TSQR, A^tA, Cholesky QR, and Iterative Refinement using Hadoop on Magellan

Austin R. Benson
CS 267 Spring 2011 | arbenson@berkeley.edu

May 9, 2011

1 Introduction

1.1 The problem and project goals

The QR factorization is an important decomposition in the areas of scientic computing and numerical linear algebra. In particular, tall and skinny (TS) matrices, where the number of rows is much larger than the number of columns, have several applications. Communication-avoiding algorithms for QR decompositions of TS matrices have taken on their own development, and we refer to these algorithms in general as TSQR. Demmel et al. provide a thorough explanation of these algorithms in the parallel and sequential cases in [5]. In [3], Constantine and Gleich demonstrates how to effectively use MapReduce to perform TSQR. Furthermore, a previous CS 267 project [4] performed TSQR on Amazons EC2 service.

One component that is missing from MapReduce TSQR is how to effectively (i.e., with speed, accuracy, and a small amount of memory) generate Q explicitly. The implementation provided by [3] is natural for generating R as one can throw away intermediate Qs that are generated. So, what happens when someone needs Q? Since A = QR, we can just compute $Q = AR^{-1}$. However, this computation can have roundoff issues [6]. In this project, one of the goals was to look at iterative refinement as a method for generating Q. Another goal was to compare MapReduce implementations of TSQR to a Cholesky QR implementation. The first step of Cholesky QR is to compute A^tA, so much of the development was focused on the A^tA computation. Cholesky QR is less numerically stable because the A^tA computation squares the condition number of A, but it is known to be a faster algorithm.

While an explicit Q may not be needed for certain applications such as least-squares solving [1], there are applications where an explicit Q would be useful. One example is the TS SVD decomposition. Another example is computing the leverage scores of a matrixthe row norms divided by the number of columns. The leverage scores algorithm is outline below in Matlab (thanks to David Gleich for sending this code snippet):

- A = randn(1000, 10);
- [Q, R] = qr(A, 0); % generate explicit Q
- lscores = sqrt(sum(Q ^ 2 , 2))/size(Q, 2);

1.2 Tools

Apache Hadoop [12] was used as the MapReduce architecture, and experiments were conducted on Magellan, NERSC's cloud testbed [10]. Hadoopy [8] and Dumbo [2] were used as Python wrappers around the Hadoop environment. Hadoopy and Dumbo were both used for testing, while Dumbo was used for deploying jobs on Magellan. The map and reduce tasks were implemented using Python 2.6. NumPy [7] was used for the linear algebra functions.

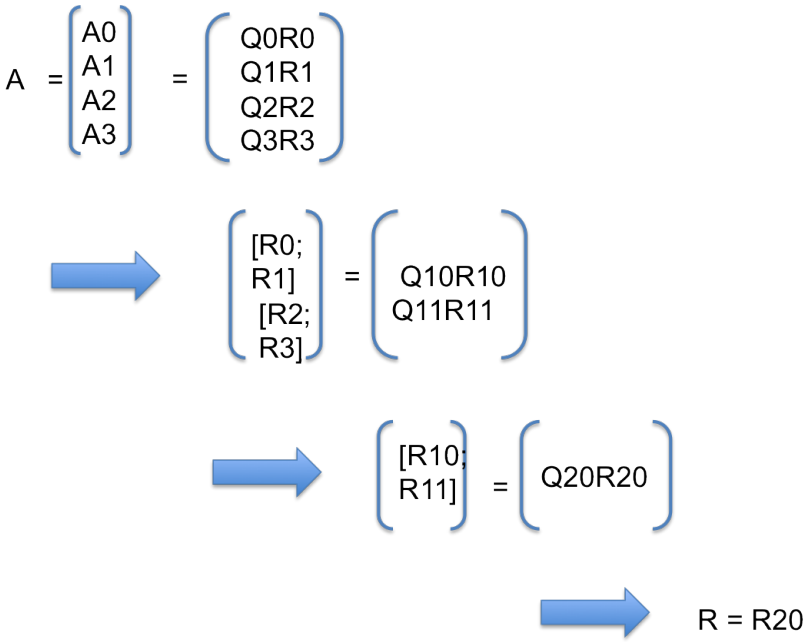

Figure 1: TSQR Algorithm.

2 Algorithms

The three main schemes used were TSQR [5], Cholesky QR, and iterative refinement. For TSQR, the matrix A is divided into several segments, and a QR decomposition is performed on each segment. The R matrices are accumulated and another QR decomposition is performed, as seen in Figure 1. For Cholesky QR, we let $B = A^tA$ and perform a Cholesky decomposition on B. Then $R = chol(B)$.

Above, we introduced the simple inverse computation for generating Q. If Q has lost orthogonality, we can attempt to regain orthogonality by performing a QR decomposition on Q. The refinement is $Q = Q_1R_1 \rightarrow Q = A(R_1R)^{-1}$. This can be applied iteratively until $||Q^tQ - I||_1$ is small enough.

3 Matlab motivation

To begin the project, a simple Matlab program was created to get an idea of how the orthogonality of Q changes with the different algorithms. Orthogonality of Q was tested by calculating $||Q^tQ - I||_1$. Simulation of the iterative refinement algorithm was performed with the qr() Matlab method. To simulate an R-solver that doesn't produce Q, the Q returned by qr() is thrown out. Q is then calculated by $Q = A/R$. Cholesky QR is used by Matlab's QR solver.

The test matrix was a rank-deficient matrix, which was used in order to exploit the numerical instability of computing R^{-1} and the Cholesky decomposition. Let A be mxn. We form A as follows:

- a = ones(m,1)
- B = rand(m, n-2)*scale, where scale is a scaling factor on the order of 10^9.
- A = [a B a]

Table 3 shows the results for the Matlab experiment.

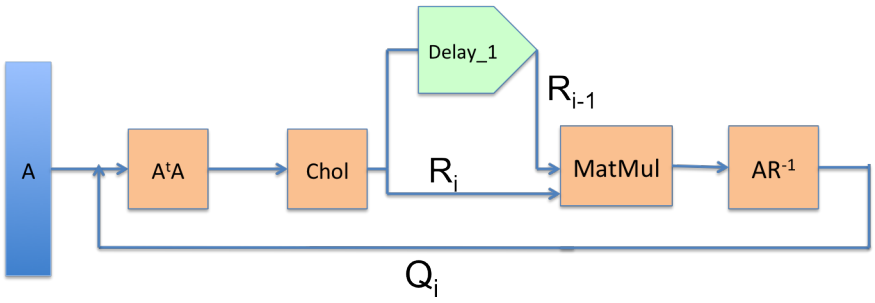

Figure 2: Iterative refinement pipeline.

m	n	norm after $Q = A/R$	norm after first refinement	norm after second refinement
2048	8	0.99944	2.11652e-13	1.33623e-15
4096	8	0.999997	3.67539e-12	2.67662e-15
8192	8	1	6.60773e-11	2.03091e-15
16384	16	0.999999	1.74721e-11	1.59887e-14
32768	16	0.999999	8.30842e-11	9.92541e-14
65536	16	1	2.51396e-09	1.99396e-13
131072	32	1	1.35582e-09	2.16716e-13
262144	32	1	4.84872e-08	8.02802e-13
524288	32	1	5.48159e-09	4.68958e-13
1048576	64	n/a	n/a	n/a

For the 1048576x64 case, the local machine did not have enough memory. This motivates the general need for more cloud-like environments. We can see that a Q = A/R solve produces a Q that is not that orthogonal. After one refinement iteration, the orthogonality is improved significantly. However, for larger matrices, $||Q^tQ - I||_1$ is still several orders of magnitude greater than machine epsilon. After a second refinement Q again becomes more orthogonal.

4 Implementation

4.1 Parallel patterns

4.1.1 13 motifs and parallel applications context

In terms of the 13 motifs, TSQR mostly falls into the dense matrix computations area. The scale of matrices that need MapReduce lead to HPC applications. Image processing, specifically PCA, can use TSQR ([1] and [3]), so practical applications exist for large-scale TSQR.

4.1.2 Structural patterns

The obvious structural pattern for the project is MapReduce, but this does not fully classify the project. In terms of generating Q, the iterator and pipe and filter structures can be applied to my TSQR strategies. Above, iterative refinement methods were discussed for generating Q, so the iterator pattern is a natural fit. For Cholesky QR, we perform A^tA first, then we perform Cholesky, then the resulting R is fed to the AR^{-1} and iterative refinement methods (see Figure 2).

4.2 Algorithm design

Other than the provided TSQR implementation [3], the central MapReduce implementations used were an A^tA and an AR^{-1} scheme. The A^tA scheme is illustrated in Figure 3 and is used for the Cholesky decomposition and the norm checking of Q^tQ. The AR^{-1} scheme is illustrated in Figure 4.

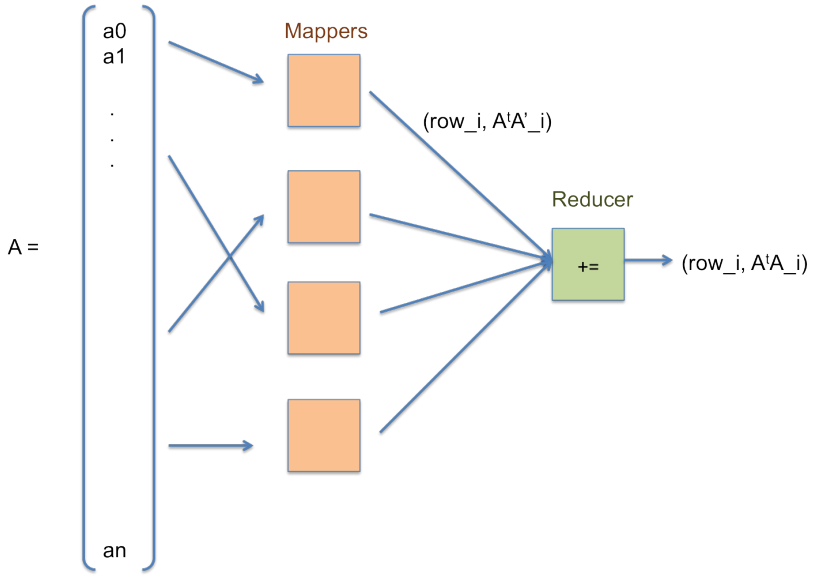

Figure 3: MapReduce A^tA scheme.

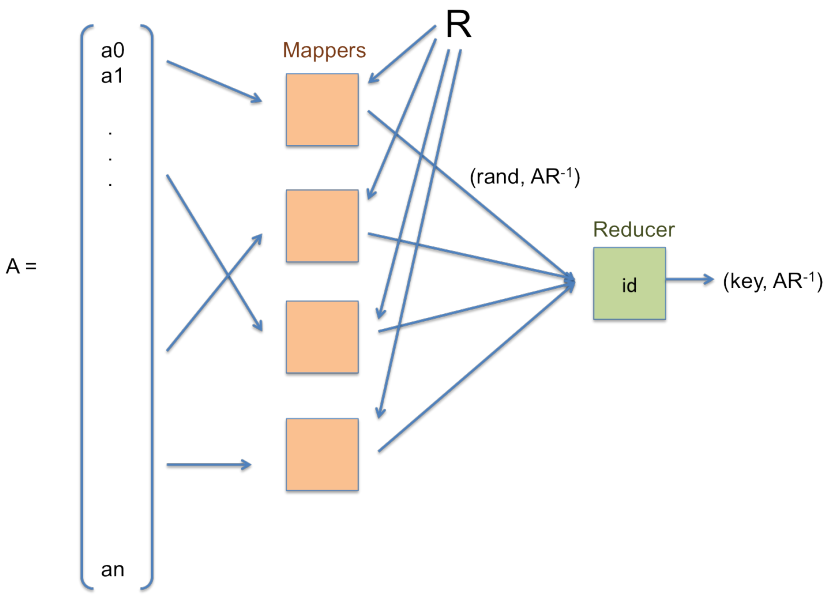

Figure 4: MapReduce AR^{-1} scheme.

The ARInv() process is outlined as follows:

input: file path to R matrix, stream of A matrix data
output: computed AR^{-1}

while not at end of R data file **do**
  read a row of input
  append row to R
end while

Compute $Rinv = R^{-1}$
Initialize buffer size = b
Initialize buffer Abuf

while there is A data **do**
  read a row of A
  append row to Abuf to empty
  if number of rows in Abuf >= b **then**
    output (random key, Abuf*Rinv) //flush buffer
    Set Abuf to empty
  end if
end while

output (random key, Abuf*Rinv) //flush buffer

4.3 Local testing

For local testing, Cloudera's Distribution including Apache Hadoop (CDH) was used. CDH provides an Ubuntu virtual machine for testing Hadoop applications. Scaling is difficult to do since the VM is simply run locally. However, it was much simpler to run basic tests and develop code in a local framework. There are a couple of tricky installation issues, and some instructions for setting up CDH and the necessary python frameworks are explained in Appendix A.

5 Magellan experiments

5.1 Machine specifications [10], [11]

Magellan is a distributed cloud environment with support for Hadoop, run by NERSC. The interface to Magellan runs through Carver, another one of NERSC's machines. Magellan typically has around 80 nodes available for use. Each node consists of two quad-core Intel Xeon 550 2.67 GHz processor. The Hadoop system supports up to 6 map tasks and 2 reduce tasks per node. The Carver nodes are connected with InfiniBand and have 32 Gb/sec point-to-point bandwidth.

5.2 R/W bandwidth communication costs and iterative refinement

Magellan uses SATA drives which can achieve bandwidth up to 3 GB/s. However, actual read and write bandwidths are much smaller. Experimental results show bandwidth $\approx$ 315 MB/s for reads and $\approx$ 157 MB/s writes. A 2.5 B x 50 matrix takes $\approx$ 1 TB on disk. For one node on Magellan, we get 6 map and 2 reduce tasks. Thus, with 10 nodes, the 1 TB matrix takes 52.9 seconds to read and 318.47 seconds to write. With 80 nodes (approximate full Magellan use), the read time would be 6.61 seconds and the write time 39.81 seconds.

Unfortunately, a write to disk for a large matrix at each step of iterative refinement is infeasible. The write times begin to take a large proportion of the MapReduce job time. We can combine steps of the

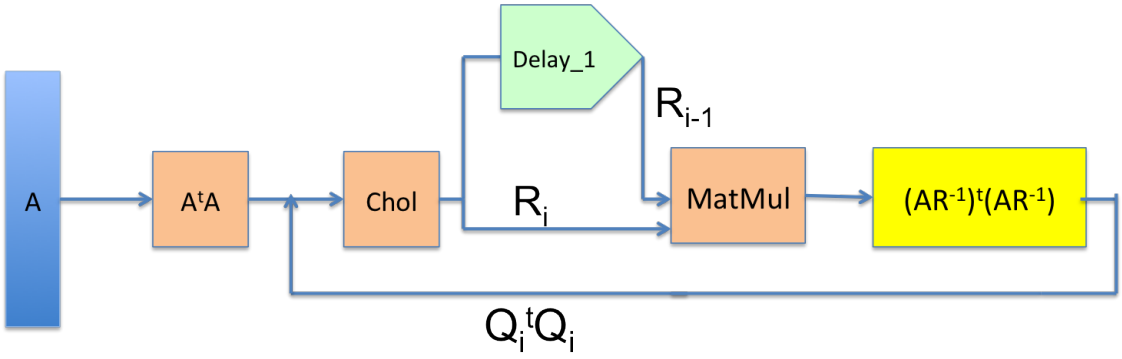

Figure 5: Communication-avoiding iterative refinement pipeline.

AR^{-1} and A^tA stages to only write out $O(ncols^2)$ per refinement. Figure 5 outlines the pipeline for this communication-avoiding variant of the pipeline in Figure 2.

An experiment was run to compare the total job completion times to the reduce task completion times for both versions of a stage in the iterative refinement. The reduction step for the naive version is an identity write to disk. Figure 6 presents the results. The 500M x 4 matrix was $\approx$ 80 GB on disk, which was on the smaller end of matrices that were tested. For larger matrices, the write-time would take significantly longer.

5.3 Scaling matrix size

Several experiments were conducted on Magellan which compared the running times of the TSQR scheme and the A^tA scheme. Unfortunately, NumPy does not have a BLAS interface to the symmetric rank-k update (SYRK) function. A simple model was used to predict the running time of A^tA with SYRK. The model worked as follows: Let t_c be the total time to completion (in seconds), t_m be the mapping time (in seconds), s be the size of the matrix on disk (in MB), and b the read bandwidth (in MB/second). The saved time is approximated by $t_{saved} = (t_m - b/s)/2$. Then the running time is predicted by $t_{c,syrk} = t_c - t_{saved}$. Experiments were conducted for ncols = 10, 50, and 200 while scaling the number of rows. TSQR was fastest in all trials except for the matrix size 1,000,000,000 x 10, where the predicted SYRK was fastest. Experiments were run with 77 nodes. The number of map tasks was 77*6 = 462 and the number of reduce tasks was min(ncols, 77*2). Figures 7, 8, and 9 show the results.

5.4 Buffer size tuning

In the A^tA scheme, we set a fixed buffer size and compute $A_{buf}R^{-1}$, where A_{buf} consists of a number of rows of A that fill the buffer. We can tune the buffer size to see what runs the fastest. Experiments were run with a smaller matrix (10,000,000 x 4) and a small number of nodes (2). The small number of nodes was used to keep the map time long in order to measure the total cost of the map stage in mapper-seconds. The same experiment was conducted for the AR^{-1}. Figure 10 shows the results of these experiments. In general, larger buffers were better. This makes sense because it reduces communication to the reduce tasks.

5.5 Iterative refinement

From discussions with Professor Demmel, Grey Ballard, and Michael Anderson, the de facto rule of iterative refinement is "twice is enough", meaning that performing two QR decompositions (one refinement) will produce a good error on $||Q^tQ - I||_1$. According to the Matlab experiments, the first refinement brings the error down to reasonable levels, as expected. To verify this for a larger matrix, a similar experiment was performed on a 500 million x 4 matrix. The process for generating A was:

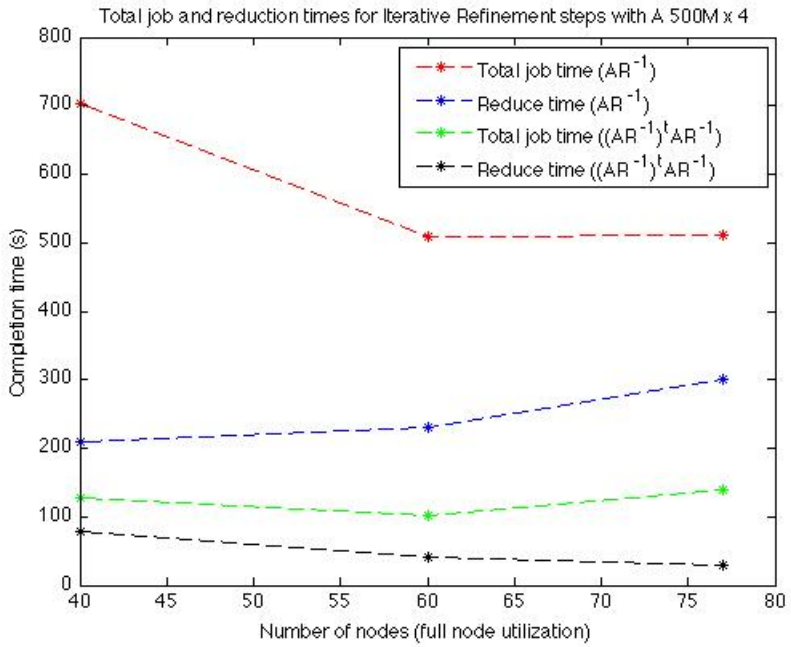

Figure 6: Total job and reduce step completion times for iterative refinement.

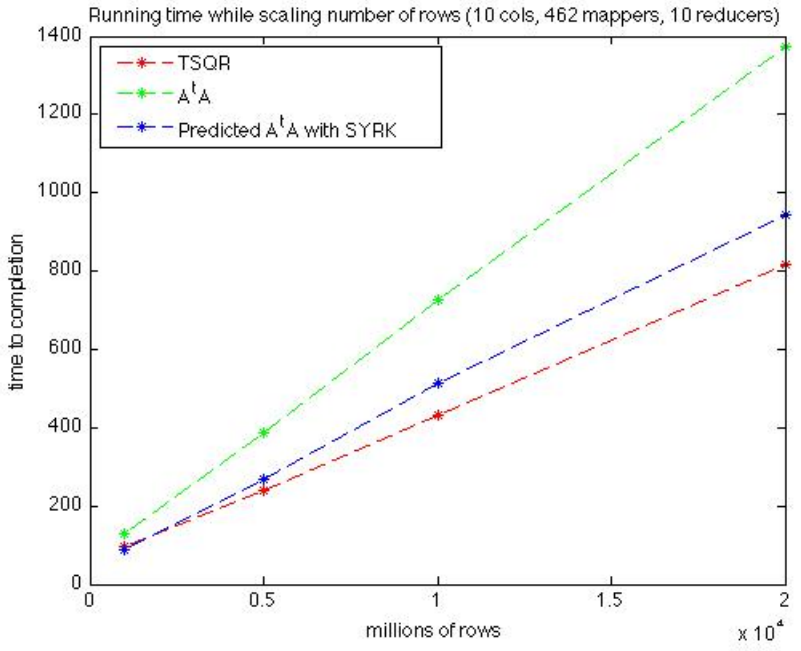

Figure 7: Scaling number of rows with fixed number of columns (10).

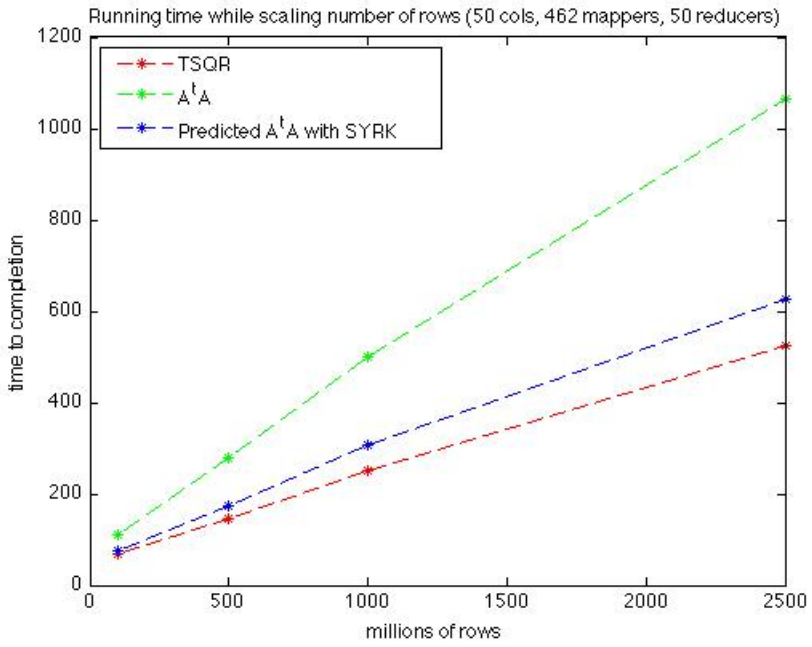

Figure 8: Scaling number of rows with fixed number of columns (50).

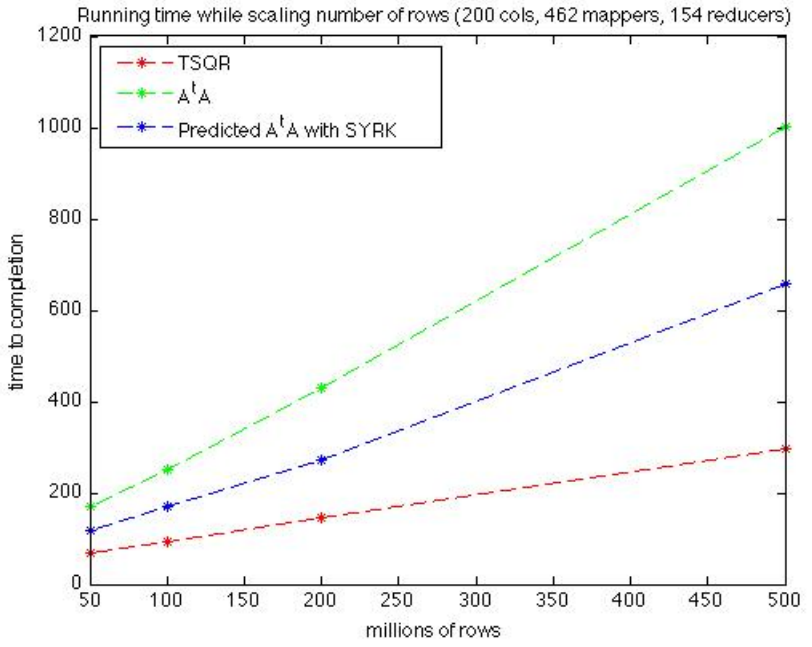

Figure 9: Scaling number of rows with fixed number of columns (200).

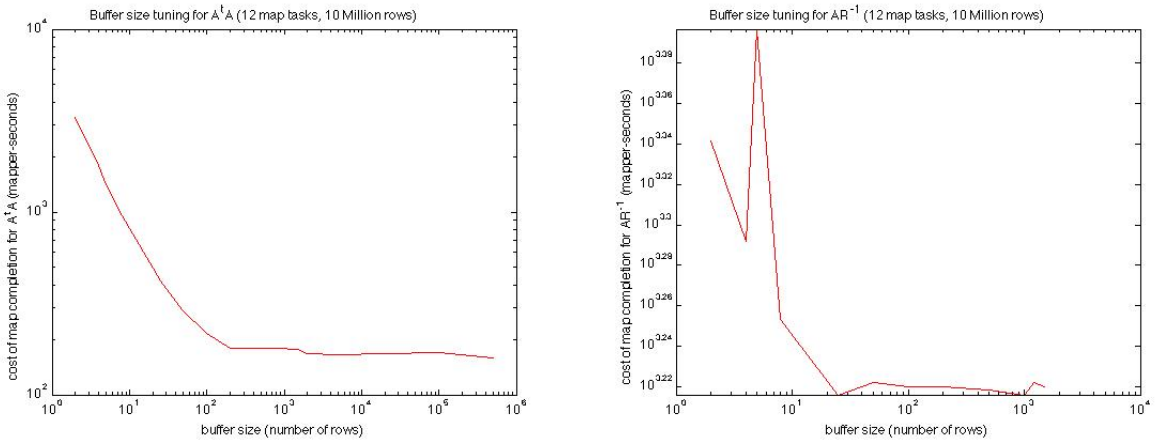

Figure 10: Left: Buffer size tuning for A^tA. Right: Buffer size tuning for AR^{-1}.

- $a = ones(4 * 10^6, 1)$
- $B = rand(4 * 10^6, 2) * 10^7$
- $c = [ones(3.8 * 10^6); zeros(2 * 10^5, 1)]$
- $A_{partial} = [a, B, c]$
- Feed $A_{partial}$ 125 times to Hadoop mappers as A

Figure 11 shows the experimental results for this matrix, and we see that the "twice is enough" rule applies. However, the norm with 0 iterations is still fairly small, $\approx 10^{-7}$. In future experiments, more ill-conditioned matrices will be tested.

6 Performance Analysis

6.1 Peak performance of Magellan

We can compute the peak performance of the system when performing arithmetic operations. Let n_s be the number of nodes available on the system. Then:

- peak performance = 2.67 GFlop/second/task * 8 tasks/node * n_s nodes = $21.36n_s$ GFlop/s.

Figure 12 shows the Flop/s performance of computing A^tA with larger matrices ($\approx$ 1-1.5 TB). With a smaller number of columns, the performance goes way down because of the serial row sums that occur in the reduction stage. With a larger number of columns, more reducers are used and the performance increases substantially. The tests were run with 77 nodes, so the peak performance is 21.36*77 = 1644.72 GFlop/s. We are far below this peak performance. However, there is a significant amount of communication overhead in reading large matrices from disk, writing matrices to disk, and communicating between mappers and reducers. In addition, we spend some time parsing strings in the reduction step.

For the matrices with 10 and 50 columns, we are not utilizing all of the reducers available because 10, 50 $<$ 77*2 = 154. For the matrix with 200 columns, all reducers are utilized. It is important to note that the peak performance is an idealized upper bound and it is extremely difficult to get close to peak performance on the Magellan MapReduce architecture. The Magellan Hadoop configuration is dedicated 6 mappers and 2 reducers per node. Because of this, we have 6 cores per node sitting idle during part of the reduction time. A more accurate upper bound would look at the shuffle, sort, and reduce stages of a reduction task. This is an area of future work.

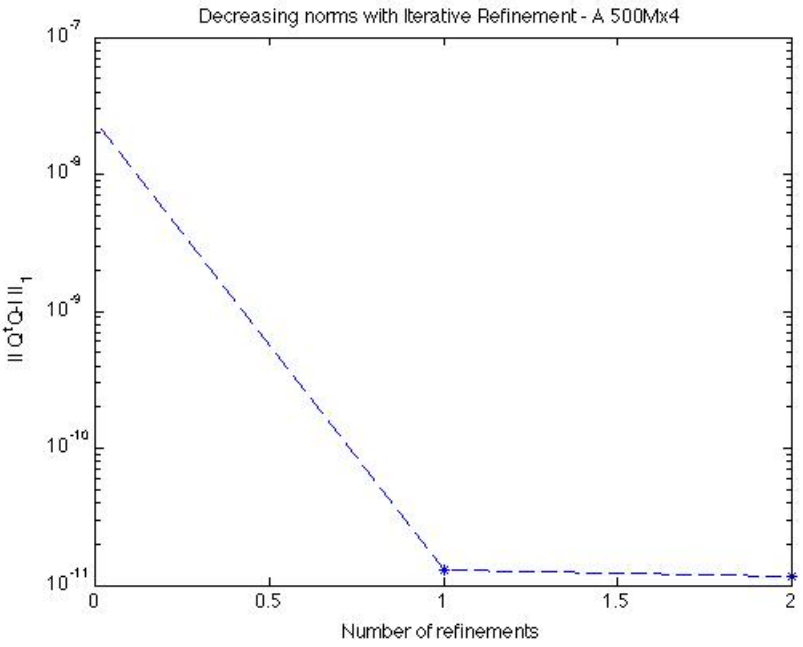

Figure 11: Iterative refinement norms for A 500,000,000 x 4.

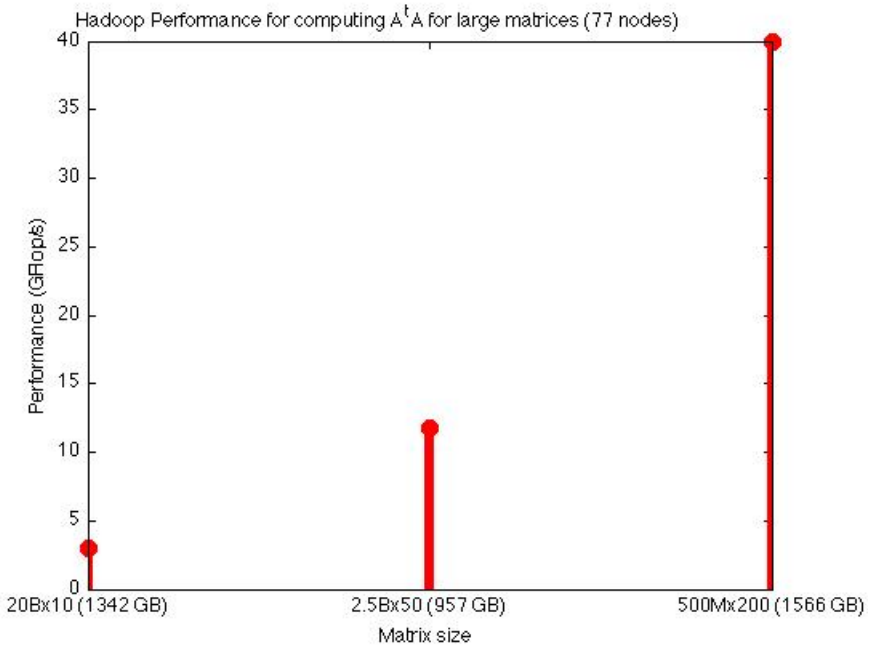

Figure 12: Performance for large matrices.

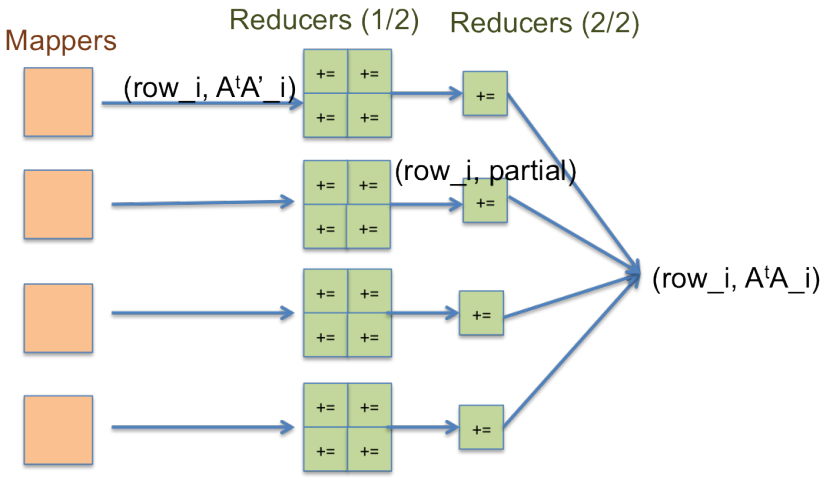

Figure 13: Optimized reduction steps for the A^tA scheme.

6.2 Reduction tree optimization for A^tA

Unlike MPI, reduction trees are not automatically formed in Hadoop. Experiments were run with a naive reduction scheme — one reducer per row. An optimization would be to add an extra step to the reduction tree. In the first stage, reducers compute partial row sums. In the second stage, the partial row sums are reduced to a final row sum.

More formally, suppose ncols $\leq$ nnodes*2 = nreducers. Suppose ncols divides nreducers evenly and let p = nreducers/ncols . We can reroute key-value pairs with key k uniformly to keys $k_0 \ldots k_{p-1}$. Reducers perform row sums and then reducer k_j re-emits (k, partial row sum) to another row sum reducer. Figure 13 illustrates the idea.

To model the performance improvement for this new scheme, data was collected for the reduce task times of a 500,000,000 x 4 matrix. Because the number of row sums is the same ($O(nmappers)$) for a fixed number of rows, the predicted time of a running time of a reduction step of a matrix with n columns was $t_n = t_4 * n/4$. The predicted running time is $n/(nnodes * 2) * t_n + o$, where o is the overhead of launching another reduction step. The $n/(nnodes*2)$ factor represents an idealized reduction load balance, which is an idealized distribution of rows to all of the available reduction tasks. Figure 14 shows the predicted speedups of the reduction steps. Predictions were calculated for o = 1, 5, and 10 seconds.

6.3 TSQR vs. A^tA

TSQR runs much faster than A^tA in almost all cases. There are several possible reasons for this. For one, the A^tA code is not optimized, and there is much room for improvement there. With some data structure changes, the reduction code can avoid some parsing and save time. In addition, reduction tree optimizations can be employed. Another important observation is that TSQR reduction steps are simply more QR decompositions, which can use BLAS 3. On the other hand, A^tA reductions are row sums, which use BLAS 1.

7 Future work

There are a number of areas to improve upon and experiment with the MapReduce schemes described above. The number one priority for the A^tA scheme would be to implement SYRK. There are a couple of options for this. First, the cvxopt package has a SYRK interface. However, the library has to link to ATLAS and BLAS, which can be a pain. Second, more advanced matrix algebra can be done by splitting a given a block

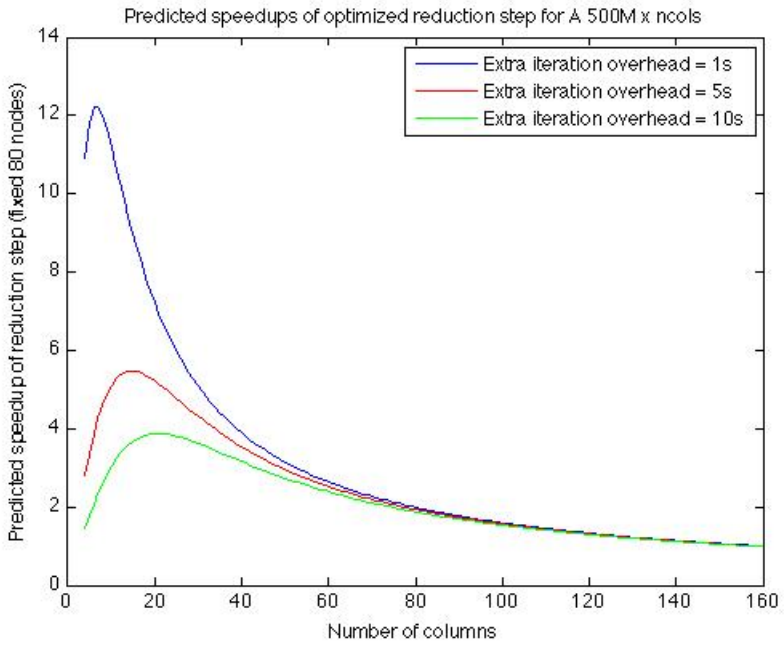

Figure 14: Predicted speedups of reduction time with optimized tree reduction for A^tA.

of A into 4 pieces and computing the upper left, upper right, and lower right quadrants of the resulting A^tA matrix. Another priority for the A^tA scheme is the aforementioned reduction tree optimization.

For TSQR, there still needs to be an implementation for the communication-avoiding iterative refinement. This should follow the Cholesky QR iterative refinement scheme closely. Development has started on this but was not yet completed by the end of the semester.

In addition, there are several other ideas that can be used:

1. Improve data structures used for A^tA to avoid string parsing
2. On Magellan, use the Hadoopy wrapper (written in Cython) instead of Dumbo
3. More detailed analysis of peak performance of Magellan
4. Different node configurations in terms of dedicated mappers and reducers
5. Run on other cloud environments, e.g., EC2
6. Run on cloud infrastructures, e.g., Spark [9]
7. Use C++, Java, or other programming language implementations (Spark uses Scala).
8. Analyze data from example applications

8 Conclusion

The cloud is an exciting environment for numerical linear algebra. As larger data sets emerge and we look to compute larger matrices, it is necessary to scale to bigger machines. Cloud computing provides a dynamic environment to handle such scaling of numerical linear algebra. In addition, Hadoop provides a convenient

framework for rapid development. The TSQR algorithm is particularly fit for MapReduce schemes. With the correct optimizations, Cholesky QR could become a competitive algorithm and produce results faster in certain situations.

9 Appendix A: Setting up CDH

Unfortunately, CDH is not as simple as it claims to be for running python-driven Hadoop jobs. This appendix aims to guide others in setting up CDH for python-driven Hadoop work. After installing the CDH VM, you need to update Ubuntu. This allows us to use apt-get install. Without the update, apt-get install will produce IP address errors. From the terminal run:

- sudo do-release-upgrade

The above upgrade may take around a half hour. After this upgrade, we should have access to gcc and various other tools that we need. Make sure that we have gcc and git:

- sudo apt-get install gcc
- sudo apt-get install git

We will git the following software packages:

1. git clone git://github.com/numpy/numpy.git numpy → The numerical methods package.
2. git clone https://github.com/bwhite/hadoopy/ → One of the python hadoop packages.
3. git clone https://github.com/klbostee/dumbo.git → The other python hadoop package.
4. git clone https://github.com/dgleich/mrtsqr.git → David Gleich's MRTSQR package.

(1), (2), and (3) will need to be installed. First, we will install easy_install, which, by its namesake, will let us easily install these packages. Get the setuptools egg file from:

- http://pypi.python.org/pypi/setuptools#files

Make sure that the setuptools egg file matches the version of python you want to use! This will prevent annoying build/install errors in the future. In fact, I recommend deleting the other python directories in /usr/include to make future building errors more obvious. Run the following to install easy_install:

- sudo sh setuptools-0.6c11-py2.6.egg (or whatever egg file you downloaded)

Run the following commands from the directory above the package directory:

- sudo easy_install numpy (this may produce some warnings, but it should be OK)
- sudo easy_install hadoopy
- sudo easy_install dumbo

10 Appendix B: Setting up Magellan

We need to get around the permissions on Magellan in order to setup Dumbo. To accomplish this, we use Python's virtual env module. First, change shells to bash and load the modules:

- bash
- module load tig
- module load hadoop/magellan
- module load python

Now, get the virtual env software:

- wget http://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.6.tar.gz#md5=b1c585e0bbe4fca56bfa0b34aff8
- tar xzvf virtualenv-1.6.tar.gz
- cd virtualenv-1.6
- python setup.py install –prefix $HOME/local

Now, get all of the other software through git:

1. git clone https://github.com/klbostee/dumbo.git → The other python hadoop package.
2. git clone https://github.com/dgleich/mrtsqr.git → David Gleich's MRTSQR package.

NumPy should be available in the python 2.6 module with the "module load python" command. Now we need to setup a virtual environment to run Dumbo.

- cd virtualenv-1.6/ENV/bin/
- python easy_install path/to/dumbo .

Finally, we run our virtual environment. We need to run this command each time we log in to Magellan. To do this, use the command:

- . virtualenv-1.6/ENV/bin/activate

We will need to activate the virtual environment each time we log in to Magellan. I suggest making a bash script which will load the modules and activate the environment. In addition, we need to setup the .dumborc file to run Dumbo. For convenience, the necessary lines are listed below (thanks to David Gleich for sending me this):

- [common]
- python: /usr/common/usg/python/2.6.4/bin/python
- [hadoops]
- nersc: /usr/common/tig/hadoop/hadoop-0.20.2+228/
- [streaming_nersc]
- pypath: .

11 Acknowledgements

Thanks to David Gleich for providing TSQR code, project ideas, and help with the software engineering in Hadoop. Thanks to the GSIs—Grey Ballard and Michael Anderson—for several discussions on QR decompositions. Thanks to Prof. Demmel and Prof. Yelick for the CS 267 course and thanks to Prof. Demmel for project ideas and mathematical discussions.

References

[1] Michael Anderson and Stephan Ritter. *Parallel Robust PCA*. CS 267 final project, Spring 2010, UC-Berkeley.
⟨ http://www.cs.berkeley.edu/ agearh/cs267.sp10/final.html ⟩

[2] Klaas Bosteels. Dumbo. ⟨ https://github.com/klbostee/dumbo ⟩

[3] Paul G. Constantine and David F. Gleich. *Tall and Skinny QR factorizations in MapReduce architectures*. MAPREDUCE 2011.

[4] Benjamin Dagnon. *TSQR on EC2 Using the Nexus Substrate*. CS 267 final project, Spring 2010, UC-Berkeley. ⟨ http://www.cs.berkeley.edu/ agearh/cs267.sp10/final.html ⟩

[5] James Demmel, Laura Grigori, Mark F. Hoemmen, and Julien Langou. *Communication-optimal parallel and sequential QR and LU factorizations*. UCB/EECS-2008-89. August 2008.

[6] James Demmel. Applied Numerical Linear Algebra. SIAM. 1997.

[7] Eric Jones, Travis Oliphant, Pearu Peterson and others. *SciPy: Open Source Scientific Tools for Python*. 2001. ⟨ http://www.scipy.org ⟩

[8] Brandyn A. White. Hadoopy. ⟨ https://github.com/bwhite/hadoopy ⟩

[9] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker and I. Stoica. *Spark: Cluster Computing with Working Sets*. HotCloud 2010

[10] Magellan at NERSC. ⟨ http://magellan.nersc.gov ⟩

[11] Carver at NERSC. ⟨ http://www.nersc.gov/users/computational-systems/carver/configuration/ ⟩

[12] Hadoop version 0.21. ⟨http://hadoop.apache.org/⟩