SPARSE TRIANGULAR SOLVES FOR ILU REVISITED: DATA LAYOUT CRUCIAL TO BETTER PERFORMANCE

BARRY SMITH[*] AND HONG ZHANG[†]

[*]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4844, USA (bsmith@mcs.anl.gov).

[†]Computer Science Department, Illinois Institute of Technology, 10 West 31st Street, Chicago, IL 60616, USA (hzhang@mcs.anl.gov).

Sparse Triangular Solves for ILU Revisited

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439-4844, USA
Phone: 630-252-9174
Fax: 630-252-5986
Email: `bsmith@mcs.anl.gov`

Hong Zhang (corresponding author)
Computer Science Department
Illinois Institute of Technology
10 West 31st Street
Chicago, IL 60616, USA
Phone: 630-252-3978
Fax: 630-252-5986
Email: `hzhang@mcs.anl.gov`

Abstract. A key to good processor utilization for sparse matrix computations is storing the data in the format that is most conducive to fast access by the memory system. In particular, for sparse matrix triangular solves the traditional compressed sparse matrix format is poor, and minor adjustments to the data structure can increase the processor utilization dramatically. Such adjustments involve storing the L and U factors separately and storing the U rows "backwards" so that they are accessed in a simple streaming fashion during the triangular solves. Changes to the PETSc libraries to use this modified storage format resulted in over twice the floating-point rate for some matrices. This improvement can be accounted for by a decrease in the cache misses and TLB (transaction lookaside buffer) misses in the modified code.

Key words. sparse triangular solve, ILU-factorization, matrix-vector product, data access pattern, data layout

1. Introduction.

Many of the basic computational kernels in numeric software libraries were developed and implemented based on decades-old algorithms and techniques without serious consideration of computer architecture, for example, the complex memory layout and data-fetching behavior. Traditionally, numerical algorithms and the associated programming subroutines are evaluated based on such factors as the mathematical error analysis, the rate of algorithmic convergence, and the floating point (flop) counts. Hence many applications fail to achieve the anticipated speedup because of a mismatch between the data access patterns in the code and the data access patterns that are fastest on the given hardware.

As PETSc developers [2, 3], we have long been aware of the various memory bottlenecks in sparse matrix computation. We feel strongly that the *data access pattern* should become a standard in the evaluation of numerical algorithms and their implementations. Recently, we explored this concept on a computational kernel in PETSc: the *sparse triangular solve.* **In particular, our focus is on *sparse triangular solves* that arise from incomplete LU factorizations, where the cost of the factorization is relatively low and there are many *sparse triangular solves*, hence decreasing the time of the *sparse triangular solves* is crucial for fast solvers.** Through a simple reorganization of the data structure during the matrix factorization, we witnessed over 100 percent acceleration **(that is, more than a doubling of the flop rate)** in the *sparse triangular solve* on a single core as a result of much better utilization of the memory subsystem. We are not changing how much data is accessed, nor are we changing the numerical algorithm. We are changing only the locations where the data is stored **and the accessing pattern** so that **the data** accessing is as fast as possible. Essentially, we are decreasing the number of cache misses, **optimizing the** cache line usage (rather than having parts of cache lines not needed in the next step of computation), and reducing the number of TLB (transaction lookaside buffer) misses. The TLB is the mechanism used to map virtual memory addresses to their physical address that is then used by the hardware to load the data from memory. Since the TLB is generally small, addressing virtual memory addresses that are "near" each other is faster than randomly accessing very different virtual memory addresses.

The compressed sparse row (CSR) format is the most commonly used sparse matrix storage format. For sparse matrices with no additional structure (small dense blocks or values along certain diagonals, for example) the CSR format is appropriate for the sparse matrix-vector product kernel. Many sparse matrix software packages provide LU and/or ILU factorization and triangular solver kernels, **which are implemented by using some simple variant of the CSR format without considering the data accessing pattern employed in the successive triangular solves. For example, the Euclid [13, 14], SPARSKIT ILUT [17], and the PETSc**

version 3[2] ILU implementations store the L and U factors interlaced by row, that is, as $[L(1,:), U(1,:), L(2,:), U(2,:), ..., L(n,:), U(n,:)]$. AztecOO with IFPACK [19] and hypre's pilut [5, 8] store the L and U separately, but still have the rows starting with the first $[L(1,:), L(2,:), ..., L(n,:)]$ and $[U(1,:), U(2,:), ..., U(n,:)]$ in the order of matrix factorization. The new ILU factorization code in SuperLU [15] uses a more sophisticated storage for L and U, but the data layout still requires "jumping around" the data during the *sparse triangular solves* that could perhaps be optimized.

All these variants are poor for system utilization because they result in slow memory access patterns on the back triangular solves. Hence the triangular solves as implemented have much lower floating-point rates than do their corresponding matrix-vector products. By a simple change in the data layout we show that one can bring the floating-point rate of the triangular solves closer to that of the sparse *matrix-vector products.*

In this paper we measure the efficiency of the triangular solves by their flop rates. High flop rates are not the ultimate goal; faster time to solution is the ultimate goal. But since for triangular solves the number of floating point operations remains the same, the flop rate is a good measure of the overall utilization of the compute node. Sparse matrix computations are always memory bandwidth limited. That is, some upper bound on the speed of computation is determined by the raw speed at which the memory can provide data for the process; see, for example, [1, 9, 10]. In this paper the focus is on how to get a particular computation closer to the memory bandwidth limit by taking into account other aspects of the memory system than simple raw bandwidth.

2. Modifications to the Data Structure.

The *sparse triangular solve* and *matrix-vector product* are the dominating computational kernels in many large-scale iterative solvers. When an application uses Krylov subspace iteration with matrix-based preconditioners, for example, the incomplete LU (ILU) preconditioner, the *matrix-vector product* and *sparse triangular solve* are called repeatedly for generating Krylov subspaces; they often consume 70% or more of the total execution time.

For a sparse matrix A stored in the conventional compressed row format, a practical way of implementing the ILU preconditioner is described in [18].

Algorithm *sparse LU factor*:
Input: sparse matrix A
Output: sparse matrix factors L and U

```
For $i = 1, 2, \cdots, n$ Do:
    w := A(i, :)
    For $k = 1, 2, \cdots, i-1$ Do:
        $multiplier := A(i,k)/A(k,k)$
        update $w := w - multiplier * U(k,:)$
    EndDo
    store $w$ in $L(i,:)$ and $U(i,:)$
EndDo
```

For the ILU algorithm, the arithmetic operations described above occur only in the matrix entries that fall into a given nonzero sparse pattern. **A straight forward** implementation of the *sparse LU factor* algorithm uses the standard compressed row format for the input matrix A. The entries of the output matrix factors L and U are naturally stored together in a single array with their rows interlaced:

$$[L(1,:), U(1,:), L(2,:), U(2,:), ..., L(n,:), U(n,:)]. \tag{2.1}$$

The *sparse triangular solve* then follows the numerical factorization.

Algorithm *sparse triangular solve*:
Input: matrix factor L and U, right-hand side vector b
Output: solution vector x such that LUx approximates b

```
# forward substitution
For $i = 1, 2, \cdots, n$ Do
    $x(i) := b(i) - L(i, 1 : i-1) * [x(1), \cdots, x(i-1)]^T$
EndDo

# backward substitution
For $i = n, n-1, \cdots, 1$ Do
    $x(i) := (x(i) - U(i, i+1 : n) * [x(i+1), \cdots, x(n)]^T)/U(i,i)$
EndDo
```

For each call of *sparse triangular solve*, the array of factor values in (2.1) is accessed twice: first, in the order of

$$L(1,:) \to L(2,:) \to \cdots \to L(n,:)$$

during the forward substitution, then in the order of

$$U(n,:) \to U(n-1,:) \to \cdots \to U(1,:)$$

during the backward substitution. Here, for simplicity we present the solves without row and column permutations.

The flop counts for the *sparse matrix-vector product* are equal to twice the number of nonzero entries in A, which is the same as the flop counts for the *sparse triangular solve* when matrices L and U are obtained from the ILU(0) matrix factorization. However, because of the difference in the data layout of the matrix entries in the original matrix A and the matrix factors L and U, that is, **linear, contiguous array access** in A and non-contiguous array access in L and U, the execution time of a *sparse triangular solve* usually takes twice as long as that of the *sparse matrix-vector product* **(see Tables 3.1 - 3.5)**. The delay in the *sparse triangular solve* becomes more significant for larger matrices.

A careful examination of the *sparse triangular solve* algorithm reveals that the memory access can be improved through a simple reorganization of the matrix entries in the factored matrix L and U. Instead of storing the rows of L interlaced with U's in the order being computed from the *sparse LU factor*, we arrange them in the **order of accessing** by the *sparse triangular solve.* Therefore the matrix entries are stored contiguously as

$$[L(1,:), L(2,:), ..., L(n,:), U(n,:), ..., U(2,:), U(1,:)]. \tag{2.2}$$

With this data layout, the *sparse triangular solve* reads the array of factor values, in a linear fashion, (2.2) only once, from beginning to end, in comparison with two sweeps of (2.1) as in the previous implementation.

The idea of reorganizing matrix data is simple. Its implementation is simple, too: in the subroutine *sparse LU factor*, we store U entries from the end of the array of factor values instead of next to L's, and we modify the values of the row and diagonal pointers accordingly. The existing subroutine *sparse triangular solve* requires only trivial editing that ensures the correct rows of L and U are accessed during the forward and backward substitution. Yet, the numerical experiments show an amazing acceleration: up to a **50** percent reduction in execution time.

Two important variants of the basic sparse triangular solves are as follows:

- Row and column permutations that are used to reduce fill in the factors or improve the convergence of the iterative method.
- Point-block storage of the factored matrix where the matrix has "natural" small blocks that it inherits from the continuous problem. For example, the fully implicit discretization of the Euler equations leads to sparse matrices with 5 by 5 dense blocks. In this case the CSR format is modified so only a single column index is needed to indicate the block column of the entire block. Computations using this block CSR format are faster because fewer loads of the column indices are needed.

Both these extensions can handle the modified storage proposed and benefit from it in the same manner.

3. Numerical Results.

We tested the new data layout on two extreme cases – the traditional 7-point stencil on a unit cube and a matrix arising from the discretization of the compressible Euler equations – in order to measure the effect of the modified data structure on both extremely sparse matrices and those arising in applications with more nonzeros per row.

The experiments were conducted on a MacBook Pro with 2.8 GHz Intel Core 2 Duo and 1.067 GHz DDR3 memory using one core. This system has 6 megabytes of cache and a cache line of 64 bytes. The STREAMS TRIAD benchmark performance [16] is 4.97 gigabytes per second of achieved memory bandwidth with dynamically allocated memory and a slightly higher 5.06 gigabytes per second with statically allocated memory. ILU(0) with no row and column permutations was used for both CSR and block CSR formats. We are currently modifying all of the PETSc ILU solvers to use the new format and have obtained similar performance improvements with those as well. Our performance results were obtained by running the entire GMRES algorithm and profiling the relevant *matrix-vector products* and *sparse triangular solves.* We have done so because standalone benchmarking often produces unreasonably optimistic performance projections since much of the data is already in cache, whereas in the actual application it will not be. The numerical results were obtained by making multiple runs and using the average value; all runs had results within 5 percent of the average.

The floating-point rates were obtained by using the Intel ICC compiler version 10.1 with -O3 optimization. The cache misses and TLB misses were obtained by using the Apple profiling package Shark, Apple's gcc version 4.0.1 with the -O3 option, and the Intel hardware counters L2_CACHE_LINE_IN.DEMAND and DTLB_MISSES.

We considered two cases:

- **Extremely Sparse Matrix**, created with the 7-point stencil finite-difference scheme on a 65 by 65 by 65 cube. It ran for 46 iterations. The performance results are given in Table 3.1.
- **Block Sparse Matrix**, the Jacobian matrix obtained from a fully implicit compressible Euler code on a mapped C-H mesh [11, 12]. It has a natural block size of 5. For this matrix we ran two studies. The first, where we use the traditional compressed sparse row format, is given in Table 3.2. The

TABLE 3.1
Performance Improvements for the Extremely Sparse Matrix

	Matrix-Vector Product (SpMV)	Triangular Solves	
		Non-contiguous Array	Contiguous Array
Flop rate (megaflops)	537	261 (49% of **SpMV**)	447 (83% of SpMV)
L2 cache misses (1,000,000)	19.6	33.6	20.4
TLB misses (1,000)	635	$1,300$	750

TABLE 3.2
Performance Improvements for the Block Sparse Matrix

	Matrix-Vector Product (SpMV)	Triangular Solves	
		Non-contiguous Array	Contiguous Array
Flop rate (megaflops)	620	260 (42% of SpMV)	660 (106% of SpMV)
L2 cache misses (1,000,000)	12.6	17.5	13
TLB misses (1,000)	500	950	500

second, that uses the block compressed sparse row format, is given in Table 3.4.

TABLE 3.3
Performance Improvements of Triangular Solver With Intermediate Forms of the Implementation

	Original Implementation	Separate L and U	Traverse Rows of U Backward	New Implementation
Extremely Sparse	260	428	428	447
Block Sparse	260	387	645	660

We next consider the question of how much of the performance improvement comes from separating the storage of L and U and how much comes from storing rows of U in reversed order and thus allowing linear, contiguous array access. Table 3.3 shows the mega-flop rate for

- ***Original Implementation:*** **The rows of L and U are stored interlaced. Each row is accessed from the beginning to end.**
- ***Separate L and U:*** **The L and U are stored separately as**

$$[L(1,:), L(2,:), ..., L(n,:), U(1,:), ..., U(n-1,:), U(n,:)].$$

  The rows of U are accessed as in the original implementation.
- ***Traverse rows of U backward:*** **The L and U are stored separately as above. Each row of U is accessed from end to beginning, instead of beginning to end.**
- ***New Implementation:*** **The L and U are stored separately with the rows of U stored backward as (2.2), which are then accessed in a linear pattern from beginning to end during triangular solve.**

This computation is run on the same system as those in Tables 3.1 and 3.2. For the very sparse matrix most of the benefit comes from separating the L and U factors, no benefit comes traversing each row of U backwards and a small benefit comes from storing the rows of U backwards. We note

that the rows of U in the very sparse matrix case are so small that they almost always fit within a single cache line. For the more representative sparse block sparse matrix, the largest benefit comes from either traversing the rows of U backwards or, slightly better, reversing the order of the rows of U and traversing them forward.

Table 3.4
Performance Improvements for Block Sparse Matrix Using Block Compressed Sparse Row Storage

	Matrix-Vector Product (SpMV)	Triangular Solves	
		Non-contiguous Array	Contiguous Array
Flop rate (megaflops)	890	468 (53% of SpMV)	758 (85% of SpMV)
L2 cache misses (1,000,000)	7.5	9.3	7.2
TLB misses (1,000)	350	610	330

In Table 3.5 we provide the flop rates for the three studies in the previous tables for the IBM Blue Gene/P core, which is a PowerPC 450 running at 850 MHz with memory running at 425 MHz.

We also selected 25 square matrices from the University of Florida's sparse matrix collection [6] with a minimum of 200,000 rows. The smallest improvement in the flop rate was 30 percent and the largest 148 percent over all the matrices. The average improvement was 98 percent (essentially twice as fast), with 64 percent of the matrices at least doubling in floating-point performance of the triangular solves. These computations were all computed on the Intel system.

Table 3.5
Flop Rate (in megaflops) Performance Improvements for the IBM Blue Gene/P Core

	Matrix-Vector Product (SpMV)	Triangular Solves	
		Non-contiguous Array	Contiguous Array
7-point stencil	98	46 (47% of SpMV)	62 (63% of SpMV)
Euler	148	99 (66% of SpMV)	126 (85% of SpMV)
Euler with block CSR	303	198 (65% of SpMV)	260 (86% of SpMV)

In all cases, using linear contiguous array access in the *sparse triangular solve* accelerates execution time up to 100 percent (that is, twice as fast) in comparison with the non-contiguous array storage of matrix factors. The newly proposed **linear, contiguous data access** makes the *sparse triangular solve* almost as fast as the *sparse matrix-vector product.* **In our experiments, the time required for the sparse ILU factorization with the newly proposed storage of L and U remains almost same as the original implementation,** so this is not a matter of simply moving the computation time from one part of the code to another.

For the first experiment we now analyze the obtained flop rates and L2 cache misses.

Flop rates: The *matrix-vector product* requires loading the double-precision matrix value as well as the integer column index for each multiply-add performed. This amounts to requiring loading 6 bytes per flop. A more careful analysis that includes the vector that must be loaded and the fact that this matrix has 7 nonzeros per row gives a more precise value of 6.57 bytes per flop. Dividing the STREAMS achieved memory bandwidth of 4.97 gigabytes/second by 6.57 bytes/flop gives a bound on the achievable floating point rate of 756 megaflops. From Table 3.1 the actual achieved value is 537 megaflops. For the *sparse triangular solves*, each row of the interlaced L and U factors is of size 7, which means that in the lower triangular solves, though only the L portions are needed since the cache line contains 8 double-precision numbers, the entire U is also loaded to the L2 cache. This situation will also hold for the upper triangular solve. Hence the triangular solve as originally implemented loads all the numerical values as well as most of the column indices twice, compared to once for the *matrix-vector product*. Not surprising, the achieved flop rate for the original triangular solves for this matrix is essentially half of the flop rate for the *matrix-vector product*.

L2 cache misses: The number of cache misses can be estimated by the total amount of data loaded divided by the cache size minus some correction factor that takes into account prefetching and any other hardware optimizations [4]. We made two sets of runs with prefetching enabled and disabled. Performance was identical, indicating that prefetching was not being utilized in this code. Each *matrix-vector product* for this matrix requires loads of $7n * 12$ bytes for the matrix entries and column indices plus $n * 8$ bytes for the vector. Multiplying this by the 46 calls to the product routine and then dividing by 64 bytes per cache line gives a lower bound on the number of cache misses of 18 million. The actual count in Table 3.1 is 19.6 million. For the *sparse triangular solve*, making the pessimistic assumption that the U values loaded during the lower triangular solve are not available in the cache for the upper triangular solves, one would expect 35 million cache misses. In fact, since the cache is 6 megabytes, many of the last rows of U will be retained in the cache, and the actual number of cache misses was a slightly lower 33.6 million. Note that in Table 3.2 where the matrix has many more nonzeros per row, the excessive number of cache misses needed by the old *sparse triangular solve* is a lower percentage than in Table 3.1 because only a portion of the U factors are brought into L2 cache during the lower triangular solve. Storing the L and U separately and the U backwards eliminates nearly all the unused portions of the cache line so that the new *sparse triangular solve* requires only a very small percentage more cache misses than the *matrix-vector product*, regardless of the number of nonzeros per row. The TLB misses exhibit a similar reduction as the cache misses do with the new data layout for similar reasons since the data is now being accessed sequentially.

4. Conclusion.

We have presented a case study in sparse matrix computations where a small change to the data structure for a sparse matrix results in a dramatic improvement in the performance of a computational kernel that uses the data structure. We note that for ILU factorizations, the factorization comes first, and traditionally that has dictated the data layout of the L and U factors. The factorization routine loads the factor values, in a natural way, from beginning to end. But this means that the solve routines access the values "backwards." **The new data structure maintains the same run time of the ILU factorization and accelerates the triangular solve significantly. This emphasizes the importance of picking a data structure based not on how it is created first but rather on how it will be accessed efficiently and frequently.**

We plan to study data structures for other sparse matrix computations such as successive overrelaxation and Eisenstat's trick [7] to improve the performance of those computations as well.

5. Acknowledgments. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under

Contract DE-AC02-06CH11357.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (”Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

REFERENCES

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of SC 99*, 1999. Winner of Gordon Bell Special Prize at SC1999.

[2] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2009.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[4] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur. Predicting memory-access cost based on data-access patterns. In *Proceedings of IEEE International Conference on Cluster Computing, San Diego*, 2004.

[5] E. Chow, A. Cleary, and R. Falgout. Design of the hypre preconditioner library. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM, 1999.

[6] T. Davis. The University of Florida sparse matrix collection. Technical report, University of Florida, 1997.

[7] S. Eisenstat. Efficient implementation of a class of CG methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[8] R. Falgout. hypre users manual. Technical Report Revision 2.0.0, Lawrence Livermore National Laboratory, 2006.

[9] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. High performance parallel implicit CFD. *Journal of Parallel Computing*, 27:337–362, 2001.

[10] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Latency, bandwidth, and concurrent issue limitations in high-performance CFD. In *Proceedings of the First MIT Conference on Computational Fluid and Solid Mechanics*, June 2001.

[11] William D. Gropp, David E. Keyes, Lois Curfman McInnes, and M. D. Tidriri. Parallel implicit PDE computations: Algorithms and software. In *Proceedings of Parallel CFD'97*, pages 333–344. Elsevier, 1998.

[12] William D. Gropp, David E. Keyes, Lois Curfman McInnes, and M. D. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. *Int. J. High Performance Computing Applications*, 14:102–136, 2000.

[13] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215.

[14] D. Hysom and A. Pothen. Euclid user manual (a scalable ILU preconditioning library for the parallel solution of sparse linear systems). Technical report, Old Dominion University, 2001.

[15] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[16] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. http://www.cs.virginia.edu/stream.

[17] Youcef Saad. SPARSKIT, a basic tool kit for sparse matrix computations. Technical Report 1029, Center for Supercomputing Research and Development, University of Illinois at Urbana-Chanpaign, 1990.

[18] Yousef Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, Philadelpha,

PA, 2003.

[19] Marzio Sala and Michael Heroux. Robust algebraic preconditioners using IFPACK 3.0. Technical Report SAND2005-0662, Sandia National Laboratories, 2005.