CS267 Final Project - Matrix Inversion Using the NumS Cloud Library

Arjun Dhamrait
UC Berkeley
adhamrait@berkeley.edu

Aatif Jiwani
UC Berkeley
aatifjiwani@berkeley.edu

Shivin Devgon
UC Berkeley
shivin302@berkeley.edu

1 Introduction

For our final project, we decided to implement various methods for computing $(A^TA)^{-1}$ and X in NumS, where A is crucially a tall-skinny matrix [1] and X is a symmetric positive definite matrix. We implemented a baseline serial algorithm first, then compared it to an algorithm utilizing LU decomposition, an algorithm using Cholesky decomposition, and an algorithm using TSQR factorization.

1.1 NumS Cloud Library

NumS is a Python library that translates n-dimensional operations across a distributed network of computers. NumS operates with Ray as a systems backend responsible for managing the memory and resources of a node cluster. Ray operates using a distributed memory model, and NumS uses this model to partition matrices across the cluster in either a block or block-cyclic format.

In this project, we used NumS's block partitioning, so n-dimensional arrays are partitioned into block-sized blocks and distributed across the cluster. Operations called using the NumS API are scheduled by the NumS backend, wherein Ray schedules task in a queue-like fashion and workers execute the tasks one by one in parallel. When workers have try to do operations on blocks that are not owned by them, the NumS backend finds which worker that block is located on and requests that block from the other worker automatically. The cost of communicating the block sizes is not negligable and tends to be the largest factor in performance. Minimizing the communication in algorithms is the primary goal when working in NumS.

The NumS API mimics Numpy API, making it approachable for Python developers to use without relearning too much.

1.2 Inversion of Large-Scale Matrices

According to [2], "matrix inversion is an essential computation task in many data scientific applications, such as signal processing, complex network analysis and collaborative recommendation. For example, in real-life social networks (like Facebook and Twitter), e-commerce websites (like Amazon and Ebay) and online-video service providers (like Youtube and Netflix), various types of matrices, including following matrices, transaction matrices and rating matrices, contain millions of distinct users and items. Inversion of these large-scale matrices is a fundamental operation for proximity measurement, link prediction and personalized recommendation tasks." Because these companies have access to a massive number of process, including thousands of computers each with dozens of cores and multiple threads per core, it is ideal to devise algorithms to perform matrix inversion that can evenly distribute the workload across all the processors. Dense matrix inversion is also useful for data science, earth science, scientific computing, according to [3].

1.3 A^TA inversion

The matrix $(A^TA)^{-1}$ is useful in machine learning and statistics because it is used in least squares fitting and maximum likelihood estimation (QDA in particular). It is also useful in stochastic systems, control theory, and reinforcement learning when dealing with covariance matrices for tasks such as Kalman Filtering. A key property of this matrix is that it is always positive semi-definite, and usually positive definite. Positive semi-definite matrices also occur in graphs for tasks such as spectral graph clustering, which require processing the Laplacian matrix. For big data applications, it is standard to have a tall-skinny matrix $A \in \mathbb{R}^{n\times d}, n >> d$, composed of millions of data points, where each data point is a 1D feature vector that has length in the order of hundreds or thousands [4]. While an upper triangular solve could perform least squares fitting more efficiently than calculating the inverse, we still require inversion of this massive matrix in the cases described in the previous paragraph. For this, we theorize that it will be faster to decompose $A = QR$ first using an algorithm like [4] that is optimized for tall-skinny matrices, rather than compute the dot product A^TA.

1.4 A^TA inversion in NumS

Currently in NumS, the only method to compute $(A^TA)^{-1}$ is to directly compute (A^TA) and then invert that computed matrix. When A is tall and skinny, A^TA becomes a very large matrix and the computation becomes slow. Additionally, currently in NumS the matrix inversion operation is not distributed, meaning in order to invert a matrix in NumS the matrix must be communicated to a single process and then redistributed back to the grid.

2 Related Work

The LU decomposition algorithm used is block-based initially created for use with Spark on large matrices. [2] This algorithm in the original paper was tested on a cluster of machines with 64GB memory, two 2.1GHz Intel Xeon CPUs with 12 physical cores, eight 7200 RPM hard disks, and one Gigabit ethernet card, and got inversion of a matrix of order 40960. A key difference between our experiments here and in the paper is that the paper matrices were stored on disk rather than in memory like they are in our example.

The TSQR factorization algorithm is centered around [4]. [4] introduces an approach of factorizing very-tall and very-skinny matrices using the MapReduce framework. Although this paper also provides an approach for SVD, we focus primarily on their approach for a direct TSQR and an indirect TSQR algorithm. In both indirect and direct TSQR, authors of [4] decompose the QR factorization algorithm as a set of map or reduction operations. An important note that the authors highlight is that the iterative direct TSQR algorithm, much like the indirect algorithm, has a severe bottleneck. After the first map operation, there is a single reduction operation to perform a QR factorization on a stacked set of square matrices. This reduction operation pulls all data onto a single node for computation. As a result, they offer a recursive algorithm that serves to widen the bottleneck for improved speeds. Then, a key difference between our experiments and the experiments laid out in [4] is that we use much wider matrices. Since a large portion of this paper surrounds the idea of improving the speed of computing $(A^TA)^{-1}$, it would not be effective to analyze results when (A^TA) is very small. Therefore, we experiment with matrix sizes beyond the maximum 100 columns as discussed in [4].

The Upper Triangular Inversion and Cholesky decomposition algorithms were based off algorithms described in [5]. [5] introduces a list of standardized operations that should be part of every distributed linear algebra library, and then describes and profiles various ways to implement in-place Cholesky decomposition $X = R^TR$, upper triangular inversion R^-1, and dot product $X^{-1} = R^{-1}R^{-T}$.

[3] also introduces an algorithm for distributed large-scale matrix inversion incorporating Strassen's scheme. [3] claims to beat [2] because their algorithm requires less operations. [6] also proposes an algorithm incorporating Strassen's scheme, but it does not compare its results to or cite [3].

3 Problem Statement

In this project, we will explore different methods of computing $(A^TA)^{-1}$ and X^{-1} efficiently in NumS for tall skinny A. In particular, we will use a naive method of computing $(A^TA)^{-1}$ directly, using LU decomposition for inversion, using TSQR decomposition to find $(A^TA)^{-1}$ given A, and also computing A^TA using a dot product and then finding the inverse $(A^TA)^{-1}$ using Cholesky decomposition.

We also check the performance of finding X^{-1}, where $X \succ 0$, a positive definite square matrix, using the cholesky decomposition.

4 Serial Methods

4.1 Naive Method

To naively find $(A^TA)^{-1}$ serially, one must use a matrix transpose, multiply, and then inversion.

4.2 TSQR Factorization

Any tall-skinny $m \times n$ matrix A, where $m >> n$, can be decomposed as $A = QR$. This decomposition produces an orthogonal $m \times n$ matrix Q and an upper triangular $n \times n$ matrix R. Using these properties, we can further decompose the naive operation as such:

$$(A^TA)^{-1} = ((QR)^T(QR))^{-1} = (R^TQ^TQR)^{-1} = (R^TR)^{-1} = R^{-1}(R^{-1})^T$$

This serial TSQR Method requires a singular QR factorization, an inverse on a small $n \times n$ matrix, and then a matrix multiplication on a matrix of the same shape. By using a TSQR factorization, we have removed the need for a large matrix multiply on two matrices of size $m \times n$. Note that this serial method uses a serial, unoptimized QR factorization and inversion.

4.3 LU Decomposition

LU decomposition for matrix inversion is similar to the naive implementation. First, $A^TA = B$ is calculated. Then, B is decomposed into its P, L, and U matrices such that $PB = LU$. Then, L^{-1} and U^{-1} are calculated and $(A^TA)^{-1}$ is found as $B^{-1} = U^{-1}L^{-1}P$. At first glance this seems like it may take more steps than the naive implementation, however there are methods to calculate L^{-1} and U^{-1} directly, thus it takes 4 matrix multiplies, an inverted LU decomposition, and one transpose to complete.

$$(A^TA)^{-1} = B^{-1} = (P^{-1}LU)^{-1} = U^{-1}L^{-1}P$$

4.4 Cholesky Decomposition

The Cholesky decomposition only works for symmetric positive definite matrices. Given $X \succ 0$, it computes R, where $X = R^TR$ [5]. Then, we can compute $X^{-1} = R^{-1}R^{-T}$. [7] proves that we only need $\frac{1}{2}n^3$ multiply operations to calculate the Cholesky decomposition of $X \in \mathbb{R}^{n\times n}$. [8] shows that Cholesky decomposition is 2 to 4 times faster than QR, while having a third less standard deviation in it's times between matrices of the same dimension. Obviously, the catch is that it can only work on positive definite matrices.

5 Parallel Algorithms

5.1 Naive Implementation

The naive method of computing $(A^TA)^{-1}$ first involves a distributed matrix multiply of A^T and A. Finding A^T is done with a bit of communication natively in NumS. Then, because a distributed matrix inversion algorithm does not yet exist, $(A^TA)^{-1}$ must be done by first sending all the distributed data of A^TA to a single node, doing a normal sequential matrix inversion on that single node, then sending the data back out to the original distributed shape. The communication steps end up being the most time-intensive of this algorithm.
Because the entire matrix must be able to fit on a single node, this naive implementation runs into memory errors on larger matrix sizes.

5.2 TSQR Factorization

The parallel implementation of the TSQR factorization to find $(A^TA)^{-1}$ follows the approaches in [5] and [4]. We briefly provide a quick overview of the algorithm.
The process of performing $(A^TA)^{-1}$ using TSQR is broken into two main steps: 1. Compute a blocked TSQR, and 2. Compute an upper triangular inversion. In [4], there are two parallel implementations of a TSQR: the indirect and direct method. The direct and indirect TSQR operate similarly mathematically, but in [4] they differentiate both methods as they require different MapReduce operations. Here, we describe the high-level mathematical overview of the blocked TSQR. The direct and indirect TSQR perform the following operations on a tall-skinny matrix A:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_b \end{bmatrix} = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_b \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_b \end{bmatrix} = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_b \end{bmatrix} \widetilde{Q}\widetilde{R}$$

Where $Q\widetilde{Q}$ is the Q factor of A. Here, we perform a QR factorization each block of A, and another QR factorization on the stacked R factors from the second equality. An important note is that [4] also has a recursive implementation of the Direct TSQR. However, by analyzing their results, the recursive implementation is on par with the iterative implementation for the block sizes we use in our experiments. Therefore, we do not consider the recursive Direct TSQR.
Once the upper-triangular R factor has been computed, we follow [5] to perform an **in-place** blocked inversion. The algorithm

proceeds as follows:

Algorithm 1: In-place Blocked Upper Triangular Inversion

```
Partition $R = \begin{bmatrix} R_{TL} & R_{TR} \\ 0 & R_{BR} \end{bmatrix}$ where $shape(R_{TL}) = 0 \times 0$
determine block shape $b \times b$
while $shape(R_{TL}) \neq shape(R)$ do
    let $R_{TL} = R_{00}$
    let $R_{TR} = [R_{01} \quad R_{02}]$
    let $R_{BR} = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$ where $R_{11}$ is $b \times b$

    perform $R_{01} = -R_{00}R_{01}$
    perform $R_{01} = R_{01}R_{11}^{-1}$
    perform $R_{11} = R_{11}^{-1}$

    redefine $R_{TL} = \begin{bmatrix} R_{00} & R_{01} \\ 0 & R_{11} \end{bmatrix}$
    redefine $R_{TR} = [R_{02} \quad R_{12}]^T$
    redefine $R_{BR} = R_{22}$
end
```

By the time the **while** loop breaks, the blocked matrix R will now be its own inverted matrix. Once we have $R' = R^{-1}$, we simply compute a blocked matrix-multiplication: $(A^TA)^{-1} = R'R'^T$. Each individual operation is scheduled by NumS and thus Ray and is then distributed across all processors.

5.3 LU Decomposition

The LU decomposition algorithm for finding $(A^TA)^-1$ is similar to the naive implementation in that it starts with calculating A^TA. The LU decomposition algorithm used for distributed memory is a block-based recursive algorithm. In the algorithm, L^{-1} and U^{-1} are computed directly so that inversion at the end is not necessary at all.
At the base recursion level, the LU decomposition of block-sized matrices is computed, then both the L and U matrices obtained are inverted. Both of these operations happen on the same node that the data is stored in so there is no communication steps necessary. The only communication overall that is necessary is in the matrix multiplication performed on every recursion step. The LU decomposition algorithm can be found in [2].

5.4 Cholesky Decomposition

Based off [5], the distributed Cholesky decomposition algorithm can be done in-place, and involves computing the serial Cholesky and a solve operation on smaller blocks. There are three variants, and I choose variant 3, which only requires distributed addition, subtraction, multiplication on large matrices, and solve operations on block sized matrices.

6 Experiments

The ultimate goal of these experiments is to compare and contrast calculating $(A^TA)^{-1}$ using the different decompositions.

6.1 Types of Experiments

1. Naive inversion: Given $A \in \mathbb{R}^{n\times d}, n >> d$, find $(A^TA)^{-1}$ directly. Needs 1 Matrix Multiplication, and 1 Naive Inversion
2. Cholesky decomposition: Given $A \in \mathbb{R}^{n\times d}, n >> d$, perform A^TA , find $(A^TA)^{-1}$ using Cholesky. Needs 1 large matrix multiplication, 1 small Cholesky Decomposition, 1 small upper triangular inverse, and 1 small triangular matrix multiplication.
3. Cholesky decomposition: Given $X \in \mathbb{R}^{n\times n}, X \succ 0$, find X^{-1} using Cholesky. Needs 1 Cholesky Decomposition, 1 upper triangular inverse, 1 triangular matrix multiplication,.
4. LU decomposition: Given A, find $B = A^TA$ and then return $B^{-1} = L^{1-}U^{-1}P$. Needs 3 Matrix Multiplication and 1 LU decomposition.
5. TSQR decomposition: Given A, perform $A = QR$ using Direct TSQR, perform R^{-1} using upper triangular matrix inversion, and return $R^{-1}(R^{-1})^T$. Needs 1 TSQR factorization, 1 Upper Triangular Inverse, and 1 Matrix Multiplication.

6.2 Parameters of Nodes/Matrix Sizes to Use

6.2.1 Naive and LU

Due to the generality of the Naive and LU implementations, they ran much slower and took more memory. Because of this, the matrix sizes used were smaller and block sizes used were larger.

1. **Number of Nodes:** Intra (1)
2. **Row Sizes Used** $N = 2^k$ for $k = [10, 11, ..., 14]$
3. **Block Sizes to Use** $B = 2^l$ for $l = [9, 10, ..., 14]$

6.2.2 Cholesky

The Cholesky implementation is more optimized for finding $(A^T A)^{-1}$, and thus can operate on much larger matrices and smaller block sizes.

1. **Number of Nodes:** Intra (1)
2. **Row Sizes Used** $N = 1024 * 2^k$ for $k = [7, 8, ..., 12]$
3. **Block Sizes to Use** $B = 128 * 2^k$ for $k = [0, 1, ..., 4]$

6.2.3 TSQR

The Indirect and Direct TSQR methods assume that the block-size of the tall-skinny matrix A is **greater than or equal to** the number of columns in A. Because of this block-size constraint, we follow a slightly modified set of parameters for experimentation. We perform the same experiment for both indirect and direct TSQR.

1. **Number of Nodes:** Intra (1), Inter: 2
2. **Row Sizes Used** $N = 1024 * 2^k$ for $k = [7, 8, ..., 12]$
3. **Block Sizes to Use** $B = c2^k$ for $c = [1, 2, 4, 6, 8, 12, 16, 24]$ where k is selected beforehand.

6.3 Procedure

The following general process was repeated 5 times for each block size-matrix size combination:

1. Initialize NumS/Ray backend. For inter-node experiments, we launch Ray on multiple nodes and connect it to the master node's IP address.
2. Generate a random matrix A size N-by-$N/1024$
3. Start timer
4. Find $(A^T A)^{-1}$ from one of the above experiments
5. Perform the fetch operation on the remote matrix result to ensure all NumS operations have executed
6. Stop timer

The times are then averaged, outliers are removed, and they are plotted with error bars.

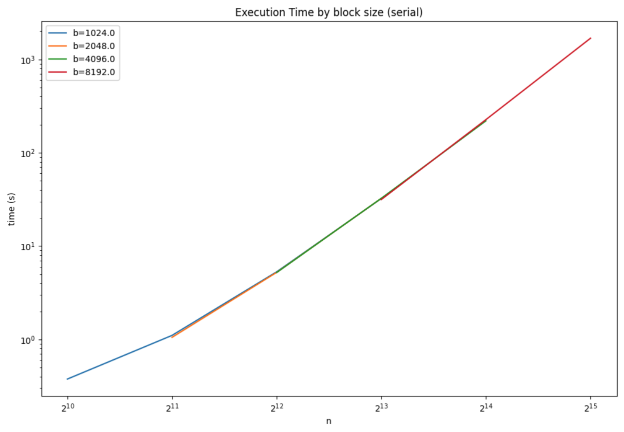
6.4 Machine Specs

The experiments were run on AWS machines, each equipped with a Platinum 8175M CPU @ 2.50GHz. There are 2 sockets with 16 cores each, multithreading up to 64 threads total. The total memory on each machine is 505.5 GB. When using multiple nodes, we used Ray and connected two nodes via their local IPs. Because the machines were reserved on a cluster, they are close together and the data does not have to make too many jumps to communicate between the two.

7 Results

7.1 Naive Implementation

Figure 1



The naive implementation has expected results (1). Block size has minimal impact on the execution time, in fact the different block sizes are almost indistinguishable.

7.2 TSQR Factorization

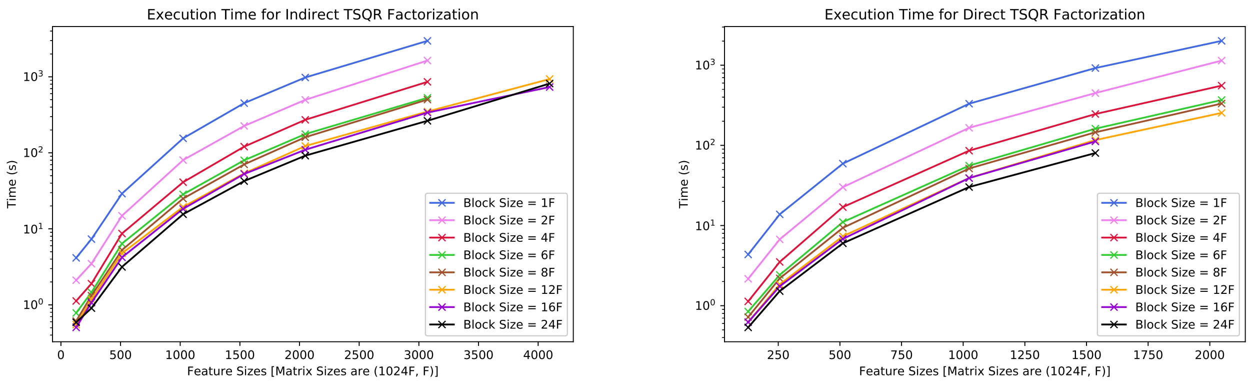
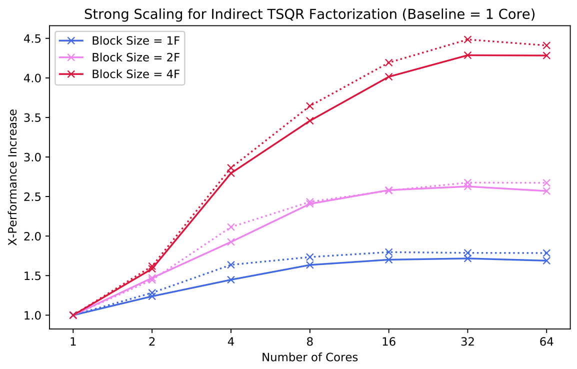


Figure 2: Execution Time Comparisons for Indirect and Direct TSQR factorizations. Discontinuities in the plots indicate that **out-of-memory** exceptions occurred during the experiment.

Between both figures in Figure 2, we can see that **larger block sizes** means **faster execution time**. As with the naive implementation, this idea is generally known because with smaller block sizes, we have more individual blocks to fit into memory. Since memory latency is a massive bottleneck, the time to retrieve many blocks outweighs the time to perform computation. Therefore, as we increase computational intensity by increasing the block size, and we decrease our execution time. Also, notice that for small block sizes and large matrix sizes, the experiment runs out of memory. But as we increase the blocksize to $B = 16F$ and beyond, the experiment does not exceed the node's memory limit.

However, with regards to indirect vs. direct TSQR, we came upon some surprising results. First, with direct TSQR, experiments were running out of memory when reaching a matrix size of $A : 1024 * 3072 \times 3072$. Then, by comparing both methods side-by-side, Indirect TSQR vastly outperforms Direct TSQR especially on large A matrices. This is interesting because we had an initial hypothesis that Direct TSQR would outperform Indirect TSQR. However, as indicated by [4], the reduction operation (QR factorization of stacked R factors) becomes a severe bottleneck in the Direct TSQR algorithm. Thus, we recognize Indirect TSQR as being the superior alternative.



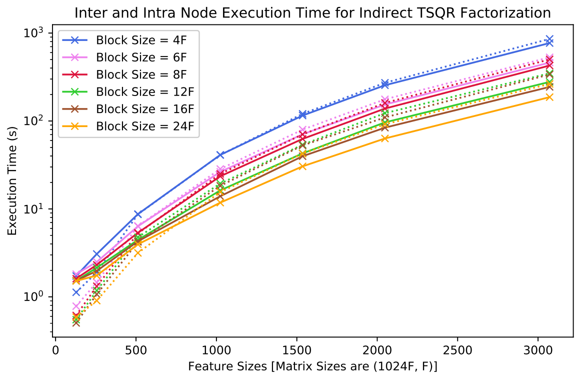


Figure 3: **Left:** Strong Scaling for Indirect TSQR. The solid line represents matrices with 1024 columns, and the dashed line represents matrices with 1536 columns. The number of rows is the number of columns scaled by 1024. **Right:** Comparison of execution times for inter and intra node experiments. The dashed line indicates single-node, and solid line indicates multi-node (2).

The strong-scaling with regards to the number of processors scales decently for 2 and 4 cores. As soon as we use 8 cores, though, the scaling starts to flatten. It seems as though as we increase the number of cores beyond 8, the efficiency of all cores starts to decrease. It isn't until we reach 64 cores that performance actually starts to decrease. We believe this is because our AWS machines are technically 32 core machines, and can have 64 cores with hyper-threading. However, hyper-threading did not prove to be helpful, and in fact hurt us more in the end.

By looking at the sub-figure on the right in 3, we found a few interesting results. The first, and the most obvious, is that multi-node execution did not outperform single-node execution for very small matrices. But, as the matrix A gets larger, notice how the dashed line soon becomes on top of the solid line indicating that multi-node execution provides benefit for large data. Then, we noticed that multi-node execution did not provide much performance gains for small block sizes. We relate this back to the idea presented in the previous section that larger block sizes are more beneficial. We especially see this fact when scaling to multi-node execution.

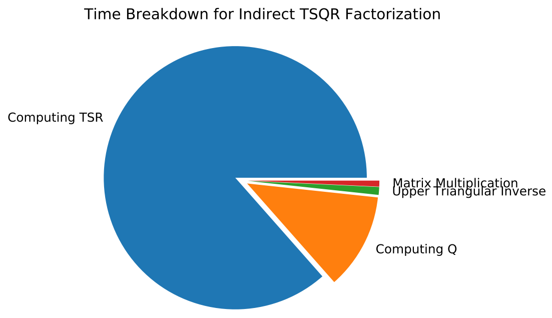
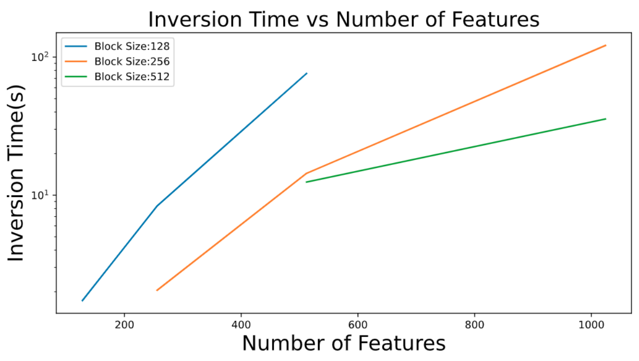


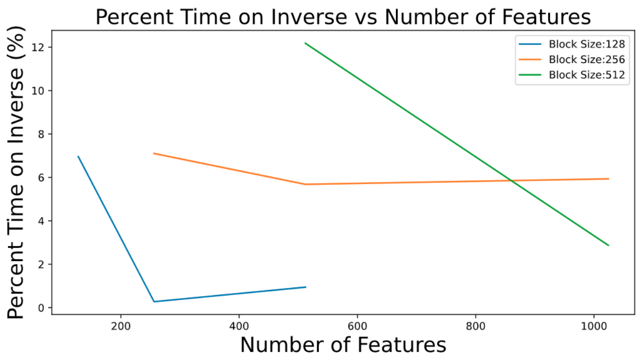
Figure 4: Pie chart showing the time breakdowns of Indirect TSQR.

Finally, we provide a time breakdown for the Indirect TSQR Factorization in 4. As we suspected, the TSR computation, where we perform a QR factorization on stacked R blocks, takes up most of the execution time. Interestingly, the upper triangular inversion and the final matrix multiplication step take approximately the same time. Although we do not display these results, we found that the naive inversion takes longer than the matrix multiplication. Moving forward, we could improve this operation by focusing most of our attention in optimizing the TSR computation.

7.3 Cholesky Decomposition - Tall Skinny Matrices



(a) Graph showing runtimes of Cholesky Inversion on Tall Skinny Matrices A of varying sizes. Here, $A \in \mathbb{R}^{n \times d}$, where d is the number of features and $n = d * 1024$. Due to the nature of the problem, we run out of memory at $d = 2048$



(b) Graph showing percentage of time spent on Cholesky Inversion on Tall Skinny Matrices A of varying sizes. We observe a loose trend where as the number of features gets larger, we spend a lower percentage of time on the cholesky inversion and more on the matrix multiplication $A^T A$

7.4 Cholesky Decomposition - Positive Definite Matrices

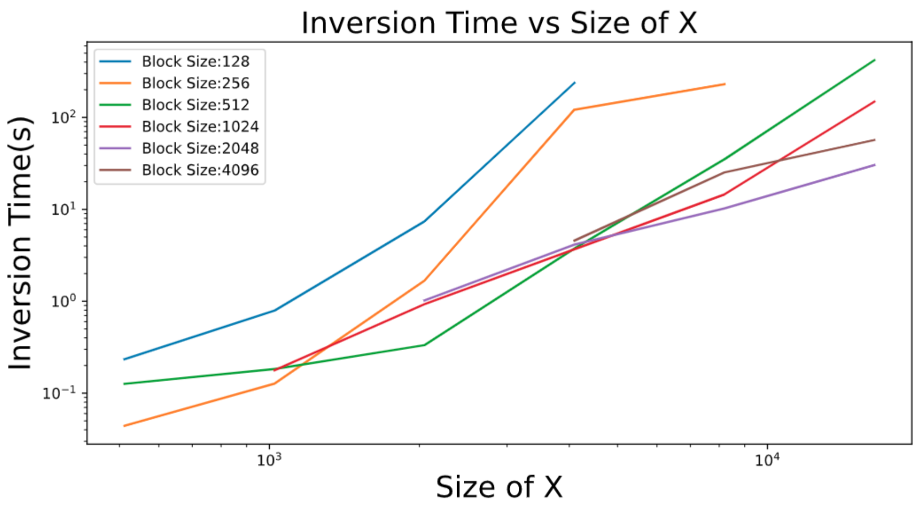
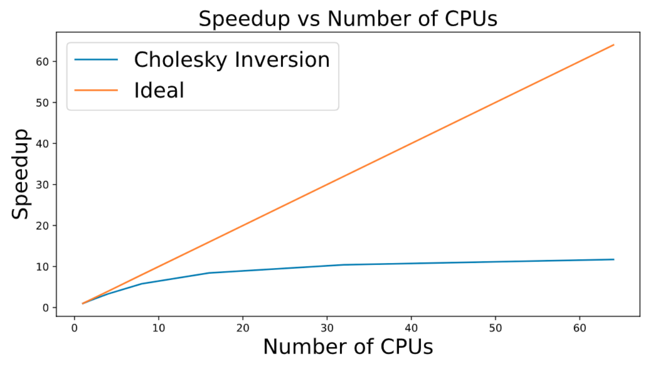
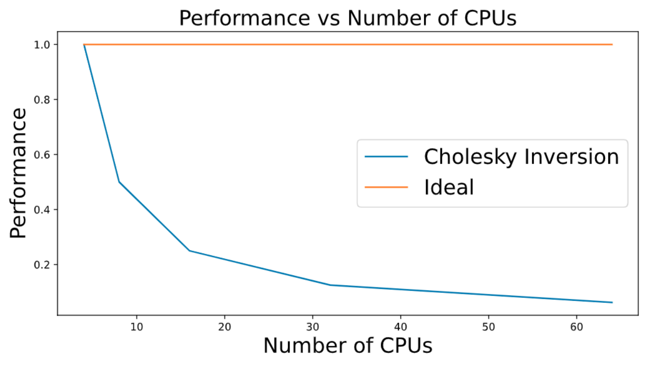


Figure 6: Graph showing runtimes of Cholesky Inversion on positive definite matrices X of varying sizes. We observe that for each size there is an optimal block size. For example, in the case of matrix of size 2048, we see that a block size of 512 outperforms other block sizes larger and smaller than it.



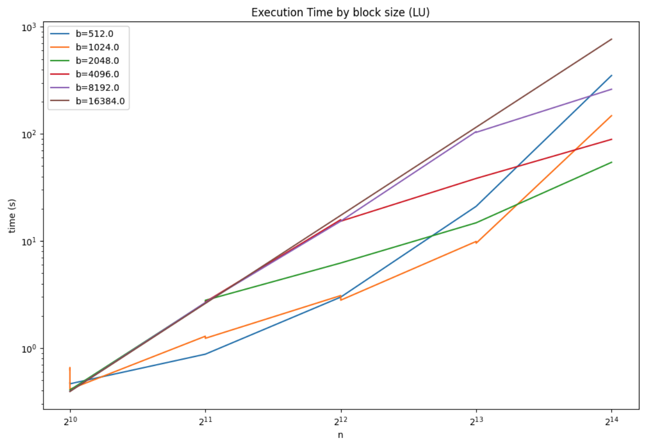
(a) Graph showing strong scaling of Cholesky Inversion on positive definite matrices X of varying sizes. Here we use a matrix size of 8192 and a block size of 1024. We observe that the maximum speedup we can get from the implementation of Cholesky is approximately 10x. According to Amdahl's law, this indicates that the unparallelizable code is approx 10% of the code, and to get better speedups we need to address the serial code.



(b) Graph showing weak scaling of Cholesky Inversion on positive definite matrices X of varying sizes. Here, performance is defined as the fraction of time observed versus the time we would expect given perfect scaling with the number of processors

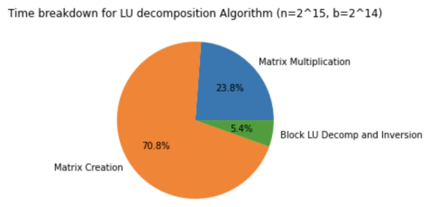
7.5 LU Decomposition

Figure 8

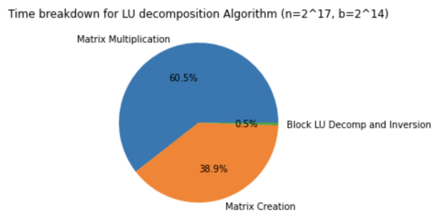


The LU Decomposition implementation has mixed results when it comes to block size. When the entire matrix fits into one block, the execution time is almost exactly linear with n, however it is always more efficient to have smaller block sizes than n. Different matrix sizes have different best block sizes. That probably has to do with the differences in how those matrices are broken up into blocks.

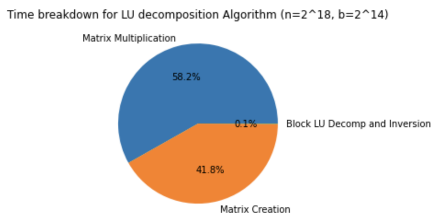
(a)



(b)

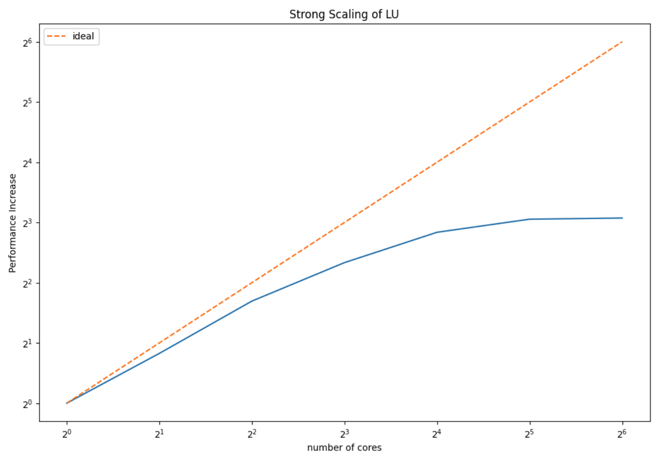


(c)



For small recursion depths, the time spent creating the intermediate matrices takes the most time while for larger recursion depths the matrix multiplication takes the cake. It's worth noting that as the recursion depth increases the relative time spent doing the actual LU decomposition of the blocks goes down

Figure 10



Finally, looking at the strong scaling of LU decomposition we get promising results. Because the hardware this was run on has only 32 physical cores (64 after hyperthreading), up to 32 processes we see consistent improvements in performance. While it is not quite ideal, this signifies that the algorithm could scale very well for large clusters.

8 Conclusion

Performing these experiments grants us insight on what algorithms to use when. For smaller matrices it seems that the naive implementation has the best results even if the matrix is distributed. The naive method cannot be used to compute matrices that are too large as it is possible that the single worker cannot fit the entire matrix in memory. Especially in clusters with less memory than ours (which is very high), this can make a big impact. For slightly larger matrices that are distributed, it is a good idea to use LU decomposition assuming you have many processes available. LU decomposition has good strong scaling compared to the naive implementation (which had reverse scaling), and because of that if you have the same number of CPUs as blocks it is a good choice. Both Cholesky and TSQR work very well on large matrices, and TSQR has good scaling so for big distributed matrices TSQR is a good choice for an inversion algorithm.

Another thing we noticed from the results is that the strong-scaling for LU decomposition and TSQR factorization are very similar. In fact, LU decomposition has a slightly better TSQR factorization. We believe that in this domain of parallelizing $(A^TA)^{-1}$, there is an increasing difficulty in trying to increase the efficiency of each worker. With very large A matrices, there is a lot of data to move around in memory. Thus, with an increasing number of processors, its very easy to get caught up in moving memory back in forth as the program attempts to perform blocked linear algebra operations.

References

[1] nums-project/nums: A library that translates python and numpy to optimized distributed systems code. `https://github.com/nums-project/nums`. (Accessed on 05/13/2021).

[2] Yang Liang, Jun Liu, Cheng Fang, and Nirwan Ansari. Spark-based large-scale matrix inversion for big data processing. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 718–723, 2016.

[3] Chandan Misra, Swastik Haldar, Sourangshu Bhattacharya, and Soumya Ghosh. Spin: A fast and scalable matrix inversion method in apache spark. pages 1–10, 01 2018.

[4] Austin R. Benson, David F. Gleich, and James Demmel. Direct qr factorizations for tall-and-skinny matrices in mapreduce architectures. *2013 IEEE International Conference on Big Data*, Oct 2013.

[5] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1), July 2008.

[6] HouZhen Wang, Yan Guo, and HuanGuo Zhang. A method of ultra-large-scale matrix inversion using block recursion. *Information*, 11(11), 2020.

[7] Aravindh Krishnamoorthy and Deepak Menon. Matrix inversion using cholesky decomposition. *CoRR*, abs/1111.4144, 2011.

[8] A. A. Trindade M. Lira, R. Iyer and V. Howle. Qr versus cholesky: A probabilistic analysis. *International Journal of Numerical Analysis and Modeling*, 13(1):114–121, 2016.