LSRN: A Parallel Iterative Solver for Strongly Over- or Under-Determined Systems

Xiangrui Meng

Joint with Michael A. Saunders and Michael W. Mahoney

Stanford University

June 19, 2012

Strongly over- or under-determined least squares

We are interested in computing the unique min-length solution, denoted by x^*, to

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \|Ax - b\|_2,$$

where $A \in \mathbb{R}^{m \times n}$ with $m \gg n$ or $m \ll n$ and $b \in \mathbb{R}^m$. A could be rank deficient. When the system is under-determined, we may want to solve it with Tikhonov regularization:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2}\|Ax - b\|_2^2 + \frac{\lambda}{2}\|x\|_2^2,$$

where $\lambda > 0$ is a regularization parameter.

Strongly rectangular data

	m	n
SNP	number of SNPs (10^6)	number of subjects (10^3)
TinyImages	number of pixels in each image (10^3)	number of images (10^8)
PDE	number of degrees of freedom	number of time steps
sensor network	size of sensing data	number of sensors
NLP	number of words and *n*-grams	number of principle components

Traditional algorithms: singular value decomposition

It is well known that $x^* = V\Sigma^{-1}U^T b$, where $U\Sigma V^T$ is A's economy-sized SVD. The time complexity is $\mathcal{O}(mn^2 + n^3)$.

- Pros:
  - High precision and robust to rank deficiency.
  - Implemented in LAPACK.
- Cons:
  - Hard to take advantage of sparsity.
  - Hard to implement in a parallel environment.
  - Incapable of implicit inputs.

Traditional algorithms: iterative methods

Iterative methods are widely used to solve large-scale sparse linear systems. Practical methods include, for example, CGLS and LSQR.

- Pros:
  - Low cost per iteration.
  - Taking A as a linear operator.
  - Easy to implement in a parallel environment.
  - Capable of computing approximate solutions.
- Cons:
  - Hard to predict the number of iterations needed.

Traditional algorithms: iterative methods

The convergence rate of an iterative method is affected by the *condition number* of A, denoted by $\kappa(A) = \sigma_{\max}(A)/\sigma^{+}_{\min}(A)$. For a family of iterative methods, we have

$$\frac{\|x^{(k)} - x^*\|_{A^TA}}{\|x^{(0)} - x^*\|_{A^TA}} \leq 2\left(\frac{\kappa(A)-1}{\kappa(A)+1}\right)^k.$$

However, estimating $\kappa(A)$ is generally as hard as solving the least squares problem itself. For ill-conditioned problems (e.g., $\kappa(A) \approx 10^6$), the convergence speed would be very low.

Before LSRN: random sampling (Drineas, Mahoney, and Muthukrishnan 2006)

- Random sample some rows of A based on its leverage scores:

$$u_i = \|U_{i*}\|_2^2, \quad i = 1, \ldots, m,$$

  where U is an orthonormal basis matrix of range(A).
- If the sample size $s > \mathcal{O}(n^2 \log(1/\delta)/\epsilon^4)$, with probability at least $1 - \delta$, the subsampled solution $\hat{x}$ gives an $(1 + \epsilon)$-approximation:

$$\|A\hat{x} - b\|_2 \leq (1 + \epsilon)\|Ax^* - b\|_2.$$

- How to compute or estimate the leverage scores?

Before LSRN: row mixing (Drineas, Mahoney, Muthukrishnan, and Sarlós 2007)

- Mix rows of A via randomized Hadamard transform to make leverage scores distributed uniformly:

$$\tilde{A} = HDA,$$

  where D is a diagonal matrix with diagonal elements chosen randomly from $+1$ and -1, and H is the Hadamard matrix.
- Sample $s = \mathcal{O}(d \log(nd)/\epsilon)$ rows of $\tilde{A}$ uniformly and solve the subsampled problem:

$$\hat{x} = (SHDA)^{\dagger}(SHD)b,$$

  where S is the sample matrix.
- With probability, say, at least 0.8, $\hat{x}$ gives an $(1+\epsilon)$-approximation.
- Running time: $\mathcal{O}(mn \log m + n^3 \log(mn)/\epsilon)$.

Before LSRN: iteratively solving (Rokhlin and Tygert 2008)

- Combine random Givens rotations and the randomized Fourier transform for row mixing.
- In stead of solving the subsampled problem, use the QR factorization of the subsampled matrix to create preconditioners.
- Solve the preconditioned system $\min_x \|AR^{-1}y - b\|_2$ iteratively.
- In theory, choosing $s \geq 4n^2$ guarantees that the condition number is at most 3 with high probability. In practice, choosing $s = 4n$ produced a condition number less than 3 in all their test problems.
- Running time: $\mathcal{O}(mn \log n + mn \log(1/\epsilon) + n^3)$.

Before LSRN: Blendenpik (Avron , Maymounkov , and Toledo 2010)

- High-performance black-box solver.
- Extensive experiments on selecting randomized projections.
- Outperforms LAPACK on strongly over-determined problems.

Before LSRN: what are missing?

- Rank deficiency.
- Sparse A or implicit A.
- Under-determined problems (with Tikhonov regularization).
- Parallel implementation.

Algorithm LSRN (for strongly over-determined systems)

1: Choose an oversampling factor $\gamma > 1$, e.g., $\gamma = 2$. Set $s = \lceil \gamma n \rceil$.
2: Generate $G = \text{randn}(s, m)$, a Gaussian matrix.
3: Compute $\tilde{A} = GA$.
4: Compute $\tilde{A}$'s economy-sized SVD: $\tilde{U}\tilde{\Sigma}\tilde{V}^T$.
5: Let $N = \tilde{V}\tilde{\Sigma}^{-1}$.
6: Iteratively compute the min-length solution $\hat{y}$ to

$$\text{minimize}_{y \in \mathbb{R}^r} \quad \|ANy - b\|_2.$$

7: Return $\hat{x} = N\hat{y}$.

Algorithm LSRN (for strongly under-determined systems)

1: Choose an oversampling factor $\gamma > 1$, e.g., $\gamma = 2$. Set $s = \lceil \gamma m \rceil$.
2: Generate $G = \text{randn}(n, s)$, a Gaussian matrix.
3: Compute $\tilde{A} = AG$.
4: Compute $\tilde{A}$'s economy-sized SVD: $\tilde{U}\tilde{\Sigma}\tilde{V}^T$.
5: Let $M = \tilde{U}\tilde{\Sigma}^{-1}$.
6: Iteratively compute the min-length solution $\hat{x}$ to

$$\text{minimize}_{x \in \mathbb{R}^n} \quad \|M^T Ax - M^T b\|_2.$$

7: Return $\hat{x}$.

Why we choose Gaussian random projection

Gaussian random projection

- has the best theoretical result on conditioning,
- can be generated super fast,
- uses level 3 BLAS on dense matrices,
- speeds up automatically on sparse matrices and fast operators,
- still works (with an extra "allreduce" operation) when A is partitioned along its bigger dimension.

Preconditioning for linear least squares

Given a matrix $N \in \mathbb{R}^{n \times p}$, consider the following least squares problem:

$$\underset{y \in \mathbb{R}^p}{\text{minimize}} \quad \|ANy - b\|_2.$$

Denote its min-length solution by y^*. We proved that $x^* = Ny^*$ if $\text{range}(N) = \text{range}(A^T)$. Similarly, we proved that x^* is the min-length solution to

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \|M^T Ax - M^T b\|_2,$$

if $\text{range}(M) = \text{range}(A)$.

Theoretical properties of LSRN

- In exact arithmetic, $\hat{x} = x^*$ almost surely.
- The distribution of the spectrum of AN is the same as that of the pseudoinverse of a Gaussian matrix of size $s \times r$.
- $\kappa(AN)$ is independent of all the entries of A and hence $\kappa(A)$.
- For any $\alpha \in (0, 1 - \sqrt{r/s})$, we have

$$\mathcal{P}\left(\kappa(AN) \leq \frac{1+\alpha+\sqrt{r/s}}{1-\alpha-\sqrt{r/s}}\right) \geq 1 - 2e^{-\alpha^2 s/2},$$

  where r is the rank of A.

It means that, if we choose $s = 2n \geq 2r$ for a large-scale problem, we have $\kappa(AN) < 6$ with high probability and hence we only need around 100 iterations to reach machine precision.

Tikhonov regularization

If we want to solve an under-determined system with Tikhonov regularization:

$$\text{minimize} \quad \frac{1}{2}\|Ax - b\|_2^2 + \frac{\lambda}{2}\|x\|_2^2.$$

we can first re-write it as

$$\text{minimize} \quad \frac{1}{2}\left\|\begin{pmatrix} z \\ r \end{pmatrix}\right\|_2^2 \quad \text{s.t.} \quad (A/\sqrt{\lambda}, I)\begin{pmatrix} z \\ r \end{pmatrix} = b,$$

which is asking for the min-length solution of an under-determined system. We have $x^* = z^*/\sqrt{\lambda}$, where (z^*, r^*) solves the above problem.

Implementation

- Shared memory (C++ with MATLAB interface)
  - Multi-threaded ziggurat random number generator (Marsaglia and Tsang 2000), generating 10^9 numbers in less than 2 seconds using 12 CPU cores.
  - A naïve implementation of multi-threaded dense-sparse matrix multiplications.
- Message passing (Python)
  - Single-threaded BLAS for matrix-matrix and matrix-vector products.
  - Multi-threaded BLAS/LAPACK for SVD.
  - Using the Chebyshev semi-iterative method (Golub and Varga 1961) instead of LSQR.

A comparison of least squares solvers

solver	min-len solution to		taking advantage of	
	under-det?	rank-def?	sparse A	operator A
LAPACK's DGELSD	yes	yes	no	no
Matlab's backslash	no	no	yes	no
Blendenpik	yes	no	no	no
LSRN	yes	yes	yes	yes

Table: Matlab's backslash uses different algorithms for different problem types. For sparse rectangular systems, it uses SuiteSparseQR, which tries to compute a sparse QR decomposition of A by analyzing A's pattern.

$\kappa(AN)$ and number of iterations

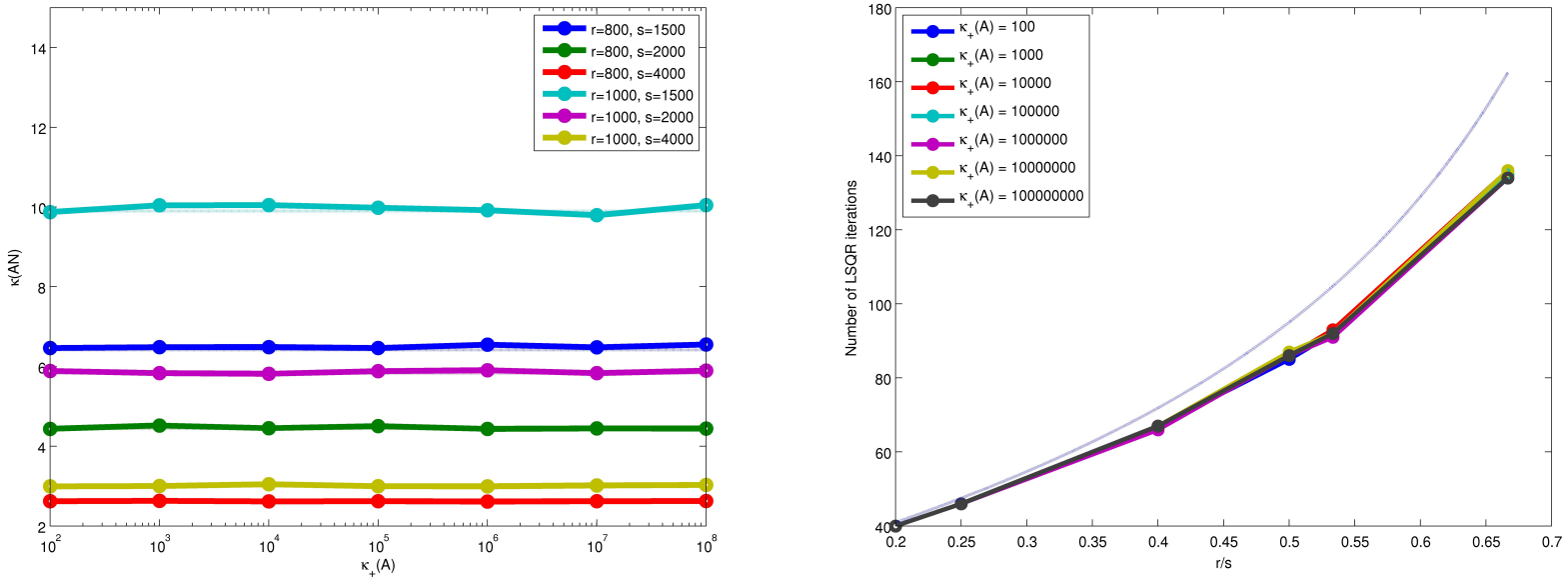

Figure: Left: $\kappa_+(A)$ vs. $\kappa(AN)$ for different choices of r and s. $A \in \mathbb{R}^{10^4 \times 10^3}$ is randomly generated with rank r. For each (r, s) pair, we take the largest value of $\kappa(AN)$ in 10 independent runs for each $\kappa_+(A)$ and connect them using a solid line. The estimate $(1 + \sqrt{r/s})/(1 - \sqrt{r/s})$ is drawn in a dotted line for each (r, s) pair, if not overlapped with the corresponding solid line. Right: number of LSQR iterations vs. r/s. The number of LSQR iterations is merely a function of r/s, independent of the condition number of the original system.

Tuning the oversampling factor

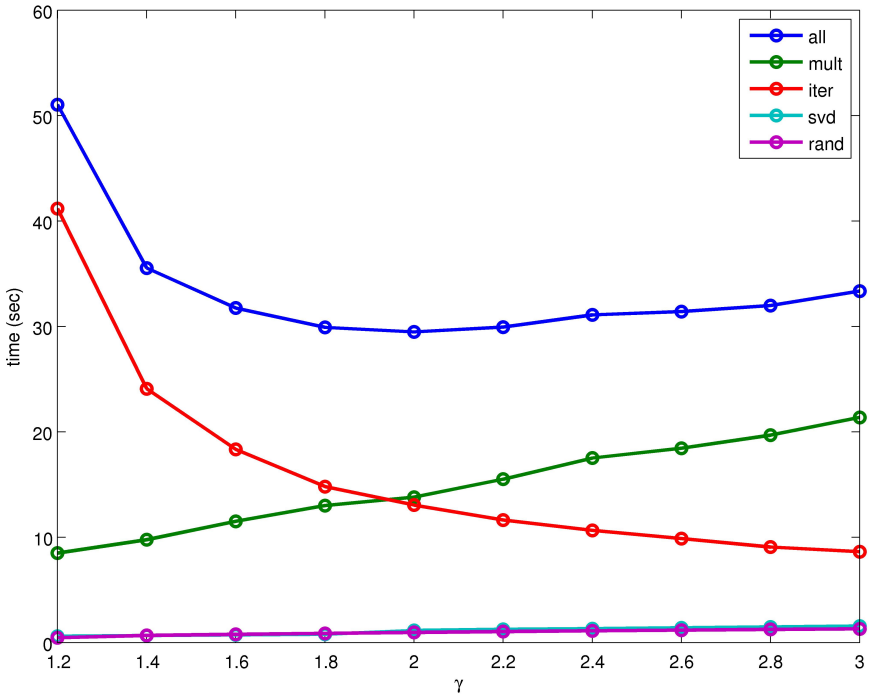

Figure: The overall running time of LSRN and the running time of each LSRN stage with different oversampling factor γ for a randomly generated problem of size $10^5 \times 10^3$. For this particular problem, the optimal γ that minimizes the overall running time lies in $[1.8, 2.2]$.

Solving dense least squares

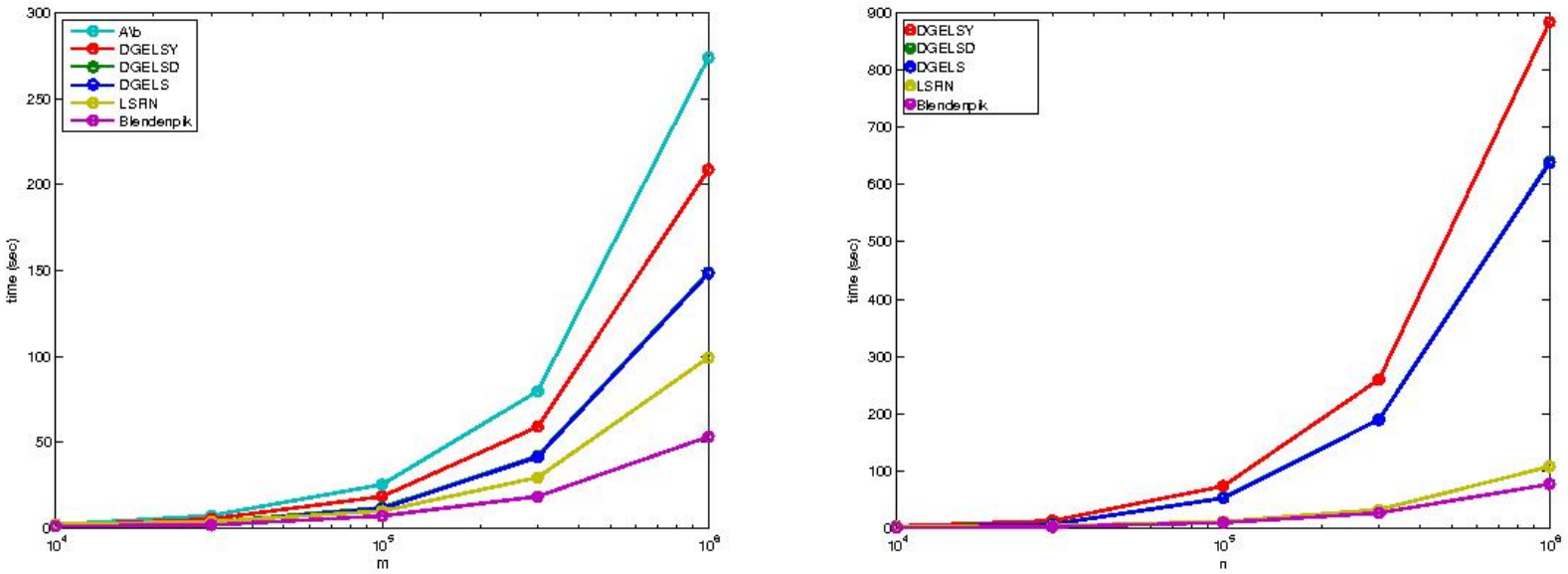

Figure: Running times on $m \times 1000$ dense over-determined problems with full rank (left) and on $1000 \times n$ dense under-determined problems with full rank (right). Note that DGELS and DGELSD almost overlap. When $m > 3e4$, we have Blendenpik $>$ LSRN $>$ DGELS/DGELSD $>$ DGELSY $> A\backslash b$ in terms of speed. On under-determined problems, LAPACK's performance decreases significantly compared with the over-determined cases. Blendenpik's performance decreases as well. LSRN doesn't change much.

Solving sparse least squares

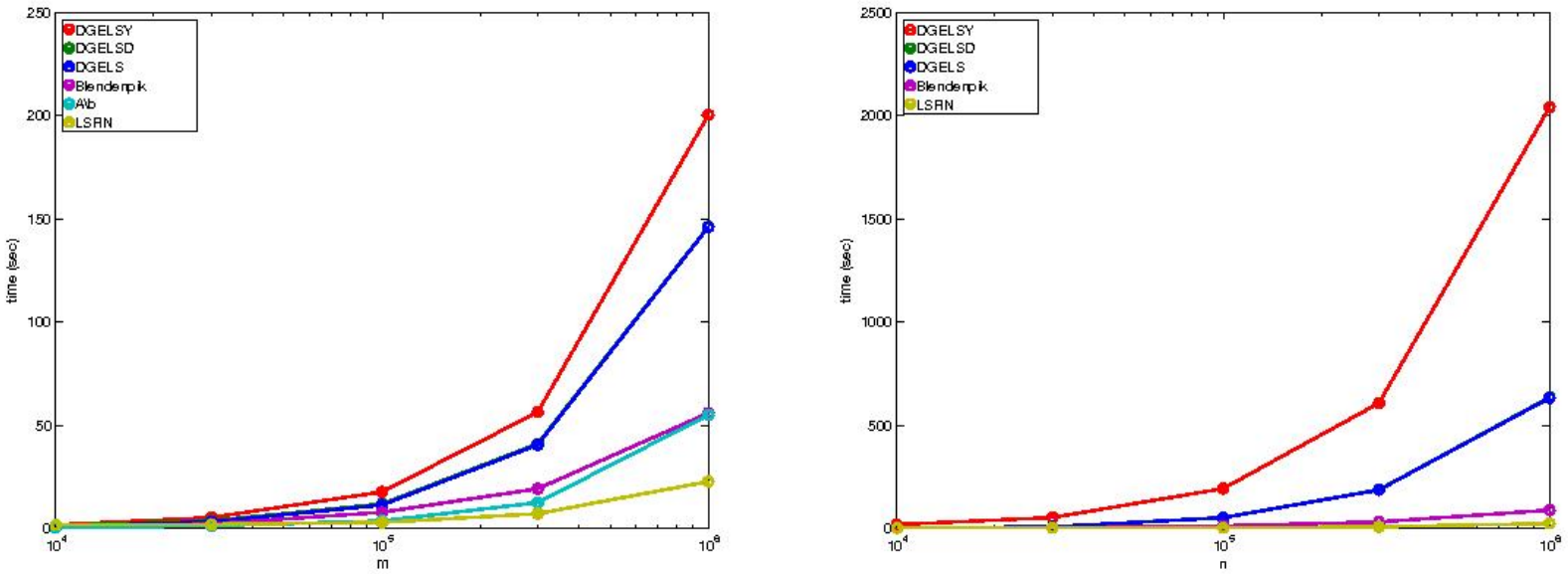

Figure: Running times on $m \times 1000$ sparse over-determined problems with full rank (left) and on $1000 \times n$ sparse under-determined problems with full rank (right). DGELS and DGELSD overlap with each other. LAPACK's solvers and Blendenpik perform almost the same as in the dense case. MATLAB's backslash speeds up on sparse problems, and performs a little better than Blendenpik, but it is still slower than LSRN. LSRN leads by a huge margin on sparse problems.

Solving real-world problems

matrix	m	n	nnz	rank	cond	DGELSD	$A\backslash b$	Blendenpik	LSRN
landmark	71952	2704	1.15e6	2671	1.0e8	29.54	0.6498*	-	17.55
rail4284	4284	1.1e6	1.1e7	full	400.0	$>$ 3600	1.203*	OOM	136.0
tnimg_1	951	1e6	2.1e7	925	-	630.6	1067*	-	36.02
tnimg_2	1000	2e6	4.2e7	981	-	1291	$>$ 3600*	-	72.05
tnimg_3	1018	3e6	6.3e7	1016	-	2084	$>$ 3600*	-	111.1
tnimg_4	1019	4e6	8.4e7	1018	-	2945	$>$ 3600*	-	147.1
tnimg_5	1023	5e6	1.1e8	full	-	$>$ 3600	$>$ 3600*	OOM	188.5

Table: Real-world problems and corresponding running times. DGELSD doesn't take advantage of sparsity. Though MATLAB's backslash may not give the min-length solutions to rank-deficient or under-determined problems, we still report its running times. Blendenpik either doesn't apply to rank-deficient problems or runs out of memory (OOM). LSRN's running time is mainly determined by the problem size and the sparsity.

Scalability and choice of iterative solvers on clusters

solver	N_{nodes}	$N_{\text{processes}}$	m	n	nnz	N_{iter}	T_{iter}	T_{total}
LSRN w/ CS	2	4	1024	4e6	8.4e7	106	34.03	170.4
LSRN w/ LSQR						84	41.14	178.6
LSRN w/ CS	5	10	1024	1e7	2.1e8	106	50.37	193.3
LSRN w/ LSQR						84	68.72	211.6
LSRN w/ CS	10	20	1024	2e7	4.2e8	106	73.73	220.9
LSRN w/ LSQR						84	102.3	249.0
LSRN w/ CS	20	40	1024	4e7	8.4e8	106	102.5	255.6
LSRN w/ LSQR						84	137.2	290.2

Table: Test problems on an Amazon EC2 cluster and corresponding running times in seconds. When we enlarge the problem scale by a factor of 10 and increase the number of cores accordingly, the running time only increases by a factor of 50%. It shows LSRN's good scalability. Though the CS method takes more iterations, it actually runs faster than LSQR by making only two cluster-wide synchronizations per iteration.

LSQR vs. the Chebyshev semi-iterative method

LSQR code snippet:

```
u     = A.matvec(v) - alpha*u
beta  = sqrt(comm.allreduce(np.dot(u,u)))
...
v     = comm.allreduce(A.rmatvec(u)) - beta*v
```

Chebyshev code snippet:

```
v  = comm.allreduce(A.rmatvec(r)) - beta*v
x += alpha*v
r -= alpha*A.matvec(v)
```

Pros and cons of LSRN

- Pros:
  - A high-precision iterative solver with predictable running time.
  - Accelerating automatically on sparse matrices and fast operators.
  - Capable of solving rank deficient problems and even taking advantage of rank deficiency.
  - Embarrassingly parallel (multi-threading or MPI) and scalable.
  - Possible to use the Chebyshev semi-iterative method.
  - Capable of handling Tikhonov regularization.
- Cons:
  - Large overhead on small-scale problems.
  - Randomized normal projection is slow on dense problems.
  - The smaller dimension cannot be too large.