Using Random Butterfly Transformations to Avoid Pivoting in Sparse Direct Methods

Marc Baboulin[1], Xiaoye S. Li[2] and François-Henry Rouet[2]

[1] University of Paris-Sud, Inria Saclay, France
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract. We consider the solution of sparse linear systems using direct methods via LU factorization. Unless the matrix is positive definite, numerical pivoting is usually needed to ensure stability, which is costly to implement especially in the sparse case. The Random Butterfly Transformations (RBT) technique provides an alternative to pivoting and is easily parallelizable. The RBT transforms the original matrix into another one that can be factorized without pivoting with probability one. This approach has been successful for dense matrices; in this work, we investigate the sparse case. In particular, we address the issue of fill-in in the transformed system.

1 Introduction

When solving the linear systems using the LU or the LDL^T factorizations, numerical pivoting is often needed to ensure stability. Pivoting prevents division by zero or by small quantities by permuting on the fly the rows and/or columns of the matrix so that the pivotal element is relatively large in magnitude. Pivoting involves irregular data movement and can significantly impact the speed of the factorization, especially on large parallel machines. This issue arises in both unsymmetric and symmetric cases, and for both dense and sparse factorizations. The ScaLAPACK [7], MAGMA [17] and PLASMA [14] dense linear algebra libraries contain a Cholesky factorization for positive definite matrices, for which no pivoting is required, but they do not contain an LDL^T factorization. They contain an LU factorization with partial pivoting (i.e. $PA = LU$, where P is a permutation matrix), but partial pivoting can significantly slow down the speed. For example, on a hybrid CPU/GPU system, the LU algorithm in the MAGMA library spends over 20% of the factorization time in pivoting even for a large random matrix of size $10,000 \times 10,000$.

Pivoting poses additional problems in sparse factorizations because of the *fill-in*, which corresponds to the new nonzeros generated in the factored matrices L and U. For sparse Cholesky, where pivots can be chosen on the diagonal, we often use a sparsity-preserving ordering algorithm, such as minimum degree or nested dissection, to reorder the matrix first so that the Cholesky factor of the permuted matrix PAP^T has less fill-in than that of A. For sparse LU, we often factorize PAQ^T with both row and column permutation matrices P and Q. The purpose is to preserve sparsity as well as to maintain numerical stability. There are complex interplays between ordering (for sparsity) and pivoting (for stability). Often, the two objectives cannot be well achieved simultaneously. Several relaxed pivoting schemes, other than partial pivoting, have been developed to trade off stability and sparsity, which allow larger pivot growth while maintaining better sparsity. These include *threshold pivoting* [8], *restricted pivoting* [16], and *static pivoting* [13].

One difficulty with dynamic pivoting, either partial pivoting or threshold pivoting, is that the fill-ins are produced on the fly depending on the permutation at each step. It is thus not possible to have the separate ordering and symbolic preprocessing algorithms that precisely minimize the number of fill-ins and forecast the fill-in positions. A good ordering strategy to accommodate dynamic row pivoting is to apply any ordering algorithm to the graph of the symmetrized matrix $A^T A$ which gives a fill-reducing permutation Q. Then, Q is applied to the columns of A *before* performing the LU factorization with row pivoting: $P(AQ^T) = LU$. The rationale behind this is that the nonzero structure of the Cholesky factor R of $A^T A = R^T R$ upper bounds the nonzero structures of L^T and U of $PA = LU$, for *any* row permutation P [12]. That is, the Cholesky factor R_q of $(AQ^T)^T(AQ) = R_q^T R_q$ upper bounds the L_q^T and U_q of $P(AQ^T) = L_q U_q$, and R_q contains smaller amount of fill than that of R. In essence, the column ordering Q tends to minimize an upper bound on the actual fill-ins in the LU factors, taking into account all the possible row pivotings. This strategy can be pessimistic when most pivots happen to be on the diagonal (e.g. diagonally dominant matrices). The sequential SuperLU library uses this ordering strategy together with partial pivoting [11]. This is our comparison baseline to be used in Section 3 about the numerical results.

The cost of dynamic pivoting in parallel is even more dramatic than in the dense case. For example, for matrix nlpkkt80 of a KKT system from nonlinear optimization, the parallel factorization with threshold pivoting using MUMPS [1] took 639 seconds with 128 processes. After the matrix is modified to be diagonally dominant with the same sparsity structure, the parallel factorization without pivoting took only 87 seconds, even though the size of the LU factors and the flop count are roughly the same in both cases.

In the parallel direct solver SuperLU_DIST [13], a static pivoting strategy is used to enhance scalability. Here, P is chosen *before* factorization based solely on the values of the original A. A maximum weighted matching algorithm and the code MC64 [9] is currently employed. The algorithm chooses P to maximize the magnitude of the diagonal entries of PA. During factorization, the pivots are chosen on the diagonal and the tiny ones are replaced by a fixed value. Since this does not involve dynamic row permutation, a sparsity-reducing algorithm can be applied to the graph of another symmetrized matrix $PA + (PA)^T$, producing the permutation matrix Q. This tends to minimize the amount of fill in the L and U of $Q(PA)Q^T = LU$. The static pivoting improves speed and scalability but it might fail for very challenging problems. MC64 is sequential in nature and there is no good parallel algorithm yet. Riedy [15] suggests a parallel auction algorithm but concludes that parallel performance is too unpredictable to make it a black-box tool. Therefore, the pre-pivoting phase will be a severe obstacle for solving larger problems on extreme-scale parallel machines.

In 1995, Parker introduced a randomization algorithm to *eliminate the need for pivoting* [10]. In this approach, the Random Butterfly Transformation (RBT) is used to transform the original system into an "easier" one such that, with probability one, the LU factorization of the transformed matrix can be performed without pivoting. This technique was successfully applied and implemented into the dense libraries for LU and LDL^T factorizations [2,6]. In this work, we investigate the potential of the RBT method for sparse cases.

2 Random Butterfly Transformations

In this section we recall the main concepts and definitions related to RBT where the randomization of the matrix is based on a technique initially described by Parker [10] and recently revisited by Baboulin et al. [2] for general dense systems. The procedure to solve $Ax = b$, where A is a general matrix, using a random transformation and the LU factorization is:

1. Compute $A_r = U^T AV$, with U, V random matrices,
2. Factorize $A_r = LU$ (without pivoting),
3. Solve $A_r y = U^T b$ and compute $x = Vy$.

The random matrices U and V are chosen among a particular class of matrices called *recursive butterfly matrices.* A *butterfly matrix* is an $n \times n$ matrix of the form

$$B^{<n>} = \frac{1}{\sqrt{2}} \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix}$$

where R_0 and R_1 are random diagonal $\frac{n}{2} \times \frac{n}{2}$ matrices. A *recursive butterfly matrix* of size n and depth d is defined recursively as

$$W^{<n,d>} = \begin{bmatrix} B_1^{<n/2^{d-1}>} & & \\ & \ddots & \\ & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{bmatrix} \cdot W^{<n,d-1>}, \text{ with } W^{<n,1>} = B^{<n>}$$

where the $B_i^{<n/2^{d-1}>}$ are butterflies of size $n/2^{d-1}$, and $B^{<n>}$ is a butterfly of size n.

In the original work by Parker, $d = \log_2 n$; he shows that, given two recursive butterfly matrices U and V, the matrix $U^T AV$, where A is the original matrix of the system to be solved, can be factored into LU without pivoting with probability 1 in exact arithmetic, or with probability $1 - O(2^{-t})$ using t-bit floating point numbers. For symmetric problems, $V = U$ and the same result holds with LDL^T. Baboulin et al. studied extensively the use of RBT for dense matrices and showed that in practice, $d = 1$ or 2 is enough; in most cases a few steps of iterative refinement can recover the digits that have been lost. They also showed that random butterfly matrices are cheap to store and to apply ($O(nd)$ and $O(dn^2)$ respectively) and they proposed implementations on hybrid multicore/GPU systems for the unsymmetric [2] case. For the symmetric case, they proposed a tiled algorithm for multicore architectures [3] and more recently a distributed solver [4] combined with a runtime system [5]. As was demonstrated, the preprocessing by RBT can be easily parallelized with good scalability.

3 Using RBT in sparse direct solvers

We first describe and compare different strategies and parameters when applying RBT to the sparse LU factorization. We carry out the experiments on a large set of sparse matrices in order to identify the best practical strategy.

3.1 Influence of the degree d

In the dense case, the use of RBT incurs small amount of extra operation and memory. The cost is limited to storing and applying RBT *prior to* the factorization. However, in the sparse case, applying RBT modifies the *nonzero structure* of the transformed matrix. The number of nonzeros in the transformed matrix $U^T AV$ can be up to 4^d times the number of nonzeros in A in the worst case. This increase in nonzeros may lead to an even larger increase in the size of the LU factors and thus to prohibitive costs. We therefore limit our investigation to small degrees: $d = 1$ or 2, which correspond to the practical setting used by Baboulin et al. in the dense case [2,3,4].

3.2 Combining RBT and fill-reducing permutations

Fill-reducing ordering is critical to preserve sparsity. This operation is usually performed after all the preprocessings that modify the sparsity pattern of the input matrix (e.g., MC64). At first glance, it seems that the most natural way of combining RBT with a fill-reducing permutation is:

1. transform the original matrix A into $U^T AV$,
2. permute with a fill-reducing algorithm (then factorize).

However, one can show that the matrix resulting from steps 1. and 2. is not guaranteed to be factorizable without pivoting. We provide an example here. Let A be a 4×4 matrix; A can be written in a 2×2 form as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Let U and V two recursive butterflies of size 4 and degree 2. By Parker's theorem, if A is non-singular then $U^T AV$ is factorizable without pivoting. Let p be the permutation vector $[1\ 3\ 2\ 4]$ and P the associated permutation matrix. We consider $B = PU^T AVP^T$. One can show that if $\sum A_{11} = \sum A_{22} = \sum A_{21} = \sum A_{12}$ then the leading submatrix $B_{1:2,1:2}$ is singular, regardless of the random values in U and V. Therefore B is not factorizable without pivoting (B_{22} becomes

0 after eliminating the first pivot, in the absence of roundoff errors). One can easily build a non-singular matrix satisfying this property, e.g., $A_{11} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$, $A_{12} = A_{21} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $A_{22} = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$ (leading to $\det(A) = -4$, i.e., A non-singular).

As a consequence, the strategy consisting in permuting for sparsity after the transformation may not work in theory, but we still wish to investigate its practical performance. We compare the following two strategies:

Strategy 1: the matrix is permuted using a fill-reducing (or bandwidth minimization) heuristic then transformed with RBT. This guarantees that the factorization would succeed for $d = \log_2 n$ but it might yield large number of fill-ins in the factors. The first step is an attempt to minimize the nonzeros in the transformed matrix and the fill-ins.

Strategy 2: the matrix is transformed with RBT then permuted using a fill-reducing heuristic. This might fail even for $d = \log_2 n$ but it provides a much better control of fill-in.

3.3 Evaluation of the different strategies and parameters

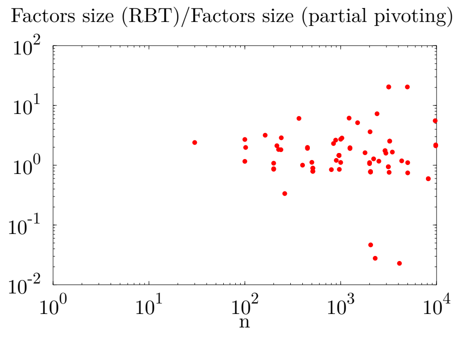
The experiments were carried out on 90 non-singular matrices with size $n \leq 10,000$. Table 1 shows the success rate of the factorization, the increase in nonzeros and the increase in the size of the LU factors with respect to partial pivoting. We use the partial pivoting code SuperLU [11]; for RBT, pivoting is disabled and the factorization is stopped whenever a zero diagonal pivot is found (although a possibility could be to replace it by a small perturbation such as $\varepsilon \|A\|$). The random values we use are $e^{r/10}$ with r randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$ from a uniform distribution. This guarantees a small condition number for U and V [2].

Strategy and degree		Success	Increase in nonzeros				Increase in factors			
		rate	min	geo	avg	max	min	geo	avg	max
Strategy 1	$d = 1$	81.1%	1.00	2.97	3.14	3.99	1.12	9.92	21.07	362.32
	$d = 2$	92.2%	2.01	9.53	10.52	15.79	1.14	19.35	45.41	635.84
Strategy 2	$d = 1$	82.2%	1.00	2.02	2.25	4.00	0.03	1.55	2.62	20.42
	$d = 2$	80.0%	1.50	4.95	5.98	15.01	0.06	2.96	6.78	144.49

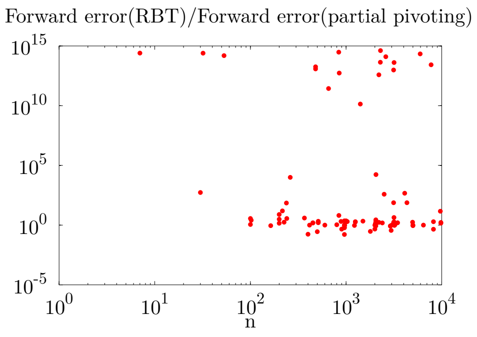
Table 1: Influence of the different strategies and parameters for 90 matrices with size $n \leq 10,000$. "Success rate" is the percentage of matrices for which the factorization completes. "Increase in nonzeros" is the ratio $nnz(U^T AV)/nnz(A)$ and "Increase in factors" is the ratio $nnz(LU(U^T AV))/nnz(LU(A))$; we report the minimum, geometric mean, arithmetic mean, and maximum.

We make the following observations: 1) Strategy 1 and Strategy 2 have similar success rates. Although both strategies lead to an increase in the size of the LU factors (with respect to partial pivoting), this increase is much more limited with Strategy 2. Therefore, Strategy 2 will be our method of choice. 2) Similar to what was observed in the dense case, most matrices succeed with $d = 1$. With Strategy 1, $d = 2$ yields a near-perfect success rate at the price of a large increase in the size of factors; the effect is less clear with Strategy 2.

Using Strategy 2 with $d = 1$ seems to be the most practical setting. Fig. 1 shows how this approach compares with partial pivoting. Fig. 1(a) shows how the size of the factors varies when RBT is used. 37 out of 90 matrices have a smaller size of LU factors; as explained in the introduction, this is due to the fact that partial pivoting relies on a fill-reducing permutation that can only aim at minimizing an upper bound of the fill-in, since the order in which variables are eliminated is not known in advance. On the other hand, not doing pivoting allows the fill-reducing permutation to focus on the right problem (minimizing the actual fill-in). For 30 matrices, the increase (due to the larger structure of the transformed matrix) is moderate (larger than one but less than two). Although this means that the number of operations with RBT might be larger than with partial pivoting, RBT may catch up since doing no pivoting yields better flop rate and scalability. For 23 matrices, the increase is large (between 2 and 20), which means it is more unlikely that RBT will yield better runtime. Fig. 1(b) shows the ratio between the forward error $||x - x_{\text{true}}||/||x_{\text{true}}||$ with RBT and that with partial pivoting. For 69 out of 90 problems, the ratio is less than 10^2 i.e. at most 2 digits are lost when using RBT instead of partial pivoting. The loss in accuracy found for some matrices is due to a larger growth factor with RBT, meaning that some elements found during the factorization become very large (relative to the elements in the matrix to be factored) and lead to inaccuracies. In most cases, a few steps of iterative refinement recover the lost digits. Overall, we found that 48 out of 90, i.e. 53.3% have both a moderate increase in the factors size (less than twice) and a moderate loss in accuracy (less than 2 digits).



(a) Difference in factors.



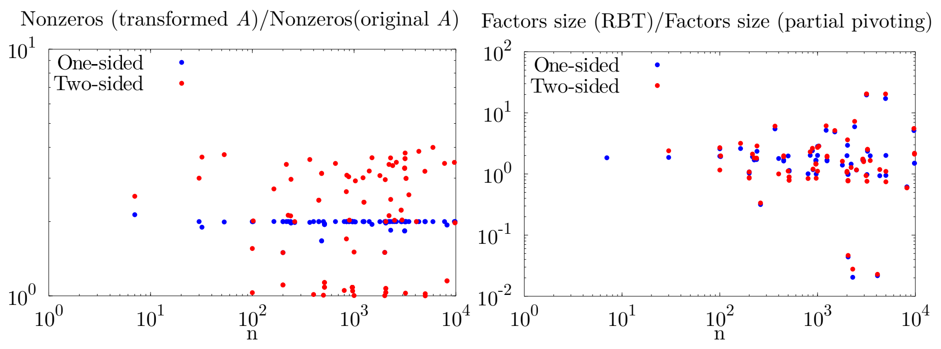
(b) Difference in forward error.

Fig. 1: RBT (Strategy 2, $d=1$) vs partial pivoting for 90 matrices sorted by size.

3.4 One-sided transformation

The original approach proposed by Parker relies on a two-sided transformation $U^T AV$. We showed that a one-sided transformation is sufficient to maintain the main numerical property, i.e., $U^T A$ can be factorized without pivoting when U is a recursive butterfly matrix with degree $d = \log_2 n$. The benefit is that the number of nonzeros in $U^T A$ (and the LU factor size) can be less than the number of nonzeros in $U^T AV$. Through private communication, Parker mentions that it is analogous to using partial pivoting rather than complete pivoting, i.e., although no zero pivot appears, the growth factor may be larger.

We experimented this one-sided approach, and found that, with $d = 1$, the success rate of the one-sided and two-sided approaches are similar. For $d = 2$, the success rate is marginally higher with the two-sided approach. Fig. 2 illustrates how the two approaches influence the size of the transformed matrix and the size of the factors. We observed that the one-sided approach marginally decreases the size of the factors on average, but the results are problem-dependent.



(a) Nonzeros. (b) Factors.

Fig. 2: One-sided vs two-sided (Strategy 2, $d{=}1$) for 90 matrices sorted by size.

4 Conclusion and perspectives

For sparse direct solvers using LU factorization, a serious scalability bottleneck is numerical pivoting. A number of relaxed pivoting algorithms have been developed, but none of them have shown promise of scalable implementation. In this exploratory work, through large number (90) of real-world test matrices, we demonstrated that the Random Butterfly Transformation is a good alternative to pivoting, especially with properly chosen ordering strategies and transformation parameters. RBT is particularly appealing for extreme-scale systems because it is highly parallelizable. This opens the possibilities of several avenues of new research, such as application of RBT to the LDL^T factorization, classification of the problems according to various RBT strategies, and investigation of RBT's impact on the scalability of existing parallel direct solvers.

Acknowledgement

We would like to thank Stott Parker for insightful discussions about the one-sided transformation. Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics). We used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Computing, 32(2):136–156, 2006.
2. M. Baboulin, J. J. Dongarra, J. Hermann, and S. Tomov, *Accelerating Linear System Solutions using Randomization Techniques*, ACM Transactions on Mathematical Software, 39(2), 2013.
3. M. Baboulin, D. Becker, and J. J. Dongarra, *A Parallel Tiled Solver for Dense Symmetric Indefinite Systems on Multicore Architectures*, Parallel & Distributed Processing Symposium (IPDPS), 2012.
4. M. Baboulin, D. Becker, G. Bosilca, A. Danalis, and J. J. Dongarra, *An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems*, Parallel Computing, to appear, 2014.
5. G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. J. Dongarra, *DAGuE: A generic distributed DAG engine for high performance computing*, Parallel Computing, 38(1&2):37–51, 2011.
6. D. Becker, M. Baboulin, and J. J. Dongarra, *Reducing the amount of pivoting in symmetric indefinite systems*, Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), 133–142, 2012.
7. L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, SIAM, 1997.
8. I. S. Duff and I. M. Erisman and J .K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, London, 1986.
9. I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Analysis and Applications, 20(4):889–901, 1999.
10. D. S. Parker, *Random Butterfly Transformations with Applications in Computational Linear Algebra*, Technical Report, UCLA, 1995.
11. J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *a Supernodal Approach to Sparse Partial Pivoting*, SIAM Journal on Matrix Analysis, and Applications, 20(3):720–755, 1999.
12. A. George and E. Ng, *Symbolic factorization for sparse Gaussian elimination with partial pivoting*, SIAM J. Sci. Stat. Comput., 8(6):877–898, 1987.
13. X. S. Li and J. W. Demmel, *SuperLU_DIST: a Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, ACM Transactions on Mathematical Software, 29(9):110–140, 2003.
14. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures*, Version 2.3, 2010. University of Tennessee.
15. E. J. Riedy, *Making Static Pivoting Scalable and Dependable*, Technical Report, UC Berkeley, EECS-2010-172, 2010.
16. O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Future Generation Computer Systems, (20):476–487, 2004.
17. S. Tomov and J. J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Computing, 36(5&6):232–240, 2010.