# A Scalable Randomized Least Squares Solver for Dense Overdetermined Systems*

Chander Iyer
Department of Computer Science
Rensselaer Polytechnic Institute

Haim Avron
Department of Applied Mathematics
Tel Aviv University

Georgios Kollias
IBM Research - T. J. Watson Research Center

Yves Ineichen
IBM Research - Zurich Research Lab

Christopher Carothers
Department of Computer Science
Rensselaer Polytechnic Institute

Petros Drineas
Department of Computer Science
Rensselaer Polytechnic Institute

## ABSTRACT

We present a fast randomized least-squares solver for distributed-memory platforms. Our solver is based on the Blendenpik algorithm, but employs a batchwise randomized unitary transformation scheme. The batchwise transformation enables our algorithm to scale the distributed memory vanilla implementation of Blendenpik by up to $\times 3$ and provides up to $\times 7.5$ speedup over a state-of-the-art scalable least-squares solver based on the classic QR based algorithm. Experimental evaluations on terabyte scale matrices demonstrate excellent speedups on up to 16384 cores on a Blue Gene/Q supercomputer.

## CCS Concepts

•**Computing methodologies** → **Linear algebra algorithms; Distributed algorithms;** •**Mathematics of computing** → *Computations on matrices;*

## Keywords

Randomized numerical linear algebra; high-performance computing; dense least squares regression

## 1. INTRODUCTION

The explosive growth of data in the past few decades in various domains ranging from physics and biological sciences to economics and social sciences has led to a need to perform efficient and scalable analysis on these massive datasets. One of the most widely and routinely used primitives is least-squares regression. Several algorithms have been proposed to solve the large-scale least-squares problems in various distributed and parallel environments [10], giving solutions with accuracy close to machine precision. In particular, recent literature advocates the use of communication-avoiding factorizations [5]. However, while these approaches have been shown to be scalable for a variety of shared memory and distributed memory platforms [6], they are still based on the classic QR based $O(mn^2)$ algorithm, while the fastest sequential codes are based on fast $o(mn^2)$ randomized algorithms [1, 13].

Indeed, recent years has seen extensive research on so-called Randomized Numerical Linear Algebra (RandNLA). One of the core problems extensively researched by this emerging field is least-squares regression. Drineas et al. [7] introduced the first set of randomized algorithms for this problem. These algorithms are based on applying a randomized Hadamard transform to the columns of the input matrix and then using uniform random sampling and sparse random projections to generate problems of lower sizes. This was followed by the work of Rokhlin and Tygert [16], which used a subsampled randomized Fourier transform to form a preconditioner, and then used a standard Krylov subspace based iterative solver. Subsequently, Avron et al. [1] introduced *Blendenpik* which was the first practical implementation of a RandNLA dense least-squares solver that consistently and comprehensively outperformed state-of-the-art implementations of the traditional QR based $O(mn^2)$ algorithm. Since then, there has been extensive research on RandNLA algorithms for regression (least-squares and others). See Yang et al. [17] for a recent survey.

So far, most research on randomized least-squares regression algorithms has focused on the sequential setting. There are two important exceptions. Meng et al. [13] introduced *LSRN*, a distributed memory algorithm for least-square projection based on random normal projections. The algorithm still has $O(mn^2)$ complexity, and the main benefits of randomization comes in the form of reduced constants and improved parallel efficiency. Yang et al. [17] consider RandNLA in a MapReduce like framework called Spark. This framework is less appropriate for supercomputers, as it is less suitable for taking advantage of their hardware architecture.

In this work we explore the behavior of the Blendenpik algorithm in a distributed memory setting. We show that a variant of the algorithm that uses a batchwise unitary transformation leads to an algorithm that is not only faster than state-of-the-art implementation of the $O(mn^2)$ algorithm, but is also able to scale to much larger matrix sizes. In particular, we show that a Blendenpik based algorithm can solve terabyte sized matrices and above. While the

*IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other product and service names might be trademarks of IBM or other companies.



*ScalA15 November 15–20 2015, Austin, TX, USA*



ISBN 978-1-4503-4011-3/15/11…$15.00

DOI: 10.1145/2832080.2832083

gains in terms of running time (vs. the $O(mn^2)$ solver) are somewhat modest, they are tangible, and they do suggest the possibility that variants of Blendenpik that are more finely tuned for the distributed memory setting will deliver the same gains seen in the sequential setting [1].

We experiment on AMOS[1], the high-performance Blue Gene/Q supercomputer system at RPI, which has five racks, 5120 nodes / 81920 cores and 81920 GB of main memory with a peak performance of 1 PetaFLOP ($10^{15}$ floating point operations per second) and a 5-D torus network with 2 GBs/sec of bandwidth per link and 512 GB/sec to 1 TB/sec of bisection network bandwidth per rack depending on the torus network configuration. Due to runtime constraints imposed by the scheduling system for each partition of AMOS, we limit our experiments to testing up to 1024 nodes (16384 cores). The Blue Gene/Q architecture supports a hybrid communication framework that uses the MPI (Message Passing Interface) [8] standard for distributed communication and multithreading using OpenMP [3] which our implementation seeks to exploit.

To summarize, our contributions are:

- Implementation of and experimentation with Blendenpik on distributed-memory platforms.
- A batchwise transformation scheme that scales a distributed vanilla implementation of Blendenpik by up to $\times 3$ in terms of matrix sizes and provides up to $\times 7.5$ speedup over a state-of-the-art scalable least-squares solver.

The full source code of our batchwise Blendenpik implemention is available for download at https://github.com/cjiyer/libskylark/tree/batchwiseblendenpik .

The rest of this paper is structured as follows. Section 2 describes the Blendenpik algorithm and the various stages of the algorithm in detail. Section 3 highlights the distributed Blendenpik implementation in the Blue Gene/Q along with scalability issues in our implementation and describes of an approach to overcome them. Section 4 first describes experiments to tune our Blue Gene/Q environment for our evaluations and later demonstrates the outcome of our evaluations against three different criteria.

### *Notation.*

$A, B, \ldots$ denote matrices and $a, b, \ldots$ denote column vectors. Given a matrix $A \in \mathbb{R}^{m\times n}$, let $\|A\|_2 = \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2}$ be the spectral norm and $\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}^2|}$ be the Frobenius norm. Let $\sigma_1 \geq \sigma_1 \geq \sigma_2 \cdots \geq \sigma_r$ be the nonzero singular values of $A$ where $r = \text{rank}(A)$, the rank of the matrix. Then the **condition number** of $A$ is given by $\kappa(A) = \frac{\sigma_1}{\sigma_r}$. Also, let $U \in \mathbb{R}^{m\times n}$ be a matrix whose columns form an orthonormal basis for the column space of $A$ and let $U_{i,*}$ denote the $i$-th row of the matrix U as a row vector. We define the **coherence** of $A$ as $\mu(A) = \max_{i\in\{1,\ldots,m\}}\|U_{i,*}\|_2^2$. Also, we denote by $\text{nnz}(A)$, the number of nonzero entries of $A$ and $\alpha\,(A) = \frac{m}{n}$ as the aspect ratio of $A$.

## 2. THE *BLENDENPIK* ALGORITHM FOR DENSE OVERDETERMINED SYSTEMS

*Blendenpik* is a least-squares solver for dense overdetermined full column rank systems that computes the approximate solution with a high degree of precision. Given a dense overdetermined matrix $A \in \mathbb{R}^{m\times n}$ and a column vector $b \in \mathbb{R}^m$, the algorithm computes the solution using three stages:

1. A preconditioner is constructed by applying a randomized unitary transform $F$ to the input matrix $A$ and then sampling a small number of rows, $M_s$ from the transformed matrix $FA$.
2. A QR factorization of the sampled matrix $M_s$ is computed giving an orthogonal matrix $Q_s$ and an upper triangular matrix $R_s$. The latter is then used as a preconditioner for the input matrix.
3. The preconditioner is used in conjunction with LSQR (an iterative method for solving least squares problems) to compute an approximate solution $\hat{x}$ to the original problem.

**Algorithm 1** The Blendenpik algorithm [1]

**Input:** $A \in \mathbb{R}^{m\times n}$ matrix, $m \gg n$ and rank $(A) = n$.
$b \in \mathbb{R}^m$ vector.
$F \in \mathbb{R}^{m\times m}$ random unitary transform matrix.
$\gamma(\geq 1)$ - Sampling factor.
**Output:** $\hat{x} =$ Solution of $\min_x \|Ax - b\|_2$.
**while** Output not returned **do**
  $M = FA$
  Let $\mathcal{S} \in \mathbb{R}^{m\times m}$ be a random diagonal matrix:

$$\mathcal{S}_{ii} = \begin{cases} 1 & \text{with probability } \frac{\gamma n}{m} \\ 0 & \text{with probability } 1 - \frac{\gamma n}{m} \end{cases}$$

  $M_s = \mathcal{S}M$ ; $M_s = Q_s R_s$
  $\hat{\kappa} = \kappa_{\text{estimate}}(R_s)$
  **if** $\hat{\kappa}^{-1} > 5\epsilon_{\text{machine}}$ **then**
    $y = \min_z \|AR_s^{-1}z - b\|_2$
    Solve $R_s\hat{x} = y$
    **return** $\hat{x}$
  **else**
    **if** # iterations $> 3$ **then**
      solve using Baseline Least squares and return
    **end if**
  **end if**
**end while**

A sketch of Blendenpik is described in Algorithm 1. We give a brief overview on each of the stages below.

### *Randomized Unitary transformation.*

The randomized unitary transformation matrix $F$ is constructed as the product of a random diagonal matrix $D$ with i.i.d. Rademacher random diagonal entries ($\Pr(D_{ii} = \pm 1) = 1/2$) and a fixed unitary transformation. This results in a randomly shuffled, low coherence matrix $M$ from which a preconditioner is constructed by sampling rows of the matrix. We experiment mainly on the Discrete Cosine Transform (DCT) as the underlying fixed unitary transformation. A detailed implementation of the DCT and subsequent limitations imposed by overdetermined terascale matrices are explained in Section 3.

### *Row Sampling and QR Preconditioning.*

The algorithm then creates a lower dimensional matrix $M_s$ by selecting rows with probability $\gamma(n/m)$ from $M$, where $\gamma$ is a sampling factor that can be tuned. It is observed experimentally that a small number of rows $\gamma n$ is sufficient for generating a well-conditioned preconditioner. This sampled matrix $M_s$ is then used

[1]https://secure.cci.rpi.edu/wiki/index.php/Blue_Gene/Q

to compute the preconditioner $R_s$ for the input matrix by a QR factorization. Since the upper triangular matrix $R_s$ returned by QR may be ill-conditioned, we run the unitary transformation and row sampling steps more than once (usually three times as mentioned in the algorithm suffices). While selecting a smaller sample of rows can improve the QR preconditioning runtime, there is a tradeoff as it slows down the convergence rate of the iterative solver (since the preconditioner is of lower quality).

*Iterative Solution.*

The solution is computed using a Krylov-subspace method called LSQR [14], with $R_s$ serving as a preconditioner. Given an iterate $x_j$ and the corresponding residual error $r_j = b - Ax_j$, the algorithm uses the following criteria in LSQR to test convergence (this is a standard test in LSQR)

$$\frac{\|(AR^{-1})^T r_j\|_2}{\|AR^{-1}\|_F \|r_j\|_2} \leq \rho$$

where $\rho$ is a tolerance value that determines the backward error at which the iterative solver terminates. This guarantees a backward stable solution to $y_j = \min_z \|AR_s^{-1}z - b\|_2$ and returns the estimate $x_j = R_s^{-1}y_j$. The residual error at convergence is used to compute the final backward error estimate, given by $\|A^T r_j\|_2$. The runtime of LSQR is affected by how well conditioned the preconditioned system $AR_s^{-1}$ is, which in turn is determined by the row size of $M_s$. The lower the number of rows sampled, the greater the coherence of $M_s$ leading to a high condition number of the preconditioned system, and hence greater the time it takes to converge.

# 3. DISTRIBUTED-MEMORY BLENDENPIK: OUR ALGORITHM

| | | |
|---|---|---|
| Distribution formats for a 2-D process grid | $\sim$ | $[M_C, M_R]$ & $[M_R, M_C]$<br>$[V_R, \star]$ & $[\star, V_R]$<br>$[V_C, \star]$ & $[\star, V_C]$<br>$[\star, \star]$ |
| Distribution order within each grid dimension | $M_C$ | Matrix column |
| | $M_R$ | Matrix row |
| | $V_C$ | Vector in column major order |
| | $V_R$ | Vector in row major order |
| | $\star$ | Stored on every process |
| Description | $[X, Y]$ | Distribute [columns, rows] with scheme [X, Y] |
| | $[M_C, M_R]$ | Distribute [columns, rows] equally among processes |
| | $V_C/V_R$ | Distribute over processes in column/row major wrapping |

Table 1: Elemental data distribution overview.

The algorithm is implemented on top of the Elemental library [15]. Given a distributed environment over $p$ processes, any dense matrix $A \in \mathbb{R}^{m \times n}$ is partitioned in Elemental into rectangular grids of sizes $r \times c$ in a 2D cyclic distribution such that $p = r \times c$ and $r, c = O(\sqrt{p})$. Elemental allows a matrix to be distributed in more than one way. An overview of various data distributions available in Elemental is given in Table 1 (not exhaustive). We use the standard distribution $[M_C, M_R]$ listed in Table 1 for dense input matrices to exploit operations that are communication intensive. For columnwise/rowwise vector operations that require local computations to be performed, we use a $[\star, V_C/V_R]$ or a $[V_C/V_R, \star]$ distribution that assigns each column/row vector to a single process. In some cases, we require a matrix or a column vector to be present across all processes which is done using the $[\star, \star]$ format. The notations used henceforth are adapted from Elemental for convenience. The attendant paper gives a comprehensive insight on these notations describing different data distributions and the communication costs involved in redistribution.

## 3.1 Randomized Unitary Transformation

The first step in the Blendenpik sketch performs a random unitary transformation $M = FA$ on the input matrix. As mentioned in Section 2, a random unitary transformation is the product of a random diagonal matrix and a fixed unitary transformation. We now describe how the randomized unitary transformation is implemented, with DCT as our fixed unitary transformation below.

Essentially, we use the DCT implementation in FFTW [9], a highly optimized implementation of FFT tuned for underlying architectures that can work on multidimensional data. For our purposes we use the 1-D extensions of DCT that operate on data distributions of Elemental. The $[M_C, M_R]$ Elemental distribution for input matrices is not a suitable format to apply FFTW's DCT, since the data distributed across multiple nodes columnwise as well as rowwise is locally non-contiguous, while the implementation expects contiguously distributed data across either of the dimensions. We resolve this problem by redistributing the data such that all elements of a column/row vector are owned locally by a process, using either the $[V_R/V_C, *]$ or $[*, V_R/V_C]$ distribution mentioned in Table 1. The DCT can be considered as pre multiplying the input matrix by the discrete cosine components, and hence the input matrix must have all elements of a column locally i.e. in the $[*, V_R/V_C]$ distribution. The Elemental pseudocode for the unitary transformation in Blendenpik can be described in the following steps:

$$M = FA \Longleftrightarrow \begin{cases} A[\star, V_R] & = & A[M_C, M_R] \\ M[\star, V_R] & = & F[\star, V_R]A[\star, V_R] \\ M[M_C, M_R] & = & M[\star, V_R] \end{cases}$$

The $A[*, V_R] \longleftarrow A[M_C, M_R]$ redistribution can be thought of as a `MPI_Scatter` and an `MPI_Gather` collective pair operation. The current MPI specifications (MPI 3.0) support sending/receiving upto `INT_MAX` $(2^{31} - 1)$ elements for any collective operation. This places memory constraints for terascale dense overdetermined systems as the row sizes increase and more column elements get bunched together inside a single process. Hammond et. al. [11] present an excellent discussion and demonstrate a library implementation called BigMPI as a wrapper to the current MPI specifications to resolve this problem. However, porting this as a wrapper to the local MPI implementation on AMOS is a cumbersome task and beyond the scope of our work. We instead overcome this problem using a batchwise unitary transformation as explained in the following section.

## 3.2 Batchwise Unitary Transformation

The grid distribution by Elemental being cylic in nature, imposes an additional overload during redistribution in that the memory for each process is now shared by other columns of the matrix too. Thus, not only are we constrained by the row size i.e. requirement for the entire matrix column to be present locally to apply the transform, but also being inhibited by the column size i.e. memory shared between additional columns. This tradeoff requires a flexibility in choosing to do a bulk redistribution and delayed tranformation than a piecemeal redistribution and immediate transformation. Since we are concerned with terascale overdetermined systems, a piecemeal redistribution, wherein we select only a few columns to transform at a time that enables faster transform is generally preferred. Another limiting factor that inhibits Blendenpik performance for terascale matrices is the communication cost in redistributing matrices after batchwise transformation. However,

one can observe that we sample the batchwise transformed matrix before redistribution to generate the preconditioner.

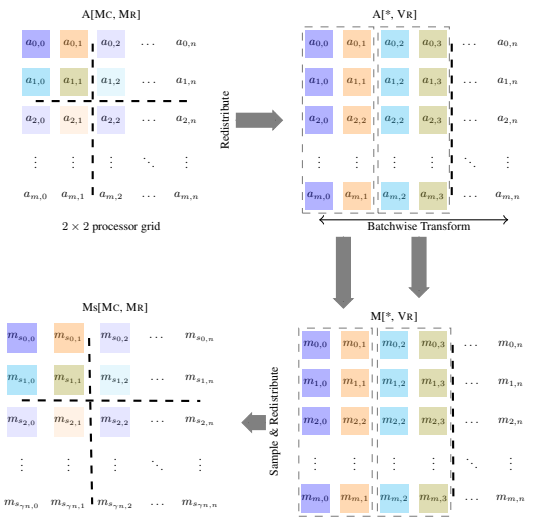


Figure 1: Batchwise unitary transformation in Blendenpik.

Figure 1 illustrates the batchwise transformation done on a piecemeal basis, i.e. only selecting a limited number of columns whose flexibility can be suitably chosen. Let a matrix $A \in \mathbb{R}^{m \times n}$ be distributed in an $[M_C, M_R]$ format and let there be four MPI processes (say) in our distributed environment (denoted by the color scheme) given by $p_0, \ldots, p_3$. We divide the matrix $A$ columnwise into $b$ submatrices given by $A^{(1)}, A^{(2)}, \ldots, A^{(b)}$, redistribute each submatrix $A^{(i)}[\star, V_R] \leftarrow A^{(i)}[M_C, M_R] : i \in \{1, \ldots, b\}$, perform random unitary transformation on each submatrix $M^{(i)}[\star, V_R] \leftarrow F[\star, V_R]A^{(i)}[\star, V_R] : i \in \{1, \ldots, b\}$, sample from each of the transformed submatrix $M_s^{(i)}[\star, V_R] \leftarrow \mathcal{S}[\star, V_R]M^{(i)}[\star, V_R] : i \in \{1, \ldots, b\}$, redistribute back each of the sampled $b$ submatrices $M_s^{(i)}[M_C, M_R] \leftarrow M_s^{(i)}[\star, V_R] : i \in \{1, \ldots, b\}$ and finally merge in the $M_s[M_C, M_R]$ matrix. The number of columns in each submatrix and thus effectively, the number of submatrices $b$ can be tuned as per the dimensions of the matrix and the number of IBM® BG/Q nodes used in our evaluations.

A pseudocode description is given in Algorithm 2.

Algorithm 2: Elemental pseudocode for batchwise transformation.

**Input:**
$A[M_C, M_R] \in \mathbb{R}^{m \times n}$.
$F[\star, V_R] \in \mathbb{R}^{m \times m}$.
$\mathcal{S}[\star, V_R] \in \mathbb{R}^{\gamma n \times m}$.
**Output:**
$M_s[M_C, M_R]$ - the transformed matrix.

$\left(A^{(1)}[M_C, M_R] \,|\, A^{(2)}[M_C, M_R] \,|\, \cdots \,|\, A^{(b)}[M_C, M_R]\right) \longleftarrow A[M_C, M_R]$
**for** $i = 1, 2, \ldots, b$ **do**
  $A^{(i)}[\star, V_R] = A^{(i)}[M_C, M_R]$
  $M^{(i)}[\star, V_R] = F[\star, V_R] \quad A^{(i)}[\star, V_R]$
  $M_s^{(i)}[\star, V_R] = \mathcal{S}[\star, V_R] \quad M^{(i)}[\star, V_R]$
  $M_s^{(i)}[M_C, M_R] = M_s^{(i)}[\star, V_R]$
**end for**
$M_s[M_C, M_R] \longleftarrow$
$\left(M_s^{(1)}[M_C, M_R] \,|\, M_s^{(2)}[M_C, M_R] \,|\, \cdots \,|\, M_s^{(b)}[M_C, M_R]\right)$
**return** $M_s[M_C, M_R]$

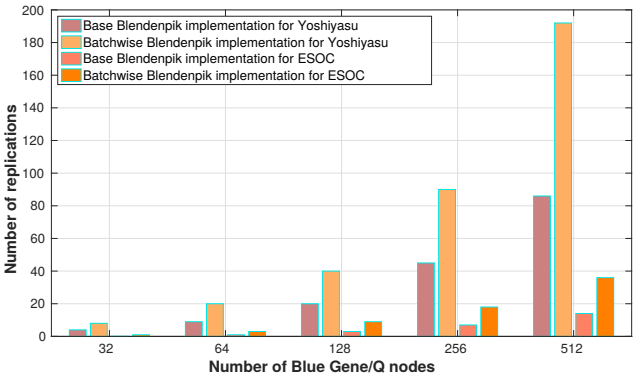


Figure 2: A comparison of the maximum number of replications achieved using the batchwise transformation implementation in Blendenpik and the base Blendenpik implementation for Yoshiyasu and ESOC dense matrices.

Figure 2 compares the scalability of batchwise unitary transformation in Blendenpik with the base distributed implementation of Blendenpik in terms of data sizes for dense matrices for increasing Blue Gene/Q nodes. For this purpose, we generate terascale dense matrices of the base Yoshiyasu Mesh and ESOC Springer sparse matrices by densifying and replicating them as explained in Section 4. The batchwise unitary transformation scales approximately by a factor of 2 for the Yoshiyasu matrix and approximately 3 times for the ESOC matrix over the base Blendenpik implementation that performs unitary transformation in a single stage. This scaling effect becomes more pronounced when the columns of the matrix increase as observed for the ESOC matrix for increasing BG/Q nodes, which supports our choice of using batchwise transformation to scale up Blendenpik.

## 3.3 Other Stages

The sampled matrix $M_s$ generates the preconditioner $R_s^{-1}$ obtained using Elemental's QR solver. Finally, the iterative solution is obtained using an LSQR implementation. The parameters for LSQR, the tolerance value $\rho$ and the iteration limit $N_{\text{iter}}$ can be suitably tuned depending upon the magnitude of accuracy needed and the speedup desired from our Blendenpik solver. Usually a tolerance value $\rho$ closer to $\epsilon_{\text{machine}}$ is chosen for a better backward stable solution.

# 4. EVALUATION

To generate terascale dense matrices, we randomly perturb sparse matrices with values generated from a standard normal distribution i.e. $\mathcal{N}(0, 1)$. We further replicate matrices for a chosen number of times depending upon the size of the matrix to generate. We rely on the UFL Sparse matrix collection [4] for obtaining matrices of different conditioning and coherence values.

We choose the Yoshiyasu Mesh[2], ESOC Springer[3] and the Rucci[4] matrix from the UFL Sparse matrix collection for creating our datasets. Table 2 shows the datasets used for our evaluation after densification and replication steps on the sparse matrices. Each dataset is suffixed with the number of replications done on the base dense

[2]http://www.cise.ufl.edu/research/sparse/matrices/Yoshiyasu/mesh_deform.html
[3]http://www.cise.ufl.edu/research/sparse/matrices/Springer/ESOC.html
[4]http://www.cise.ufl.edu/research/sparse/matrices/Rucci/Rucci1.html

| Data Set | Number of rows (Million) | Number of columns | Number of entries (Billion) | Total size(TB) |
|---|---|---|---|---|
| Yoshiyasu-1 | 0.234 | 9393 | 2.198 | 0.016 |
| Yoshiyasu-24 | 5.616 | | 52.756 | 0.384 |
| Yoshiyasu-48 | 11.233 | | 105.512 | 0.768 |
| Yoshiyasu-72 | 16.849 | | 158.268 | 1.152 |
| Yoshiyasu-96 | 22.466 | | 211.025 | 1.535 |
| Yoshiyasu-120 | 28.082 | | 263.781 | 1.919 |
| Yoshiyasu-144 | 33.699 | | 316.537 | 2.303 |
| Yoshiyasu-168 | 39.315 | | 369.294 | 2.687 |
| Yoshiyasu-192 | 44.932 | | 422.050 | 3.070 |
| ESOC-1 | 0.327 | 37830 | 12.373 | 0.090 |
| ESOC-8 | 2.616 | | 98.982 | 0.720 |
| ESOC-16 | 5.233 | | 197.964 | 1.440 |
| ESOC-24 | 7.849 | | 296.946 | 2.160 |
| ESOC-32 | 10.466 | | 395.928 | 2.880 |
| ESOC-40 | 13.082 | | 494.910 | 3.600 |
| ESOC-48 | 15.698 | | 593.892 | 4.321 |
| ESOC-56 | 18.315 | | 692.874 | 5.041 |
| ESOC-64 | 20.931 | | 791.856 | 5.761 |
| Rucci-1 | 1.978 | 109900 | 217.369 | 1.581 |
| Rucci-2 | 3.956 | | 434.739 | 3.163 |
| Rucci-3 | 5.933 | | 652.108 | 4.744 |

Table 2: Datasets used in Blendenpik evaluation.

matrix e.g. Yoshiyasu-24 indicates that the Yoshiyasu Mesh dense matrix has been replicated 24 times. We choose a sampling factor of $\gamma = 2$ for the dense matrices based on scalability tests with the DCT transform to generate a relatively well-conditioned preconditioner.

*Evaluation Metrics.*

Let $A \in \mathbb{R}^{m \times n}$ be the input matrix, $b \in \mathbb{R}^m$ be the right hand side vector and let:

$\hat{x} \longleftarrow \min_x \|Ax - b\|_2$, the batchwise Blendenpik min-norm solution.

$x^* \longleftarrow$ the exact solution.

$\hat{r} \longleftarrow$ Defined as $b - A\hat{x}$.

$\hat{t}_{\text{run}} \longleftarrow$ running time of Blendenpik.

$t^*_{\text{run}} \longleftarrow$ running time of baseline (Elemental).

We evaluate the Blendenpik algorithm for large scale dense matrices using the following metrics.

**Speedup :** Speedup is given by $\frac{t^*_{\text{run}}}{\hat{t}_{\text{run}}}$.

**Accuracy :** Accuracy is defined in terms of the relative error for the min-norm solution $\hat{x}$ given by $\frac{\|A\hat{x} - Ax^*\|_2}{\|Ax^*\|_2}$ and the backward error given by $\|A^T\hat{r}\|_2$.

## 4.1 AMOS Environment Setup

Our goal in this work is to provide a thorough evaluation of Blendenpik as a terascale dense least squares solver from three primary viewpoints: scalability, performance and numerical stability. We tune AMOS to evaluate the Blendenpik implementation against optimum baseline performance and scalability. We select that combination of number of OpenMP threads per node and number of MPI processes per node that gives maximum performance for the baseline least squares solver (Elemental) on AMOS. We measure the average time taken by the baseline solver over 5 runs for all possible MPI/OpenMP combinations for two dense matrices: a random matrix generated uniformly at random between $[-1, 1]$ of dimensions $\mathbb{R}^{1320000 \times 38000}$ and the ESOC-4 replicated matrix (see above for the replication process) of dimensions $\mathbb{R}^{1308248 \times 37830}$. The AMOS system can execute upto 64 MPI processes with 1 OpenMP thread or 1 MPI process with 64 OpenMP threads or combinations of the same in a single node. We use a bgclang/LLVM build of the baseline Elemental solver for our evaluations.

As seen in Table 3, the performance of the baseline Elemental solver improves as the number of OpenMP threads per node increases for a single MPI process per node. Similarly, the performance also increases with increasing MPI processes per node for a single OpenMP thread. However, the performance drops when many MPI processes compete for CPU resources with several OpenMP threads in the system. The maximum performance is realized for 32 OpenMP threads for a single MPI process as seen in Table 3, while it immediately degrades when the number of threads reach 64. One possible reason could be because of possible thread synchronization during Floating point operations in the BG/Q CPU cores. The choice of 1 MPI process and 32 OpenMP threads per Blue Gene/Q node is the standard configuration we select for our evaluations and henceforth we use the number of nodes interchangeably for the number of cores.

## 4.2 Runtime environment evaluation

One of the key factors that influence scalability considerations is the performance of the runtime environment in the AMOS Blue Gene/Q system. The AMOS system supports both standard GNU (4.7.2) as well as LLVM/bgclang[5] environments [12]. Figure 3 demonstrates the speedup for the baseline (Elemental) least squares solver built with bgclang over the baseline solver built with GNU for various replications of the Yoshiyasu and the ESOC matrix for increasing row sizes on 512 BG/Q nodes. The baseline (Elemental) solver achieves a significant speedup in the LLVM/bgclang environment due to the highly optimized linear algebraic routines implemented in the BG/Q Math libraries, ESSL (Engineering Scientific Subroutine Library) and MASS (Mathematical Acceleration Subsystem).

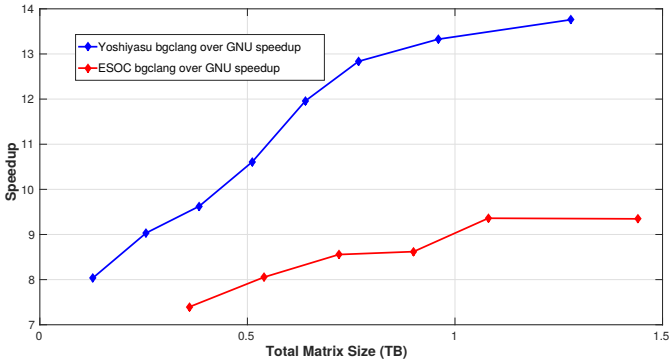


Figure 3: Speedup defined as $t^*_{\text{gnu}}/t^*_{\text{bgclang}}$ of the baseline solver as a function of matrix size in TB for 512 nodes.

To solve the least squares regression problem, the baseline Elemental solver computes the QR for the preconditioner. The QR can be considered as a sequence of matrix multiplications of orthogonal matrices with the input matrix, and ESSL outperforms the BLAS routines built with the standard GNU compiler for the Blue Gene/Q in this respect. As we see, the speedup for the Yoshiyasu is much better than the one achieved for the ESOC Springer matrix as the matrix is highly overdetermined and with increasing columns, QR

[5]http://trac.alcf.anl.gov/projects/llvm-bgq

| OpenMP threads per node | $A \in \mathbb{R}^{1.32M \times 38K}$ random matrix; $\mu(A) = 0.0054 \pm 1.5 * 10^{-4}$; $\kappa(A) = 1.4084 \pm 1.1 * 10^{-4}$ | | | | | | | $A \in \mathbb{R}^{1.3082M \times 38K}$ ESOC-4 matrix; $\mu(A) = 0.998 \pm 0$; $\kappa(A) = 1.4594 * 10^{6} \pm 11.0268$ | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MPI processes per node | | | | | | | | | | | | | |
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 1 | 8734.48 | 4613.62 | 2996.67 | 2187.48 | 1477.89 | 1878.74 | 1755.57 | 9304.71 | 5205.42 | 2737.43 | 2067.88 | 1378.84 | 1809.77 | – |
| 2 | 4499.6 | 2637.91 | 1762.49 | 1580.48 | 1240.34 | 1826.52 | – | 4831.84 | 2922.68 | 1640.01 | 1542.24 | 1133.12 | 1741.58 | – |
| 4 | 2349.32 | 1565.67 | 1156.55 | 1334.04 | 1168.86 | – | – | 2567.38 | 1741.82 | 1091.47 | 1236.9 | 1023.93 | – | – |
| 8 | 1276.73 | 1097.84 | 922.341 | 2261.12 | – | – | – | 1463.21 | 1207.34 | 898.165 | 1219.18 | – | – | – |
| 16 | 800.443 | 1057.67 | 1193 | – | – | – | – | 961.26 | 1026.97 | 891.234 | – | – | – | – |
| 32 | **628.171** | 1081.64 | – | – | – | – | – | **742.075** | 919.704 | – | – | – | – | – |
| 64 | 686.517 | – | – | – | – | – | – | 744.563 | – | – | – | – | – | – |

Table 3: Runtime analysis to select an optimal hybrid MPI/OpenMP configuration for our scaling experiments. Each run was done on 128 BG/Q nodes for 5 iterations. All runtime values mentioned are in seconds(lower runtime values are preferred indicating faster execution).

worsens quadratically (since the QR for a matrix $A \in \mathbb{R}^{m \times n}$ require $O(mn^2)$ floating point operations.

## 4.3 Scalability evaluation

One of the primary goals is to demonstrate Blendenpik as a scalable solver for terascale overdetermined systems. Figures 4 and 5 show the scalability of our solver on 512 nodes and 1024 nodes respectively on AMOS. The base Elemental least squares solver scales quite well for highly overdetermined dense matrices like the Yoshiyasu matrix. Hence the speedup observed for the Blendenpik solver compared to the baseline solver is not as significant as seen in Figure 4. However for matrices that are less overdetermined, the runtime and thus the speedup improves considerably. This effect is observed for the ESOC Springer matrix as shown in Figure 4.

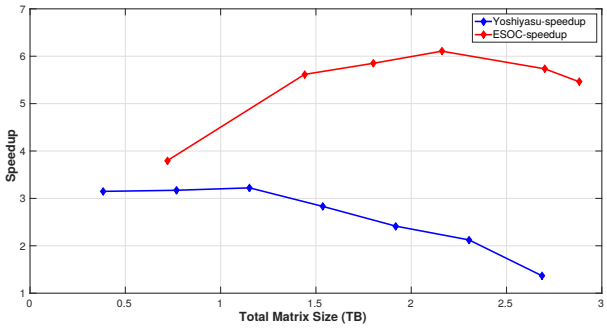


Figure 4: Speedup analysis for Yoshiyasu Mesh and ESOC Springer dense matrices as a function of increasing matrix sizes for 512 BG/Q nodes.

One observation that is particularly significant is the effect of batchwise transformation on the speedup as the matrix size increases. The number of columns transformed in a single batch depends upon the number of rows of the matrix and the minimum space available across all processes to allocate the columns. Thus, as the number of rows increases, fewer and fewer columns fit in a batch making the batchwise transformation step slower. This is the reason why the speedup peaks at a point where the entire transformed matrix is able to fit into memory and beyond this stage, the batchwise processing kicks in. This effect is more pronounced as seen for increasing replications of the ESOC Springer matrix and Rucci matrix in Figure 5 for 1024 BG/Q nodes. In general, the Blendenpik solver scales excellently as compared to the baseline Elemental solver. While the baseline solver fails to execute for Yoshiyasu-192 (Table 2) & ESOC-36 (ESOC matrix with 36 replications) do not run in 512 nodes and ESOC-68 (ESOC matrix with 68 replications) in 1024 nodes, the Blendenpik solver is able

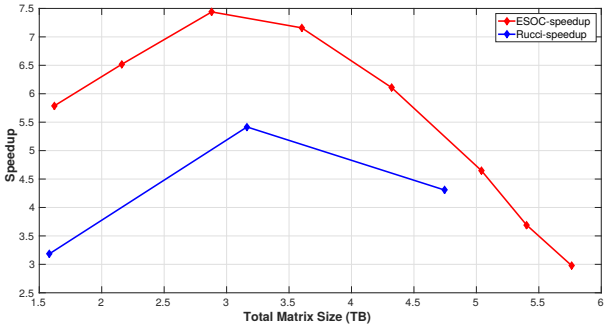


Figure 5: Speedup analysis for ESOC Springer and Rucci dense matrices as a function of increasing matrix sizes for 1024 BG/Q nodes.

to scale to matrix sizes of these replications.

## 4.4 Performance Evaluation

We also evaluate our Blendenpik solver for assessing performance in terms of both strong scaling and weak scaling for increasing Blue Gene/Q nodes. Figure 6 shows the strong scaling performance of the solvers on the base Yoshiyasu Mesh and ESOC Springer dense matrices respectively. As we see, the speedup of the Blendenpik solver rises almost linearly with increasing BG/Q nodes which is more pronounced for the Yoshiyasu Mesh matrix than the ESOC Springer matrix. However, this advantage is offset as the baseline Elemental solver performs comparably to the batchwise Blendenpik solver for 512 BG/Q nodes. This slowdown in the batchwise Blendenpik solver is mainly on account of the QR preconditioning phase that does not scale as well as the unitary transform and the LSQR stages with increasing number of BG/Q nodes. Similarly, the strong scaling runtime for the Rucci matrix is given in Table 4. The Rucci matrix shows better speedup than the ESOC Springer and Yoshiyasu Mesh matrices for increasing number of BG/Q nodes.

| **Number of BG/Q nodes** | **Batchwise Blendenpik (seconds)** | **Elemental-Baseline (seconds)** | **Speedup** |
|---|---|---|---|
| 512 | 645.511 | 1916.85 | 2.9695 |
| 1024 | 413.897 | 1318.65 | 3.1859 |

Table 4: Strong scaling runtime analysis for the Rucci matrix ($1977885 \times 109900$) for increasing Blue Gene/Q nodes.

Figure 7 shows the improved runtime achieved by the baseline Elemental solver and the batchwise Blendenpik solver for the max-

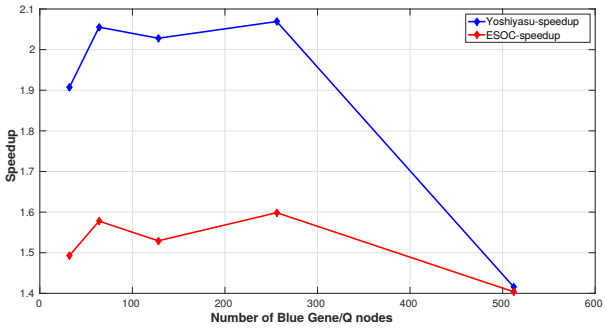


Figure 6: Strong scaling speedup analysis for the Yoshiyasu Mesh matrix (234023 $\times$ 9393) and ESOC Springer matrix (327062 $\times$ 37830) for increasing Blue Gene/Q nodes.

imum number of BG/Q nodes for each of the matrices (512 nodes for the Yoshiyasu Mesh & ESOC Springer matrices and 1024 nodes for the Rucci matrix) which is still slower compared to the Blendenpik solver.

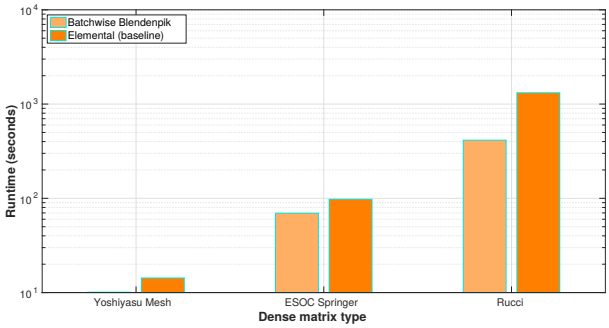


Figure 7: Runtime plot for the Yoshiyasu Mesh matrix (234023 $\times$ 9393), ESOC Springer matrix (327062 $\times$ 37830) and Rucci matrix (1977885 $\times$ 109900) for max. number of BG/Q nodes.

Finally, the weak scaling performance of the Yoshiyasu Mesh and ESOC Springer matrices are shown in Figures 8 and 9 respectively. As seen, the Yoshiyasu Mesh runtime for the batchwise Blendenpik solver is approximately constant for increasing matrix sizes and for increasing BG/Q nodes while there's a significant bump in the running time for the baseline solver. Interestingly, the runtime of the ESOC matrix keeps reducing in spite of the number of rows increasing with increasing number of BG/Q nodes. As mentioned in the strong scaling analysis, the primary bottleneck for the reduction in performance for increasing BG/Q nodes was the performance of the QR preconditioning stage. However, the size of the sampled matrix which is the input for the QR preconditioner still remains the same even when the problem size increases. When more computing resources are assigned, the QR preconditioner performance improves. This gives a boost to the QR speedup leading to improved overall runtime. The weak scaling runtime for the Rucci matrix is given in Table 5 whose behavior is similar to the ESOC matrix.

## 4.5 Numerical Stability Evaluation

We evaluate the numerical stability of the Blendenpik solver as matrix size increases. The numerical stability is captured by the relative error and the backward error defined earlier in our accuracy metric. We show the numerical stability with respect to the relative error for increasing matrix sizes for 512 Blue Gene/Q nodes for the

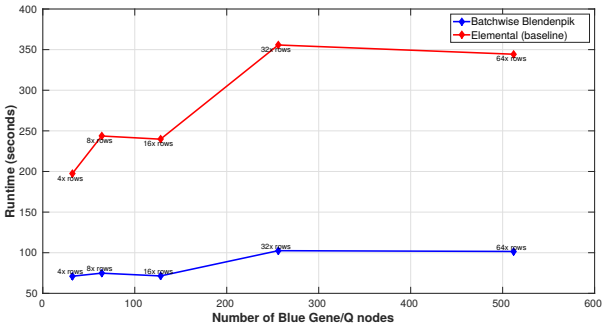


Figure 8: Weak scaling runtime analysis for the Yoshiyasu Mesh matrix (234023 $\times$ 9393) for increasing matrix sizes and increasing Blue Gene/Q nodes.

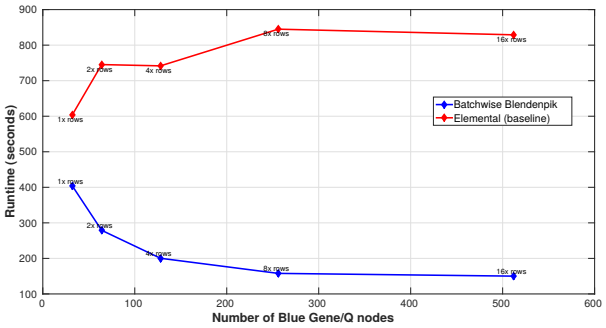


Figure 9: Weak scaling speedup analysis for the ESOC Springer matrix (327062$\times$37830) for increasing matrix sizes and increasing Blue Gene/Q nodes.

| Number of BG/Q nodes | Batchwise Blendenpik (seconds) | Elemental-Baseline (seconds) | Speedup |
|---|---|---|---|
| 512 | 645.511 | 1916.85 | 2.9695 |
| 1024 | 473.565 | 2564.85 | 5.416 |

Table 5: Weak scaling runtime analysis for the Rucci matrix (1977885 $\times$ 109900) for increasing row sizes and increasing Blue Gene/Q nodes.

Yoshiyasu Mesh and the ESOC matrix in Figure 10. The accuracy in terms of relative error for both the ESOC and Yoshiyasu matrices are much better than $O(\sqrt{\epsilon_{\text{machine}}})$ which is well within the bounds on the relative error given by Drineas et al. [7].

The numerical stability defined by backward error is as shown in Figure 11 for both the baseline Elemental solver and the Blendenpik solver. As observed, the ESOC Springer matrix has a worse backward error than the Yoshiyasu Mesh matrix ($\approx$ 5 orders of magnitude worse) for increasing matrix sizes due to it's high condition number due to which the LSQR solver quickly stagnates. However, the backward error of the Blendenpik solver is comparable to the backward error of the baseline solver. This error can be improved by either using more than one preprocessing and mixing stage or by selecting a larger sample size for the preconditioning stage that leads to a worse overall running time and reduced speedup for increasingly large matrices. Thus there is a tradeoff between the numerical stability and the speedup desired to compute least squares for dense terascale matrices.

*Summary.*

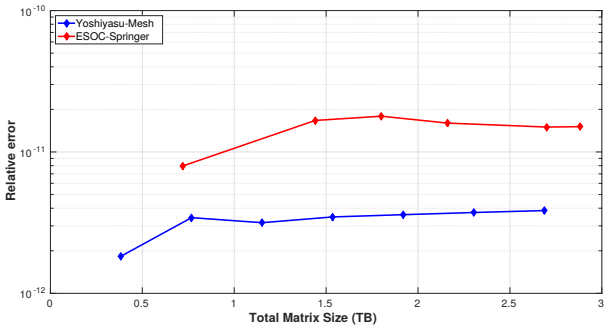


Figure 10: Accuracy analysis in terms of relative error as a function of increasing matrix size for Yoshiyasu Mesh and ESOC Springer matrices for 512 BG/Q nodes.

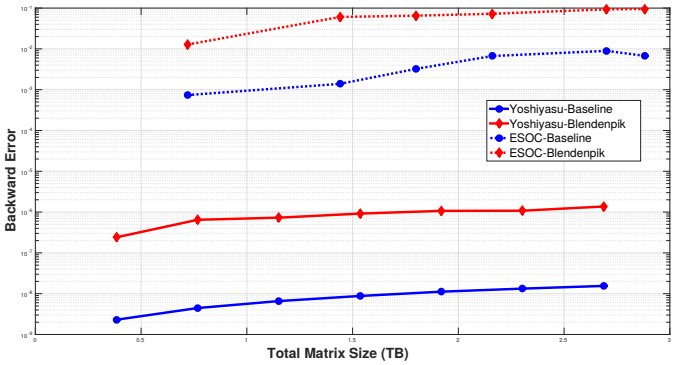


Figure 11: Accuracy analysis in terms of backward error as a function of increasing matrix size for Yoshiyasu Mesh and ESOC Springer matrices for 512 BG/Q nodes.

- The scalability of the batchwise Blendenpik implementation is determined by the number of columns in each batch of the DCT transform which in turn is determined by the number of rows of the matrix. As the number of rows increases, the runtime of the batchwise unitary transformation stage worsens leading to a reduced speedup.
- The batchwise Blendenpik solver demonstrates appreciable strong scaling and weak scaling comparable to the baseline Elemental solver for all matrices.
- The Blendenpik solver demonstrates excellent numerical stability in terms of the forward error for increasing matrix sizes. The backward error however is worse, though this is comparable to the backward error achieved by the baseline Elemental solver.

# 5. CONCLUSIONS AND FUTURE WORK

In this paper we take the first steps towards designing an highly scalable distributed memory least squares solver based on the Blendenpik algorithm. Our solver, which is based on a somewhat straightforward implemention of the Blendenpik algorithm in a distributed setting coupled with a batchwise unitary transformation, is already able to beat state-of-the-art algorithm based on the classical algorithm both in runtime and in the sheer size of matrices that can be solved.

In the future, we plan to design a more finely tuned Blendenpik-based algorithm for distributed memory platforms. This mostly involves reducing the communication overhead involved in the randomized unitary transformation. One exciting idea is to do the randomized unitary transformation only after an initial reduction of row space using input-sparsity sketching, as suggested by Clarkson and Woodruff [2]. This row-reduced space also enables us to choose a larger sample size for the preconditioning stage that can lead to a significant improvement in the numerical stability.

# Acknowledgments

We would like to thank Jack Poulson for his helpful discussions on Elemental and Hal Finkel for his help with the llvm/bgclang build. Haim Avron's work was conducted while at IBM T.J. Watson Research Center. The work is partially supported by NSF IIS-1302231, and by the XDATA program of the Defense Advanced Research Projects Agency (DARPA), administered through Air Force Research Laboratory contract FA8750-12-C-0323.

# 6. REFERENCES

[1] H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging lapack's least-squares solver. *SIAM J. Scientific Computing*, 32(3):1217–1236, 2010.

[2] K. L. Clarkson and D. P. Woodruff. Low rank approximation and regression in input sparsity time. *CoRR*, abs/1207.6365, 2012.

[3] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[4] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[5] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *ArXiv e-prints*, Aug. 2008.

[6] J. Demmel and K. Yelick. Communication avoiding (CA) and other innovative algorithms. *The Berkeley Par Lab: Progress in the Parallel Computing Landscape*, pages 243–250.

[7] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós. Faster least squares approximation. *Numer. Math.*, 117(2):219–249, Feb. 2011.

[8] M. P. Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[9] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, Washington, 1998.

[10] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32(1):54–135, 1990.

[11] J. R. Hammond, A. Schäfer, and R. Latham. To `INT_MAX`... and beyond!: Exploring large-count support in MPI. In *Proceedings of the 2014 Workshop on Exascale MPI*, ExaMPI '14, pages 1–8, Piscataway, NJ, USA, 2014. IEEE Press.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] X. Meng, M. A. Saunders, and M. W. Mahoney. LSRN: A parallel iterative solver for strongly over- or under-determined systems. *CoRR*, abs/1109.5981, 2011.

[14] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, 8(1):43–71, Mar. 1982.

[15] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, Feb. 2013.

[16] V. Rokhlin and M. Tygert. A fast randomized algorithm for overdetermined linear least-squares regression. *Proc. Natl. Acad. Sci. USA*, 105(36):13212–13217, 2008.

[17] J. Yang, X. Meng, and M. W. Mahoney. Implementing randomized matrix algorithms in parallel and distributed environments. *CoRR*, abs/1502.03032, 2015.