CS C267 Homework 4

Kevin Fong, Usman Jamshed, Anna Weber

May 12, 2023

Abstract

1 Problem Statement

For this assignment, we were given a naive implementation to solve the a sparse linear system $A\mathbf{x} = \mathbf{b}$. The conjugate gradient algorithm is an iterative method that can be used to solve linear systems where the matrix is symmetric and positive-definite. However as the convergence of the conjugate gradient algorithm can deteriorate significantly when A is ill-conditioned, we can use a preconditioner to address this lack of robustness. The idea of preconditioned iterative methods is to transform the initial problem $A\mathbf{x} = \mathbf{b}$, for example by left-multiplying the equation by an invertible matrix P, where P is chosen so that $cond(PA)$ is closer to 1 than $cond(A)$. Matrix P is called the preconditioner, and the idea is to choose P such that P is an approximation of A^{-1} while being "cheaper" to compute.

```
**function** $precondCG(A, P, \mathbf{b}, \epsilon_{TOL})$
$\epsilon = \epsilon_{TOL}|\mathbf{b}|_2$
$\mathbf{x} = 0$
$\mathbf{r} = \mathbf{b}$
$\mathbf{z} = P\mathbf{b}$
$\mathbf{p} = \mathbf{z}$
**while** $|\mathbf{r}|_2 \geq \epsilon$ **do**
    $\alpha = \mathbf{r}^T\mathbf{z}/|\mathbf{p}|_A^2$
```

```
        $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
        $\mathbf{r} = \mathbf{r} - \alpha A \mathbf{p}$
        $\mathbf{z} = P\mathbf{r}$
        $\beta = \mathbf{r}^T \mathbf{z} / (\alpha |\mathbf{p}|_A^2)$
        $\mathbf{p} = \mathbf{z} + \beta \mathbf{p}$
    end while
    return ($\mathbf{x}$)
end function
```

Specifically the block Jacbobi Preconditioner restricts the matrix to p block diagonals where each MPI process holds a subset of $n := N/p$ rows of the matrix and vector. Thus each process can independently compute a Cholesky factorization of its diagonal block to parallelize the preconditioned conjugate gradient algorithm.

2 Implementation

2.1 Eigen SparseMatrices

To build and hold our sparse matrices we leveraged the C++ library, *Eigen*. It has a built in class called *SparseMatrices* which can be created in row or column major format. It is especially convenient for its built in functionality with respect to matrix-vector multiplication and obtaining block submatrices $.middleCols()$, which were necessary for our implementation.

2.2 1D Approach

Distributing the total $N \times N$ matrix amongst n processors in a 1D approach was done by first calculating the number of rows per processor nr and the row offset for each processor. We were then able to to build local matrices in parallel where each processor only built the part of the $N \times N$ matrix that it was responsible for (Figure 2). The result is n matrices of size $nr \times N$.

Each processor then creates local x and b vectors of size nr which are sent along with the local matrix and the row offset to the conjugate gradient (CG) function. In the conjugate gradient we block the diagonals in each local matrix using the row offset to create an $nr \times nr$ matrix and pass it directly to the Eigen::SimplicalCholesky class (Figure 3). We can then perform the matrix-

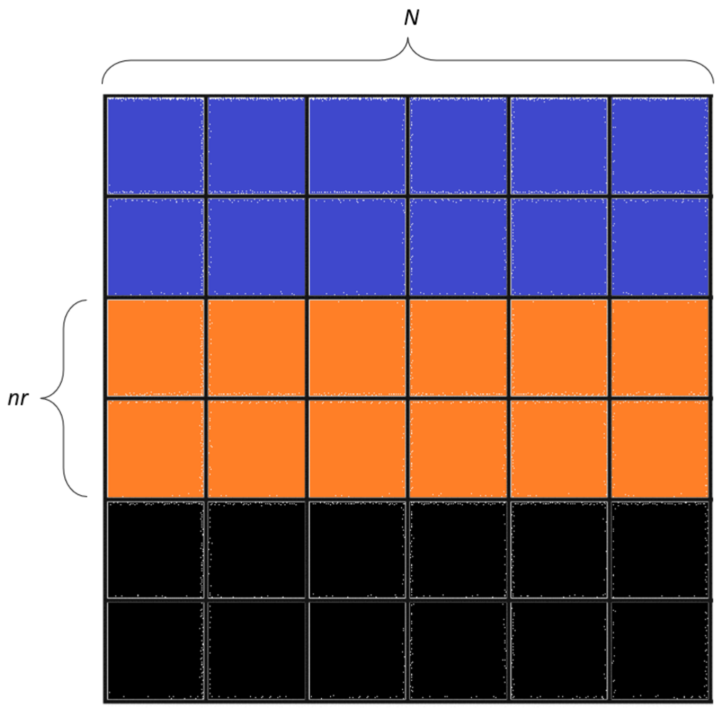

Figure 1: A 6×6 matrix partitioned in 1D on 3 processors.

vector multiply and compute the CG until it has converged. This process is much more efficient than looping through the Map data structure to build out the block diagonal of the current processor as seen in the naive implementation. Since different rows contribute to different sections of the vector product, it is a reasonable strategy to parallelize this operation because processors will not need to handle any communication or synchronization.

Once all processors have converged their CG, we calculate the local r for each processor using the local block diagonal matrix and the converged local x and b vectors. At this point we have 2 MPI_Gather calls which gather all the r and b values from the processors onto two vectors on the rank 0 processor $total\_r$ and $total\_b$. Then we simply calculate the norm of each and divide them to get our final error for the process.

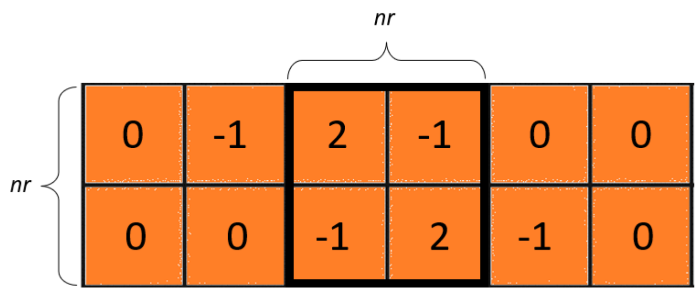

Figure 2: A 2×6 matrix with its $nr \times nr$ diagonal block highlighted.

2.3 2D Initialization

In the construction the two-dimensional process layout for matrix handling, an $m \times n$ grid of processors can be used to represent a square grid. However specifically for the Jacobi block preconditioned conjugate gradient algorithm, since we are interested in the diagonal blocks of the matrix, the layout is limited to a perfect grid of an $n \times n$ processors. so initial argument handling must be taken before setting up the system to only use the largest square number p smaller than the allocated processes, where $p = s \times s$ and s is the number of processors along the length of a matrix. Similar to the 1D approach, where nr is the number of rows per processor, each processor will build its local view of the $N \times N$ matrix. The result is p matrices of size $nr \times nr$.

2.4 Matrix-Vector Multiplication in 2D

Parallelization of matrix-vector multiplication can be implemented in the 2D grid similarly to 1D, but with additional communication. Because rows of the matrix are distributed, several processes will contribute to the a corresponding subvector of the matrix-vector product. The local matrix-vector product contributions can be computed in parallel, then all the contributions are serially sent to the first processor in its row and added. This reduces the product vector to s sub-vectors across s processors, similarly to 1D into rows. the sub-vectors are then broadcast specifically to the contributing processors for that row.

2.5 2D Parallelization

In reconciling the Jacobi block-conditioner, we reduce the initial matrix to the block diagonals. Thus our first design choice was to only call the Jacobi block-conditioner on a local matrix if the processor rank was a diagonal rank. Otherwise the local matrix is transformed into the 0 matrix.

Similar to 1D each processor undergoes the Conjugate Gradient loop until the calculated residue falls within the tolerance. However the should exit the loop when checking if the residue of the diagonal block convergesl, rather than an off diagonal. This is important to handle as the CG algorithm implements MPI_Send and MPI_Recv to expect a explicitly defined number of calls to conduct matrix-vector multiplication. Missing send calls from exited processors would induce a lock. This is handled by signalling a processor to exit the loop only when its block diagonal converges. This is handles by MPI_Send and MPI_Recv, but an alternative to define smaller communicator groups and implement a Broadcast was also considered.

Once all processors have converged their CG, we calculate the local r for each processor along the local block diagonal matrix and the converged local x and b vectors. We define a specific communicator group containing only the diagonal ranks. Just as in 1D again we use 2 MPI_Gather calls to gather all the r and b values onto two vectors on the rank 0 processor *total_r* and *total_b*. Then we simply calculate the norm of each and divide them to get our final error for the process.

3 Results and Evaluation

3.1 1D vs Map Performance

Comparing performance between the Map and the 1D implementation we see that they are quite close in terms of timing on a matrix with a side length of 100,000. We do see a constant marginal improvement between the two across all processors (∼1.5x), which may continue to be the case as we scale to a greater number of processors (Fig. 3).

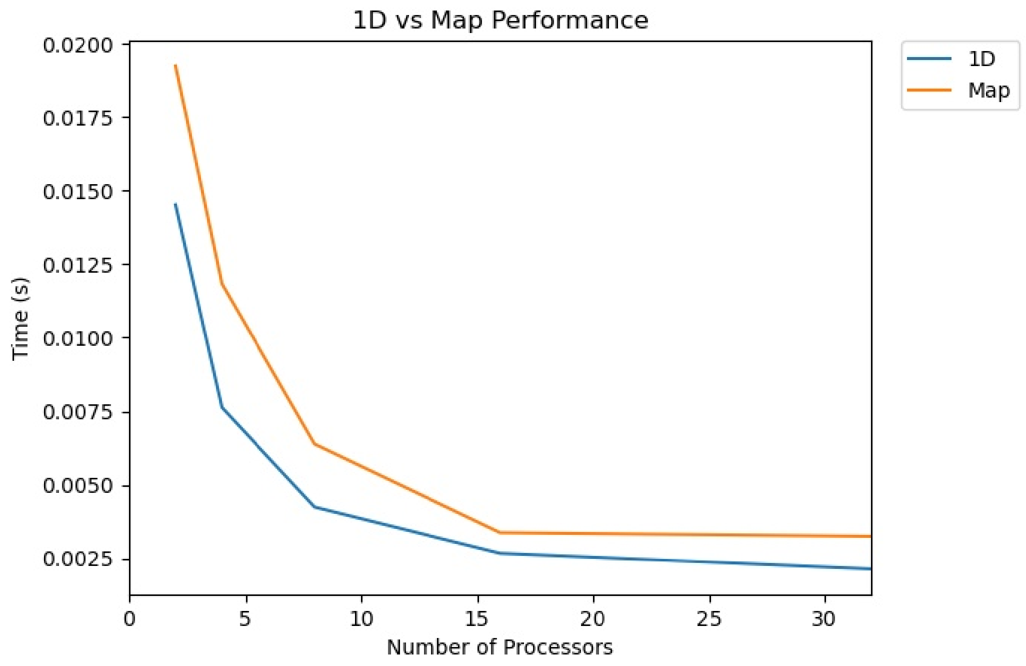

Figure 3: 1D Strong Scaling on a 100000×100000 matrix from 2-32 processors.

3.2 1D Strong Scaling

To evaluate how our code would perform under strong scaling conditions we held the side length of the matrix constant at 100 000, and varied the number of processors starting from 2 and doubling until 32 (Fig. 4). By simply having 4 processors our time taken drops by nearly half and we continue decreasing the time taken until around 16 processors, after which further increases in processor count do not yield significant improvements in the time taken to run the program. This is indicative of good strong scaling at low processor counts but shows poor strong scaling at higher processor counts. In total we were able to achieve a $7\times$ performance gain when at 32 processors compared to 2.

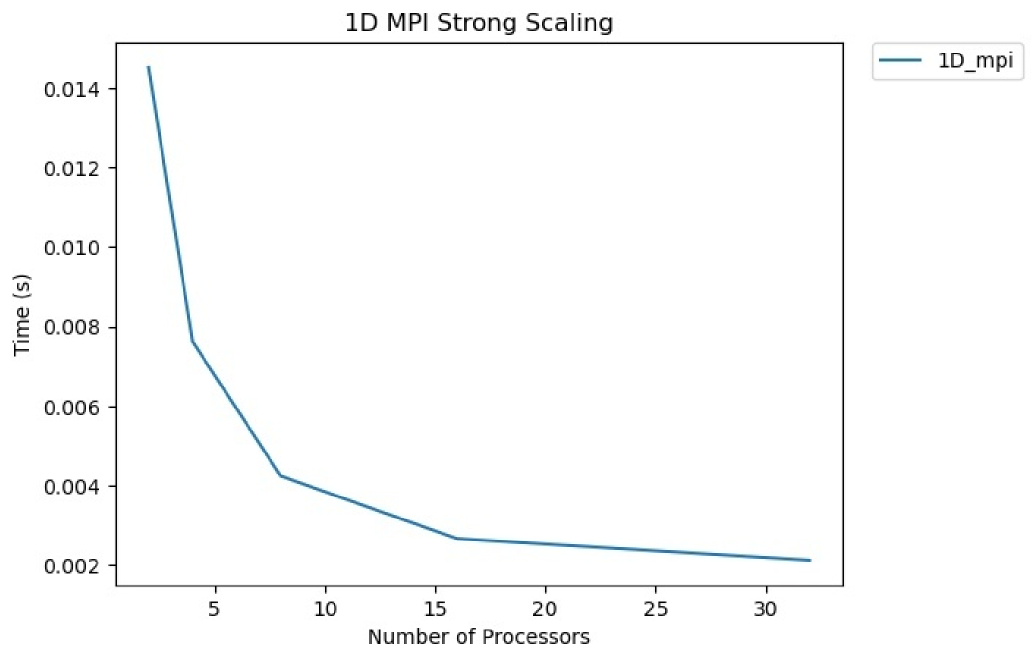

Figure 4: 1D Strong Scaling on a 100000 × 100000 matrix from 2-32 processors.

3.3 Matrix Creation Time

Creating the matrices locally using a Map, 1D or 2D incurred a decent amount of time and required the use of an MPI_BARRIER (Fig. 5). Across all processors, distributing the matrix in a 2D fashion was the slowest. One reason why the 2D implementation took longer at low processor counts was due to the double for loop required, compared to the single for loop found in the 1D and Map implementations. At low processor counts 1D was the fastest but gets overtaken by the Map at high processor counts. It is important to note that at high processor counts, all the distribution patterns take roughly the same time with marginal timing differences.

3.4 2D Performance

Although we were successful in the compilation of the 2D algorithm, we were not able to achieve accurate results. The error of the norms differed from our test cases when performed in the naive case and in our 1D algorithm.

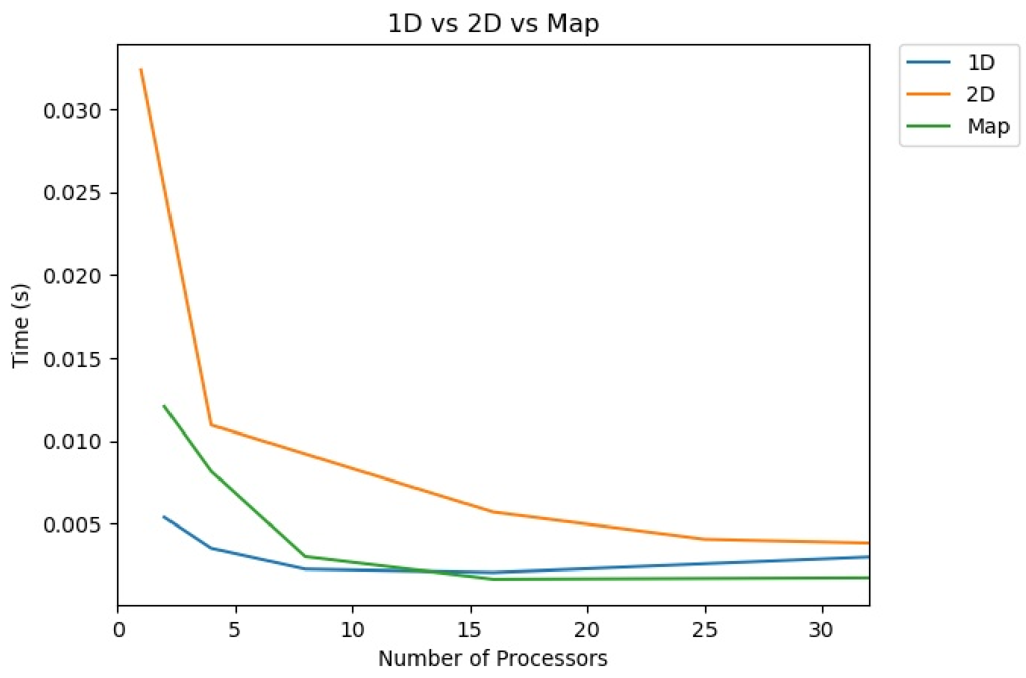

Figure 5: Comparing matrix creation time in 1D, 2D and Map.

Furthermore, we were unsure what methodology of comparison would be appropriate. This is because the only point at which any computation is distributed and parallelized is during matrix-vector multiplication, and the conjugate gradient loop. However the If the Jacobi Block conditioner is transforming any off diagonal blocks to 0, the matrix vector multiplication would be adding vectors of 0 to our actual product. This reduces the actual problem to only our block diagonal matrix.

Because of this reduction, it made it unclear to compare the performance of 1D on p processors to 2D on p where the number of processors is the same but the block diagonals are partitioned differently, or compare the performance of 1D on p processors to 2D on $p \times p$ where the block diagonals are the same, but essentially more processors have been added to the problem. In looking at preliminary data, because no computation from the 1D problem is even being parallelized when implementing a 2D process layout, we believe that 2D would be slowed down by the strict synchronization that was implemented to handle dependencies in the computation. Therefore we believe that 1D is a more appropriate implementation to handle the Conjugate Gradient for the Jacobi block pre-conditioner algorithm.

4 Contributions

Kevin Fong - 2D Code implementation
Usman Jamshed - Eigen SparseMatrix , 1D Code implementation
Anna Weber - Report Writeup