Accelerating linear system solutions using randomization techniques

Marc Baboulin[1], Jack Dongarra[2,3,4], Julien Herrmann[1], and Stanimire Tomov[2]

[1] INRIA / Université Paris-Sud, France
[2] University of Tennessee, USA
[3] Oak Ridge National Laboratory, USA
[4] University of Manchester, United Kingdom

LAPACK Working Note 246
May 2011

Abstract. We illustrate how linear algebra calculations can be enhanced by statistical techniques in the case of a square linear system $Ax = b$. We study a random transformation of A that enables us to avoid pivoting and then to reduce the amount of communication. Numerical experiments show that this randomization can be performed at a very affordable computational price while providing us with a satisfying accuracy when compared to partial pivoting. This random transformation called Partial Random Butterfly Transformation (PRBT) is optimized in terms of data storage and flops count. We propose a solver where PRBT and the LU factorization with no pivoting take advantage of the latest generation of hybrid multicore/GPU machines and we compare its Gflop/s performance with a solver implemented in a current parallel library.

Keywords: dense linear algebra, linear systems, LU factorization, randomization, multiplicative preconditioning, Graphics Processing Units.

1 Introduction

Pivoting is a classical method to ensure stability in linear system solutions. It aims at preventing divisions by zero or too-small quantities in the process of Gaussian Elimination (GE). The complete pivoting procedure permutes rows and columns of the input matrix so that large nonzero matrix elements are moved to the diagonal to be used as "pivot". There is no floating-point operation in pivoting but it involves irregular movements ($\mathcal{O}(n^3)$ comparisons for the complete pivoting, where n is the matrix size). To reduce this overhead, the usual technique is Gaussian Elimination with Partial Pivoting (GEPP) where at each stage of the elimination, the pivot is searched within a column and only rows are permuted, reducing the number of comparisons to $\mathcal{O}(n^2)$. Note that there also exists an intermediate pivoting strategy called "rook pivoting" where the search for a pivot requires a number of comparisons comprised between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ ([11, p. 159]). The stability of GE is strongly related to the growth factor [11, p. 165] that measures how large the entries of the matrix become in the process of elimination. As in many numerical linear algebra algorithms, the choice of a pivoting strategy is the result of a trade-off between stability concerns and Gflop/s performance. In respect with that, a good GE algorithm should minimize the growth factor (to provide backward stability) and the amount of pivoting (to avoid penalizing performance). The upper bounds on the growth factor for GEPP might be much larger than for complete and rook pivoting (see [11, p. 169]) and it can be unstable for some very specific examples [23]. However GEPP turns out to be very stable in practice and has been implemented in standard linear algebra libraries (e.g. LAPACK [1]).

With the advent of architectures such as multicore processors or Graphics Processing Units (GPU), the growing gap between communication and computation efficiency made the communication overhead due to pivoting more critical. Moreover, in the LAPACK implementation of GEPP, rows are swapped at once during pivoting, which inhibits the exploitation of more asynchronicity between block operations. Several pivoting techniques, potentially less stable than partial or complete pivoting, can be used to minimize the communication like pairwise pivoting [20] or threshold pivoting [7] (see [21] for a stability analysis of these pivoting techniques). In particular pairwise pivoting has been implemented in algorithms for multicore machines [4] but this generates a significant overhead since the rows are swapped in pairs of blocks. We also mention, for multithreaded architectures, a pivoting technique called incremental pivoting in [17] based on principles used for out-of-core solvers. Another pivoting technique has been proposed in [10] that minimizes the number of messages exchanged during the factorization, leading to a new class of algorithms often referred to as "communication-optimal" algorithms. More specifically for GPUs, the pivoting overhead was reduced by using an innovative data structure [22].

To illustrate the cost of pivoting, we plot in Figure 1 the percentage of time due to pivoting in LU factorization (MAGMA[5] implementation) for several sizes of random matrices on a current hybrid CPU/GPU machine. We observe that pivoting can represent more than 40% of the global factorization time for small matrices and although the overhead decreases with the size of the matrix, it still represents 17% for a matrix of size $10,000$.

The fact that pivoting remains a bottleneck for linear system solutions is a motivation to present in this paper an alternative to pivoting thanks to randomization.

Statistical techniques have been widely used in linear algebra for instance for solving linear systems using Monte Carlo methods [5] or computing condition estimates [2, 13]. Statistical properties of Gaussian elimination have also been studied for the non pivoting case [25] and for the partial and complete pivoting case [21]. In this paper, we describe an approach based on randomization where the original matrix A is transformed into a matrix that would be sufficiently "random" so that, with a probability close to 1, pivoting is not needed.

[5] Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/`

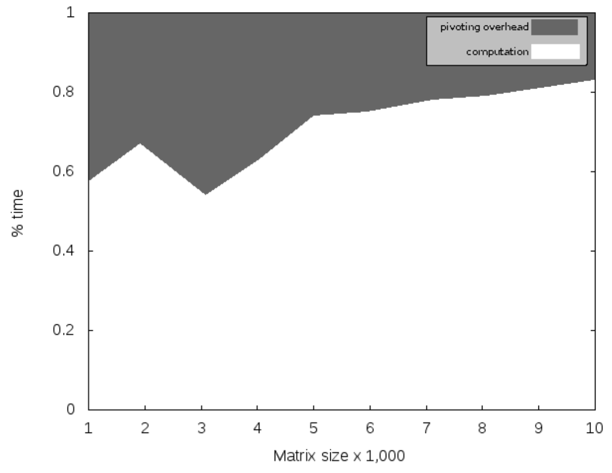

Fig. 1. Cost of pivoting in LU factorization (CPU 1 × Quad-Core Intel Core2 Processor Q9300 @ 2.50 GHz GPU C2050 — 14 Multiprocessors (× 32 CUDA cores) @ 1.15 GHz).

This technique has been initially proposed in [15, 16] where the randomization is referred to as Random Butterfly Transformation (RBT). It consists of a multiplicative preconditioning U^TAV where the matrices U and V are chosen among a particular class of random matrices called *recursive butterfly matrices*. Then Gaussian Elimination with No Pivoting (GENP) is performed on the matrix U^TAV and, to solve $Ax = b$, we instead solve $(U^TAV)y = U^Tb$ followed by $x = Vy$. Note that, if we know in advance that we are not going to pivot, GENP can be implemented as a very efficient fully BLAS 3 [6] algorithm. We study here the random transformation with *recursive* butterfly matrices and minimize the number of recursion steps which are required to get a satisfying accuracy. The resulting transformation will be called Partial Random Butterfly Transformation (PRBT). We propose a packed storage for the recursive butterfly matrices and we show that the multiplication by U^T and V can be efficiently computed by taking advantage of the particular structure of the recursive butterflies.

We also show that in practice, at most two levels of recursion are required for recursive butterflies to obtain an accuracy close to that of GEPP. Thus in most cases we can avoid the computation of a full RBT (that would require about $n^2log(n)$ flops) and the cost for the preconditioning reduces to $\sim 8n^2$ operations, which is negligible when compared to the cost of pivoting.

For the sake of stability we add some iterative refinement steps in the working precision where the stopping criterion is the component-wise relative backward error. For matrices that we use in our experiments, we never need more than one iteration. An important observation is that the 2-norm condition number of the initial matrix A is kept almost unchanged in the PRBT randomization.

Finally we present performance results for a PRBT solver on a state-of-the-art hybrid multicore/GPU machine and we compare the Gflop/s performance of this solver with a solver from the parallel library MAGMA.

2 Randomization

2.1 Definitions

We define here two types of matrices that will be used in the random transformation. We follow the definitions given in [15] in the particular case on real arithmetic entries.

Definition 1 *A butterfly matrix is defined as any n-by-n matrix of the form:*

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix}$$

where $n \geq 2$ and R_0 and R_1 are random diagonal and nonsingular $n/2$-by-$n/2$ matrices.

Note that a butterfly matrix B can also be expressed as

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{pmatrix} \begin{pmatrix} R_0 & 0 \\ 0 & R_1 \end{pmatrix},$$

where $I_{n/2}$ denotes the identity matrix of size $n/2$ i.e. B is a product of an orthogonal matrix and a random diagonal matrix. Then the possible orthogonality properties of B depend on how the random diagonal is obtained.

Definition 2 *A recursive butterfly matrix of size n and depth d is a product of the form*

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times ... \times \begin{pmatrix} B_1^{<n/4>} & 0 & 0 & 0 \\ 0 & B_2^{<n/4>} & 0 & 0 \\ 0 & 0 & B_3^{<n/4>} & 0 \\ 0 & 0 & 0 & B_4^{<n/4>} \end{pmatrix}$$
$$\times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B^{<n>},$$

where the $B_i^{<n/2^{k-1}>}$ are butterfly matrices of size $n/2^{k-1}, k = 1, \ldots, d$.

Note that this definition requires that n is a multiple of 2^{d-1} which can be always obtained by "augmenting" the matrix A with additional 1's on the diagonal. Note also that it differs from the definition of a recursive butterfly given in [15] which corresponds to the special case where $d = log_2 n + 1$, i.e. the first term of the product expressing $W^{<n,d>}$ is a diagonal matrix of size n.

For instance, if $n = 4$ and $d = 2$, then the recursive butterfly $W^{<4,2>}$ would be defined by

$$
\begin{aligned}
W^{<4,2>} &= \begin{pmatrix} B_1^{<2>} & 0 \\ 0 & B_2^{<2>} \end{pmatrix} \times B^{<4>} \\
&= \frac{1}{2} \begin{pmatrix} r_1^{<2>} & r_2^{<2>} & 0 & 0 \\ r_1^{<2>} & -r_2^{<2>} & 0 & 0 \\ 0 & 0 & r_3^{<2>} & r_4^{<2>} \\ 0 & 0 & r_3^{<2>} & -r_4^{<2>} \end{pmatrix} \begin{pmatrix} r_1^{<4>} & 0 & r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & r_4^{<4>} \\ r_1^{<4>} & 0 & -r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & -r_4^{<4>} \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} r_1^{<2>}r_1^{<4>} & r_2^{<2>}r_2^{<4>} & r_1^{<2>}r_3^{<4>} & r_2^{<2>}r_4^{<4>} \\ r_1^{<2>}r_1^{<4>} & -r_2^{<2>}r_2^{<4>} & r_1^{<2>}r_3^{<4>} & -r_2^{<2>}r_4^{<4>} \\ r_3^{<2>}r_1^{<4>} & r_4^{<2>}r_2^{<4>} & -r_3^{<2>}r_3^{<4>} & -r_4^{<2>}r_4^{<4>} \\ r_3^{<2>}r_1^{<4>} & -r_4^{<2>}r_2^{<4>} & -r_3^{<2>}r_3^{<4>} & r_4^{<2>}r_4^{<4>} \end{pmatrix},
\end{aligned}
$$

where the $r_i^{<j>}$ are real random entries.

Our motivation here is to minimize the computational cost of the RBT defined in [15] by considering a number of recursions d such that $d \ll n$ resulting in the transformation defined below.

Definition 3 *A partial random butterfly transformation (PRBT) of depth d of a square matrix A is the product:*

$$A_r = U^T A V$$

where U and V are recursive butterflies of depth d.

Then, the process to solve the general linear system $Ax = b$ is the following:

1. Compute the randomized matrix $A_r = U^T A V$, with U and V recursive butterflies
2. Factorize A_r with GENP
3. Solve $A_r y = U^T b$
4. Solution is $x = V y$

We recall that the GENP algorithm which is performed on A_r is unstable, due to a possibly large growth factor. We can find in [15] explanations about how RBT might modify the growth factor of the original matrix A. To ameliorate this potential instability, we systematically add in our method iterative refinement in the working precision as indicated in [11, p. 232].

2.2 Packed storage for recursive butterfly matrices

We describe here how a butterfly matrix and a recursive butterfly matrix can be stored compactly using respectively a vector and a matrix.

Following Section 2.1, a butterfly matrix has the form

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

where R_0 and R_1 are diagonal random matrices. Then $B^{<n>}$ can be stored compactly in a vector w of size n, where the $n/2$ first values are the coefficients of R_0 and the $n/2$ last ones are the coefficients of R_1.

Let us now consider a recursive butterfly of depth d expressed using butterfly matrices as the product

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times ... \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B^{<n>}.$$

We observe that each term of the product can be stored in a vector of size n. Thus $W^{<n,d>}$ can be stored compactly in a matrix W_p of size $n \times d$ where the k-th column represents the matrice $\begin{pmatrix} B_1^{<n/2^{k-1}>} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{k-1}}^{<n/2^{k-1}>} \end{pmatrix}$, which means that each vector $W_p((i-1) * \frac{n}{2^{k-1}} + 1 : i * \frac{n}{2^{k-1}}, k)$ stores the butterfly matrix $B_i^{<n/2^{k-1}>}$. As a result, $W^{<n,d>}$ can be obtained at once by choosing randomly the corresponding n-by-d matrix W_p.

2.3 Computational cost of the randomized matrix

In the computation of $U^T AV$, where U and V are recursive butterflies, the elementary operation is a multiplication of a dense matrix A to the left and to the right by a butterfly matrix.

Let $B = \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix}$ and $B' = \begin{pmatrix} R_0' & R_1' \\ R_0' & -R_1' \end{pmatrix}$ be two butterfly matrices stored in vectors w and w' using the packed storage defined in Section 2.2. We observe that a multiplication on both sides of A by B and B' can be expressed as

$$B^T A B' = \frac{1}{2} \begin{pmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{pmatrix} A \begin{pmatrix} R'_0 & R'_1 \\ R'_0 & -R'_1 \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} R'_0 & R'_1 \\ R'_0 & -R'_1 \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} R_0 & 0 \\ 0 & R_1 \end{pmatrix} C \begin{pmatrix} R'_0 & 0 \\ 0 & R'_1 \end{pmatrix}$$
$$= \frac{1}{2} \operatorname{diag}(w)\ C\ \operatorname{diag}(w'),$$

where $C = \begin{pmatrix} A_{11} + A_{12} + A_{21} + A_{22} & A_{11} - A_{12} + A_{21} - A_{22} \\ A_{11} + A_{12} - A_{21} - A_{22} & A_{11} - A_{12} - A_{21} + A_{22} \end{pmatrix}$.
Then $(B^T A B')_{i,j} = w_i C_{i,j} w'_j$ and the computation of $B^T A B'$ requires $4n^2$ flops. This kernel corresponds to a PRBT of depth 1 and will be applied for computing the successive products of the form $B^T A B'$ involved in the PRBT. For instance, for $d = 2$ we have

$$W^{<n,2>} = B^T \begin{pmatrix} B_1^T & 0 \\ 0 & B_2^T \end{pmatrix} A \begin{pmatrix} B'_1 & 0 \\ 0 & B'_2 \end{pmatrix} B'$$

$$= B^T \begin{pmatrix} B_1^T A_{11} B'_1 & B_1^T A_{12} B'_2 \\ B_2^T A_{21} B'_1 & B_2^T A_{22} B'_2 \end{pmatrix} B',$$

which involves four elementary products of the form $B^T A B'$ with butterflies of size $n/2$ and one with butterflies of size n. This requires $8n^2$ flops.

More generally, let A be a square matrix of size n and $M(n)$ the computational cost of a multiplication $B^T A B'$ with B and B' butterflies of size n, then the number of operations involved in the computation of A_r by a PRBT of depth d is

$$C(n,d) = \sum_{k=1}^{d} ((2^{k-1})^2 \times M(n/2^{k-1})) = \sum_{k=1}^{d} ((2^{k-1})^2 \times 4(n/2^{k-1})^2) = \sum_{k=1}^{d} (4n^2) = 4dn^2.$$

Note that the maximum cost obtained in the case of an RBT as described in [15] is
$$C(n, log_2 n + 1) \simeq 4n^2 log_2 n,$$
and then this cost is significantly reduced by considering numbers of recursions d such that $d \ll n$.

Similarly to the product of a recursive butterfly by a matrix, the product of a recursive butterfly by a vector does not require the explicit formation of the recursive butterfly since the computational kernel will be a product of a butterfly by a vector, which involves $\mathcal{O}(n)$ operations. As a result, the computation of $U^T b$ and Vy (steps (3) and (4) of the solution process given after Definition 3) can be performed in $\mathcal{O}(dn)$ flops and will be neglected in the remainder of this paper, for small values of d.

2.4 Condition number of the randomized matrix

A major concern in the multiplicative preconditioning involved in PRBT is to keep the condition number as "unchanged" as possible. Let us denote by $\text{cond}_2(A)$ the 2-norm condition number of a square matrix A and defined by $\text{cond}_2(A) = \|A\|_2 \left\|A^{-1}\right\|_2$. Then, with the notations of Section 2.1, we have

$$\text{cond}_2(A_r) \leq \text{cond}_2(U)\ \text{cond}_2(A)\ \text{cond}_2(V)\ .$$

Ideally, a recursive butterfly matrix will have a condition number close to 1 so that the condition number of A_r will be close to that of A. In general random matrices tend to be well conditioned (see [8]) but let us study here the particular case of the recursive butterfly matrices.

For an elementary butterfly matrix B of size n, we have

$$\begin{aligned} B^T B &= \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix} \\ &= \begin{pmatrix} R_0^2 & 0 \\ 0 & R_1^2 \end{pmatrix} \\ &= diag(r_1, \ldots, r_n)^2, \end{aligned}$$

where the r_i are random entries and then we obtain (using e.g. [18, p. 231])

$$\text{cond}_2(B) = \sqrt{\text{cond}_2(B^T B)} = \frac{\max |r_i|}{\min |r_i|}. \tag{1}$$

It comes from Equation (1) that the random variables r_i should not be too small to avoid having a large condition number for B.

More generally, a recursive butterfly of depth d is a product of block-diagonal matrices having the form $\mathcal{B} = diag(B_1, \ldots, B_p)$ where $1 \leq p \leq 2^{d-1}$ and the B_i are butterfly matrices of size n/p. Therefore we have

$$\mathcal{B}^T \mathcal{B} = \begin{pmatrix} B_1^T B_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_p^T B_p \end{pmatrix},$$

and $\text{cond}_2(\mathcal{B})$ can also be expressed as $\frac{\max |r_i|}{\min |r_i|}$ where the r_i are random numbers that obviously take values different from those in Equation (1).

If the r_i are such that $|r_i| \in [\alpha, \beta]$ ($\alpha > 0$), then we have $\text{cond}_2(\mathcal{B}) \leq \frac{\beta}{\alpha}$ and thus, for U being a recursive butterfly of depth d, we get

$$\text{cond}_2(U) \leq \left(\frac{\beta}{\alpha}\right)^d . \tag{2}$$

This result will motivate the type of random values used in forming the recursive butterflies. In particular, since the bound on the condition number grows with the number of recursions, $\frac{\beta}{\alpha}$ should be close to 1. In [15], the random diagonal values used in the butterflies are generated as $exp(\frac{r}{10})$, where r is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$, and this choice is justified by the fact that the determinant of a butterfly has an expected value 1. It satisfies also our requirement because $\frac{\beta}{\alpha} = e^{0.1} \approx 1.1052$. Experiments will be performed in Section 3.2 to confirm the good behaviour of this randomization process in terms of conditioning.

3 Numerical experiments

3.1 Accuracy of PRBT

In this section, we compare the accuracy of the linear system solution obtained using GEPP (as it is implemented in LAPACK) and PRBT followed by GENP (in the remainder, this solver will be simply denoted as PRBT). We also compare with GENP and QR. We recall here that the Householder QR factorization is always a good option for solving square linear systems because of its backward stability property (see [11, p. 361]) and due to the fact that we do not have to worry about large growth factors (however the computational cost is about twice that of LU).

Experiments were carried out using Matlab version 7.12 (R2011a) on a machine with a precision of $2.22 \cdot 10^{-16}$. In Table 1, we consider test matrices of size 1024 where the first 11 matrices come from the Matlab gallery and Higham's Matrix Computation Toolbox [11], the 12-th matrix comes from [9], the test cases number 13 to 16 come from [21] and the last 2 matrices are defined in [15]. Similarly to [15], the random diagonal matrices used to generate the butterfly matrices described in Definition 1 have diagonal values $exp(\frac{r}{10})$ where r is chosen from the uniform distribution in $[-\frac{1}{2}, \frac{1}{2}]$ (using the matlab instruction `rand`). For all test matrices, we consider the exact solution $x = [1\ 1 \ldots 1]$ and the right-hand side is set as $b = Ax$.

We report in Table 1 the 2-norm condition number of the original matrix (Matlab function `cond`) and the component-wise backward error resulting from the four solvers considered in this study. This error is defined in [14] and expressed by

$$\omega = \max_i \frac{|A\hat{x} - b|_i}{(|A| \cdot |\hat{x}| + |b|)_i},$$

where $\hat{x}$ is the computed solution. We also report the number of recursion steps (REC) used in the PRBT algorithm for the recursive butterflies (parameter d

in Definition 3). For better stability, we add systematically iterative refinement (in the working precision) when we use PRBT. Similarly to [3, 19], the iterative refinement algorithm is activated while $\omega > (n+1)u$, where u is the machine precision. The number of iterations (IR) in the iterative refinement process is also listed in Table 1.

We observe that we never need more than two recursions, which involves for PRBT an extra computational cost lower or equal to $8n^2$ operations. The two matrices `gfpp` [12] and `Foster` [9] are well-known pathological matrices that maximize the growth factor. For these matrices, PRBT destroys the original structure and gives very accurate results (for these two matrices, one step of iterative refinement was also required for QR to get the best accuracy). GENP fails for 4 matrices (`fiedler`, $\{-1,1\}$, $\{0,1\}$, $|i-j|$) and for each of them, PRBT is as accurate as GEPP. For the matrices `fielder`, $|i-j|$ and $max(i,j)$, PRBT gives results that are slightly better than QR.

For 3 matrices (`chebspec`, `condex`, `randcorr`), using PRBT is not useful because GENP gives a good solution. However this shows that these matrices are not degenerated by the randomization applied to them. On some matrices (`circul`, `augment`, `normaldata`, $[-1,1]$, $[0,1]$), the accuracy of GENP can be improved just by adding iterative refinement and PRBT is not useful. Iterative refinement turns out to be necessary in some cases when using PRBT but with never more than one iteration. Note that when matrices are orthogonal (`orthog` or proportional to an orthogonal matrix (`Hadamard`)), Gaussian elimination has not to be used. These 2 examples have been used only for purpose of testing. In the case of integer-valued matrices ($max(i,j)$, `Hadamard`), PRBT destroys the integer structure and transform the matrix into a real-valued one. Finally, in all test cases considered in these experiments, PRBT provides us with a satisfying accuracy while requiring an extra computational cost of $\mathcal{O}(n^2)$ operations (coming from one or two recursions and possibly one step of iterative refinement).

3.2 Tests on condition numbers

In the previous experiments we also computed, for all test matrices, the condition number of the randomized matrix. As expected from the comments in Section 2.4, $\mathrm{cond}_2(A_r)$ is of same order of magnitude as $\mathrm{cond}_2(A)$ and therefore is not listed in Table 1.

Let us now study in more details the condition number of the recursive butterflies resulting from the random distribution chosen in our experiments. We represent in Figure 2 the 2-norm condition number (computed using the Matlab function `cond`) of the recursive butterflies used in the experiments described in Section 3.1. We plot, for each recursion depth, the average condition number obtained for a sample of 500 recursive butterflies of size 1024 and the upper bound on this condition number as expressed in Equation (2). We observe that for small numbers of recursions, the average condition number is very close to its bound (e.g. for $d=2$, $\overline{\mathrm{cond}_2(U)} = 1.2026$ and $\left(\frac{\beta}{\alpha}\right)^d = 1.2214$) and that for larger numbers of recursions, the difference between these quantities becomes

Table 1. Comparison of linear system solution using PRBT with other solvers on a collection of matrices.

Matrix	Cond	GENP	GEPP	QR	PRBT	REC	IR
`augment`	$4 \cdot 10^{4}$	$1.28 \cdot 10^{-14}$	$2.28 \cdot 10^{-15}$	$2.99 \cdot 10^{-16}$	$2.81 \cdot 10^{-16}$	1	1
`gfpp`	$5 \cdot 10^{2}$	$9.01 \cdot 10^{-01}$	$6.88 \cdot 10^{-01}$	$1.06 \cdot 10^{-16}$	$1.27 \cdot 10^{-16}$	1	1
`chebspec`	$2 \cdot 10^{14}$	$1.19 \cdot 10^{-15}$	$3.29 \cdot 10^{-16}$	$5.22 \cdot 10^{-15}$	$3.23 \cdot 10^{-14}$	1	0
`circul`	$1 \cdot 10^{3}$	$1.74 \cdot 10^{-13}$	$1.66 \cdot 10^{-15}$	$2.66 \cdot 10^{-15}$	$2.66 \cdot 10^{-15}$	1	0
`condex`	$1 \cdot 10^{2}$	$7.32 \cdot 10^{-15}$	$5.98 \cdot 10^{-15}$	$8.34 \cdot 10^{-15}$	$6.50 \cdot 10^{-15}$	1	0
`fiedler`	$7 \cdot 10^{5}$	Fail	$2.11 \cdot 10^{-15}$	$1.54 \cdot 10^{-14}$	$7.90 \cdot 10^{-15}$	1	0
`Hadamard`	$1 \cdot 10^{0}$	$0 \cdot 10^{0}$	$0 \cdot 10^{0}$	$7.58 \cdot 10^{-16}$	$8.33 \cdot 10^{-15}$	1	0
`normaldata`	$3 \cdot 10^{4}$	$2.03 \cdot 10^{-12}$	$6.30 \cdot 10^{-15}$	$2.38 \cdot 10^{-16}$	$3.30 \cdot 10^{-16}$	1	1
`orthog`	$1 \cdot 10^{0}$	$5.64 \cdot 10^{-01}$	$4.33 \cdot 10^{-15}$	$3.70 \cdot 10^{-16}$	$4.31 \cdot 10^{-16}$	2	1
`randcorr`	$3 \cdot 10^{3}$	$5.12 \cdot 10^{-16}$	$4.04 \cdot 10^{-16}$	$5.73 \cdot 10^{-16}$	$5.92 \cdot 10^{-16}$	1	0
`toeppd`	$7 \cdot 10^{5}$	$2.53 \cdot 10^{-13}$	$2.60 \cdot 10^{-15}$	$8.39 \cdot 10^{-15}$	$5.71 \cdot 10^{-15}$	1	0
`Foster`	$5 \cdot 10^{2}$	$1 \cdot 10^{0}$	$1 \cdot 10^{0}$	$1.90 \cdot 10^{-16}$	$3.30 \cdot 10^{-16}$	2	1
$[-1, 1]$	$2 \cdot 10^{3}$	$2.19 \cdot 10^{-11}$	$5.19 \cdot 10^{-15}$	$2.33 \cdot 10^{-16}$	$2.35 \cdot 10^{-16}$	1	1
$[0, 1]$	$4 \cdot 10^{4}$	$1.97 \cdot 10^{-12}$	$2.85 \cdot 10^{-15}$	$2.15 \cdot 10^{-15}$	$1.79 \cdot 10^{-15}$	1	1
$\{-1, 1\}$	$4 \cdot 10^{3}$	Fail	$3.96 \cdot 10^{-15}$	$2.38 \cdot 10^{-16}$	$2.70 \cdot 10^{-16}$	2	1
$\{0, 1\}$	$5 \cdot 10^{4}$	Fail	$4.39 \cdot 10^{-15}$	$2.19 \cdot 10^{-15}$	$1.09 \cdot 10^{-15}$	2	1
$\|i - j\|$	$7 \cdot 10^{5}$	Fail	$3.33 \cdot 10^{-16}$	$1.54 \cdot 10^{-14}$	$6.05 \cdot 10^{-15}$	1	0
$max(i, j)$	$3 \cdot 10^{6}$	$2.16 \cdot 10^{-14}$	$1.21 \cdot 10^{-15}$	$1.46 \cdot 10^{-14}$	$2.27 \cdot 10^{-15}$	1	1

larger (e.g. for $d = 10$, $\overline{\text{cond}_2(U)} = 1.5183$ and $\left(\frac{\beta}{\alpha}\right)^d = 2.7183$) and then the upper bound becomes more pessimistic. This is not surprising since for small values of d the difference comes mainly from the statistics and for large values, the difference comes also from the nature of the upper bound which is a product of d bounds as explained in Section 2.4. However, as shown in Section 3.1, two recursions are in general enough to get a satisfying accuracy and in that case recursive butterflies are very well conditioned.

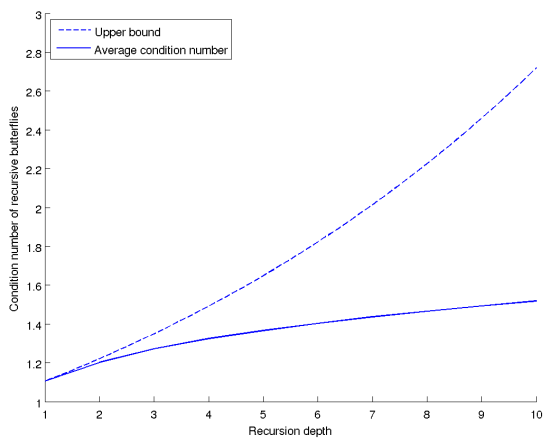

Fig. 2. Average 2-norm condition number for recursive butterfly matrices (samples of 500 matrices) for a fixed matrix size $n = 1024$.

3.3 Preliminary performance results

In this section, we present a first implementation of PRBT on a current hybrid CPU/GPU architecture. The GPU is a Fermi Tesla S2050 (1.15 GHz, 2687.4 MB memory) and its multicore host is a 48 cores system (4 sockets x 12 cores) AMD Opteron 6172 (2.1 GHz). On the multicore we use LAPACK and BLAS from MKL 10.2. The PRBT solver is compared with a GEPP solver as it is implemented in the MAGMA 1.0 library. In both cases the multicore host is involved just in the panel factorization, the update of the trailing matrix being performed on the GPU. In our implementation of PRBT, the randomization is performed on the CPU (with 2 recursions). Figure 3 shows the performance in Gflop/s for both solvers using double precision arithmetic and we observe that PRBT achieves much better performance depending on the size of the matrix. For small problems the gain is much bigger (from 100% for size 1,000 to 33% for size 3,000). In the range 4,000-8,000, the gain obtained by using PRBT is about 20% and for matrix sizes larger than 9,000, the improvement is around 10%

showing that asymptotically, the two performances should be close. We point out that these results are obtained using a GEPP implementation specifically tuned for this architecture while PRBT could be still improved by additional tuning and use of a scheduler (e.g. QUARK [24]). Improvement could also be obtained by taking advantage of the multicore in the update of the trailing matrix. In this respect, the performance results of PRBT are very promising.

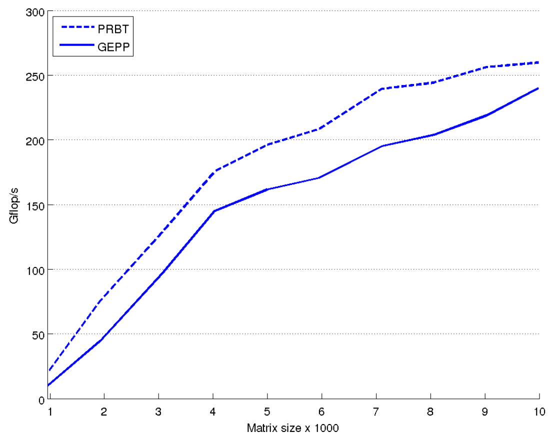

Fig. 3. Performance for PRBT and GEPP in double precision arithmetic (4 $\times$ 12-Core AMD Opteron 6172 @ 2.1 GHz - GPU Fermi Tesla S2050 @ 1.15 GHz).

4 Conclusion and future work

We proposed a linear system solver where the LU factorization is performed without pivoting on a matrix randomized by PRBT. We showed that PRBT does not alter the 2-norm condition number of the original matrix and that it requires in practice a low computational cost ($\mathcal{O}(n^2)$ operations) and a few additional data storage. We demonstrated that the obtained accuracy is similar to that of GEPP on a reasonable range of matrices. We also gave first performance results on a current hybrid CPU/GPU architecture where the pre-processing due to randomization is performed on the CPU and the LU factorization without pivoting is a hybrid CPU/GPU program. The resulting PRBT solver outperforms the GEPP solver as it is implemented in the MAGMA library. The PRBT method shall be integrated into the MAGMA library jointly with a fully BLAS 3 GENP solver. The latter could be indeed very useful to factorize efficiently matrices for which the growth factor is $\mathcal{O}(1)$ and therefore pivoting is not needed (see examples of such classes of matrices in [11, p. 166]). Further experiments will be performed on multicore architectures which will allow performance comparisons with other solvers (e.g. from the PLASMA[6] library). which are not necessarily based on GEPP and enable more extensive testing.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.
2. M. Arioli, M. Baboulin, and S. Gratton, *A partial condition number for linear least-squares problems*, SIAM J. Matrix Anal. and Appl. **29** (2007), no. 2, 413–433.
3. M. Arioli, J. W. Demmel, and I. S. Duff, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Anal. and Appl. **10** (1989), no. 2, 165–190.
4. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing **35** (2009), 38–53.
5. I. Dimov, *Monte carlo methods for applied scientists*, Word Scientific, 2008.
6. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw. **16** (1990), 1–17.
7. I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Clarendon Press, Oxford, 1986.
8. A. Edelman, *Eigenvalues and condition numbers of random matrices*, SIAM J. Matrix Anal. and Appl. **9** (1988), no. 4, 543–560.
9. L. V. Foster, *Gaussian elimination with partial pivoting can fail in practice*, SIAM J. Matrix Anal. and Appl. **15** (1994), no. 4, 1354–1362.
10. L. Grigori, J. W. Demmel, and H. Xiang, *Communication avoiding Gaussian elimination*, (2008), In Proceedings of the IEEE/ACM SuperComputing SC08 Conference.
11. N. J. Higham, *Accuracy and stability of numerical algorithms*, SIAM, 2002, Second edition.
12. N. J. Higham and D. J. Higham, *Large growth factors in Gaussian elimination with pivoting*, SIAM J. Matrix Anal. and Appl. **10** (1989), no. 2, 155–164.
13. C. S. Kenney, A. J. Laub, and M. S. Reese, *Statistical condition estimation for linear least squares*, SIAM J. Matrix Anal. and Appl. **19** (1998), 906–923.
14. W. Oettli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numerische Mathematik **6** (1964), 405–409.
15. D. S. Parker, *Random butterfly transformations with applications in computational linear algebra*, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.
16. D. S. Parker and B. Pierce, *The randomizing FFT: an alternative to pivoting in Gaussian elimination*, Technical Report CSD-950037, Computer Science Department, UCLA, 1995.
17. G. Quintana-Orti, E. S. Quintana-Orti, R. A. van de Geijn, F. G. van Zee, and E. Chan, *Programming algorithms-by-blocks for matrix computations on multithreaded architectures*, ACM Trans. Math. Softw. **36** (2009), no. 3, 1–26.
18. Y. Saad, *Iterative methods for sparse linear systems*, SIAM, 2000, Second edition.

[6] Parallel Linear Algebra for Scalable Multi-core Architectures, `http://icl.cs.utk.edu/plasma/`

19. R. D. Skeel, *Iterative refinement implies numerical stability for Gaussian elimination*, Math. Comput. **35** (1980), 817–832.
20. D. C. Sorensen, *Analysis of pairwise pivoting in gaussian elimination*, IEEE Trans. Comput. **34** (1984), 274–278.
21. L. N. Trefethen and R. S. Schreiber, *Average-case stability of Gaussian elimination*, SIAM J. Matrix Anal. and Appl. **11** (1990), no. 3, 335–360.
22. V. Volkov and J. W. Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Technical Report UCB/EECS-2008-49, University of California, Berkeley, 2008, Also LAPACK Working Note 202.
23. S. J. Wright, *A collection of problems for which Gaussian elimination with partial pivoting is unstable*, SIAM J. Sci. Statist. Comput. **14** (1993), 231–238.
24. A. YarKhan, J. Kurzak, and Dongarra J., *QUARK users' guide: QUeueing And Runtime for Kernels*, Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011.
25. M. Yeung and T. F. Chan, *Probabilistic analysis of Gaussian elimination without pivoting*, SIAM J. Matrix Anal. and Appl. **18** (1997), no. 2, 499–517.