A parallel tiled solver for dense symmetric indefinite systems on multicore architectures

(LAPACK Working Note 261)

Marc Baboulin [*†], Dulceneia Becker [‡], Jack Dongarra [‡§¶]
[*] INRIA Saclay-Île de France, F-91893 Orsay, France
[†] Université Paris Sud, F-91405 Orsay, France
[‡] University of Tennessee, Knoxville, TN, USA
[§] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[¶] University of Manchester, Manchester, United Kingdom
marc.baboulin@inria.fr, dbecker7@eecs.utk.edu, dongarra@eecs.utk.edu

***Abstract*—We describe an efficient and innovative parallel tiled algorithm for solving symmetric indefinite systems on multicore architectures. This solver avoids pivoting by using a multiplicative preconditioning based on symmetric randomization. This randomization prevents the communication overhead due to pivoting, is computationally inexpensive and requires very little storage. Following randomization, a tiled LDL$^{\mathrm{T}}$ factorization is used that reduces synchronization by using static or dynamic scheduling. We compare Gflop/s performance of our solver with other types of factorizations on a current multicore machine and we provide tests on accuracy using LAPACK test cases.**

I. Introduction

Symmetric indefinite linear systems are commonly encountered in physical applications, *e.g.* Navier-Stokes discretized equations coming from incompressible fluid simulations, and optimization problems coming from physics of structures, acoustics, and electromagnetism. In the particular case of dense systems, an important example of application of such systems is the linear least-squares problem when it is solved via the augmented system method [5, p. 77]. In this case, we solve the linear least-squares problem

$$\min_{x\in\mathbb{R}^n} \|Cx-d\|_2$$

where $d \in \mathbb{R}^m$ and $C \in \mathbb{R}^{m\times n}$ by considering the equivalent linear system

$$\begin{pmatrix} I & C \\ C^T & 0 \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} d \\ 0 \end{pmatrix} \Leftrightarrow Ax = b$$

where $r = Cx-d$ is the residual, I is the identity matrix and the symmetric matrix $A \in \mathbb{R}^{m+n}$ is indefinite.

Dense symmetric indefinite systems (in complex arithmetic) also arise in electromagnestism when the Maxwell equations are discretized using the Boundary Element Method (BEM). For all these applications the usual problem size may be a few hundreds of thousands variables. This requires the use of parallel algorithms that minimize the number of floating-point operations per second, optimize the data storage, and achieve concurrency and scalability on current multicore architectures.

A symmetric matrix A is called indefinite when the quadratic form x^TAx can take on both positive and negative values. By extension, a linear system $Ax = b$ is called symmetric indefinite when A is symmetric indefinite. Even though symmetric indefinite matrices can be factorized using methods for general matrices such as LU or QR, methods that exploit the symmetry of the system are generally favored since the flop count becomes half that for a general matrix ($n^3/3$ vs $2n^3/3$ operations where n is the matrix size).

A classical way to solve these systems is based on so-called diagonal pivoting methods [8] where a block-LDL$^{\mathrm{T}}$ factorization is obtained such as

$$PAP^T = LDL^T \tag{1}$$

where P is a permutation matrix, A is a symmetric square matrix, L is unit lower triangular and D is block-diagonal, with blocks of size 1×1 or 2×2; all matrices are of size $n\times n$. If no pivoting is applied, *i.e.* $P = I$, D becomes diagonal. The solution x can be computed by successively solving the triangular or block-diagonal systems $Lz = Pb$, $Dw = z$, $L^Ty = w$, and ultimately we have $x = P^Ty$. This pivoting method turns out to be very stable in practice and is implemented in current serial dense linear algebra libraries (*e.g.* LAPACK [1]). This pivoting method requires between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons.

While many implementations of the $\mathrm{LDL}^{\mathrm{T}}$ factorization have been proposed for sparse solvers on distributed and shared memory architectures [11], [12], [14], [22], there is no parallel implementation in the current dense linear algebra libraries SCALAPACK [6], PLASMA [20], MAGMA [26], and FLAME [13]. These libraries have implemented solutions for the common Cholesky, LU and QR factorizations but none of them introduced a solution for indefinite symmetric matrices in spite of the gain of flops it could provide for these cases. The main reason for this comes from the algorithms used for pivoting in $\mathrm{LDL}^{\mathrm{T}}$, which are difficult to parallelize efficiently. To our knowledge, the only research in the subject has been done by Strazdins [24] and the procedure is available in the OpenMP version of MKL [16].

One of the differences between symmetric and nonsymmetric pivoting is that, independently from the pivoting technique used, columns and rows must be interchanged in the symmetric case while only columns must be swapped in the nonsymmetric case. This in itself makes pivoting more expensive in terms of data movement for symmetric matrices. Interchanging rows and columns also compromises data locality since noncontiguous data blocks must be moved however data are stored. There is also an increase of data dependencies, which inhibits parallelism, both in interchanging columns/rows and in searching for pivots. For nonsymmetric matrices, pivots are most commonly searched in a single column (partial pivoting) while for symmetric matrices the search may be extended to the diagonal and further.

With the advent of architectures such as multicore processors [25] and Graphics Processing Units (GPU), the growing gap between communication and computation efficiency made the communication overhead due to pivoting even more critical since it might represent more than 40% of the global factorization time, depending on the matrix size (see *e.g.* [3] for the case of general linear systems). This is why we propose in this paper a solver that eliminates the communication overhead due to pivoting by considering a randomization technique initially proposed in [21] and developed in [3] for the LU factorization, with preliminary results in [4] for symmetric indefinite systems. According to this transformation, the original matrix A is transformed into a matrix that would be sufficiently "random" so that, with a probability close to one, pivoting is not needed. This transformation is a multiplicative preconditioning by means of random matrices called *recursive butterfly matrices* resulting, when A is symmetric, in a so-called Symmetric Random Butterfly Transformation (SRBT). The $\mathrm{LDL}^{\mathrm{T}}$ factorization without pivoting is then applied to the preconditioned matrix. It has been observed in [4] on a collection of matrices that the computational overhead resulting from the randomization is reduced to $\sim 8n^2$ operations (which is negligible when compared to the communication overhead due to pivoting) while providing an accuracy close to that of $\mathrm{LDL}^{\mathrm{T}}$ with Bunch-Kaufman pivoting strategy as it is implemented in LAPACK and Matlab.

In this paper we propose an implementation of SRBT that uses a packed storage for the butterfly matrices and takes advantage of their particular structure to efficiently compute the random transformations. We also present a tiled $\mathrm{LDL}^{\mathrm{T}}$ factorization with no pivoting, which can reasonably strive for a "Cholesky speed" $\mathrm{LDL}^{\mathrm{T}}$ solver on multicore architectures.

Tiled algorithms are based on decomposing the computation into small tasks in order to overcome the sequential nature of the algorithms implemented in LAPACK. These tasks can be executed out of order, as long as dependencies are observed, rendering parallelism. Furthermore, tiled algorithms make use of a tile data-layout where data are stored in contiguous blocks, which differs from the column-wise layout used by LAPACK, for instance. The tile data-layout allows the computation to be performed on small blocks of data that fit into cache, and hence exploits cache locality and re-use. The so-called tiled $\mathrm{LDL}^{\mathrm{T}}$ factorization described in this paper is based on these principles. Note that such a factorization, *i.e.* without pivoting, could be used even without preliminar randomization to factorize efficiently symmetric matrices for which the growth factor is $\mathcal{O}(1)$ and therefore pivoting is not needed (see examples of such classes of matrices in [15, p. 166]).

For more stability, we systematically add a few iterative refinement steps in working precision where the stopping criterion is the componentwise relative backward error. For the matrices used in the experiments, no more than one iteration is ever needed. An important observation is also that the 2-norm condition number of the initial matrix A is kept almost unchanged after SRBT.

We can summarize the method used by our solver as:

1) multiplicative preconditioning of the original matrix using SRBT with in general a maximum of two recursions ($\mathcal{O}(n^2)$ flops),
2) efficient tiled $\mathrm{LDL}^{\mathrm{T}}$ factorization without pivoting ($n^3/3$ flops),
3) iterative refinement in working precision (one or two iterations in practice) requiring $\mathcal{O}(n^2)$ flops.

This paper is organized as follows. In Section II, we describe the SRBT algorithm used prior to the $\mathrm{LDL^T}$ in order to avoid pivoting. In Section III, the $\mathrm{LDL^T}$ algorithm is introduced and the tiled $\mathrm{LDL^T}$ algorithm is detailed, as well as both dynamic and static scheduling. Performance results regarding a time comparison among $\mathrm{LDL^T}$, Cholesky and LU factorization, the $\mathrm{LDL^T}$ scalability, tile size performance and numerical accuracy are presented in Section IV. Conclusions are presented in Section V.

II. Transforming symmetric indefinite systems to avoid pivoting

A. Definition

For solving the symmetric indefinite system $Ax = b$, we first apply a random transformation to the matrix A so that pivoting is not needed in the $\mathrm{LDL^T}$ factorization. This technique was initially proposed by Parker [21] in the context of general linear systems where the randomization is referred to as Random Butterfly Transformation (RBT). Then a modified approach has been described in [3] for the LU factorization that reduces the computational cost of the transformation.

We have adapted this technique to symmetric systems[1]. The procedure to solve $Ax = b$, where A is symmetric, using a random transformation and the $\mathrm{LDL^T}$ factorization is:

1) Compute $A_r = U^T A U$, with U a random matrix,
2) Factorize $A_r = LDL^T$ (without pivoting),
3) Solve $A_r y = U^T b$ and compute $x = Uy$.

The matrix U is chosen among a particular class of random matrices called *recursive butterfly matrices* and the resulting transformation will be referred to as **Symmetric Random Butterfly Transformation** (SRBT). Obviously such a transformation is interesting only if the multiplicative preconditioning $U^T A U$ is "cheap" enough.

Let us first recall the definitions of two types of matrices used in SRBT. These definitions are based on [21] in the particular case of real-valued matrices.

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

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix} \tag{2}$$

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

[1]Note that when A is positive definite, randomization is not relevant since the factorization can be computed in a stable and efficient manner using a Cholesky algorithm resulting in a factorization $A = LL^T$ with L lower triangular.

We define a recursive butterfly matrix U of size n and depth d as a product of the form

$$U = U_d \times \cdots \times U_1, \tag{3}$$

where U_k $(1 \leq k \leq d)$ is a block diagonal matrix expressed as

$$U_k = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_{2^{k-1}} \end{pmatrix} \tag{4}$$

each B_i being a butterfly matrix of size $n/2^{k-1}$. In particular U_1 is a butterfly as defined in Formula (2).

Note that this definition requires that n is a multiple of 2^{d-1} which can always be obtained by "augmenting" the matrix A with additional 1's on the diagonal. Note also that the definition of U differs from the definition of a recursive butterfly matrix given in [21], which corresponds to the special case where $d = \log_2 n + 1$.

B. Storage of recursive butterfly matrices

A butterfly matrix as well as a recursive butterfly matrix can be stored compactly using a vector and a matrix, respectively. From Formula (2) and given $m = n/2$, a butterfly matrix B of size $n \times n$ can be stored compactly in a vector w of size n, such as the top m elements and the bottom m elements are, respectively, the coefficients of R and S:

$$w = [r_{11} r_{22} \ldots r_{mm}\ s_{11} s_{22} \ldots s_{mm}]^T$$

Let us now consider a recursive butterfly U of depth d, as expressed in Formulas (3) and (4). We observe that each term U_k can be stored in a vector of size n. Thus U can be stored compactly in a matrix W of size $n \times d$ where the kth column represents the matrix U_k expressed in Formula (4), which means that each matrix B_i is stored as

$$B_i \rightarrow W\left((i-1) \cdot \frac{n}{2^{k-1}} + 1\ :\ i \cdot \frac{n}{2^{k-1}}\ ,\ k\right)$$

As a result, the recursive butterfly U can be obtained at once by choosing randomly the corresponding n-by-d matrix W.

Note that due to the symmetry of the transformation $A_r = U^T A U$, this represents half the storage required for the butterflies used in LU factorization.

C. Computation of the randomized matrix

In the following, we describe how SRBT can be efficiently computed by taking advantage of the symmetry and also estimate the number of floating-point operations required. The computational cost of SRBT depends on

the order of the matrix to be transformed n and on number of recursion levels d.

Given that

$$A_r = U^T A U = \prod_{i=1}^{d} U_i^T \; A \; \prod_{i=d}^{1} U_i$$

for each recursion level k, $U_k^T Q\, U_k$ must be computed as a block matrix of the form

$$\begin{pmatrix} B_1^T Q_{11} B_1 & \cdots & B_1^T Q_{p1}^T B_p \\ \vdots & \ddots & \vdots \\ B_p^T Q_{p1} B_1 & \cdots & B_p^T Q_{pp} B_p \end{pmatrix} \tag{5}$$

where $p = 2^{k-1}$ and Q is a partial random transformation of A (levels d to $k+1$) given by

$$Q = \prod_{i=k+1}^{d} U_i^T \; A \; \prod_{i=d}^{k+1} U_i$$

Equation (5) requires two computational kernels:

1) symmetric $B^T C B$ with C symmetric, and
2) general $B^T C B'$.

For the general (nonsymmetric) kernel:

$$\begin{aligned} B^T C B' &= \frac{1}{2} \begin{pmatrix} R & R \\ S & -S \end{pmatrix} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \begin{pmatrix} R' & S' \\ R' & -S' \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} R & 0 \\ 0 & S \end{pmatrix} D \begin{pmatrix} R' & 0 \\ 0 & S' \end{pmatrix} \\ &= \frac{1}{2} \operatorname{diag}(w)\; D \; \operatorname{diag}(w'), \end{aligned}$$

where $D =$

$$\begin{pmatrix} (C_{11}+C_{22})+(C_{21}+C_{12}) & (C_{11}-C_{22})-(C_{12}-C_{21}) \\ (C_{11}-C_{22})-(C_{21}-C_{12}) & (C_{11}+C_{22})-(C_{21}+C_{12}) \end{pmatrix}$$

and w and w' are the vectors storing compactly B and B', respectively, as described in Section II-B. The computation of $B^T C B'$ requires $4n^2$ flops.

For the symmetric kernel, $w' = w$ and therefore

$$B^T C B = \frac{1}{2} \operatorname{diag}(w)\; D \; \operatorname{diag}(w),$$

D is symmetric and hence so are $C_{11}+C_{22}$, $C_{21}+C_{12}$, and $C_{11}-C_{22}$. The computation of D and each multiplication by w require approximately $n^2 + \mathcal{O}(n)$ flops (considering the symmetry). Finally the computation of $B^T C B$ requires $2n^2 + \mathcal{O}(n)$ flops.

Each matrix expressed in (5) requires p symmetric kernels and $p(p-1)/2$ general (nonsymmetric) kernels operating on matrices of size n/p. Therefore, the number of operations involved in randomizing A by an SRBT of depth d is

$$\begin{aligned} C(n,d) &\simeq \sum_{k=1}^{d} \left(p \cdot 2(n/p)^2 + p(p-1)/2 \cdot 4(n/p)^2 \right) \\ &= 2dn^2 \end{aligned}$$

As expected, SRBT requires half the flop count required for the randomization applied to the nonsymmetric case (see [3]). This cost is minimized by considering numbers of recursions d such that $d \ll n$. Numerical tests described in [4] and performed on a collection of matrices from the Higham's Matrix Computation Toolbox [15] have shown that, in practice, $d = 2$ enables us to achieve satisfactory accuracy.

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 Uy (step 3 of the solution process given at the beginning of Section II-A) can be performed in $\mathcal{O}(dn)$ flops and will be neglected in the remainder of this paper, for small values of d.

D. Generation of the butterflies and conditioning issues

We generate the random diagonal values used in the butterflies as $e^{\rho/10}$, where ρ is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$. This choice is suggested and justified in [21] by the fact that the determinant of a butterfly has an expected value 1. Then the random values r_i used in generating butterflies are such that

$$e^{-1/20} \leq r_i \leq e^{1/20}.$$

Let us now evaluate how the multiplicative preconditioning involved in SRBT might affect the 2-norm condition number of the original matrix defined by $\operatorname{cond}_2(A) = \|A\|_2 \left\|A^{-1}\right\|_2$.

For an elementary butterfly B of size n, we have

$$\begin{aligned} B^T B &= \begin{pmatrix} R^2 & 0 \\ 0 & S^2 \end{pmatrix} \\ &= diag(r_1, \ldots, r_n)^2. \end{aligned}$$

Then we have

$$\operatorname{cond}_2(B) = \sqrt{\operatorname{cond}_2(B^T B)} = \frac{\max r_i}{\min r_i},$$

and thus

$$\operatorname{cond}_2(B) \leq e^{1/10}. \tag{6}$$

Let $\mathcal{B} = diag(B_1, \ldots, B_p)$ be one of the random block diagonal matrices as expressed in Formula (4) with $1 \leq p \leq 2^{d-1}$ and the B_i's butterflies of size n/p. Then, we have $\mathcal{B}^T \mathcal{B} = diag({B_1}^T B_1, \ldots, {B_p}^T B_p)$ and, using Equation (6), we also have $\operatorname{cond}_2(\mathcal{B}) \leq exp\left(\frac{1}{10}\right)$, and for a a recursive butterfly U of depth d, we get $\operatorname{cond}_2(U) \leq e^{d/10}$. Then, since the condition number of the randomized matrix A_r verifies

$$\operatorname{cond}_2(A_r) \leq \operatorname{cond}_2(U)\, \operatorname{cond}_2(A)\, \operatorname{cond}_2(U)\ ,$$

we get

$$\mathrm{cond}_2(A_r) \ \leq e^{d/5}\mathrm{cond}_2(A) \ = 1.2214^d \mathrm{cond}_2(A) \ . \tag{7}$$

Since d is in general close to 2, Formula (7) shows that the condition number of the original matrix is kept almost unchanged by SRBT. However, we recall that the LDL$^{\mathrm{T}}$ algorithm without pivoting is potentially unstable [15, p. 214], due to a possibly large growth factor. We can find in [21] 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 a few steps of iterative refinement in the working precision as indicated in [15, p. 232].

III. Tiled LDL$^{\mathrm{T}}$ Factorization

The LDL$^{\mathrm{T}}$ factorization is given by equation

$$A = LDL^T \tag{8}$$

where A is an $N \times N$ symmetric square matrix, L is unit lower triangular and D is diagonal. For simplicity and also because pivoting is not used, the assumption that D is diagonal has been made, although D can also be block-diagonal, with blocks of size 1×1 or 2×2, when pivoting is applied. The matrix A can also be factorized as $A = UDU^{\mathrm{T}}$, where U is unit upper triangular. The algorithm for the lower triangular case (L) can straightforwardly be extended to the upper triangular case (U) and therefore only the development of the former is presented.

Algorithm 1 shows the steps needed to decompose A using a column-wise LDL$^{\mathrm{T}}$ factorization. After N steps of Algorithm 1, L and D are such that

$$A = \begin{bmatrix} 1 & & \\ l_{21} & 1 & \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} d_1 & & \\ & d_2 & \\ & & d_3 \end{bmatrix} \begin{bmatrix} 1 & l_{21} & l_{31} \\ & 1 & l_{32} \\ & & 1 \end{bmatrix}$$

As it is depicted in Figure 1, this process is intrinsically sequential.

Algorithm 1 LDL$^{\mathrm{T}}$ Factorization

1: **for** $j = 1$ to N **do**
2: **for** $i = 1$ to $j - 1$ **do**
3: $v_i = A_{j,i} A_{i,i}$
4: **end for**
5: $v_j = A_{j,j} - A_{j,1:j-1} v_{1:j-1}$
6: $A_{j,j} = v_j$
7: $A_{j+1:N,j} = \left(A_{j+1:N,j} - A_{j+1:N,1:j-1} v_{1:j-1}\right) / v_j$
8: **end for**

In order to increase parallelism, the tiled algorithm starts by decomposing A in $NT \times NT$ tiles (blocks), such as

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1,NT} \\ A_{21} & A_{22} & \dots & A_{2,NT} \\ \vdots & \vdots & \ddots & \vdots \\ A_{NT,1} & A_{NT,2} & \dots & A_{NT,NT} \end{bmatrix}_{N \times N} \tag{9}$$

Each A_{ij} is a tile of size $MB \times NB$. The same decomposition can be applied to L and D. For instance, for $NT = 3$:

$$L = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix}, \ D = \begin{bmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{bmatrix}$$

Upon this decomposition and using the same principle of the Schur complement, the following equalities can be obtained:

$$A_{11} = L_{11} D_{11} L_{11}^T \tag{10}$$
$$A_{21} = L_{21} D_{11} L_{11}^T \tag{11}$$
$$A_{31} = L_{31} D_{11} L_{11}^T \tag{12}$$
$$A_{22} = L_{21} D_{11} L_{21}^T + L_{22} D_{22} L_{22}^T \tag{13}$$
$$A_{32} = L_{31} D_{11} L_{21}^T + L_{32} D_{22} L_{22}^T \tag{14}$$
$$A_{33} = L_{31} D_{11} L_{31}^T + L_{32} D_{22} L_{32}^T - L_{33} D_{33} L_{33}^T \tag{15}$$

By further rearranging the equalities, a series of tasks can be set to calculate each L_{ij} and D_{ii}:

$$[L_{11}, D_{11}] = \mathrm{LDL}(A_{11}) \tag{16}$$
$$L_{21} = A_{21} (D_{11} L_{11}^T)^{-1} \tag{17}$$
$$L_{31} = A_{31} (D_{11} L_{11}^T)^{-1} \tag{18}$$
$$\tilde{A}_{22} = A_{22} - L_{21} D_{11} L_{21}^T \tag{19}$$
$$[L_{22}, D_{22}] = \mathrm{LDL}(\tilde{A}_{22}) \tag{20}$$
$$\tilde{A}_{32} = A_{32} - L_{31} D_{11} L_{21}^T \tag{21}$$
$$L_{32} = \tilde{A}_{32} (D_{22} L_{22}^T)^{-1} \tag{22}$$
$$\tilde{A}_{33} = A_{33} - L_{31} D_{11} L_{31}^T + L_{32} D_{22} L_{32}^T \tag{23}$$
$$[L_{33}, D_{33}] = \mathrm{LDL}(\tilde{A}_{33}) \tag{24}$$

$$A^{(4)} = \begin{bmatrix} \circ & & & & & & & \\ & \circ & & & & & & \\ & & \circ & & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \end{bmatrix}$$

Fig. 1. Sketch of elements to be calculated ($\times$) and accessed ($\circ$) at the fourth step ($k = 4$) of Algorithm 1.

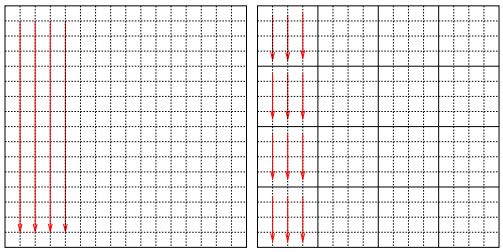


Fig. 2. Column-major and tile layout sketch.

where LDL(X_{kk}) at Equations (16), (20) and (24) means the actual LDL$^{\mathrm{T}}$ factorization of tile X_{kk}. These tasks can be executed out of order, as long as dependencies are observed, rendering parallelism.

The decomposition into tiles allows the computation to be performed on small blocks of data that fit into cache. This leads to the need of a reorganization of data formerly given in a column major layout, as depicted in Figure 2. The tile layout reorders data in such a way that all data of a single block is contiguous in memory. Thus the decomposition of the computation can either be statically scheduled to take advantage of cache locality and reuse or be dynamically scheduled based on dependencies among data and computational resources available.

A. Tiled LDL$^{\mathrm{T}}$ Algorithm

The tiled algorithm for the LDL$^{\mathrm{T}}$ factorization is based on the following operations:

`xSYTRF`: This LAPACK based subroutine is used to perform the LDL$^{\mathrm{T}}$ factorization of a symmetric tile A_{kk} of size $NB \times NB$ producing a unit triangular tile L_{kk} and a diagonal tile D_{kk}.
Using the notation *input* $\longrightarrow$ *output*, the call `xSYTRF`(A_{kk}, L_{kk}, D_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \mathrm{LDL}^{\mathrm{T}}\ (A_{kk})$$

`xSYTRF2`: This subroutine first calls `xSYTRF` to perform the factorization of A_{kk} and then multiplies L_{kk} by D_{kk}. The call `xSYTRF2`(A_{kk}, L_{kk}, D_{kk}, W_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \mathrm{LDL}^{\mathrm{T}}\ (A_{kk}),$$
$$W_{kk} = L_{kk} D_{kk}$$

`xTRSM`: This BLAS subroutine is used to apply the transformation computed by xSYTRF2 to an A_{ik} tile by means of a triangular system solve. The call `xTRSM`(W_{kk}, A_{ik}) performs

$$W_{kk}, A_{ik} \longrightarrow L_{ik} = A_{ik} W_{kk}^{-T}$$

`xSYDRK`: This subroutine is used to update the tiles A_{kk} in the trailing submatrix by means of a matrix-matrix multiply. It differs from `xGEMDM` by taking advantage of the symmetry of A_{kk} and by using only the lower triangular part of A and L. The call `xSYDRK`(A_{kk}, L_{ki}, D_{ii}) performs

$$A_{kk}, L_{ki}, D_{ii} \longrightarrow A_{kk} = A_{ik} - L_{ki} D_{ii} L_{ki}^{T}$$

`xGEMDM`: This subroutine is used to update the tiles A_{ij} for $i \neq j$ in the trailing submatrix by means of a matrix-matrix multiply. The call `xGEMDM`(A_{ij}, L_{ik}, L_{jk}, D_{kk}) performs

$$A_{ij}, L_{ik}, L_{jk}, D_{kk} \longrightarrow A_{ij} = A_{ij} - L_{ik} D_{kk} L_{jk}^{T}$$

Given a symmetric matrix A of size $N \times N$, NT as the number of tiles, such as in Equation (9), and making the assumption that $N = NT \times NB$ (for simplicity), where $NB \times NB$ is the size of each tile A_{ij}, then the tiled LDL$^{\mathrm{T}}$ algorithm can be described as in Algorithm 2. A graphical representation of Algorithm 2 is depicted in Figure 3.

Algorithm 2 Tile LDL$^{\mathrm{T}}$ Factorization

```
1:  for k = 1 to NT do
2:     xSYTRF2(A_kk, L_kk, D_kk, W_kk)
3:     for i = k + 1 to NT do
4:        xTRSM(W_kk, A_ik)
5:     end for
6:     for i = k + 1 to NT do
7:        xSYDRK(A_kk, L_ki, D_ii)
8:        for j = k + 1 to i − 1 do
9:           xGEMDM(A_ij, L_ik, L_jk, D_kk)
10:       end for
11:    end for
12: end for
```

B. Static and Dynamic Scheduling

Following the approach presented in [9], [10], [17], Algorithm 2 can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on one or several $NB \times NB$ blocks and where edges represent the dependencies among them. A dependency occurs when a task must access data outputted by another task either to update or to read them. Figure 4 shows a DAG for the tiled LDL$^{\mathrm{T}}$ factorization when Algorithm 2 is executed with $NT = 4$. Once the DAG is known, the tasks can be scheduled asynchronously and independently as long as the dependencies are not violated.

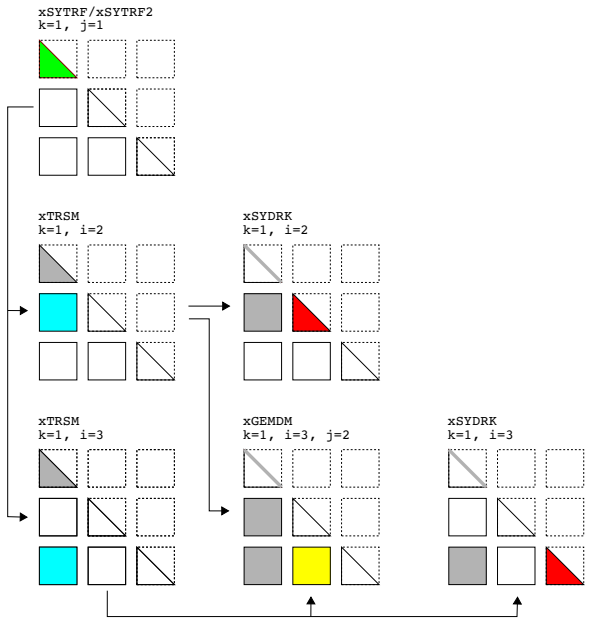


Fig. 3. Graphical representation with dependencies of one repetition of the outer loop in Algorithm 2 with $NT = 3$.

This dynamic scheduling results in an out-of-order execution where idle time is almost completely eliminated since only very loose synchronization is required between the threads. Figure 5(a) shows the execution trace of Algorithm 2 where tasks are dynamically scheduled, based on dependencies in the DAG, and run on 8 cores of the *MagnyCours-48* machine (described in Section IV). The tasks were scheduled using QUARK [27], which is the scheduler available in the PLASMA library. Each row in the execution flow shows which tasks are performed and each task is executed by one of the threads involved in the factorization. The trace follows the same color code as Figure 3.

Figure 5(b) shows the trace of Algorithm 2 using static scheduling, which means that each core's workload is predetermined. The synchronization of the computation for correctness is enforced by a global progress table. The static scheduling technique has two important shortcomings. First is the difficulty of development. It requires full understanding of the data dependencies in the algorithm, which is hard to acquire even by an experienced developer. Second is the inability to schedule dynamic algorithms, where the complete task graph is not known beforehand. This is the common situation for eigenvalue algorithms, which are iterative by nature [18]. Finally, it is almost impossible with the static scheduling to overlap simply and efficiently several functionalities like the factorization and the solve that are often called simultaneously. However for a single step, as can be seen in Figure 5, the static scheduling on a small number of cores may outrun the dynamic scheduling due to better data locality and cache reuse.

For the LDL$^{\mathrm{T}}$ factorization, it is possible to build an efficient progress table or execution path. Comparing Figures 5(a) and 5(b) one can notice that the tasks are scheduled in different ways but the execution time is similar (see Section IV for more details). It is important to highlight that developing an efficient static scheduling can be very difficult and that the dynamic scheduler notably reduces the complexity of programing tiled algorithms.

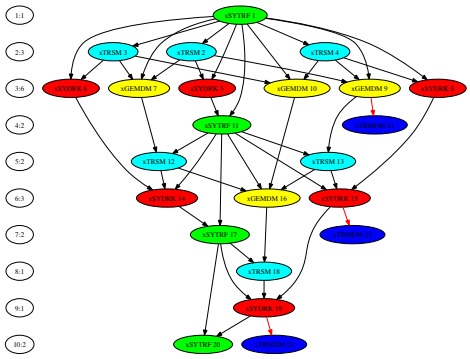


Fig. 4. DSYTRF DAG; $NT = 4$.

IV. Numerical Experiments

We present numerical experiments for our parallel LDL$^{\mathrm{T}}$ solver where the randomization by SRBT is computed as described in Section II-C with a maximum of 2 recursions and the butterflies are generated as explained in Section II-D. The LDL$^{\mathrm{T}}$ algorithm presented in Section III-A has been implemented by following the software development guidelines of PLASMA, the Parallel Linear Algebra Software for Multicore Architectures library [20]. In the remainder of this section, our solver for symmetric indefinite systems will be designated as SRBT-LDL$^{\mathrm{T}}$.

The numerical results that follow are presented for both a static and a dynamic scheduler (see Section III-B) and have been carried out using the *MagnyCours-48* system. This machine has a NUMA architecture and is composed of four AMD Opteron 6172 Magny-Cours CPUs running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. The theoretical peak of this machine is 403.2 Gflop/s (8.4 Gflop/s per core) in

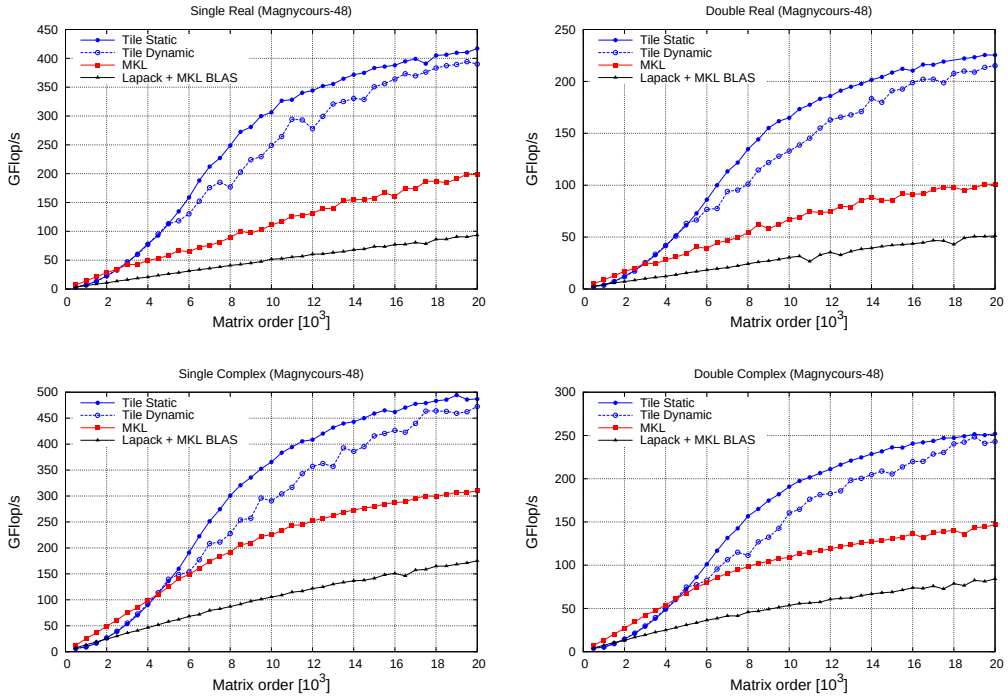
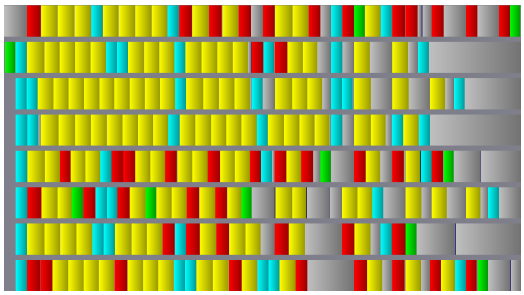
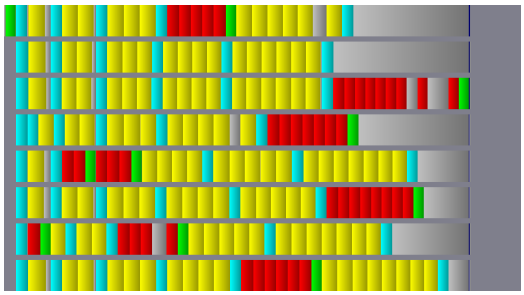


Fig. 6. Performance of SRBT-LDL$^{\mathrm{T}}$ against MKL and LAPACK.



(a) Dynamic scheduling



(b) Static scheduling

Fig. 5. Traces of tiled LDL$^{\mathrm{T}}$ (*MagnyCours-48* with 8 threads).

double precision. Comparisons are made against version 10.3.2 of the Intel MKL [16] library for multicore, and against the reference LAPACK 3.2 from Netlib, linked with the same MKL BLAS multi-threaded library. SRBT-LDL$^{\mathrm{T}}$ is linked with the sequential version of MKL for the required BLAS and LAPACK routines. This version of MKL achieves 7.5 Gflop/s on a `DGEMM` (matrix-matrix multiplication) on one core. Unless otherwise stated, the measurements were carried out using all the 48 cores of the *MagnyCours-48* system and run with `numactl -interleaved=0-#`. Also, a tile size of $NB = 250$ and an inner-blocking size of $IB = 125$.

A. Tests on accuracy

Preliminary tests were carried out in [4] to verify the accuracy of SBRT followed by LDL$^{\mathrm{T}}$ with no pivoting using a Matlab implementation and test matrices out of the Higham's Matrix Computation Toolbox [15]. We provide here additionnal tests with our multicore implementation using LAPACK test cases given in [7].

Table I describes the 10 matrices used in our experiments (size 512, all in double precision). In this table, ε denotes the machine precision and κ is the infinity-norm condition number of the matrix.

Table II shows the componentwise backward error

defined in [19] and expressed by

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

where x is the computed solution.

Three solvers are compared:

1) SRBT-LDL$^{\mathrm{T}}$,
2) LDL$^{\mathrm{T}}$ with partial pivoting (LAPACK),
3) SRBT followed by LDL$^{\mathrm{T}}$ with partial pivoting.

As mentionned in Section II-D, we add systematically iterative refinement (in the working precision) for better stability. Similarly to [2], [23], the iterative refinement algorithm is activated while $\omega > (n+1)\varepsilon$. For the LAPACK LDL$^{\mathrm{T}}$ solver (routine `DSYTRS`) in column 2, the iterative refinement is performed using the routine `DSYRFS`. The number of iterations (IR) required in the iterative refinement process is listed in Table II.

Solver 1 reports errors for matrices 6 and 9, while solver 2 for matrices 3 to 6. Matrices 3 to 6 are singular and have at least one row and column zero. These are used in [7] to test the error return codes. Nevertheless, the random transformation of A allows the LDL$^{\mathrm{T}}$ factorization to continue while the pivoting is not capable of removing the zero pivots. Matrix 9 is scaled to near underflow and hence the transformation causes it to underflow. Solver 3 has been added only to illustrate that, except for matrices 6 and 9 (which are respectively singular and scaled to near underflow), pivoting does not improve the LDL$^{\mathrm{T}}$ factorization of the randomized matrix.

TABLE I
TEST MATRICES

1	Diagonal	6	Random, $\kappa = 2$
2	First column zero	7	Random, $\kappa = \sqrt{1/\varepsilon}$
3	Last column zero	8	Random, $\kappa = 1/\varepsilon$
4	Middle column zero	9	Scaled near underflow
5	Last $n/2$ columns zero	10	Scaled near overflow

TABLE II
COMPONENTWISE BACKWARD ERROR

Matrix Type	SRBT-LDL$^{\mathrm{T}}$	LAPACK LDL$^{\mathrm{T}}$	SRBT + LDL$^{\mathrm{T}}$ PIV
1	0.8815e-13 (0)	0.1079e-15 (0)	0.1975e-13 (0)
2	0.4067e-13 (1)	0.2830e-13 (+)	0.4244e-13 (1)
3	0.2395e-13 (1)	-	0.1242e-13 (1)
4	0.2504e-13 (1)	-	0.3696e-13 (1)
5	0.5466e-13 (1)	-	0.8008e-13 (1)
6	-	-	0.1219e-13 (1)
7	0.3037e-13 (1)	0.3810e-13 (+)	0.6795e-13 (1)
8	0.6048e-13 (1)	0.2930e-13 (+)	0.5195e-13 (0)
9	-	0.5898e-13 (+)	0.2212e-13 (1)
10	0.3674e-13 (1)	0.8683e-13 (+)	0.1612e-13 (1)

(*) Iterative refinement number of iterations, where (+) stands for from 1 up to a maximum of 5 iterations

B. Performance results

Figure 6 shows the performance in Gflop/s of SRBT-LDL$^{\mathrm{T}}$ against both MKL and LAPACK LDL$^{\mathrm{T}}$ routines xSYTRS (for real and double real) and xHETRS (for complex and double complex). Note that we compare solvers that do not perform the same operations because SRBT-LDL$^{\mathrm{T}}$ does randomization and no pivoting while the other two solvers include pivoting. However, a definite matrix has been chosen for performance comparison so that no permutations are actually made by MKL and LAPACK (only search for pivot is performed). In our implementation of SRBT-LDL$^{\mathrm{T}}$, we use a tile layout where data is stored in block $NB \times NB$ (tiles) and performance is reported with dynamic and static scheduling for four arithmetic precisions (real, double real, complex, double complex). The static scheduling usually outruns the dynamic one, mostly due to the overhead of the dynamic scheduler. As mentioned before, the progress table for SRBT-LDL$^{\mathrm{T}}$ is quite efficient, exposing the overhead caused by the dynamic scheduler. We observe that SRBT-LDL$^{\mathrm{T}}$ is about twice faster than MKL and four times faster than LAPACK for all the four arithmetic precisions presented in Figure 6.

Let us now study specifically the performance of the tiled LDL$^{\mathrm{T}}$ factorization routine described in Section III-A that represents the bulk of the computation in the SRBT-LDL$^{\mathrm{T}}$ solver. This performance is compared with that of the LU (`xGETRF`) and Cholesky (`xPOTRF`) factorization routines from the version 2.4.1 of the PLASMA library. Since there is no LDL$^{\mathrm{T}}$ factorization in PLASMA and by analogy to LAPACK, the tiled LDL$^{\mathrm{T}}$ factorization routine is named here `xSYTRF` (resp. `xHETRF`) for real (resp. complex) arithmetic. Figure 7 reports the execution time of `xSYTRF`, `xPOTRF` and `xGETRF` with dynamic and static scheduling. The static scheduling scheme usually delivers the highest performance. This happens mostly because there is no overhead on scheduling the tasks and, as mentioned before, the LDL$^{\mathrm{T}}$ algorithm lends itself a quite efficient progress table. As expected, LDL$^{\mathrm{T}}$ is noticeably faster than LU and only moderately slower than Cholesky. This clearly states that it is advantageous, in terms of time, to factorize a symmetric matrix using `xSYTRF` (instead of `xGETRF`) and also that `xSYTRF` (instead of `xPOTRF`) can be used on decomposing SPD matrices (or diagonally dominant matrices, because they do not require pivoting) with no substantial time overhead.

The parallel speedup or scalability of `xSYTRF` is shown in Figure 8 for matrices of order 5000, 10000 and 20000 [N], both for dynamic and static scheduling.

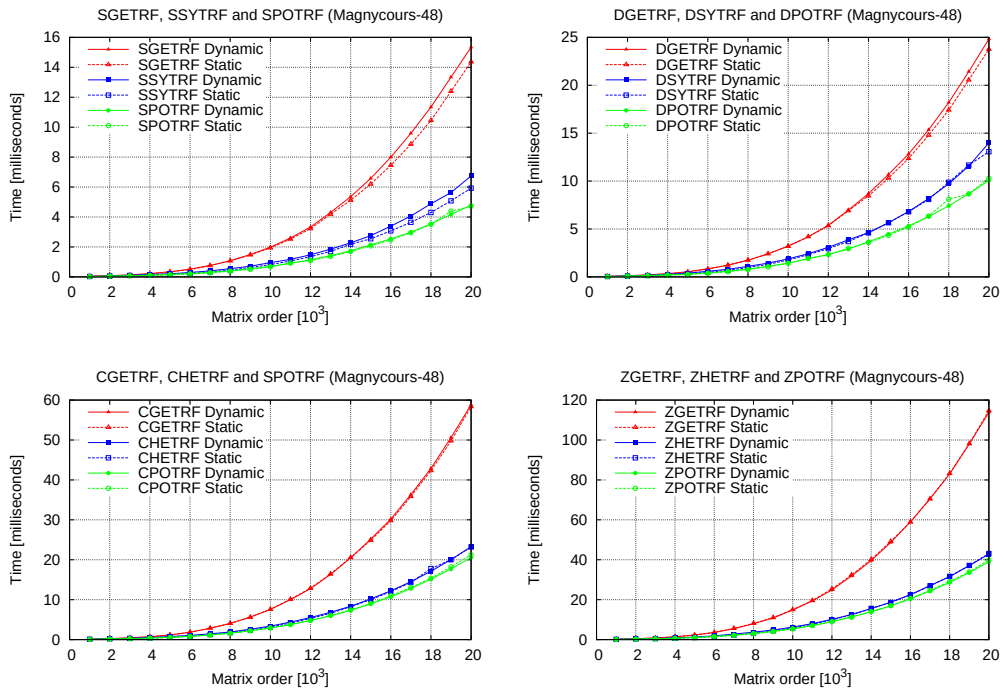


Fig. 7. Execution time of `xPOTRF`, `xSYTRF`/`xHETRF` and `xGETRF`; dynamic (solid line) and static (dashed line) scheduling.

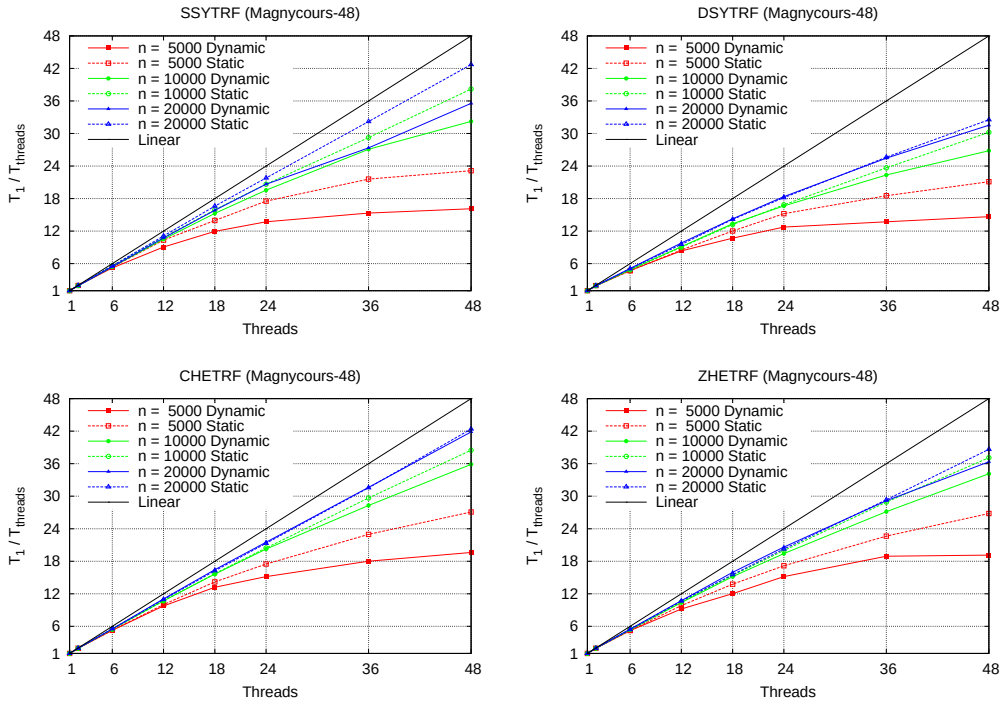


Fig. 8. `xSYTRF`/`xHETRF` scalability; dynamic (solid line) and static (dashed line) scheduling.

As anticipated, the parallel speedup increases as the matrix order increases. This happens because the bigger the matrix, the more tasks are available to be executed concurrently, resulting in higher scalability. The parallel performance actually depends on several factors, one of them being the tile size $[NB]$. The performance reported previously has been obtained with $NB = 250$ and $IB = 125$. As depicted in Figure 9, 250 is not necessarily the optimal tile size[2]. In order to achieve optimal performance, NB and other parameters must be tuned accordingly to the size of the matrix to be decomposed and the number of threads. This could be achieved through auto-tuning; this feature is still not available however.

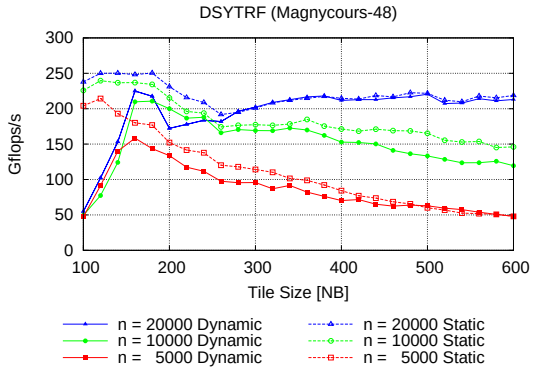


Fig. 9. Tile-size performance of tiled DSYTRF, dynamic (solid line) and static (dashed line) scheduling ($IB = 125$).

V. Conclusion

A solver for dense symmetric indefinite systems and its implementation for multicore machines were presented. This solver is based on a computationally cheap randomization technique followed by an efficient tiled $\mathrm{LDL}^{\mathrm{T}}$ factorization. It is scalable and achieves almost the same performance as the tiled Cholesky algorithm that can be considered as an upper bound for performance of $\mathrm{LDL}^{\mathrm{T}}$. Iterative refinement in working precision is added for sake of stability, and is also computationally negligible.

In addition to providing us with satisfying performance results, our solver gives accurate results in LAPACK test cases. More generally, these results illustrate that avoiding pivoting by the technique of randomization can speed up significantly linear algebra computations.

[2]Similar results were obtained for single real, single complex and double complex precision.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1999. Third edition.

[2] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. and Appl.*, 10(2):165–190, 1989.

[3] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. 2011. LAPACK Working Note 246.

[4] D. Becker, M. Baboulin, and J. Dongarra. Reducing the amount of pivoting in symmetric indefinite systems. University of Tennessee Technical Report ICL-UT-11-06 and INRIA Research Report 7621.

[5] Å. Björck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, 1996.

[6] 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 User's Guide*. SIAM, 1997.

[7] S. Blackford and J. Dongarra. Installation Guide for LAPACK. 1999. LAPACK Working Note 41, revised version 3.0.

[8] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numerical Analysis*, 8:639–655, 1971.

[9] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. 2006. In Proceedings of PARA 2006, Workshop on state-of-the art in scientific computing.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20:1573–1590, 2007.

[11] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. and Appl.*, 20(4):915–952, 1999.

[12] N. I. M. Gould, J. A. Scott, and Y. Hu. A numerical evaluation of sparse solvers for symmetric systems. *ACM Trans. Math. Softw.*, 33(2):10:1–10:32, 2007.

[13] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27(4):422–455, 2001.

[14] P. Hénon, P. Ramet, and J. Roman. On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. *In PPMA'05,* LNCS, 3911:1050–1057, 2005.

[15] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002. Second edition.

[16] Intel. *Math Kernel Library (MKL)*. http://www.intel.com/software/products/mkl/.

[17] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. 2006. LAPACK Working Note 178. Also available as University of Tennessee Technical Report UT-CS-06-581.

[18] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. 2009. LAPACK Working Note 220.

[19] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numerische Mathematik*, 6:405–409, 1964.

[20] U. of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*. 2010.

[21] D. S. Parker. Random butterfly transformations with applications in computational linear algebra. Technical Report CSD-950023, Computer Science Department, UCLA, 1995.

[22] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal.*, 23:158–179, 2006.

[23] R. D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comput.*, 35:817–832, 1980.
[24] P. E. Strazdins. A dense complex symmetric indefinite solver for the Fujitsu AP3000. Technical Report TR-CS-99-01, The Australian National University, 1999.
[25] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.
[26] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010.
[27] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011.