MapReduce TSQR: using Apache Hadoop for large-scale QR decompositions of tall and skinny matrices

Austin R. Benson
arbenson@berkeley.edu
University of California, Berkeley
Math 221: Advanced Matrix Computations, Fall 2011
Instructor: Professor James Demmel

November 27, 2011

1 NOTE

THIS IS A WORK IN PROGRESS. I am hosting this here as I have provided a link to it on my Ph.D. applications. This work will be completed by December 15, 2011.

2 Motivation

The QR decomposition is one of the most common matrix decompositions in scientific computing. QR is one of the most common methods for solving least squares problems (cite). In many situations, we get over-constrained problems, i.e., when the number of rows is larger than the number of columns. We refer to these matrices as tall and skinny (TS) and thus we have tall and skinny QR decompositions, the algorithm for which will be referred to as TSQR. In this project, we look at two TSQR algorithms, one that uses Householder QR and one that uses Cholesky QR. See sections 3 and 4.

With current trends towards "large data", we face a challenge in scaling our software to handle more data. MapReduce is a programming model for application developers to scale code. Apache Hadoop (cite) is a software framework for running MapReduce programs. Companies like Google, Facebook, IBM, and Microsoft have adopted these frameworks [10]. A major reason for the adoption of Hadoop and similar software is reliability: failed nodes in the cluster do result in a failed job. We investigate the relationship between performance and faults in section 9.

In this project, we use Hadoop for computing TSQR decompositions using NERSC's Magellan cloud testbed [12]. The ideas and code for this project are derived from similar work by Constantine and Gleich [2] [3]. We investigate large matrices on the order of 500 GB in size, which necessitates a cluster, rather than a single computer. Sections 5,6, 8, and 11 discuss implementation. Sections 8, 9, and 12 detail performance results. Lastly, section 7 looks at numerical stability.

3 Householder TSQR

The $A = QR$ factorization factors an m by n matrix A into an m by n orthogonal matrix Q and an n by n upper triangular matrix R. Let A be a $4n$ by n matrix. We formulate the Householder TSQR as follows:

$$A = \begin{pmatrix} A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \end{pmatrix} = \begin{pmatrix} Q_1R_1 \\ \hline Q_2R_2 \\ \hline Q_3R_3 \\ \hline Q_4R_4 \end{pmatrix} \rightarrow \begin{pmatrix} R_1 \\ R_2 \\ \hline R_3 \\ R_4 \end{pmatrix} = \begin{pmatrix} Q_5R_5 \\ \hline Q_6R_6 \end{pmatrix} \rightarrow \begin{pmatrix} R_5 \\ R_6 \end{pmatrix} = Q_7R_7.$$

Each intermediate Q_iR_i factorization uses a Householder QR decomposition. A_i and Q_i are n by n for $i = 1, 2, 3, 4$, Q_i is $2n$ by n for $i = 5, 6$ and Q_7 is $4n$ by n. R_i is n by n for $i = 1, 2, ..., 7$. The QR decomposition is given by $A = QR_7$, where Q is constructed from the Q_i as follows:

$$Q = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{pmatrix} \begin{pmatrix} Q_5 & \\ & Q_6 \end{pmatrix} Q_7$$

The horizontal lines in the factorization represent opportunities for parallelism. The intermediate Q_iR_i factorizations can be done in parallel at each step. This leads to a natural *logp* reduction, where A is split into $A_1, ..., A_p$.

4 Cholesky TSQR

The $A = LL^T$ Cholesky factorization factors a symmetric positive definite matrix A into a lower triangular matrix multiplied by its transpose. We use the QR factorization to solve the normal equations:

$$x = (A^TA)^{-1}A^Tb = (R^TQ^TQR)^{-1}R^TQ^Tb = (R^TR)^{-1}R^TQ^Tb = RR^{-T}R^TQ^Tb = R^{-1}Q^Tb \quad (1)$$

Equation 1 leads to a back solve of $Rx = Q^Tb$.

To solve the least squares problem with Cholesky, we note that A^TA is symmetric and assume A^TA is positive definite and write:

$$x = (A^TA)^{-1}A^Tb = (LL^T)^{-1}A^Tb \rightarrow LL^Tx = A^Tb \quad (2)$$

Equation 2 leads to a forward solve $L(L^Tx) = Ly = A^Tb$ and then a back solve $L^Tx = y$. In essence, Cholesky QR gives us the same information as Householder QR. However, computing A^TA squares the condition number of the problem, so Cholesky QR is less stable. We will investigate the trade-offs in section 11.

5 Applying the MapReduce model to TSQR

In general, the Map Reduce model follows an iterative three-step process using key-value pairs. Let k_i and v_i represent keys and values, respectively, for $i = 1, 2, 3$. Then we can represent the three-step process by the composition of three functions:

$$(k_1, v_1) \xrightarrow{map} (k_2, v_2) \xrightarrow{combine} (k_2, v_2) \xrightarrow{reduce} (k_3, v_3)$$

For simplicity, we will ignore the combine step and only consider the map and reduce steps:

$$(k_1, v_1) \xrightarrow{map} (k_2, v_2) \xrightarrow{reduce} (k_3, v_3)$$

v_2 is treated as a list of values emitted by the map function all corresponding to a given key.

For MapReduce TSQR (MRTSQR), our matrices will be stored as key-value pairs with random numbers a key and a matrix row (or several matrix rows) as a value.

5.1 Householder MRTSQR

Our Householder TSQR uses both the map and reduce steps to perform QR decomposition on leftover intermediate R_i. The functions look like:

$$(k, A_i) \xrightarrow{QR} (k, R_i) \xrightarrow{QR} (k, R_i).$$

Each map and reduce task outputs one intermediate R_i. In order to combine the results from the distributed reduce tasks, we introduce a second iteration:

$$(k, A_i) \xrightarrow{QR} (k, R_i) \xrightarrow{QR} (k, R_i) \xrightarrow{identity} (k, R_i) \xrightarrow{QR} (i, R).$$

The first iteration will use the maximum available mappers and reducers. The second iteration uses the maximum available mappers and a single reducer. A visual interpretation of the Householder MRTSQR job is given in 1

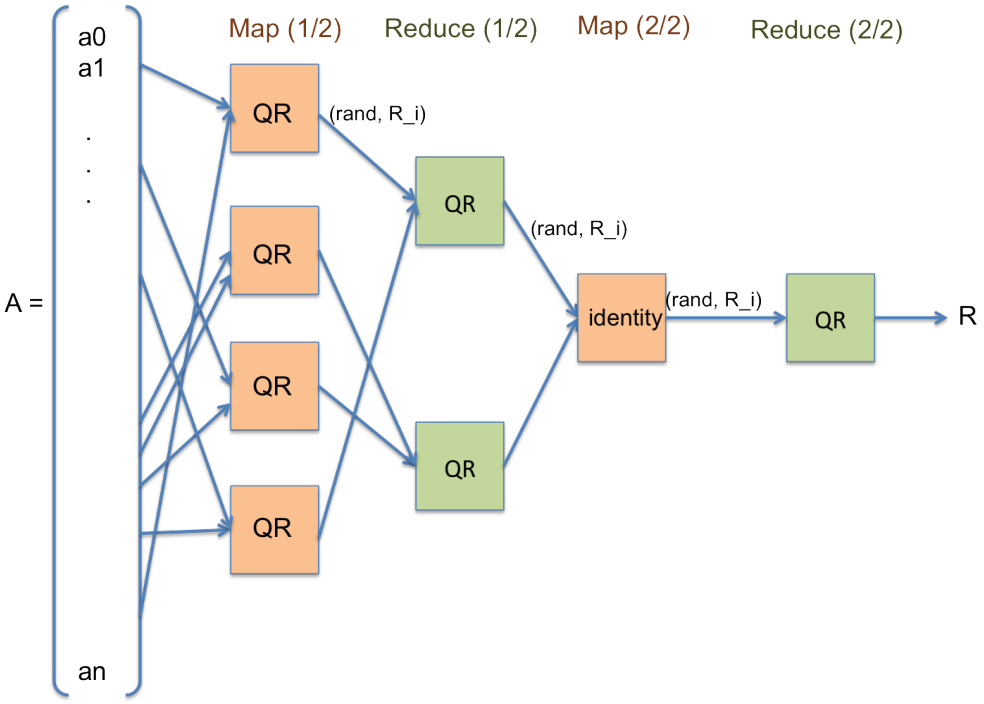

Figure 1: Householder MRTSQR scheme

5.2 Cholesky MRTSQR

The Cholesky TSQR code provides more variety to the functions:

$$(k, A_i) \xrightarrow{A^T A} (i, (A^T A)_i) \xrightarrow{rowsum} (i, (A^T A)_i) \xrightarrow{identity} (i, (A^T A)_i) \xrightarrow{Cholesky} (i, R_i).$$

Theoretically, we expect a *logp* reduction tree on the rowsums to get the best performance. However, the Hadoop overhead at each iteration makes this less appealing. We get a local $A^T A$ from each map tasks, so we only need to perform $\#(maptasks)$ vector sums per row. In our working examples, $\#(maptasks)$ is at most a few thousand.

Unlike QR, we have steps with non-trivial keys. The row sum reducers must combine all records corresponding to a given row. As in QR, the first iteration will use the maximum available mappers and reducers, and the second iteration uses the maximum available mappers and a single reducer.

A visual interpretation of the Cholesky MRTSQR job is given in 2.

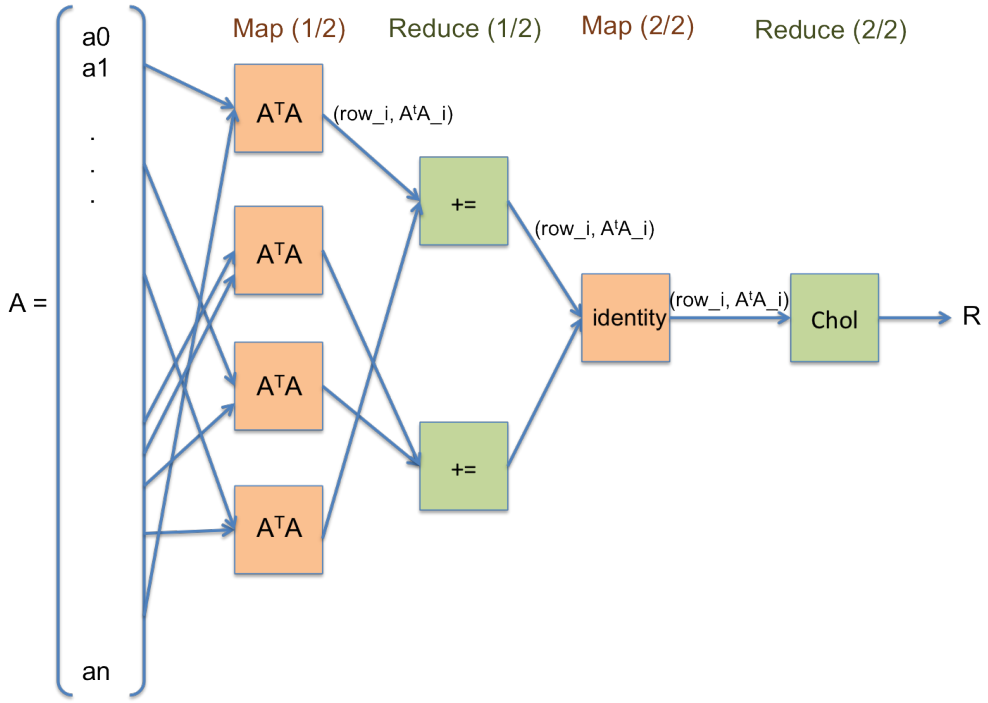

Figure 2: Cholesky MRTSQR scheme

6 Tools

6.1 Software

We employ a number of software tools. They are listed here with references for more information.

1. Apache Hadoop version 0.20.2 [9]
2. NumPy, a Python math library [6]
3. Dumbo, a Python framework for running Hadoop jobs (publicly available at [8])
4. Python TypedBytes support (publicly available at [7])
5. MRTSQR scripts (publicly available at [3] and [4])

6.1.1 Why so much Python?

Typically, high performance computing employs lower-level languages like C/C++. In this project, Python was used extensively for several reasons. First, prototyping MapReduce schemes is much easier. Full implementations of the Cholesky and Householder MRTSQR codes fit in around 100 lines of code. Second, NumPy provides interfaces to BLAS, so the computation time does not differ that much (about a factor of 3, see section xx). An investigation of the performance of C++ implementations against Python implementations for Hadoop streaming is in section 12.4.

6.2 Hardware

All experiments in the project were run on the Magellan cluster ([12]) at the National Energy Research Scientific Computing Center (NERSC). Magellan consists of an 80-node Hadoop cluster. Each node has two quad-core Intel Nehalen 2.67 GHz processors and a 1 TB (local) SATA disk. The network topology consists of local fat-trees with a global 2D mesh with InfiniBand fibre optic cables. The Hadoop cluster dedicates 6 cores/node to map tasks and 2 cores/node to reduce tasks. The theoretical peak performance of a node is about 85 GFlops [13]. Thus, the theoretical peak of the system is 80*85 = 6800 GFlops. We investigate fraction of peak performance in section 12.1.

7 Numerical stability

In order to test numerical stability, we can strategically pick our matrix A to control its condition number. We start with a m by n matrix A' with matrix entries taken from $N(0,1)$. Then we compute $A' = Q'R'$. Q' also has matrix entries from $N(0,1)$. Let D be a diagonal matrix with $D_{jj} > 0$ and let $A = Q'D$. $Q'D$ is the QR factorization of A and $Q'DI_n$ is the singular value decomposition of A. The condition number of A is given by $\kappa(A) = \frac{\sigma_{max}}{\sigma_{min}} = \frac{\max_j D_{jj}}{\min_j D_{jj}}$ [11].

For numerical stability experiments, we simply vary the matrix D to control the condition number. We have two metrics for measuring how accurate our QR decomposition is:

1. $||Q^TQ - I_n||_2$: how orthogonal is the computed Q?
2. $||A - QR||_2$: how close to A is the computed factorization?

To get Q, we form R^{-1} and take $Q = AR^{-1}$. Computing R^{-1} is not backwards stable (cite).

7.1 MapReduce scheme for $Q = AR^{-1}$

Our MapReduce scheme for $Q = AR^{-1}$ assumes that we have already computed R. The R matrix gets distributed to every map task, and the rows of A get distributed to the mappers:

$$(k, A_i) \xrightarrow{AR^{-1}} (k, Q_i) \xrightarrow{identity} (k, Q_i)$$

We only need one iteration because the entire computation is embarrassingly parallel, and we can leave the resulting Q matrix splayed across multiple directories. A visual representation of the scheme is in figure 3.

8 Data serialization optimizations

8.1 Techniques

8.2 Performance

9 Fault tolerance

The central benefit to scientific computing in cloud environments is stability. In Hadoop, failed tasks are restarted 3 times by default, and this is a tunable parameter (cite). This provides the user

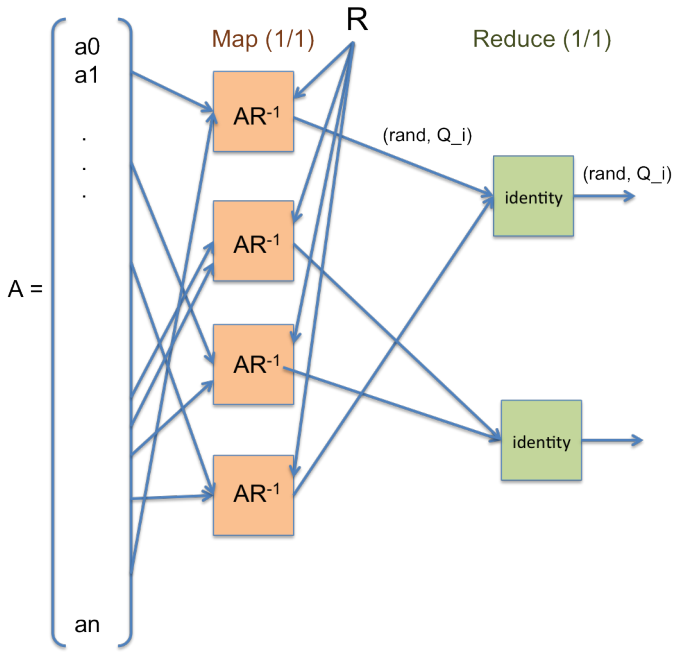

Figure 3: $Q = AR^{-1}$ scheme

with the comfort that a single point of failure will not derail entire jobs. Furthermore, consistently failing nodes are "blacklisted" and not included in future jobs.

With supercomputers moving towards exascale, fault tolerance is becoming increasingly important (cite). We hope that the experiments discussed in this section reinforce the idea that we do not need to worry too much about faults affecting performance. To benchmark performance penalties, random software faults were introduced in the Householder TSQR and we evaluate the performance as the fault rate increases. Faults were introduced in three different ways: (1) in both map and reduce tasks, (2) only in map tasks, and (3) only in reduce tasks.

Figure 4 shows the results for case (1), where faults are simulated in both map and reduce tasks. The dashed lines are a reference for the running time with no faults. For the 200 M x 200 matrix, there were around 3000 tasks, and for the 800M x 800 matrix, there were around 1000 tasks. On the left-hand side, we have a P(fault)= $\frac{1}{5}$, and the performance penalty is approximately 50%. This demonstrates the reliability of Hadoop: even with one in five tasks failing, we still finish the job relatively quickly.

Figure 5 shows results for cases (1), (2), and (3), where each experiment was run independently. As expected, map-only and reduce-only fault scenarios incur a noticeably smaller penalty in most cases. However, the noise in the data does not allow us to differentiate the effects of map faults and reduce faults when they are both present in the system.

10 Strong scaling in the cloud

Strong scaling with Hadoop is a challenge as map tasks are provided on an as-available basis, rather than an as-needed basis. In other words, map resources are greedily consumed. In order to control

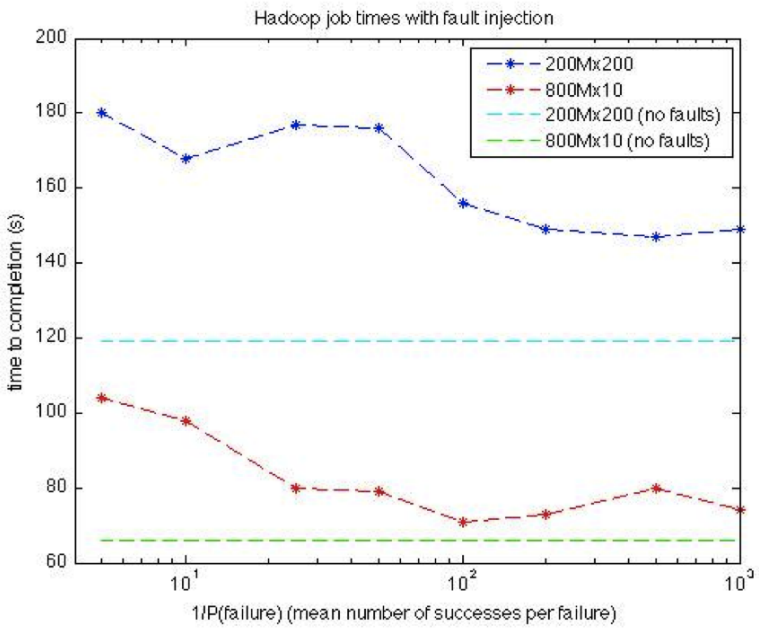

Figure 4: Measuring performance penalties induced by simulated faults in map and reduce tasks

the number of map tasks, we can manipulate the map.minimum (??) parameter, which sets the minimum number of bytes for a given map task to consume as input.

11 Benchmarking Cholesky and Householder

12 Performance

12.1 Peak performance

12.2 Challenges

12.3 Measuring Hadoop overhead

One way of measuring Hadoop overhead is to spawn a large job on a small input. In this experiment, we have a simple two-iteration MapReduce job with an input size of approximately 1 KB:

$$(k,v) \xrightarrow{identity} (k,v) \xrightarrow{identity} (k,v) \xrightarrow{identity} (k,v) \xrightarrow{identity} (k,v).$$

The first iteration uses the maximum available map and reduce tasks. The second iteration uses the maximum available map tasks and one reduce task. This is the same model for both Householder TSQR and Cholesky TSQR, but using only identity map and reduce tasks. The running time for this job over 5 trials is given in table 1.

It takes a little over a minute, just to launch the two-step iterative Hadoop jobs. Based on results results earlier in the section, this ranges from 20-50% of running time.

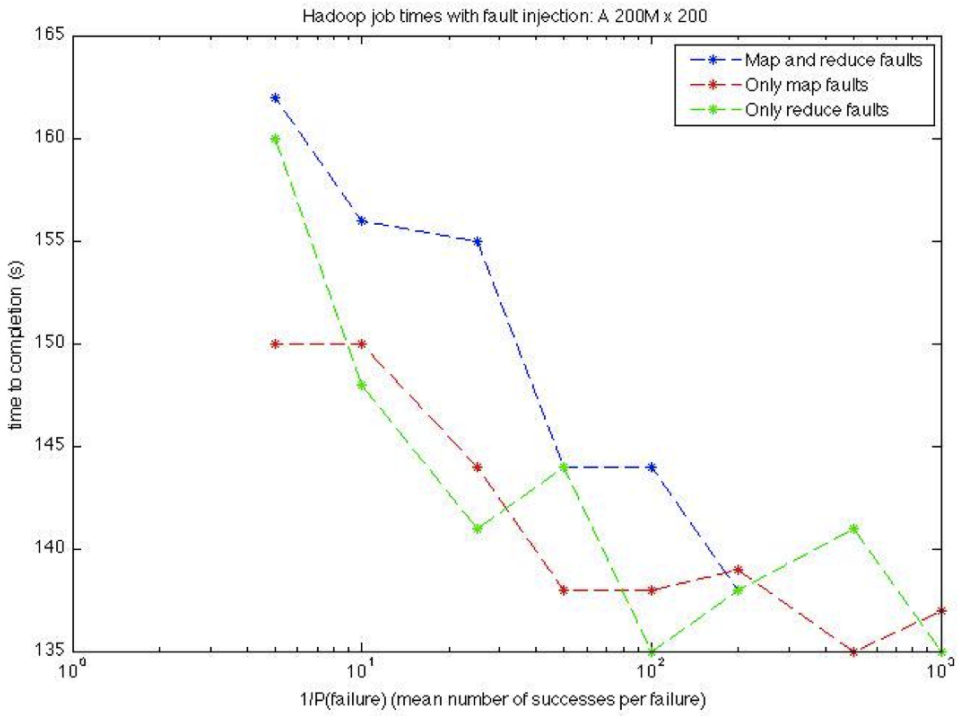

Figure 5: Measuring performance penalties induced by simulated faults in map and reduce tasks

12.4 Benchmarking C++ vs. Python

13 Conclusion

14 Future work

15 Acknowledgements

References

[1] James Demmel, Laura Grigori, Mark F. Hoemmen, and Julien Langou. *Communication-*

Table 1: Running times of identity map and reduce tasks

trial	first iteration time (s)	second iteration time (s)	total time (s)
1	35	32	67
2	42	31	73
3	36	31	67
4	33	31	64
5	34	34	68

optimal parallel and sequential QR and LU factorizations. UCB/EECS-2008-89. August 2008.

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

[3] David Gleich. MapReduce TSQR code. ⟨ https://github.com/dgleich/mrtsqr ⟩

[4] Austin Benson. MapReduce TSQR code. ⟨ https://github.com/arbenson/mrtsqr ⟩

[5] The Apache Software Foundation. *MapReduce Tutorial*. 2008.
⟨ http://hadoop.apache.org/common/docs/current/mapred_tutorial.html ⟩

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

[7] Klaas Bosteels. TypedBytes. ⟨ https://github.com/klbostee/typedbytes ⟩

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

[9] Hadoop version 0.20. ⟨ http://hadoop.apache.org/ ⟩

[10] *Hadoop Wiki: PoweredBy*. ⟨ http://wiki.apache.org/hadoop/PoweredBy ⟩

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

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

[13] Shane Corder. *High Performance Linpack on Xeon 5500 v. Opteron 2400*. 16 June 2009. ⟨ http://www.advancedclustering.com/company-blog/high-performance-linpack-on-xeon-5500-v-opteron-2400.html ⟩. Accessed 11 November 2011.