Hybrid Algorithms for Rank of Sparse Matrices

Supported by NSF grants 0098284 and 0112807

A. Duran, D. Saunders, and Z. Wan

1 Introduction

In this study we compare two distinct methods for solving basic linear algebra problems for sparse matrices over the integers and over finite fields. Here we focus on the problem of rank. One method is GSLU, an adaptation of the SuperLU method [3] and is fundamentally Gaussian elimination. The second method, called here BB for "black box", is a variant of Wiedemann's algorithm [15] and is a Krylov space method. The methods may be applied to other problems such as determinant, system solving, and Smith normal form. The observations made here apply quite directly to those problems.

The results of the computations are exact but are Monte Carlo, as discussed below. Exact methods are of interest when the matrix entries represent structural properties of some system rather than measured quantities. Frequently the matrices are incidence matrices of some kind. A 1 in the i, j position represents a relation of row object i to col object j. Blackbox methods benefit from special handling of zero-one matrices.

Our experiments provide a basis for hybrid algorithms and we present and evaluate two hybrids for use in our library LinBox[4], which may be found on the web at `linalg.org`. This library emphasizes the black box algorithms, but also uses elimination where appropriate. We have not pursued a comparison involving other elimination methods, but see [6]. It is clear that there is more to be gained by working with a variety of elimination techniques.

Portions of this work of this paper were presented at ECCAD '03[10] Specifically, the initial measurements of finite field arithmetic and basic performance of BB and GSLU algorithms.

2 The algorithms

The black box method for rank computation that we use is Wiedemann's method [15] with the preconditioning strategy of Eberly-Kaltofen [11], see also [1]. For a given matrix integer matrix A, a word size prime p is chosen and random vectore vectors $u, v \in \boldsymbol{Z}_p$ are used. The sequence $s_i = u^T A^i v$, is computed. The Berlekamp-Massey algorithm is used to determine the minimal polynomial of the sequence. With high probability this is the minimal polynomial of the matrix and, because of the preconditioning, the rank is directly determined from the degree of the minpoly and its constant term. See the above mentioned papers for details. If the trace of the preconditioned matrix is less than the prime and equals the second coefficient of the minimal polynomial it constitutes a certificate of the rank[13]. The early termination strategy is used so that just a few more than $2r$ terms of the sequence must be computed.

The run time of this algorithm, BB, is quite reliably computed a priori. Suppose the matrix has order n and nnz nonzero entries. Then matrix vector product costs nnz additions for a zero-one matrix and nnz multiply-adds in general. The algorithm uses $Theta(r)$ matrix vector products with the matrix A and $\Theta(n)$ additional work per sequence element (consisting of dot products, preconditioner (diagonal matrix) matrix-vector products, and the Belekamp-Massey step). More precisely, the version of the algorithm we tested uses $(2r + c)$ steps where r is the rank and c is a small constant, the early termination threshold. Each step uses 2 nnz $+ 4n$ arithmetic steps. On the machine used for these experiments and for the typical prime p, the arithmetic operations cost 3.7×10^{-7} seconds. Thus a conservative estimate of the BB cost for computing rank is $3.7 \times 10^{-7} \times (4n$ nnz $+ 8n^2)$.

GSLU (Generic SuperLU)[9] is adapted from SuperLU version 2.0 [3]. field arithmetic is written in the LinBoxstyle, where the field object is an explicit parameter to each operation along with the the field elements involved. This allows GSLU to be used with arbitrary fields including finite field representations from LinBoxand light wrappers on traditional floating point types (float, double. complex). The code uses C++ template parameters for the field.

SuperLU contains a set of subroutines to solve a sparse linear system $AX = B$. Consider the factorization $PAQ^T = LU$ of a sparse matrix A, using Sparse Gaussian elimination with partial pivoting, where the row ordering P is selected during factorization using standard partial pivoting and Q is a column permutation chosen with the goal of reducing fill-in. The partial pivoting is simplified slightly for finite fields, since there is no issue of numeric stability, and in any case size comparisons of field elements make no sense. One must select a column preordering, Q, so that the factorization remains as sparse as possible, regardless of choice of P. The column ordering can have dramatic impact on the number of nonzeros in L and U. SuperLU has four options for determining Q, which are: (1) MMD (Multiple elimination Minimum Degree) applied to the structure of $A^T A$, (2) MMD applied to the structure of $A + A^T$, (3) $COLAMD$, and (4)natural ($Q = I$).

MMD is a local minimization of nonzeros in the factored matrix. It is also a practical approximate solution to the NP-complete fill minimization problem. Liu[12] describes the method and gives a modification of the standard algorithm.

COLAMD (Column Approximate Minimum Degree Ordering Algorithm) [2] is based on symbolic LU factorization of the nonzero pattern of A. It is an improved version of Matlab's COLMMD. The former is faster and computes better orderings in general, with fewer nonzeros in the factors of the matrix. We found it to perform best for most of our non-symmetric examples. When it wasn't best, the MMD appied to $A^T A$ was. We report times for these two preorderings in our data below. There are cases where SuperLU has a memory problem and segmentation fault occurs. This remains true in GSLU. Also we found some cases where an erroneous rank occurs. We eliminated these matrices from our study, believing the bug fix will not likely affect the performance in the currently correct cases. Certainly clearing up these problems is desired. For the rank of an integer matrix, we choose to compute mod a word size prime. So the algorithm is Monte Carlo, with a high probability of success.

As with all elimination methods, the run time of GSLU is quite variable. It depends on the rate of fill-in, which in turn depends on the success of the fill-in avoidance method used and on the zero-nonzero pattern of the matrix. For the rank problem, elimination may stop at the r-th step. For various classes of sparse matrix, the run time varies from $\Theta(r)$ to $\Theta(rn^2)$. For example, if $\text{nnz} = 2n$ and there are exactly 2 entries per row, only $\Theta(r)$ operations are necessary. On the other hand for dense matrices and for matrix patterns in which there is rapid fill-in. the $\Theta(rn^2)$ run time is experienced. Also important for practical computation is that the overall memory requirement can vary from nnz to n^2 matrix entries, depending on fill-in.

It is very difficult to guess a-priori which method will run faster. Some generalities are that (1) BB is surperior for very large matrices, in particular when fill-in causes the matrix storage to exceed machine real memory, and that (2) GSLU is generally superior when nnz/n is very small, less than 3, say. Experience with some very large, very sparse matrices, for instance in [7], has lead one of us, Saunders, to provoke proponents of black box methods with the claim "When the matrix fits in real memory, throughout the LU computation, elimination beats black box." This is certainly not true on a sporadic basis, and this paper gives evidence that it is systematically not true for some families of matrix.

Given this uncertainty, one solution is to race the two methods, stopping the slower one when the faster one finishes. When two processors are available, the faster time is achieved. When only one (timesharing) processor is available, racing still runs in no more than twice the faster time. We implemented a general purpose racing utility and found negligible overhead. This sets the bar for any other attempt at a hybrid approach. Fortunately for the hybrid approach proposed in our last section, the GSLU algorithm is "left looking" which has the consequence that the cost of elimination steps tends to be an increasing function of step number. This gives us a chance to recognize rapid growth of step cost and switch to the BB method at a relatively early stage.

To start, let us benchmark the situation for dense matrices. Here $\text{nnz} = n^2$. Due to the non-symmetric projection used in this implementation, 2 matrix-vector products are computed for each of the $2r$ s_is, for $4n^2$ ops per s_i and $4rn^2$ ops overall. From an observation of Dumas, it is possible to reduce this by a factor of 2, [7].

In the fully dense case, GSLU takes about $(1/3)rn^2$ field ops. Thus we expect the ratio of BB time over GSLU time for dense matrices to be about 12.

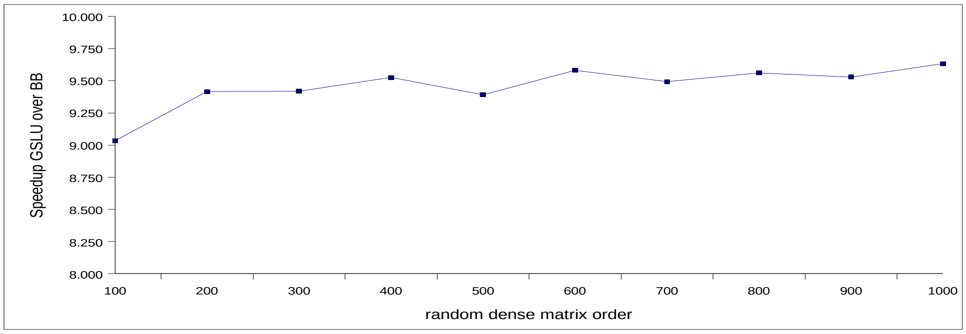

Figure 1. *comparison of black box and GSLU algorithms for dense matrices*

For random dense matrices, see figure 1, we found the actual ratio to be about 9.5. The deviation from the predicted 12 is not great and may be explained by the net faster field arithmetic for dot products versus the element by element arithmetic in row and col operations during elimination. We remark that the method of choice for dense matrices is a scheme for using floating point BLAS in an exact way, see [5]. The advantage is sufficiently great that it should also be used for sparse but small matrices.

All timings in this paper were taken on a Sun sparc running solaris, specifically a SUN4U/750 Sun-Fire V880 with 32GB main memory, SunOS 5.8.

3 The matrices chosen for experiments

Experimental measurements were done with several families of matrices and a few sporadic examples, mostly taken from the matrix market. Their properties are sketched here. Most are $0, 1$-matrices, but a few have a wider range of integers among the non-zero entries. For example the dense matrices used above had random entries in [1..100] so that full rank would result.

- The matrix `Tref[n]` is $n \times n$ with the first n primes on the diagonal and 1's wherever $|i - j|$ is a power of 2. `Tref20000` was the subject of one of Nick Trefethen's "Hundred dollar, hundred digit challenge" problems[14], and was the basis for a study of BB methods for determinant and system solving[8].
- The `TF[n]` matrices have the nonzero entres distributed near diagonal and are of almost full rank. `http://www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/Matrices/Forest`
- The `rnd[n]` random with exact order, n, number of non-zero, nnz , and approximate target rank, r. This was done by adding a sum r rank k matrices each with k^2 nonzero entries, for very small k.

- The matrices Bcsstk29, f855_mat9, Saylr3, Saylr4, and tols4000 were extracted from the MCS/NASTRAN or Boeing ATLAS structural engineering programs by Randy Cigel, Roger Grimes, John Lewis, and Ed Meyer. These are five very large problems encountered in detailed modeling of structures. `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc5/bcsstruc5.html` Although they are numeric, we used their patterns to make $0,1$-matrices for our study with properties emerging from real applications. The bibd_81_3 is the incidence matrix of a balanced incomplete block design. It has 85320 columns, but just a few hundred non-zero rows, a fact not indicated in figure 8. This accounts for the greater success of GSLU over BB on this matrix. .

Figure 8 summarizes their key parameters together with our main group of timing results. But first we study the field representation issues and then the algorithm behaviour on some of the families.

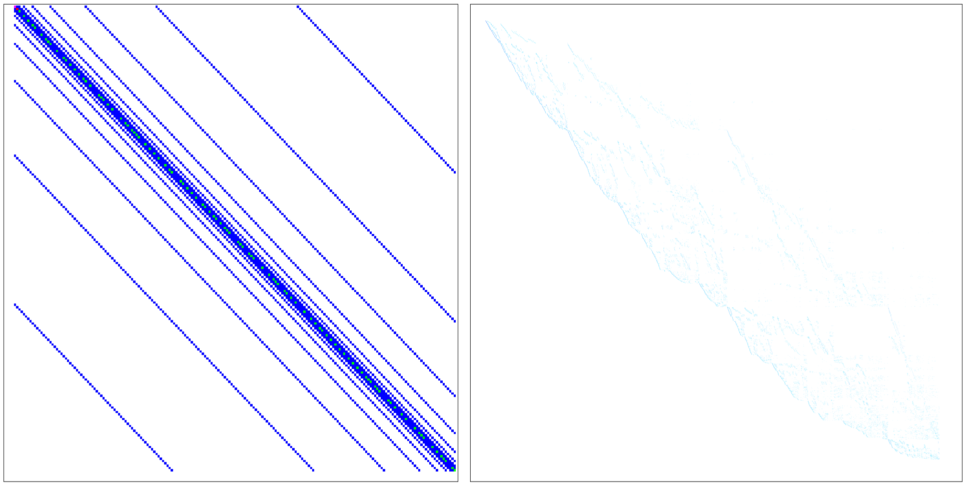

Figure 2. *Picture of trefethen and TF class matrices*

4 Field representation issues

Linbox contains several representations of finite fields, particularly of prime fields for word sized primes, i.e. primes less than 2^{32}. The fastest field arithmetic is achieved for word sized primes, so such primes are chosen in algorithms on integer matrices which use computations over modular images, i.e. over a $\boldsymbol{Z}/\boldsymbol{Z}p$, for some p. The results are then lifted and/or combined by the Chinese remainder algorithm to achieve the integer solutions. For multiplication of prime field elements x, y, the result is generally normalized to r, where r is the remainder in integer division, $xy = qp + r$. In this section we compare the performance of three prime field representations. It is the reduction modulo p, not the multiplication itself, which is the dominant cost for these representations. The first is that of the NTL package by Victor Shoup. It's dominant performance enhancement is a fast modular reduction using a floating point representation of $1/p$ with suitable adjustment to achieve an exact result. In linbox this representation is wrapped in the field class `NTL-zz_p`. The second is that of the Givaro package by the Apache group. It's dominant performance enhancement is to avoid almost all modular reductions in dot products by summing products, detecting 32 bit integer overflow, and adjusting when it occurs. For this a prime less than 2^{16} must be used. In linbox this is wrapped in `Givaro-zpz`. The third is a `Modular<uint32>` implemented directly in LinBoxitself. It uses summation of dot product terms in a 64 bit value with a reduction modulo the prime only when in danger of overflow. The prime must be less than 2^{32}.

The latter two are more effective for the BB algorithms which heavily depend on dot products. The NTL implementation is generally faster for the GSLU algorithms which are dominated by vector axpy and do not involve long sums of products.

Figure 3 shows that the NTL field representation performs better than the Modular field on GSLU. The bar heights are Modular field time over NTL field time for primes of 3 sizes and for 4 matrices (described more fully later). The speedups are slightly better when the COLAMD preordering is used. This presumably reflects less time in the preordering stage in which there is no field arithmetic. The tols4000 matrix has a very fast run time (also shown later) and the preordering stage dominates so there is little difference due to field arithmetic. The Givaro representation performs better that the others when primes less than 2^{16} are used[6]. In our experiments with the prime 65521 the time was about 2/3 that of the NTL arithmetic.

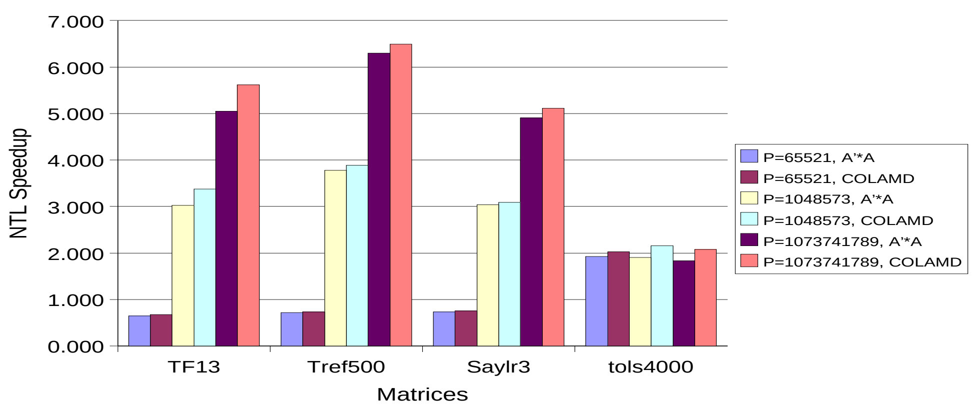

Figure 3. *Speedup of NTL field representation over Modular field for GSLU matrix rank computation with primes of three sizes and with COLAMD and A^TA preordering.*

Next we look at the BB algorithm where dot products dominate. We consider 5 primes ranging in size from 16 bits to 32 bits. The NTL representation is not sensitive to the size of the prime in this range. The points plotted in figure 4 are times divided by the NTL time thus are speedup relative to the NTL performance. We see that the speedup is highest for the smallest prime and degrades from there. This is consistent with being able to take longer sums of terms in dot products before the necessity of reduction modulo the prime.

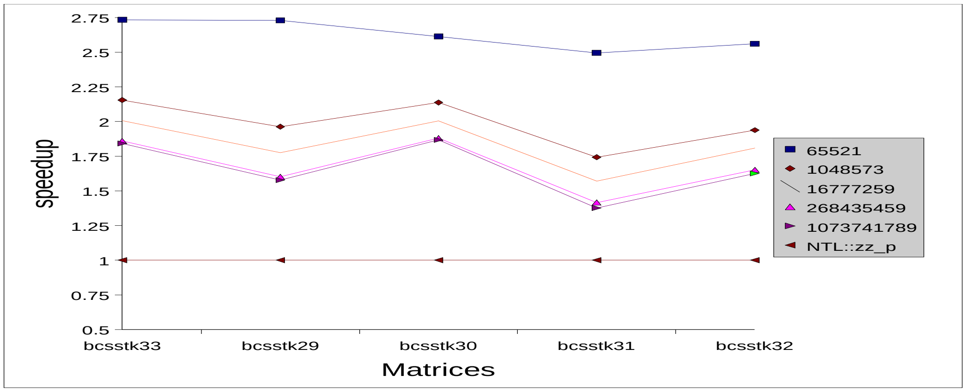

Figure 4. *speedup of Modular field representation relative to NTL field for black box matrix rank computation. As modulus increases, performance decreases.*

5 Matrix representation issues

For the GSLU computation, the matrix is initally stored in "comp col" format in which an arrays of nnz non-zero values, nnz row indices, and $n+1$ indexes which indicate where the columns begin. The algorithm then modifies the storage scheme for the final L, and U, using the "SuperMatrix" form for one of them. This storage scheme is not altered for the implementation over finite fields.

For the BB algorithm, LinBox provides a several sparse matrix representations. Each stores at least one index along with each nonzero entry. The 0,1-matrix is a common phenomenon in exact linear algebra, so we created a format, ZeroOne, which consists merely of a list of row,col pairs for the one's. They are stored as a pair of arrays of length nnz , one for row index, one for col. A "comp col" or "comp row" format could have been used as well, but this wasn't done. The result is substantially faster than the standard sparse matrix formats, taking about 2/3 of the time. We compared using the fastest field representation for each case, which turned out to be NTL for the ZeroOne class and Modular for the SparseMatrix.

6 Experimental results

Two sparse families, Tref and TF, showed particularly fast fill-in in GSLU. For these the crossover between the two methods occurs at order about 500 and 1000 respectively, as shown in figure 5 and figure 6. The number of non-zero entries is $O(\log n)$ for these families.

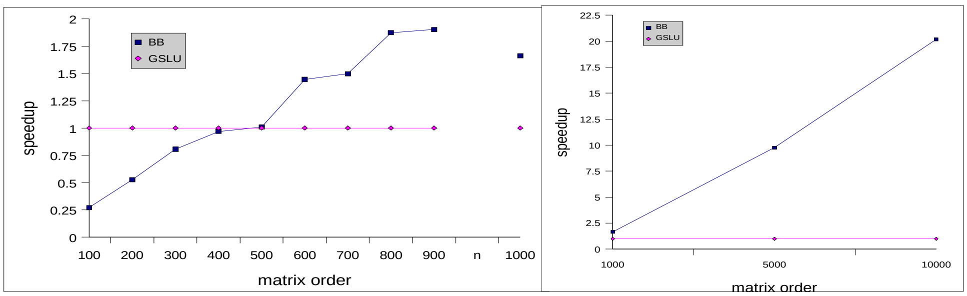

Figure 5. *Trefethen's banded matrix family. Speedup of BB over GSLU. Crossover is at order 500.*

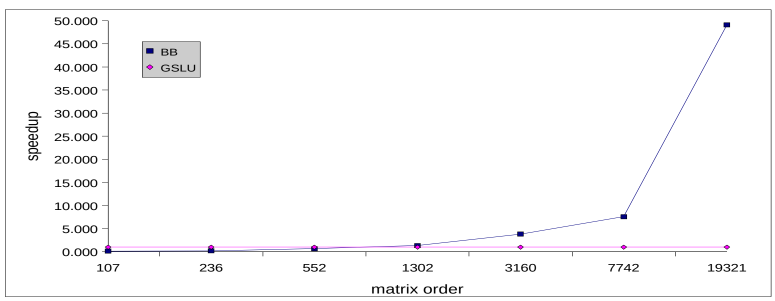

Figure 6. *TF family. Crossover is near order 1000.*

Similarly, the randomly generated matrices tend to fill-in rapidly and thwart preordering strategies, so that there again the crossover is low, provided the average number of non-zero entries per row, $\text{nnz}\ /n$, is large enough. Even for small ratio, eventually there will be show stopping fill-in for elimination methods. We studied the case of small ratio, $\text{nnz}\ /n = 3$ and found the crossover occurring at about $n = 10000$. For the random matrices with more entries per row than that, and of all sizes measured the BB performed better. For the one random matrix, `rnd6_14`, with average per row less than 3, GSLU was strikingly better (30 times better). Indeed this was also evident in the assorted group of matrices. GSLU performed best on those with $\text{nnz}\ /n <= 3$, with two exceptions where it performed within a factor of two of best.

In spite of the extremes of relative performance between GSLU and BB and the unpredictable nature of fill-in, we were able to design a hybrid algorithm that spots rising elimination cost early enough to switch to BB efficiently in almost all cases. The hybrid algorithm compares, at each few elimination steps, an estimate of the cost of continuing with elimination versus the cost of BB, starting over from scratch. The formula used to estimate cost of continuing was based simply on the assumption that per step cost would not decrease for the rest of the computation. The current the per step cost is multiplied by the number of remaining steps and this is compared to the prediction of BB cost. If BB looks cheaper, we switch. No attempt is made to predict an increase in per-step cost in future elimination steps. Interestingly, in our examinations of per-step costs we found examples where per step cost rose as would be expected in a left looking method applied to a dense matrix but we also found many examples where after an initial rise in cost, step costs remained relatively constant. Fortunately, it proved unnecessary to try to estimate continued increase in per-step cost.

Figure 7 shows the effectiveness of our hybrid strategy, labeled A3. For these measurements the COLAMD preordering was used. The switching method was similarly effective when other preordering strategies were applied.. It is put it in comparison to the GSLU method, the BB method, and the racing method. The figure shows performance of each method in terms of efficiency relative to the best of GSLU and BB for the given matrix. Higher is better. The matrices are in order of increasing size from left to right. We also see graphically demonstrated that size alone is not a good predictor of the relative performance of the GSLU and BB methods. The racing method is labeled A4 and the race is conducted via timsharing on one processor, so that it's efficiency is 50%. Of course the racing method achieves the best time when a second processor is available. Also, because of the inherent memory efficiency of the BB approach, the memory needs on the second processor are lower.

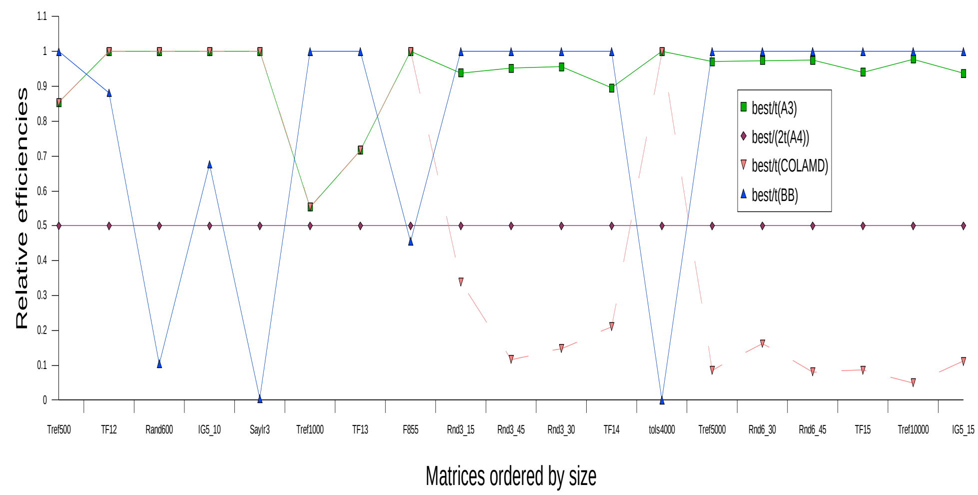

Figure 7. *Hybrid and racing algorithms compared to the best of GSLU(COLAMD preordering) and BB*

Matrices	n	nnz	Rank	nnz/n	t(GSLU)	t(BB)	BB/GSLU	Matrices	n	nnz/n	BB/GSLU
bcsstk29	13992	619488	10006	44.27	1164.77	4077.71	3.50	tols4000	4000	2.2	10273.650
Bibd_81_3	85320	255960	3240	3	97.82	3154.60	32.25	Saylr3	1000	3.75	228.050
f855_mat9	2511	171214	2456	68.19	6.95	273.22	39.30	f855_mat9	2511	68.19	39.295
Saylr3	1000	3750	998	3.75	0.12	27.37	228.05	Bibd_81_3	85320	3	32.251
TF10	107	622	99	5.81	0.02	0.12	7.67	Rnd6_14	6000	2.33	29.547
TF11	236	1607	216	6.81	0.12	0.57	4.74	Rnd12_36	12000	3	21.318
TF12	552	4231	488	7.66	2.16	3.20	1.48	Rnd6_18	6000	3	18.973
TF13	1302	11185	1121	8.59	25.41	18.92	0.74	Rnd18_54	18000	3	17.402
TF14	3160	29862	2644	9.45	438.49	114.77	0.26	TF10	107	5.81	7.667
TF15	7742	80057	6334	10.34	5495.14	723.64	0.13	TF11	236	6.81	4.744
TF16	19321	216173	15437	11.19	117664	2395.59	0.02	Rnd3000	3000	3	3.657
tols4000	4000	8784	3999	2.2	0.04	410.95	10273.65	bcsstk29	13992	44.27	3.501
Rnd3000	3000	9000	2789	3	16.35	59.79	3.66	Tref200	200	14.45	1.895
Rnd6000	6000	18000	5564	3	139.72	239.71	1.72	Rnd6000	6000	3	1.716
Rnd12000	12000	36000	11144	3	1217.55	1005.38	0.83	TF12	552	7.66	1.484
Rnd18000	18000	54000	16658	3	4154.51	2288.45	0.55	Tref300	300	15.59	1.241
Rnd6_18	6000	18000	7	3	0.04	0.70	18.97	Tref400	400	16.45	1.031
Rnd12_36	12000	36000	7	3	0.07	1.41	21.32	Tref500	500	16.96	0.990
Rnd18_54	18000	54000	5	3	0.11	1.86	17.40	Rnd12000	12000	3	0.826
Rnd3_15	3000	15000	2718	5	180.90	75.28	0.42	TF13	1302	8.59	0.744
Rnd3_30	3000	30000	2876	10	743.69	126.46	0.17	Tref600	600	17.59	0.692
Rnd3_45	3000	45000	2973	15	1409.22	178.72	0.13	Tref1000	1000	18.95	0.601
Rnd6_14	6000	14004	5010	2.33	6.50	192.08	29.55	Rnd18000	18000	3	0.551
Rnd6_30	6000	30000	5419	5	1502.21	301.54	0.20	Rnd3_15	3000	5	0.416
Rnd6_45	6000	45000	5847	7.5	4526.92	423.05	0.09	TF14	3160	9.45	0.262
Tref200	200	2890	200	14.45	0.41	0.78	1.89	Rnd6_30	6000	5	0.201
Tref300	300	4678	300	15.59	1.48	1.84	1.24	Rnd3_30	3000	10	0.170
Tref400	400	6578	400	16.45	3.38	3.49	1.03	TF15	7742	10.34	0.132
Tref500	500	8478	500	16.96	5.67	5.62	0.99	Rnd3_45	3000	15	0.127
Tref600	600	10554	600	17.59	12.12	8.38	0.69	Tref5000	5000	23.72	0.102
Tref1000	1000	18954	1000	18.95	40.15	24.14	0.60	Rnd6_45	6000	7.5	0.093
Tref5000	5000	118618	5000	23.72	7192.09	736.57	0.10	Tref10000	10000	25.72	0.050
Tref10000	10000	257234	10000	25.72	65698.30	3255.86	0.05	TF16	19321	11.19	0.020

Figure 8. *a table organized by matrix family and a table sorted by performance ratio: BB time over GSLU time. Less than 1 means BB better, greater than one means GSLU better.*

Bibliography

[1] L. Chen, W. Eberly, E. Kaltofen, W. J. Turner, B. D. Saunders, G. Villard, *Efficient Matrix Preconditioners for Black Box Linear Algebra*, LAA 343-344, 2002, pp. 119-146.

[2] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. Ng, *A column approximate minimum degree ordering algorithm*, Technical Report, CISCE, University of Florida. Oct. 2000.

[3] J. W. DEMMEL, J. R. GILBERT and X. S. LI, SuperLU User's Guide download: http://www.nersc.gov/ xiaoye/SuperLU/

[4] Dumas, Gautier, Giesbrecht, Giorgi, Hovinen, Kaltofen, Saunders, Turner, and Villard, *Linbox: A Generic Library for Exact Linear Algebra*, ICMS 2002, the International Congress of Mathematical Software, 2002, World Scientific, to appear.

[5] J-G. Dumas, T. Gautier, and C. Pernet, *Finite Field Linear Algebra Subroutines*, In Proc. 2002 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'02), ACM Press, pp. 63–74.

[6] J-G. Dumas, G. Villard, *Computing the rank of large sparse matrices over finite fields*, CASC'2002 : Computer Algebra in Scientific Computing.

[7] J-G. Dumas, B. D. Saunders, and G. Villard, *Integer Smith Form via the Valence: Experience with Large Sparse Matrices from Homology*, In Proc. 2000 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'00), ACM Press, pp. 95–105.

[8] J-G. Dumas, W. Turner, and Z. Wan, *Exact solution to large sparse integer linear systems*, ECCAD 2002, `http://www.cis.udel.edu/~wan/publication/eccad2002_abstract.ps`

[9] A. Duran and D. Saunders, *GenBLAS: Basic Linear Algebra Subroutines in C++ over Any Fields*, `http://www.cis.udel.edu/~duran/GenBLAS.pdf`, (GenBLAS version 1 download: `http://www.cis.udel.edu/~duran/GenBLAS.tar.gz`).

[10] A. Duran, D. Saunders and Z. Wan, *Rank of Sparse {0,1}-Matrices*, ECCAD 2003, `http://www.cis.udel.edu/~saunders/papers/eccad03/durana.pdf`,

[11] W. Eberly and E. Kaltofen, *On randomized Lanczos algorithms*, In Proc. 1997 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'97), pp. 176–183.

[12] J. W. H. LIU, Modification of the minimum-degree algorithm by multiple elimination. ACM Trans. Math. Software, 1:141-153, 1985.

[13] D. Saunders, A. Storjohann, and G. Villard, Matrix Rank Certification, ELA accepted, 2001.

[14] L. Trefethen *The Hundred Dollar, Hundred Digit Challenge*, SIAM News, 15, No. 6, July/August 2002.

[15] D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transf. Inform. Theory , IT-32:54–62, 1986.