{R}$. $\mathbf{Q}$ can be computed as $\mathbf{Q} = \mathbf{AR}^{-1}$, but there are also other methods to compute $\mathbf{Q}$ directly [2]. As noted above, since $\mathbf{A}$ is not square it has both QR factorizations where $\mathbf{Q}$ is $m\times m$ and where $\mathbf{Q}$ is $m\times n$. This algorithm will give a QR factorization that matches the dimensions of the second case here (called the *thin* QR factorization of $\mathbf{A}$ in this case). This algorithm can be implemented in a multi-process environment.

Instead of sequentially computing the $\mathbf{R}$ matrices from top to bottom in one process, the algorithm would be altered so that there are k processes and each one is responsible for m/k rows. Each process would then take the portion of the matrix it is responsible for and compute its QR factorization using the single-process TSQR algorithm. Essentially, the matrix is split into k sections and, using Algorithm 1, the program computes k QR factorizations in parallel. At the end of the parallel part there will be k $\mathbf{R}$ matrices that need to be combined. This is done sequentially using the original sequential TSQR algorithm.

Algorithm 2. Multi-process TSQR

```
// TSQR with p processes
// Cut matrix into p blocks,
B[] ← array s.t. B[i] is block i of A
R[] ← empty array size p
for (i = 1 to p)
    // Single-process tsqr here
    execute tsqr(B[i]) with process i
    // The result is the R  matrix
    // of B[i] = QR
    put result in R[i]
end for
M ← R[1]
for (i = 1 to t − 1)
    A ← concatenate R[i+1] below R[i]
    compute A = QR
    R[i+1] ← R
    M ← R[i+1]
end for
return M
```

As before, the resulting $\mathbf{M}$ at the end is the $\mathbf{R}$ from the factorization $\mathbf{A} = \mathbf{QR}$.

The idea behind this TSQR algorithm is that matrices that are very large may not fit in memory. In order to prevent the matrix from spilling onto the hard drive (and making the program much slower) the matrix is split into blocks. When a good blocksize is chosen, most of the block should be able to remain in the cache so that the QR factorization can be computed with minimal reads and writes to slower memory and the hard drive. If the original matrix $\mathbf{A}$ is small enough such that it can already fit in the cache without being sectioned into blocks then this algorithm should not be used over just calling the regular QR procedure in the programming language's matrix library.

2. EXPERIMENTS

2.1 Overview of Experiments

The experiments described in this paper make use of three different programs to compute the QR factorization of a matrix. The first program is `numpy(A)`, which simply computes the QR factorization by calling `numpy.linalg.qr` on the matrix. This is a standard QR decomposition implementation in a Python library that uses the Householder Transformation method to create the decomposition [1]. The second of these programs is `tsqr(A, blocksize)`, which computes the QR factorization of $\mathbf{A}$ by splitting it into blocks of size

blocksize and using the single TSQR algorithm (Algorithm 1). The third program is `mtsqr(A, blocksize, numProcesses)`, which computes the QR factorization of **A** using the TSQR algorithm with the given blocksize and number of processes (Algorithm 2). The way the algorithm has been coded for this study requires that the blocksize divides $\frac{n}{numProcs}$, that is, that the blocksize evenly divides the rows each process is responsible for.

All experiments are run on a machine using a 4-core, non-hyperthreaded Intel Core i5-4570 processor.

2.2 Performance of Multiprocessing

The first experiment is designed to analyze any possible performance increases in performing the TSQR algorithm in parallel. For this we chose two matrices of sizes 50000×50 and 500000×100. The large discrepancy in the sizes of the two matrices is to investigate if there are any changes in how well the algorithm performs compared to a simple call to the numpy QR procedure when the size of the matrix varies.

Figure 1: Graph showing runtimes for the program with various blocksizes and number of processes on a $50,000 \times 50$ matrix. In the following figures and tables, P-X and B-Y mean the program was run on X processes with a blocksize of Y.

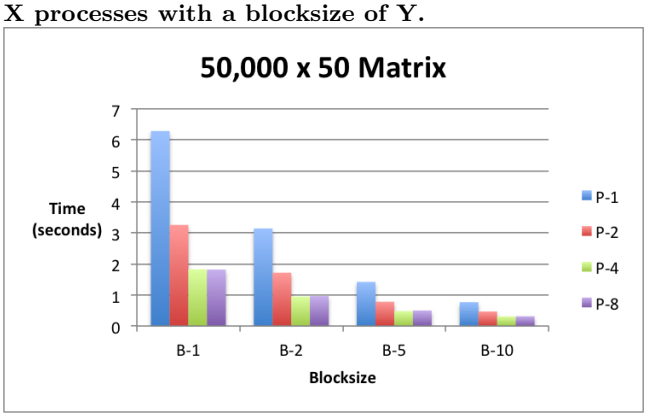
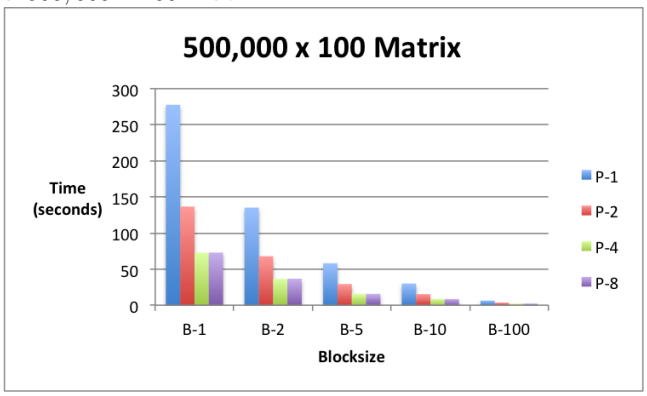


Figure 2: Graph showing runtimes for the program with various blocksizes and number of processes on a $500,000 \times 100$ matrix.



For the 50000×50 case, in our results the single-process TSQR algorithm showed a modest 22% decrease in computation time over computing it with a single numpy call. For the 500000×100 case, we see a much larger decrease of approximately 74% when run with a blocksize of 500 and 4 processes.

Table 1: Data from the graph in Figure 1.

Blocksize	P-1	P-2	P-4	P-8
B-1	6.285	3.259	1.826	1.818
B-2	3.144	1.718	0.948	0.964
B-5	1.421	0.782	0.482	0.5
B-10	0.768	0.468	0.311	0.314
B-5000	0.203	0.185		
B-12500	0.215	0.221		

Table 2: Data from the graph in Figure 2.

Blocksize	P-1	P-2	P-4	P-8
B-1	277.608	136.597	72.773	72.858
B-2	135.184	67.843	36.387	36.697
B-5	58.14	29.266	15.812	15.686
B-10	30.036	15.35	8.549	8.517
B-100	6.44	3.577	2.196	2.228
B-500	4.384	2.541	1.653	1.683
B-2500	4.209	2.466	1.758	1.903
B-32250	5.295	4.033	3.558	
B-125000	6.107	5.073		

When the number of processes is increased from 1 to 2, the speedup is close to twice as fast when the time required is large, but it becomes increasingly negligible as the computation takes less time overall. This is seen best for the 500000×100 matrix with blocksize 1. The single-process TSQR program takes approximately 277 seconds to compute this decomposition, whereas the multi-process TSQR program with 2 processes takes approximately 136 seconds, which is roughly twice as fast. On the other hand, for the 50000×50 matrix, when run with a blocksize of 12500 actually sees a speed decrease, going from 0.215 seconds to 0.221 seconds.

Observing speedup improve by approximately a factor of 2 is in line with our expectations because the work being done by each process is mostly sequential with little wait-time. This is due to the fact that the work is mostly numerical computations. As such we do not see speedup by a larger scale than the scale at which the number of processors was increased. Additionally, as seen in Table 2 above, there are no gains for increasing the number of processors from 4 to 8. In fact, the program, on average, takes longer to compute the QR factorization when utilizing 8 processes instead of 4. This is not surprising as the work as mostly sequential and the machine only has 4 cores. For this reason there is no benefit to having 2 processes run on the same core, as there is no opportunity to increase efficiency by having one process continue working while the other one is waiting. The slight worsening in time taken could be due to the additional overhead of creating the extra 4 processes and combining

their results.

Because the program is written in Python, these experiments all test the use of multiple processes instead of multiple threads in improving the performance of the TSQR algorithm. The reason for this is that all Python threads are required to run on the same process due to its *Global Interpreter Lock* (GIL). Originally this program was actually implemented with threads instead of processes. The data for the 500000×100 and 50000×50 cases with varying numbers of threads is included below.

Figure 3: Graph showing runtimes on a $50,000 \times 50$ matrix and highlighting the differences between threads and processes. Here T-X means the program was run on X threads. Note: Blocksizes 5000 and 12500 were dropped from the figure because the values are very small and distort the image. The data for these blocksizes is in Table 3.

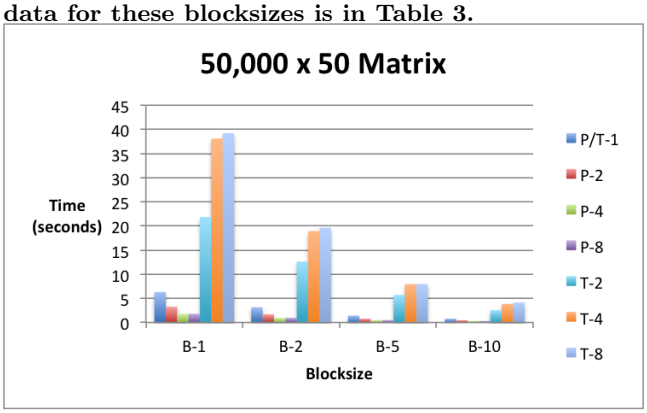


Table 3: Table for the thread data in Figure 3 (the process data is in Table 1).

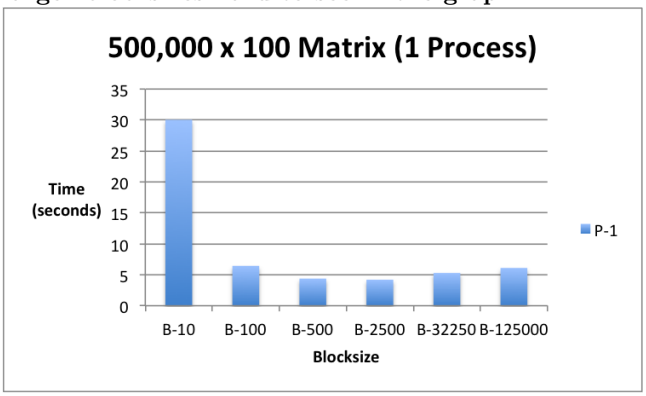
Blocksize	T-1	T-2	T-4	T-8
B-1	6.285	21.843	38.084	39.19
B-2	3.144	12.608	18.935	19.637
B-5	1.421	5.715	7.949	7.96
B-10	0.768	2.547	3.841	4.128
B-5000	0.203	0.207		
B-12500	0.215	0.217		

As seen in the tables above, there no decrease in the amount of time the program takes when increasing the number of threads. The results are actually more extreme than that; as we increase the number of threads, the time required to compute the QR factorization increases by a significant amount. Due to the Global Interpreter Lock, regardless of how many threads are created, the program is only using one process. Since the work is sequential, there are no performance benefits with having more than one thread per core. In this case we are actually seeing the overhead of creating the thread and then joining the results of each thread significantly hinder the performance of the program when the blocksize is small. One thing to note, however, is that the hinderance of threads is negligible for the runs at a blocksize of 5,000 and 12,500. This is possibly due to the fact that at these blocksizes there is very little work being done per thread. There are two threads and each is given 25,000 rows. With a blocksize of 5,000 and 12,500 the threads only split their sections into 5 and 2 blocks respectively. There are far more calls to the numpy QR function when the blocksize is small. With more calls to the QR function, there may be more opportunities for the threads to compete for system resources on one process, thereby slowing the program down.

2.3 Varying the Blocksize Parameter

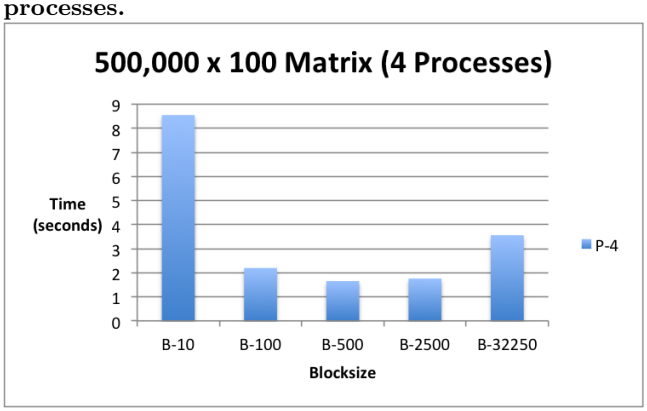
In the above section, we found that the program performs best with 4 processes because the machine has 4 cores. As such, the following experiments will only be run with 1 and 4 processes. The 2 and 8 process cases will be dropped since we are primarily concerned with finding the optimal blocksize. In these experiments, we investigate if we find the same results as in [3] without the MapReduce environment. The results found in that paper suggest that the ideal blocksize is neither as large nor as small as possible, but rather somewhere in between. Without this MapReduce environment we still expected to see this pattern take fold. We expect smaller blocksizes to decrease computation time because smaller matrices are more likely to be able to fit into cache. However, making the blocksize *too small* could also be negative to performance. This is because decreasing the blocksize means increasing the number of QR decompositions we have to compute since we are increasing the number of blocks. Since the key gains from decreasing the blocksize are coming from no longer having to put part of the block in RAM when doing the QR factorizations on the blocks, we do not expect to see much additional improvement from making the matrix even smaller, once it is small enough to fit into cache. At this point the negative effects of having to compute the QR factorization of more blocks may begin to outweigh the benefits of decreasing the size of the blocks.

Figure 4: Graph for the $500,000 \times 100$ matrix for 1 process. Note: In the graphs for this section, data for very small blocksizes are dropped. This is because the program's runtime is much higher for these cases and would make the pattern visible at larger blocksizes hard to see in the graph.



For the $500,000 \times 100$ matrix, we find that the ideal blocksize is not at either extreme of being as large or as small as possible, but somewhere in between the two. For the single-process version, the program ran fastest with a blocksize

Figure 5: Graph for the $500,000 \times 100$ matrix for 4 processes.



input of 2,500. For the four-process version, the fastest run was with a blocksize of 500, though this only narrowly beat the runs with a blocksize of 2,500. While this difference may not be significant, the run with a blocksize of 500 still comfortably beat the run with a blocksize of 32,250, which is consistent with the hypothesis that the ideal blocksize is neither too large or too small.

Figure 6: Graph for the $50,000 \times 1,000$ matrix for 1 process.

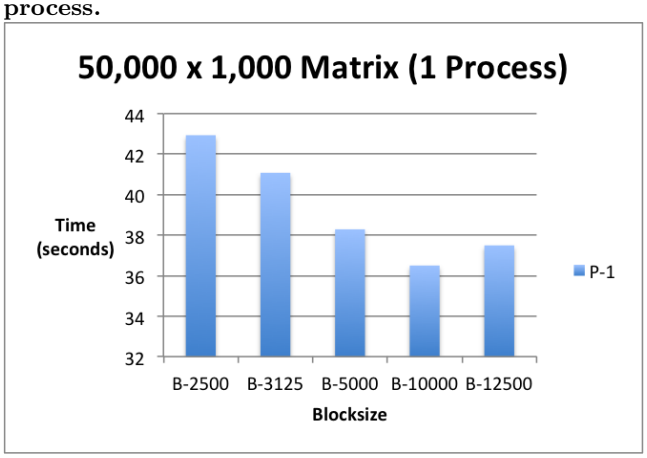
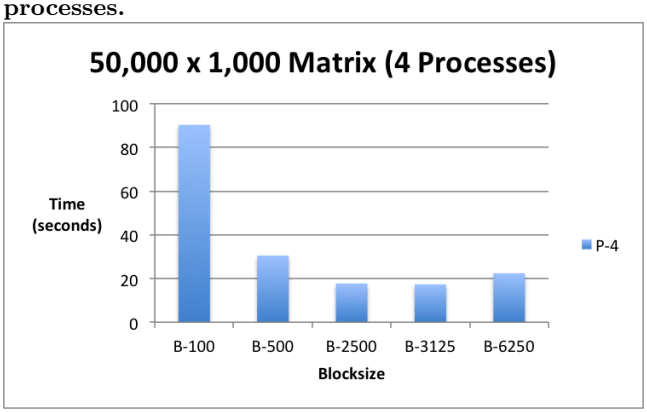
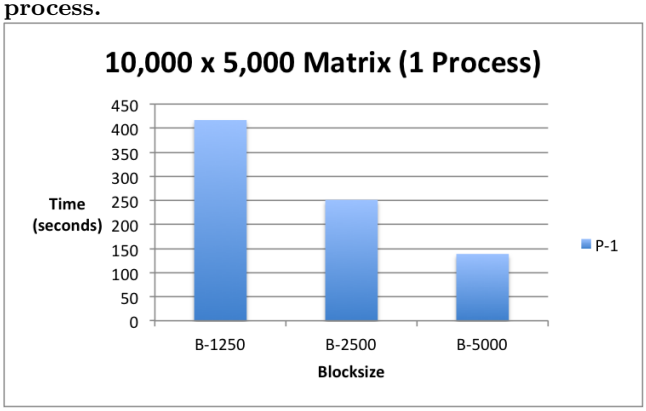


Figure 7: Graph for the $50,000 \times 1,000$ matrix for 4 processes.



For the $50,000 \times 1,000$ matrix, we again observe that the increasing the blocksize when the blocksize is small will improve performance, but only up to a certain point. Here the ideal blocksize for the single-process version of the program is 10,000, as the program took longer to compute the decomposition with both smaller and larger blocksizes. When computing the QR factorization with 4 processes,For the $50,000 \times 1,000$ matrix, we also leave off blocksizes below 100 because the runtime of the program with those inputs is much higher. For example, computing the QR decomposition with a blocksize of 10 and 4 processes on this matrix took approximate 770 seconds.

Figure 8: Graph for the $10,000 \times 5,000$ matrix for 1 process.



For the $10,000 \times 5,000$ matrix we no longer observe the pattern in varying the blocksize that was visible at matrices with a higher ratio of rows to columns. For the single-process version of the code, we find that the ideal blocksize is 5,000, which is the largest possible blocksize for a matrix with this many rows.

2.4 Comparison to MapReduce Implementation

Before comparing the two implementations, it is important to note that the use of the term *blocksize* is not identical between the two implementations. In the Python implementation in this paper, for a matrix of size $m \times n$, a blocksize of b partitions the matrix into blocks with b rows in each. However, in the MapReduce implementation, the blocks each have $b * n$ rows. So in order to make accurate comparisons, if the MapReduce implantation is run with a blocksize of b, then the Python implementation in this paper must be run with a blocksize of $b * n$.

Compared to the MapReduce implementation from Gleich and Constantine's paper, the multi-process TSQR algorithm was able to perform better on both the $50,000 \times 50$ and $500,000 \times 100$ matrices.

These results are ultimately not surprising. The MapReduce code was developed for the purposes of extremely large matrices, ones where the whole matrix would not be able to fit in memory and several reads and writes to the hard disk drive would be required. In [3], the largest matrix has dimensions $1,000,000,000 \times 50$, which has 50 billion cells. In comparison, the largest matrix in this paper has 50 million

Table 4: Comparison of the MapReduce implementation with the Python implementation. Note: In the cells for blocksize, X/Y means a blocksize of X in the MapReduce implementation, and a blocksize of Y in the Python implementation.

Matrix	Blocksize	Time in Python (P-1)	Time in MapReduce
50000x50	100/5000	0.203	6.311
50000x50	250/12500	0.215	6.31
500000x100	5/500	4.384	47.646
500000x100	25/2500	4.209	46.671

cells; approximately one-thousandth the size of that matrix. Running the multi-process QR program on the machine used in the experiments in this paper was not possible for a matrix of that size. As the matrices in this paper are much smaller, it is likely that we are observing the overhead of the MapReduce (dumbo and hadoop) be a large factor in the time needed to compute the factorization. As the overhead costs are fixed, this cost would be less of a factor for larger matrices. In the future it would be interesting to run both implementations of the TSQR algorithm on a much larger matrix using a machine with more memory.

The MapReduce code on this machine also failed to compute the factorization faster than a simple call to `numpy.linalg.qr` for these two matrices. Computing the factorization with just a call to the numpy QR function on the $50,000 \times 50$ matrix took, on average, 0.236 seconds. The numpy call took an average of 6.327 seconds on the $500,000 \times 100$ matrix.

3. CONCLUSION

Computing the the QR factorization of a matrix using the TSQR algorithm yields similar results when the MapReduce environment is dropped. The same pattern of the optimal blocksize being neither too large nor too small is present in the data collected with this version of the algorithm. Additionally, for the matrices in this experiment, the version without MapReduce was able to perform the computation faster than the MapReduce implementation when using the same number of rows per block. The matrices in this experiment however, were significantly smaller in size than those the MapReduce version was written for. In future work it would be interesting to study how the version without MapReduce fairs on much larger matrices.

References

[1] Scipy documentation. `http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.qr.html#numpy.linalg.qr`. [Online].

[2] A. Benson, D. Gleich, and J. Demmel. Direct qr factorizations for tall-and-skinny matrices in mapreduce architectures. *IEEE International Conference on Big Data*, 2013.

[3] P. Constantine and D. Gleich. Tall and skinny qr factorizations in mapreduce architectures. *Proceedings of the the second internation workshop on MapReduce and its applications*, 2011.

[4] S. Leon. *Linear Algebra With Applications*. Pearson-/Prentice Hall, 2009.

[5] D. Montgomery and E. Peck. *Introduction to Linear Regression Analysis*. Wiley, 2012.

4. APPENDIX

4.1 Code for Multi-Process TSQR

```
import numpy
import Queue
import multiprocessing

# The original, single-process, TSQR algorithm
def seq_tsqr(A, blocksize, resultQueue):
    array = []
    m = A.shape[0]
    i = 0
    numBlocks = m / blocksize
    while (i < numBlocks):
        array.append(A[(i * blocksize):((i + 1) * blocksize), :])
        i = i + 1
    i = 1
    temp = numpy.empty((2 * blocksize, A.shape[1]))
    temp[0:blocksize, :] = array[0]
    temp[blocksize:(2 * blocksize), :] = array[1]
    theR = numpy.linalg.qr(temp, 'r')
    n = A.shape[1]
    while (i < numBlocks - 1):
        temp = numpy.empty((blocksize + theR.shape[0], n))
        temp[0:theR.shape[0]] = theR
        temp[theR.shape[0]:(theR.shape[0]+blocksize)] = array[i + 1]
        theR = numpy.linalg.qr(temp, 'r')
        i = i + 1
    resultQueue.put(theR)

# Requirements for computation on an m x n matrix:
# blocksize must divide (m / numProcs)
# m divides numProcs
# Example: (A is 100x10, 5, 10) is valid since 5 divides 10
# Example: (A is 100x10, 15, 5) is invalid since 15 does not divide 20
def tsqr(A, blocksize, numProcs):
    (m, n) = A.shape
    rowsPerThread = m / numProcs
    array = []
    manager = multiprocessing.Manager()
    rQueue = manager.Queue()
    procs = []

    # Split the matrix into chunks, one chunk per process
    # Compute the QR of each block with the single-process algorithm
    for i in range(0, numProcs):
        thisBlock = A[(i * rowsPerThread):((i + 1) * rowsPerThread), :]
        array.append(thisBlock)
        procs.append(multiprocessing.Process(
            target=seq_tsqr, args=(thisBlock, blocksize, rQueue, )))
        procs[i].start()
    for i in range(0, numProcs):
        procs[i].join()
    print rQueue.qsize()
    theR = rQueue.get()
    curr = theR

    # Combine the results
    while (not rQueue.empty()):
        r1 = curr.shape[0]
        next = rQueue.get()
```

```
        r2 = next.shape[0]
        temp = numpy.empty((r1 + r2, n))
        temp[0:r1] = curr
        temp[r1:(r1 + r2)] = next
        curr = numpy.linalg.qr(temp, 'r')
        theR = curr
    return theR
```

4.2 Code to convert numpy matrix into format for use by MapReduce implementation

```
from __future__ import print_function
import numpy
import sys

A = numpy.random.rand(int(sys.argv[1]), int(sys.argv[2]))
f = open('temp.tmat', 'wc')

for row in A:
    for cell in row:
        f.write(str(cell))
        f.write("    ")
    f.write("\n")
```