# Group 4: The Kaczmarz Algorithm

Wei Deng, Nicole Eikmeier, Nate Veldt, and Xiaokai Yuan

April 28, 2016

## 1 Introduction

Project Objective:

Given a large sparse linear system $Ax = f$, of order $n = 10^6$ that can be effectively reduced to a banded matrix using the reordering scheme "reverse Cuthill-McKee", use the method of Row Projection, accelerated via the Conjugate Algorithm to yield a parallel solver. Compare the robustness and parallel scalability/speed with preconditioned Krylov subspace methods preconditioned via approximate LU-factorization.

The Kaczmarz Algorithm is a row projection method which is equivalent to solving $AA^Ty = f$ using the Gauss-Siedel iteration, with $x = A^Ty$. In Kaczmarz we consider each equation as a hyperplane:

$$S_i = \{x : A_i x - b_i = 0\}$$

for $i = 1, 2, ...,$ where $A_i$ and $b_i$ are $i$th row of matrix $A$ and vector $b$. So the problem is transformed into finding the coordinates of the point of intersection of these hyperplanes. See figure 1 for a visualization of the problem.

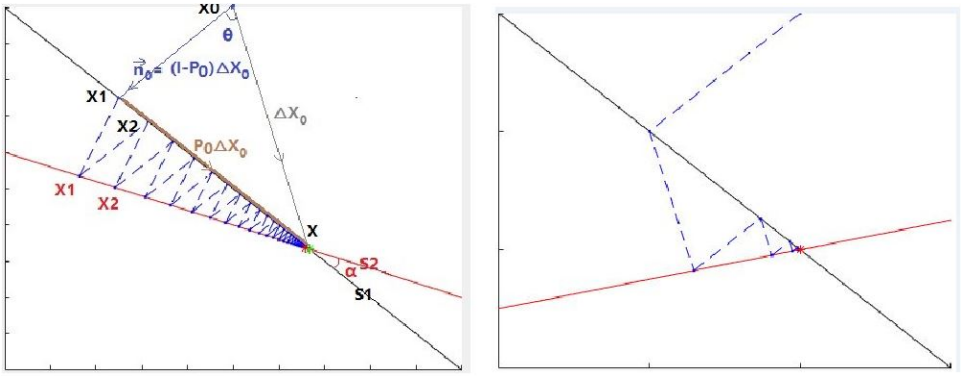



Figure 1: 2-Partition Case of the Kaczmarz Algorithm

From this basic idea we use symmetrization and acceleration via the CG method to improve the procedure. The details will be given carefully in the following sections.

# 2 Implementation

## 2.1 Reverse Cuthill-McKee and the Woodbury Formula

The first step in our implementation is to transform the sparse matrix $A$ to a banded form using Reverse Cuthill-McKee. We can perform a symmetric permutation with matrix $P$ so that $P^TAP$ is banded. In figure 3, we see the result from Matlab of Reverse Cuthill Mckee on the stomach data set. The band size is nice and small.

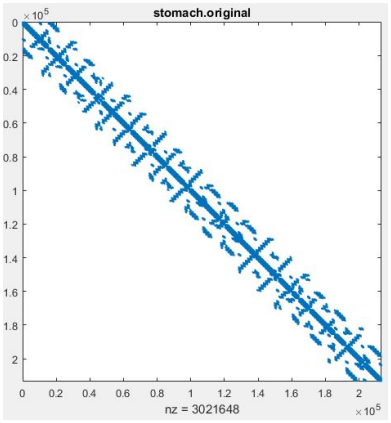



Figure 2: $A$ sparse, non-symmetric

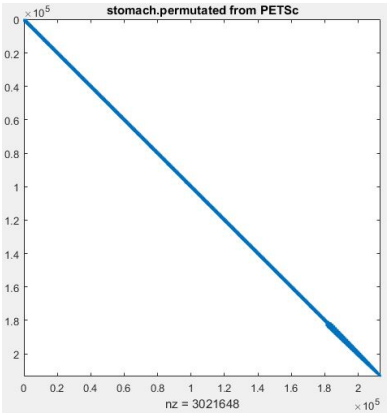



Figure 3: $P^TAP$ (banded)

For some problems the bandwidth could be too large after using Reverse Cuthill Mckee. See for example figure 5, where we see the result of RCM on the lns_131 data set, again in Matlab. If this is the case, instead we could choose $P$ so that $P^TAP$ is narrow banded plus a low rank matrix. See figure 6.

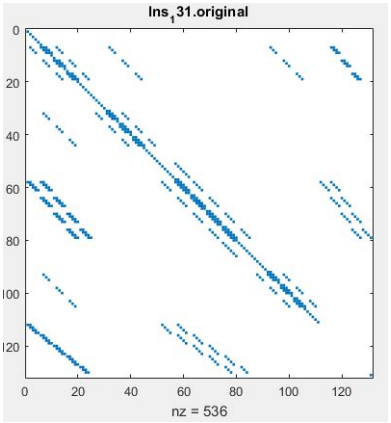



Figure 4: $A$ is sparse and non-symmetric

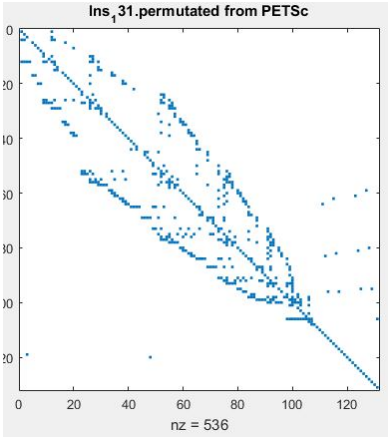



Figure 5: $P^TAP$ narrow band+low rank

In this case we use the Woodbury Formula to solve $Ax = b$. Recall first that $Ax = b$ implies $x = A^{-1}b$. Then,

$$\begin{aligned} A^{-1} &= (B - USV^T)^{-1} \\ &= B^{-1} - B^{-1}UTV^TB^{-1} \end{aligned}$$

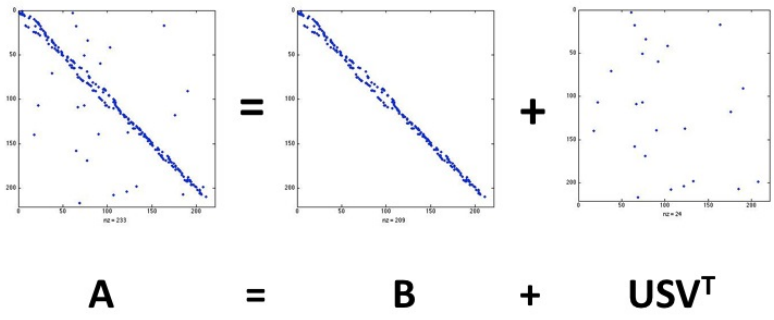



Figure 6: Breaking up $A$ into a banded matrix plus a low rank matrix

| matrix | size | Matlab | PETSc | rcm.cpp |
|---|---|---|---|---|
| lns | 131 | 32 | × 111 | × 113 |
| Jac2-db | 21,982 | 545 | × | × |
| bayer01 | 57,735 | × 18,322 | × | × |
| venkat25 | 62,424 | 1,515 | 1,515 | 1,495 |
| stomach | 213,360 | 1,133 | 2,216 | 2,239 |
| atmosmodd | 1,270,432 | 7,772 | 7,772 | 7,772 |

Table 1: Bandsize after RCM using various codes

where $T = (V^T B^{-1} U - S^{-1})^{-1}$. And

$$\begin{aligned} x &= A^{-1} b \\ &= \mathbf{B^{-1}b} - B^{-1} U T V^T \mathbf{B^{-1}b} \\ &= \mathbf{a} - B^{-1} U T (V^T \mathbf{a}) \\ &= \mathbf{a} - B^{-1} U T \mathbf{c} \quad (\text{ solve } (V^T B^{-1} U - S^{-1})\mathbf{d} = \mathbf{c}) \\ &= \mathbf{a} - B^{-1} U \mathbf{d} \\ &= \mathbf{a} - B^{-1} \mathbf{h} \end{aligned}$$

The Woodbury formula is useful because all systems involving $B$ are relatively easy to solve. We did not implement the Woodbury formula in our experiments, due to time contraints. This means that we transformed $A$ to a completely banded matrix, no matter how large the band size. We expect there would be improvements in our final results if the Woodbury formula was implemented.

In table 1, the band size after permutation is shown. Matlab performed the best across the board, but all three permutations did well for our largest test case. You can see also that for the bayer01 matrix, our band size is very large, and so we did not get reasonable results in our parallel Kaczmarz implementation.

## 2.2 The Kaczmarz method

Once we have a banded matrix, $A$, we move on to the Kaczmarz algorithm, which was introduced earlier. Suppose we partition $A$ into two pieces

$$A = \begin{pmatrix} A_1^T \\ A_2^T \end{pmatrix}$$

and we also partition the right hand size vector $f^T = (f_1^T, f_2^T)$.

In the classical Kaczmarz method at each iteration we compute:

$$x_k = x_k + \overrightarrow{n_k} = x_k + \frac{r_k^i}{||A_i||^2} A_i^T$$

where $\overrightarrow{n_k} = \triangle x_k cos\theta = \frac{\langle A_i, \triangle x_k \rangle}{||A_i||^2} A_i^T = \frac{b_i - \langle A_i, x_k \rangle}{||A_i||^2} A_i^T = \frac{r_k^i}{||A_i||^2} A_i^T$.

$$\begin{cases} AA^T y = f \\ x = A^T y \end{cases} \implies AA^T = \begin{pmatrix} A_1^T \\ A_2^T \end{pmatrix} (A_1, A_2)$$

From Gauss-Seidel, use $x^{k+1} = \begin{pmatrix} A_1, A_2 \end{pmatrix} \begin{pmatrix} y_1^{k+1} \\ y_2^{k+1} \end{pmatrix}$.

$$\begin{pmatrix} A_1^T A_1 & 0 \\ A_2^T A_1 & A_2^T A_2 \end{pmatrix} \begin{pmatrix} y_1^{k+1} \\ y_2^{k+1} \end{pmatrix} = \begin{pmatrix} 0 & -A_1 A_2 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} y_1^{k+1} \\ y_2^{k+1} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

Replacing some parameters with projection operator $P_i$, we can get:

$$x^{k+1} = A_1 y_1^{k+1} + A_2 y_2^{k+1} = Q x_k + b$$

where $Q = (I - P_2)(I - P_2)$. Similarly, for an m-partition,

$$x^{k+1} = Q_u x^k + f_u = (I - P_m)(I - P_{m-1}) \dots (I - P_1) x^k + f_u$$

.

However, the spectral radius of $Q_u$ may interfere with the convergence speed, and the distribution of eigenvalues could also influence the performance. To handle this, we can symmetric $Q_u$ to get an accelerated iteration:

$$x^{k+1} = Q(\omega) x^k + Tf$$

where $Q(\omega) = (I - \omega P_1)(I - \omega P_2) \dots (I - \omega P_m) \dots (I - \omega P_2)(I - \omega P_1)$. Since $I - Q(\omega)$ is symmetric positive definite. The conjugate gradient method is suitable to accelerate the basic scheme.

Once we have a banded matrix, we considered using various partitions of the rows. A very simple example is shown in figure 7. A permutation like this gives several benefits. The first is that we can have an outer level of parallelism for each projection. Also, when we split the matrix this way we create several small independent least squares problems, which reduces the time.

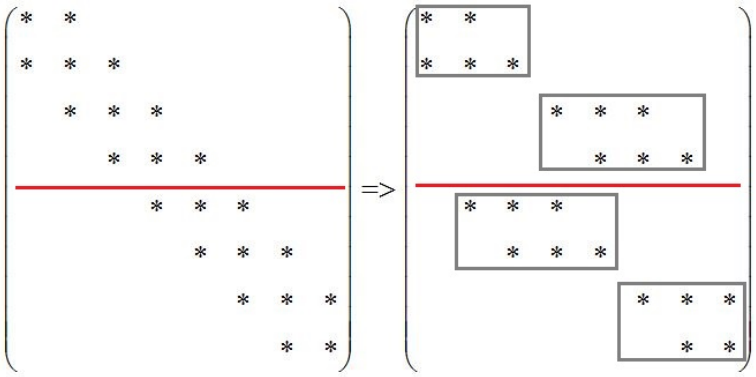


Figure 7: Re-ordering the banded matrix

In figure 2.2 we consider different types of partitions, both in the inner and outer levels. We found that have more independent blocks resulted in faster implementation of Kaczmarz. We also found that more partitions, i.e. taking 4 partitions:

$$A = \begin{pmatrix} A_1^T \\ A_2^T \\ A_3^T \\ A_4^T \end{pmatrix}$$

instead of 2 partitions is slower. To further illustrate this, it proved that the optimal value of $\omega$ is 1.0 if we use two partitions, and $Q(1)$ has the minimal spectral radius. For these reasons we decided to use 2 outer partitions for our implementation. In this case the problem can be simplified as follows:

$$(I - Q(1))x = Tf$$

## 2.3 Symmetrized Kaczmarz

Unfortunately, the spectral radius of $Q$ may interfere with the convergence speed and the distribution of eigenvalues could also influence the performance. To handle this, we can symmetrize $Q$ to get an accelerated iteration. In our symmeterized version, we wish to solve $(I - Q)x = c$, where $c = Tf$, and

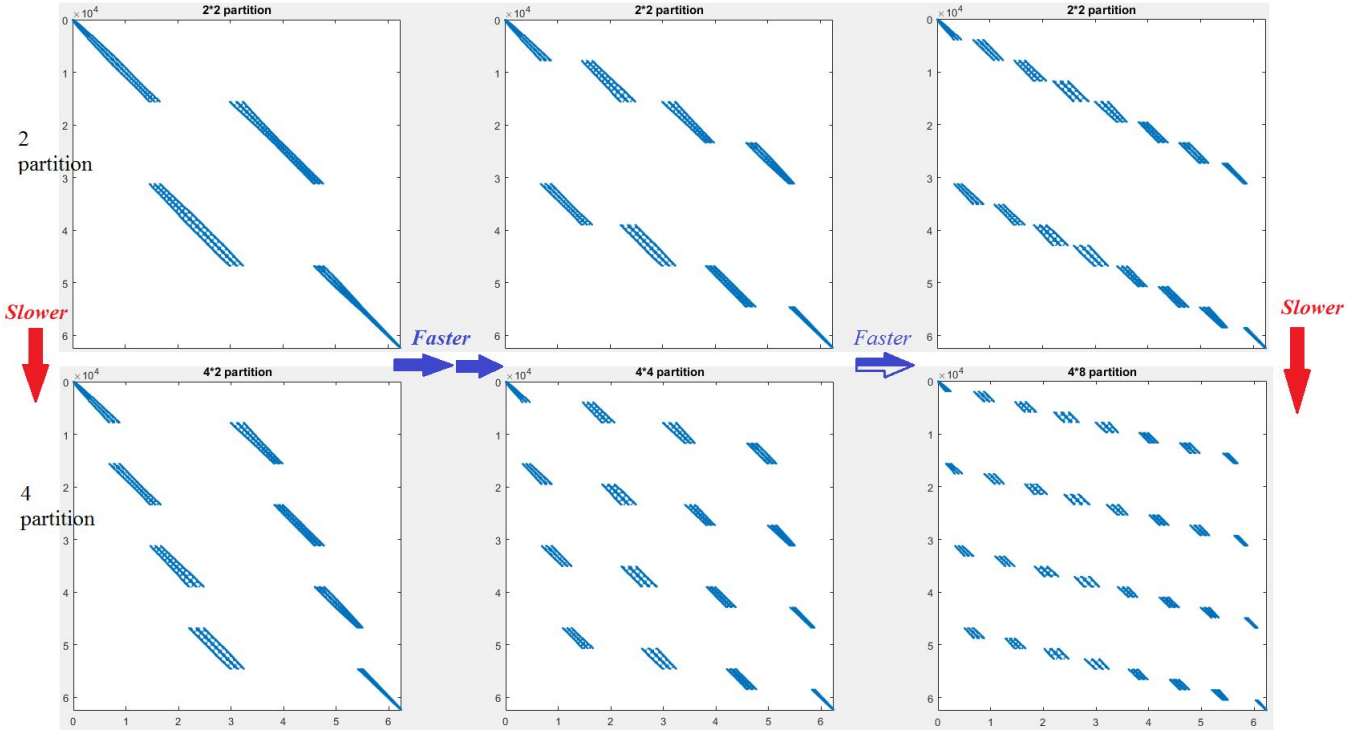
$$Q = (I - P_1)(I - P_2) \cdots (I - P_m)^2 \cdots (I - P_2)(I - P_1)$$

with $P_i = A_i(A_i^T A_i)^{-1} A_i^T$, and $T = A^T(D + L)^{-T} D(D + L)^{-1}$.

The algorithm to solve this system follows:

- Step1: chose $x_0$; set $k = 0$.
- Step2: Compute $x_k = Qx_k$.

$$x_k = (I - P_i)x_k, \quad i = 1, 2, ..., m - 1, m, m - 1, ..., 2, 1$$




- Step3: $x_k = x_k + c$;
- Step4: If a convergence criterion is satisfied, terminate the iterations; else set k=k+1 and go to Step2.

It is not stable to form $P_i$ directly since it will square the condition number of $A_i$, and at the same time will cost time on solving $(A_i^T A_i)^{-1}$. Instead, given a vector $u$ obtain $v = (I - P_j)u \Leftrightarrow \min_v ||u - A_j w||_2$. Then solve $\min_v ||u - A_j w||_2$ directly.

In Petsc, we use CG method on normal equation: $A_j^T A_j w = A_j^T u$. The parallel step is:

$$
\begin{aligned}
&v = \min_v ||u - A_j w||_2 \Leftrightarrow \min_{v_i} ||u_i - A_{j,i} w_i||_2, \\
&v^T = [v_1^T, v_2^T, ..., v_k^T], \\
&w^T = [w_1^T, w_2^T, ..., w_k^T].
\end{aligned}
$$

## 2.4 Acceleration via the CG Method

As stated previously, since $(I - Q)$ is symmetric positive definite, we accelerate solving:

$$(I - Q)x = Tf$$

using the conjugate gradient method. This is the framework of our entire algorithm. The outline follows:

**Step 1** : $x_0 = c$
Compute $r_0 = Tf - \boxed{(I - Q)c} = Qc$

Set $p_0 = r_0, i = 0$
**Step 2**: Compute:
$\alpha_i = (r_i, r_i)/(p_i, \boxed{(I-Q)p_i})$
$x_{i+1} = x_i + \alpha_i p_i$
$\beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
$p_{i+1} = r_{i+1} + \beta_i p_i$
**Step 3**: If convergence criterion is satisfied, terminate the iterations; else set $i = i+1$ and return to Step 2.

We can take a convergence criterion as :

$$\frac{||r_i||}{||r_0||} \leq \epsilon$$

The boxed out portions in our framework are very important. Remember that whenever we see:

$$(I-Q)u$$

for any vector $u$, we really mean to solve multiple least squares problems:

$$(I-Q)u = (I-P_1)(I-P_2)(I-P_1)u$$

$$(I-P_i)u \Rightarrow \min_v ||u - A_i w||$$

## 2.5 LSQR

In the end, in order to solve the least squares problem, we implemented LSQR. LSQR is an iterative method for solving $Ax = b$ where $A$ is large, sparse, and overdetermined. It is equivalent at each iteration to CG.

**Algorithm 1** LSQR

$\beta_1 u_1 = b, \alpha_1 v_1 = A^T u_1, w_1 = v_1, x_0 = 0, \bar{\phi_1} = \beta_1, \bar{\rho_1} = \alpha_1$
**for** $i = 1, 2, 3, \cdots$ **do**
    Bidiagonalization:
    $\beta_{i+1} u_{i+1} = A v_i - \alpha_i u_i$
    $\alpha_{i+1} v_{i+1} = A^T u_{i+1} - \beta_{i+1} v_i$
    Orthogonal Transformation:
    $\rho_i = \sqrt{(\bar{\rho_i}^2 + \beta_{i+1}^2)}$
    $c_i = \bar{\rho_i}/\rho_i,$
    $s_i = \beta_{i+1}/\rho_i,$
    $\theta_{i+1} = s_i \alpha_{i+1},$
    $\bar{\rho_{i+1}} = -c_i \alpha_{i+1},$
    $\phi_i = c_i \bar{\phi_i},$
    $\bar{\phi_{i+1}} = s_i \bar{\phi_i}.$
    Update x:
    $x_i = x_{i-1} + (\phi_i/\rho_i) w_i,$
    $w_{i+1} = v_{i+1} - (\theta_{i+1}/\rho_i) w_i.$

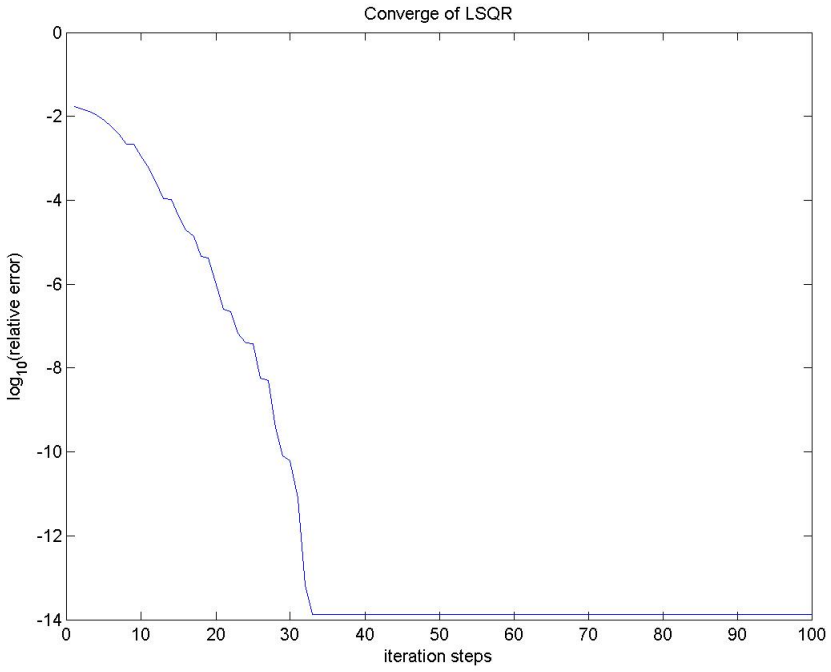



Figure 8: Convergence of LSQR

# 3 Results and Conclusions

## 3.1 Results

We will now discuss the results of our Parallel Solver. In figure 9, we show the run-times for 3 smaller test-cases. The x-axis shows the number of MPI-processes, while the y-axis is the run-time in seconds. We see that we do have decent scalability up to 4 nodes. At 8 nodes we do not see much improvement, or have a longer run-time. There are many reasons this could be. Our method is implemented in Petsc which is not optimized for efficiency. Also there may be too much communication in our method to fully take advantage of 8 nodes. We get similar results for our largest test-case, shown in figure 10.

Note: these results are the time taken to converge to with a tolerance of $10^{-8}$.

To find a benchmark of how well our method performed, we compared with a Krylov Subspace method. The implementation of this consisted of first using the MC64 software to re-order the matrix. This software finds a permutation which brings non-zeros to the diagonal. Once this permutation is used, then we called a Krylov Subspace method with ILU pre-conditioner in Petsc. The Krylov subspace method used is GMRES.

We quickly realized that the ILU preconditioner failed on some of our data. The reason for this could be that the eigenvalues of these matrices are distributed too much across the positive and negative spectrum. The ILU preconditioner is not well equipped to handle these so called "highly indefinite" matrices. Table 2 shows which of our data failed with this method.

Finally, we compared the run-time of this Krylov method to our results on 4 MPI

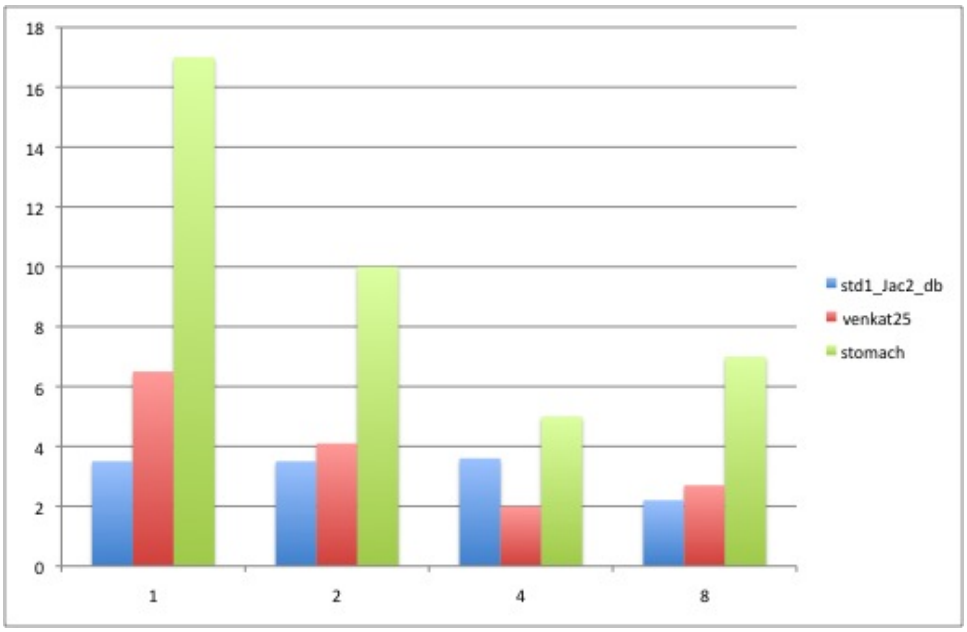



Figure 9: std1_Jac2bd: $n = 21982$; venkat25: $n = 62424$, stomach: $n = 213360$.

| matrix | size | Converged? |
|---|---|---|
| lns | 131 | × |
| std1-Jac2-db | 21,982 | × |
| bayer01 | 57,735 | × |
| venkat25 | 62,424 | ✓ |
| stomach | 213,360 | ✓ |
| atmosmodd | 1,270,432 | ✓ |

Table 2: ILU preconditioner

processors. We compared both the number of iterations and the time to converge to a tolerance of $10^{-8}$. The results are shown in Table 3. Overall our parallel solver outperforms the Krylov Subspace Method. The data set Stomach is an exception; this data set has very nice properties that allows the Krylov Method to converge very fast.

## 3.2 Conclusions

In conclusion, it was a mistake to use Petsc. Petsc is better for high level implementation, not for optimization and efficiency. Petsc was also not well suited specifically for grabbing the exact sub-matrices that we needed in order to call the parallel Least Squares.

In the end we were able to salvage our project by implementing LSQR, which greatly reduced our run-time and even produced some nice scalability. Even so, if we had left Petsc out of our project we believe our results would have been even better.

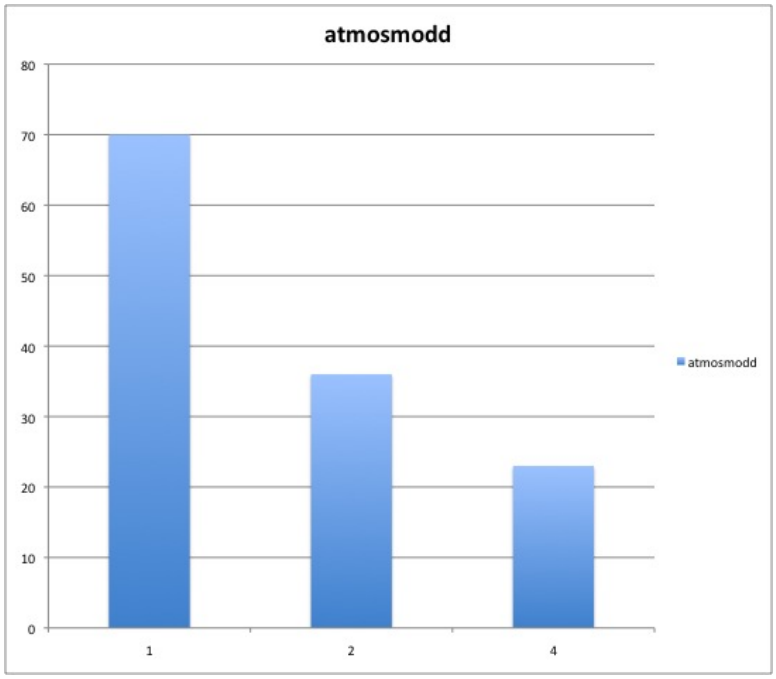



Figure 10: Runtime in seconds for largest test case ($n = 1270432$).

| | | Our Parallel | Solver | Krylov | ILU |
|---|---|---|---|---|---|
| matrix | size | num-its | time (s) | num-its | time (s) |
| lns | 131 | 10 | .1 | × | × |
| Jac2-db | 21,982 | 13 | .8 | × | × |
| bayer01 | 57,735 | × | × | × | × |
| venkat25 | 62,424 | 12 | 1.5 | 374 | 14.17 |
| stomach | 213,360 | 15 | 4 | 16 | 2.21 |
| atmosmodd | 1,270,432 | 14 | 21 | 266 | 130.72 |

Table 3: Run times for 4 MPI Processors to converge with tol = $10^{-8}$