An Implementation of Parallel Incomplete LU Factorization

Marcos Botto Tornielli *

May 11, 2021

Abstract

An implementation of the parallel incomplete LU (ILU) factorization method developed by Chow and Patel [1] is presented, using C++ and OpenMP. The relevance of this topic to science and engineering is the ILU factorization's usefulness as a preconditioner to aid in the solution of sparse linear systems that can arise in many different fields. The algorithm shows near ideal speedup up to 16 threads and sustains good speedup even up to 56 threads, suggesting that it is a good candidate to improve the efficiency of preconditioning on a shared memory architecture.

1 Introduction

Linear systems that feature sparse matrices arise commonly when modeling physical processes with applied mathematics. For complex problems, solving these systems can involve computing millions of unknowns, leading to high computational cost in terms of both operation count and memory usage. One way to approach this problem is to use iterative methods, which involve refining an initial guess for the unknowns in order to reach an approximate solution. To implement iterative methods efficiently, computational algorithms must be developed that take advantage of both the sparsity of the system and the potential for parallel computation. These algorithms can involve intermediate steps that modify the system so that it will be better suited for numerical computation; this process is known as preconditioning. Finding a preconditioner requires the computation of an additional matrix or matrix factorization, and therefore also benefits from algorithms that are aware of sparsity and parallelism.

One example of such an algorithm is the parallel ILU (PILU) factorization method developed by Chow and Patel [1] (LU factors of a matrix are an option for preconditioning). This method features an iterative approach to computing the approximate L and U factors of a matrix. The method is significant from a computational perspective because it shows a stronger capacity for parallelization than other ILU algorithms, and it does so without requiring the reordering of the matrix in question. Chow and Patel note that the algorithm is not intended to be used on very large sparse matrices, but rather on subdomains of a larger matrix that may be distributed across nodes of an HPC system. Therefore, parallelization is done within a shared memory framework using OpenMP. The PILU algorithm shows good speedup up to 56 threads, which is the number of cores on a node of the Frontera HPC system where this implementation was tested.

2 ILU Iterative Algorithm

Chow and Patel's methodology for deriving the ILU iterative algorithm will not be described in full detail here (refer to sections 2.1 and 2.2 of [1] for the full discussion), but the mathematical result will be presented before the discussion of how it can be implemented in code. The calculation of ILU factors L and U of matrix A can be formulated with the following equations:

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \tag{1}$$

*University of Texas at Austin, Department of Aerospace Engineering and Engineering Mechanics. Report developed as part of a final course project for Parallel Computing for Science and Engineering.

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \tag{2}$$

Equation 1 is valid for $i > j$, while Equation 2 is valid for $i \leq j$. Both equations are computed only over the desired sparsity structure for the L and U factors. To implement these equations as an iterative method, initial guesses are defined for L and U, and the above equations are computed for every entry in the sparsity structure; this computation process is called a "sweep." Repeated sweeps should refine the approximation to the L and U factors, although like any iterative method, there is a possibility that the algorithm will diverge.

2.1 Setup

The implementation of the ILU iterative method in code will now be discussed. The language chosen for this implementation was C++, due to the availability of the Eigen library which includes support for sparse linear algebra. Parallelization of computationally intensive loops was done with OpenMP. The source of the matrices used to test the PILU algorithm is the SuiteSparse Matrix Collection, which contains a large variety of matrices from applications in many different fields [2]. Matrices were downloaded from the SuiteSparse collection in MatrixMarket format, with input being handled by the Eigen library.

Only symmetric positive definite (SPD) matrices were used for the analysis in this report, although Chow and Patel's algorithm can be used for more general matrices. The algorithm assumes that the matrix A to be factorized has been scaled so that it has a unit diagonal. To achieve this, the C++ code computes a diagonal matrix D which has elements that are the square root of the diagonal elements of A; then the multiplication DAD scales the matrix A appropriately.

This implementation uses the "standard" initial guess for L and U. This means that the initial L is filled with the strictly lower triangular elements of A and a unit diagonal, and the initial U is filled with the upper triangular elements of A.

2.2 Main Loop

Before the main loop is discussed, there will be a brief description of the mechanism that the Eigen library uses to loop through the nonzeros of a sparse matrix. The following code shows this mechanism:

```
SparseMatrix<double> mat(rows,cols);
for (int k=0; k<mat.outerSize(); ++k)
  for (SparseMatrix<double>::InnerIterator it(mat,k); it; ++it)
  {
    //work goes here
  }
```

The default sparse storage format in Eigen is compressed sparse column (CSC), so `mat.outerSize()` is the number of columns in the matrix, and the outer `k` loop iterates through the columns. The `InnerIterator` is an object provided by Eigen, which in this case iterates the inner loop through the nonzeros in each column of the matrix. The iterator object includes accessor methods to obtain important information such as the value, row index, and column index of the current entry.

The following code shows the main loop of the PILU algorithm which implements equations 1 and 2:

```
#pragma omp parallel for schedule(dynamic)
for (int i=0; i<A.outerSize(); ++i)
    {
    for (Eigen::SparseMatrix<double>::InnerIterator it(A,i); it; ++it)
    {
        if (it.row() > it.col())
        {
            double div = 1.0/U.coeffRef(it.col(),it.col());
            SpMat in_prod_mat = L.block(it.row(),0,1,it.col()) * U.block(0,it.col(),it.col(),1);
            double in_prod = in_prod_mat.coeffRef(0,0);
            L.coeffRef(it.row(),it.col()) = (it.value() - in_prod) * div;
        }else{
            SpMat in_prod_mat = L.block(it.row(),0,1,it.row()) * U.block(0,it.col(),it.row(),1);
            double in_prod = in_prod_mat.coeffRef(0,0);
```

```
            U.coeffRef(it.row(),it.col()) = it.value() - in_prod;
        }
    }
}
```

This code is based on Algorithm 2 in [1]. The `SpMat` object is defined previously in the program as:

```
typedef Eigen::SparseMatrix<double> SpMat;
```

The first detail to note is that this implementation iterates through the sparsity pattern of A. This corresponds to an ILU "level 0" factorization, where the LU factors are only allowed to have fill-in within the original sparsity pattern of A. Since the procedure involves computing the dot products $\sum_{k=1}^{j-1} l_{ik}u_{kj}$ and $\sum_{k=1}^{i-1} l_{ik}u_{kj}$ between partial rows and columns of the LU factors, L is stored in compressed sparse row (CSR) format and U is stored in CSC format. Then, the dot products are computed by first using Eigen's `block()` method to extract the appropriate partial rows and columns and then multiplying them using the `*` operator, which is overloaded by Eigen to mean matrix-matrix multiplication (or matrix-vector multiplication, etc.). This algorithm is repeated by an additional outer loop (not shown in the code above) which controls how many sweeps are performed.

At first glance, it may seem that the iterative ILU algorithm is not a suitable candidate for parallelization because the iterations are not independent. For the inner product computation, a thread may have to use an element of L or U that has already been updated by another thread, since L and U are shared among threads. However, the usage of updated elements is a valid way to perform LU factorization iteratively, and therefore the dependency between iterations does not prevent the algorithm from being parallelized. Note that the algorithm is not expected to compute the same factorization in serial mode as in parallel, and it is not even expected to compute the same factorization in parallel between runs with the same number of threads.

The variables used to compute the inner products are local to each thread since they are declared inside the scope of a block that is within the parallel region. The same argument should hold to ensure that Eigen's `InnerIterator` object is local to each thread. This expected behavior was verified by iterating in parallel through a small sparse matrix, setting the nonzeros in each column to the column number, and finally printing out the results to confirm that the elements had the expected values.

2.3 Convergence Criteria

Although the algorithm is repeated for a specified number of sweeps rather than until convergence, there are nevertheless two convergence criteria that can be calculated to evaluate that the LU factors are indeed being refined. These criteria are defined by Chow and Patel as the "nonlinear residual norm" and the "ILU residual norm."

The nonlinear residual norm is defined by:

$$\sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{min(i,j)} l_{ik}u_{kj} \right| \tag{3}$$

where S is the sparsity pattern. The meaning of this norm is the degree to which the factorization meets the original constraints which were used to develop the ILU algorithm. Therefore, it is a measure of the convergence of the factorization, and should decrease towards zero as more and more sweeps are performed. Note that this process involves computing almost the same inner products as in the main loop, except that the entry corresponding to the current row and column is now also included (the summation is up to i or j instead of $i-1$ or $j-1$). The implementation of this norm in code uses a similar procedure as the main loop, and is included in the Code Appendix section for reference.

On the other hand, the ILU residual norm is defined by:

$$\|A - LU\|_F \tag{4}$$

where F denotes the Frobenius norm. The meaning of this residual norm is different from the meaning of the nonlinear residual norm. The ILU residual norm will not decrease to zero because the ILU factorization is by definition an approximation of the true LU, so there will always be a discrepancy between A and LU

due to the approximation. Instead, this residual norm is used to deduce the quality of the ILU factorization as a preconditioner, although Chow and Patel note that this correlation is only reliable for SPD matrices. When the ILU residual norm converges to a nonzero value, this means that the LU factorization is a good preconditioner and it will not improve much in this regard with further sweeps. The following code is used to compute the ILU residual norm, using Eigen's overloaded definitions of subtraction and multiplication:

```
resM = A - L*U;
ilu_res_norm = resM.norm();
```

3 Testing Parallelism in the ILU Algorithm

3.1 Summary

The main loop of the PILU is parallelized using OpenMP so that different threads compute entries of the L and U factors at the same time. In this section, results of scaling experiments will be reported to show how parallelism reduces the runtime of the algorithm. The input matrix for these experiments was the `Pres_Poisson` matrix from the SuiteSparse collection, which is an n by n SPD matrix with $n = 14,822$ and 715,804 nonzeros, and the algorithm uses the original sparsity structure of this input matrix to perform an ILU level 0 factorization. The HPC system used to run the experiments was the Frontera supercomputer at the Texas Advanced Computing Center. C++ source code was compiled using the Intel C++ compiler with flags `-qopenmp`, `-O2`, and `-xhost`.

3.2 Testing of Scheduling Types

Before running scaling experiments, tests of different scheduling types and chunk sizes for the main loop were run to determine the preferred configuration. It was expected that dynamic scheduling would perform better than static scheduling, since the amount of work per iteration of the main loop is not consistent; different rows and columns of a sparse matrix have different numbers of nonzeros. Table 1 shows that the best-performing schedule type was dynamic scheduling with a chunk size of 8, so this configuration was used for the scaling experiments.

Scheduling Type	Chunk Size	Average Time (s)
`static`	8	2.085
`static`	100	2.045
`static`	1024	2.019
`static`	4096	2.510
`dynamic`	8	1.787
`dynamic`	100	1.816
`dynamic`	1024	2.009
`dynamic`	4096	2.530
`guided`	8	1.810
`guided`	100	1.860
`guided`	1024	1.982
`guided`	4096	2.513

Table 1: Testing of different scheduling types for the main algorithm loop with 4 threads

3.3 Scaling Analysis

Table 2 shows a summary of results for a strong scaling analysis up to 56 threads for the main loop of the PILU algorithm. Each run performed 3 sweeps of the iterative method, and timings were averaged through the sweeps for each run to generate the third column of Table 2. Note that if higher levels of ILU were used and more elements of L and U were allowed to be filled in with successive sweeps, timings could not be

done in this same way because the amount of work would not be comparable between sweeps. In that case, timings would have to be averaged between separate runs, and not between sweeps within a single program run.

Figure 1 shows the average timings vs. thread count, and Figure 2 shows the speedup. The PILU algorithm has very good parallel speedup; although the speedup starts to deviate from the ideal at around 16 threads, it still achieves a substantial 40x increase for 56 threads. These results show that the algorithm has a strong capacity for parallelism and makes good use of the full amount of cores in a compute node.

Threads	Minimum Time (s)	Average Time (s)
1	6.5170	6.5174
2	3.5347	3.5487
4	1.8295	1.8406
8	0.9464	0.9493
16	0.4816	0.4887
32	0.2561	0.2656
56	0.1628	0.1636

Table 2: Strong scaling experiment for main algorithm loop

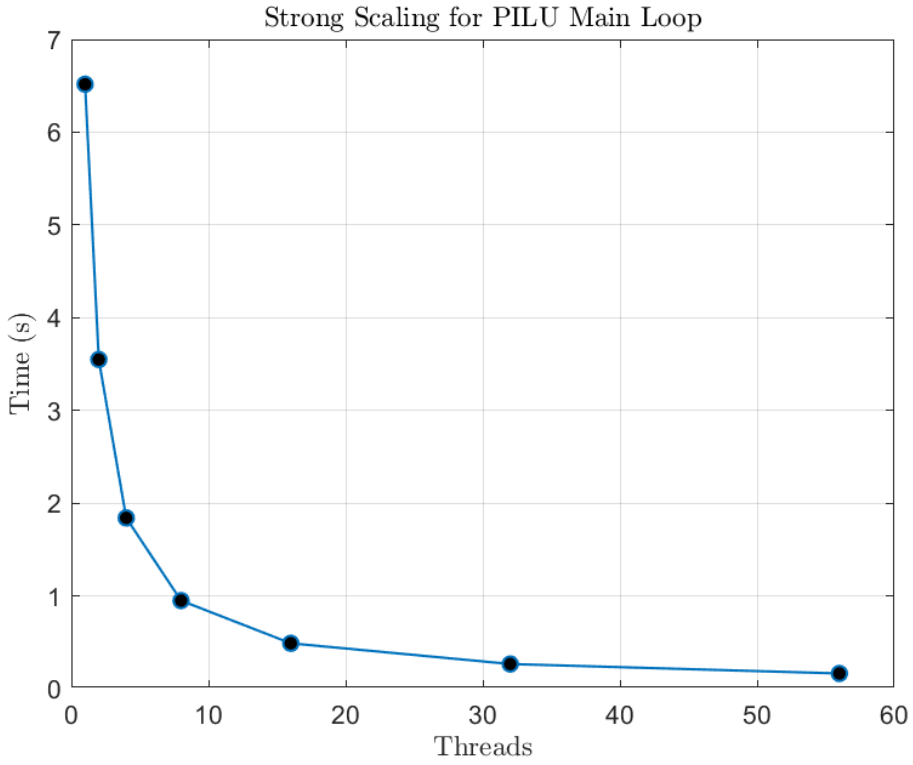

Figure 1: Strong scaling experiment for main loop of PILU algorithm up to 56 OMP threads

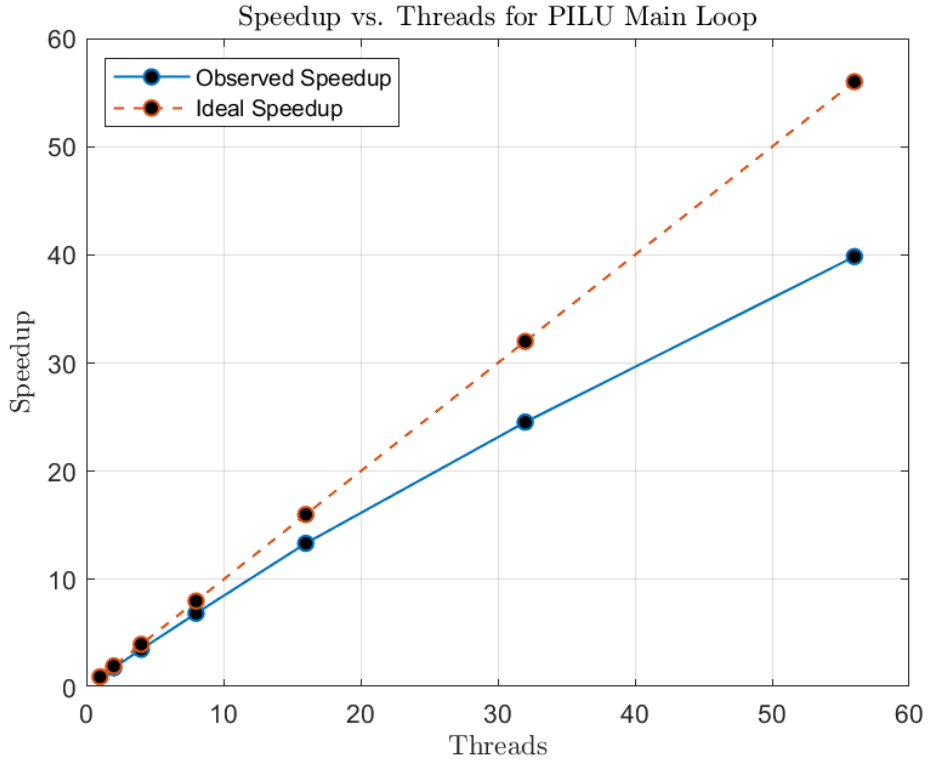

Figure 2: Speedup for main loop of PILU algorithm up to 56 OMP threads

3.4 Analysis of Residual Norms

Table 3 illustrates the expected behavior of the residual norms described in Section 2.3 for a run with 8 threads. As the number of sweeps increases, the nonlinear residual norm tends to zero, indicating the convergence of the iterative factorization algorithm. The ILU residual converges to around 11.4 by the third sweep, which means that the ILU factorization does not improve further as a preconditioner after three sweeps. This behavior shows why it is not necessary to perform the ILU algorithm until convergence of the nonlinear residual norm, if ILU factorization will be used as a preconditioner. After the third sweep, the nonlinear residual is still around 4.3, which means the iterative method has not converged yet; however, since the ILU residual has converged, further sweeps are not useful.

Sweeps	Nonlinear Residual	ILU Residual
0	3.882e+4	94.3256
1	1.525e+3	13.9613
2	7.509e+1	11.4247
3	4.295e+0	11.4054
4	0.279e+0	11.4055
5	0.018e+0	11.4055

Table 3: Convergence of residual norms for the PILU algorithm using 8 threads

If a higher level of ILU factorization is computed (more elements of L and U are allowed to fill in), then the ILU residual norm will converge to a smaller nonzero number, because the factorization will be a better approximation to the true LU factors.

4 Conclusion and Further Work

The main advantages of the PILU algorithm are its relatively simple implementation (no matrix reordering required) and its strong potential for parallelism, which was shown by the scaling analysis in this report. It is important to note that since the factorization computed by the PILU algorithm is meant to be used as a preconditioner, a discussion of PILU's benefits is not complete without considering the solution of the linear system that the computed ILU will act as a preconditioner for. In practice, Chow and Patel note that using an ILU factorization as a preconditioner for the solution of a system of equations requires the solution of sparse triangular systems. Therefore, for the PILU algorithm to be truly worthwhile in a broader context, the later triangular solves also need to be parallelized efficiently; otherwise, the factorization will be computed quickly with PILU but the triangular solves will be much slower. If further research can be done to make these triangular solves fast, then the PILU method may have great potential in the future as a good choice for precoditioning, helping to solve important problems in science and engineering more efficiently.

There is some additional work that could be done to this implementation of PILU to make it more efficient. Firstly, it may be possible to compute the inner products in the main loop as blocks of several partial rows and columns rather than one row and column at a time; in this way, fewer loop itrerations would be necessary, and the algorithm could be faster. Also, there are some intrinsic inefficiencies with the way some computations are performed with Eigen's overloading of common operators. For instance, for the ILU residual norm computation, the code first computes `A - L*U`. In using this syntax, there is an unnecessary amount of deallocation and allocation of intermediate sparse matrix objects. If methods are used that avoid this deallocation and allocation, the program would be more efficient from a runtime and memory perspective.

5 Code Appendix

Code implementation for computation of nonlinear residual norm:

```
#pragma omp parallel for reduction(+:nonl_res_norm) schedule(dynamic)
for (int i=0; i<A.outerSize(); ++i)
{
    for (Eigen::SparseMatrix<double>::InnerIterator it(A,i); it; ++it)
    {
        if (it.row() > it.col())
        {
            SpMat in_prod_mat = L.block(it.row(),0,1,it.col()+1) * U.block(0,it.col(),it.col()+1,1);
            double in_prod = in_prod_mat.coeffRef(0,0);
            nonl_res_norm = nonl_res_norm + abs(it.value() - in_prod);
        }else{
            SpMat in_prod_mat = L.block(it.row(),0,1,it.row()+1) * U.block(0,it.col(),it.row()+1,1);
            double in_prod = in_prod_mat.coeffRef(0,0);
            nonl_res_norm = nonl_res_norm + abs(it.value() - in_prod);
        }
    }
}
```

References

[1] Chow, E. and Patel, A. (2015). "Fine-Grained Parallel Incomplete LU Factorization." *SIAM J. Sci. Comput.*, Vol. 37, No. 2, pp. C169-C193.

[2] Davis, T. and Hu, Y. (2011). "The University of Florida Sparse Matrix Collection." *ACM Transactions on Mathematical Software 38*, 1, Article 1 (December 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663