Parallel Computing 40 (2014) 213–223

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems

Marc Baboulin [a,*], Dulceneia Becker [b], George Bosilca [b], Anthony Danalis [b], Jack Dongarra [b]

[a] Laboratoire de Recherche en Informatique, Inria/University Paris-Sud, Orsay, France
[b] Innovative Computing Laboratory, University of Tennessee, Knoxville, USA

ARTICLE INFO

Article history:
Available online 28 December 2013

Keywords:
Randomized algorithms
Distributed linear algebra solvers
Symmetric indefinite systems
LDL^T factorization
PaRSEC runtime

ABSTRACT

Randomized algorithms are gaining ground in high-performance computing applications as they have the potential to outperform deterministic methods, while still providing accurate results. We propose a randomized solver for distributed multicore architectures to efficiently solve large dense symmetric indefinite linear systems that are encountered, for instance, in parameter estimation problems or electromagnetism simulations. The contribution of this paper is to propose efficient kernels for applying random butterfly transformations and a new distributed implementation combined with a runtime (*PaRSEC*) that automatically adjusts data structures, data mappings, and the scheduling as systems scale up. Both the parallel distributed solver and the supporting runtime environment are innovative. To our knowledge, the randomization approach associated with this solver has never been used in public domain software for symmetric indefinite systems. The underlying runtime framework allows seamless data mapping and task scheduling, mapping its capabilities to the underlying hardware features of heterogeneous distributed architectures. The performance of our software is similar to that obtained for symmetric positive definite systems, but requires only half the execution time and half the amount of data storage of a general dense solver.

1. Introduction

The last several years saw the development of randomized algorithms in high performance computing applications. This increased interest is motivated by the fact that the resulting algorithms are able to outperform deterministic methods while still providing very accurate results (*e.g.* random sampling algorithms that can be applied to least squares solutions or low-rank matrix approximation [1]). In addition to being easier to analyze, the main advantage of such algorithms is that they can lead to much faster solution by performing a smaller number of floating-point operations (*e.g.* [2]), or by involving less communication (*e.g.* [3]). As a result, they potentially allow domain scientists to address larger simulations.

However, to be of full interest for applications, these randomized algorithms must be able to exploit the computing capabilities of current highly distributed parallel systems, which can commonly achieve performance of more than one Tflop/s per node. Since randomized algorithms are supposed to be useful for very large problems, the main challenge for them is to exploit efficiently these distributed computing units and their associated memories. As a matter of fact, large-scale linear algebra solvers from standard parallel distributed libraries like ScaLAPACK [4] often suffer from expensive inter-node communication costs.

* Corresponding author. Tel.: +33 1 69 15 47 27.
E-mail addresses: marc.baboulin@inria.fr (M. Baboulin), dbecker7@eecs.utk.edu (D. Becker), bosilca@icl.utk.edu (G. Bosilca), danalis@icl.utk.edu (A. Danalis), dongarra@eecs.utk.edu (J. Dongarra).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.12.003

The advent of multicore processors has undermined the dominance of the SPMD programming style, reviving interest in more flexible approaches such as dataflow approaches. Indeed, several projects [5–9], mostly in the field of numerical linear algebra, revived the use of Directed Acyclic Graphs (DAG), as an approach to tackle the challenges of harnessing the power of multicore and hybrid platforms. In [10], an implementation of a tiled algorithm based on dynamic scheduling for the LU factorization on top of UPC (Unified Parallel C) is proposed. Gustavson et al. [11] uses a static scheduling for the Cholesky factorization on top of MPI to evaluate the impact of data representation structures. All these projects propose ad-hoc solutions to the challenging problem of harnessing all the computing capabilities available on today's heterogeneous platforms, solutions that do not expose enough flexibility to be generalized outside their original algorithmic design space. In the *PaRSEC* project (previously named *DAGuE* [12]) we address this problem in a novel way. Using a layered runtime system, we decouple the algorithm itself from the data distribution, as the algorithm is entirely expressed as flows of data, and from the underlying hardware, allowing the developer to focus solely on the algorithmic level without constraints regarding current hardware trends. The depiction of the algorithm uses a concise symbolic representation (similar to Parameterized Task Graph (*PTG*) proposed in [13]), requiring minimal memory for the DAG representation and providing extreme flexibility to quickly follow the flows of data starting from any task in the DAG, without having to unroll the entire DAG. As a result, the DAG unrolls on-demand, and each participant never evaluates parts of the DAG pertaining to tasks executing on other resources, thereby sparing memory and compute cycles. Additionally, the symbolic representation enables the runtime to dynamically discover the communications required to satisfy remote dependencies, on the fly, without a centralized coordination. Finally, the runtime provides a heterogeneous environment where tasks can be executed on hybrid resources based on their availability.

We illustrate in this paper how linear system solvers based on random butterfly transformations can exploit distributed parallel systems in order to address large-scale problems. In particular, we will show how our approach automatically adjusts data structures, mapping, and scheduling as systems scale up. The application we consider in this paper is the solution of large dense symmetric indefinite systems. Even though dense symmetric indefinite matrices are less common than sparse ones, they do arise in practice, for instance in parameter estimation problems, when considering the augmented system approach [14, p. 77], and in constrained optimization problems [15, p. 246].

For instance, the symmetric indefinite linear system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{1}$$

arises from the optimization problem

$$\min \frac{1}{2} \|r - b\|_2^2 \text{ subject to } A^T r = 0, \tag{2}$$

where x is the vector of Lagrange multipliers, $r = b - Ax$ is the residual, and I is the identity matrix.

Another class of applications is related to simulations in electromagnetics, when the 3-D Maxwell equations are solved in the frequency domain. The goal is to solve the Radar Cross Section (RCS) problem that computes the response of an object to the wave. In that case, the discretization by the Boundary Element Method (BEM) yields large dense symmetric complex systems which are non Hermitian [16].

Many of the existing methods do not exploit the symmetry of the system, which results in extra computation and extra storage. The problem size commonly encountered for these simulations is a few hundreds of thousands. When the matrix does not fit into the core memories of the computer, this may require the use of out-of-core methods that use disk space as auxiliary memory, which degrades performance (note that an option for very large size is iterative/multipole methods, for which the matrix does not need to be stored in memory). Our in-core solver will be an alternative to these existing methods by minimizing the number of arithmetical operations and data storage.

Symmetric indefinite systems are classically solved using a $\mathrm{LDL}^{\mathrm{T}}$ factorization

$$PAP^T = \mathrm{LDL}^{\mathrm{T}}, \tag{3}$$

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.

To our knowledge, there is no parallel distributed solver in public domain libraries (*e.g.* ScaLAPACK) for solving dense symmetric indefinite systems using factorization (3). This is why, in many situations, large dense symmetric indefinite systems are solved via LU factorization, resulting in an algorithm that requires twice as much, both arithmetical operations and storage than $\mathrm{LDL}^{\mathrm{T}}$. Moreover, as mentioned in [14], the pivoting strategy adopted in LAPACK [17] (based on the Bunch-Kaufman algorithm [18]) does not generally give a stable method for solving the problem (2), since the perturbations introduced by roundoff do not respect the structure of the system (1).

The approach we propose in this paper avoids pivoting thanks to a randomization technique initially described in [19] for the LU factorization. The main advantage of randomizing here is that we can avoid the communication overhead due to pivoting (this communication cost can represent up to 40% of the global factorization time depending on the architecture, and on the problem size [3]). The resulting distributed solver enables us to address the large dense symmetric indefinite systems

mentioned before and for which there is no implementation in ScaLAPACK. It is combined with an innovative runtime system that allows seamless data mapping and task scheduling on heterogeneous distributed architectures.

2. A distributed randomized algorithm

2.1. Theoretical background on Symmetric Random Butterfly Transformation (SRBT)

We recall in this section the main definitions and results related to the randomization approach that is used for symmetric indefinite systems. The randomization of the matrix is based on a technique initially described in [19,3] for general systems and applied in [20] for symmetric indefinite systems (we can find in [19] a small example with a 2-by-2 matrix that illustrates how randomization can be used instead of pivoting). The procedure to solve $Ax = b$, where A is symmetric, using a random transformation and the 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 random matrix U is chosen among a particular class of matrices called *recursive butterfly matrices* and the resulting transformation is referred to as **Symmetric Random Butterfly Transformation** (SRBT). The LDL^T factorization without pivoting (step 2 above) is performed using a tiled algorithm already described in [21].

We recall that a butterfly matrix is defined as any n-by-n matrix of the form:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix}, \tag{4}$$

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

A recursive butterfly matrix U of size n and depth d is a product of the form

$$U = U_d \times \cdots \times U_1, \tag{5}$$

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

$$U_k = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_{2^{k-1}} \end{pmatrix} \tag{6}$$

each B_i being a butterfly matrix of size $n/2^{k-1}$. In particular U_1 is a butterfly as defined in formula (4). Note that this definition requires that n is a multiple of 2^d which can always be obtained by "augmenting" the matrix A with additional 1's on the diagonal.

We generate the random diagonal values used in the butterflies as $e^{\rho/10}$, where ρ is randomly chosen in $\left[-\frac{1}{2}, \frac{1}{2}\right]$. This choice is suggested and justified in [19] 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

$$e^{-1/20} \leqslant r_i \leqslant e^{1/20},$$

We use the POSIX function *random()* for generating the pseudo-random number sequence and we always initialize the seed to zero (via a call to *srandom()*), for reproducibility reasons. Using these random values, it is shown in [21] that the 2-norm condition number of the randomized matrix A_r verifies

$$\mathrm{cond}_2(A_r) \leqslant 1.2214^d \mathrm{cond}_2(A) \tag{7}$$

and thus is kept almost unchanged by the randomization for small values of d. We recall also that the LDL^T algorithm without pivoting is potentially unstable [22, p. 214], due to a possibly large growth factor. We can find in [19] explanations about how recursive butterfly transformations 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 [22, p. 232]. We can find in [20] numerical experiments on a collection of matrices where SRBT + LDL^T is more accurate than non-pivoting LDL^T and has similar accuracy as the Bunch–Kaufman algorithm.

A butterfly matrix and a recursive butterfly matrix can be stored in a packed storage using a vector and a matrix, respectively. It is also shown in [21] that the number of operations involved in randomizing A by an SRBT of depth d is about $2dn^2$. We will consider a number of recursions d such that $d < \log_2 n \ll n$. Numerical tests described in [20] and performed on a collection of matrices from the Higham's Matrix Computation Toolbox [22] have shown that, in practice, $d = 2$ enables us to achieve satisfying 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. Then the computation of $U^T b$ and Uy can be performed in $\mathcal{O}(dn)$ flops and, for small values of d, can be neglected compared to the $\mathcal{O}(n^3)$ cost of the factorization.

2.2. Randomization kernels

As described in Section 2.1, the original linear system can be transformed using U of Eq. (5) such that

$$Ax = b \equiv \underbrace{U^T A U}_{A_r}\ \underbrace{U^{-1}x}_{y} = \underbrace{U^T b}_{c}. \tag{8}$$

For simplicity, n (order of matrices U and A) is supposed to be a multiple of 2^d hereafter (if not the system is augmented with additional 1's on the diagonal).

In order to transform the system $Ax = b$, two kernels are required:

$$A_r = U^T A U, \tag{9}$$

$$c = U^T b. \tag{10}$$

After solving $A_r y = c$, x is obtained by

$$x = Uy. \tag{11}$$

Eqs. (10) and (11) can be taken as particular cases of Eq. (9), where the U matrix on the right side of A is an identity matrix. This means that the data dependencies described in what follows are similar but simpler for Eqs. (10) and (11). Since the implementation uses the same principle for all three operations, only $U^T A U$ is detailed.

Using the definition of the recursive matrix U given in Eq. (5), the randomized matrix A_r can be expanded as

$$A_r = U_1^T \times U_2^T \times \cdots \times U_d^T \times A \times U_d \times \cdots \times U_2 \times U_1.$$

Note that U_i is a sparse matrix, with sparsity pattern as shown in Fig. 1, and that the matrix product of the U_i's results in a different sparsity pattern, which depends on the number of levels of recursion. To avoid storing the product of U_i and to maintain the symmetry, the computation can be performed by recursively computing

$$A_r^{(i-1)} = U_i^T A^{(i)} U_i,$$

where $A^{(d)} = A$ and $A_r = A_r^{(0)}$.

It can be observed that, for each level, $U_i^T A^{(i)} U_i$ can be written as blocks given by $B_i^T A_{ij} B_j$. For instance, for the second level

$$U_2^T A^{(2)} U_2 = \begin{bmatrix} B_1^T & \\ & B_2^T \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_1 & \\ & B_2 \end{bmatrix} = \begin{bmatrix} B_1^T A_{11} B_1 & B_1^T A_{12} B_2 \\ B_2^T A_{21} B_1 & B_2^T A_{22} B_2 \end{bmatrix}.$$

Hence, the so-called *core kernel* of a random butterfly transformation is given by

$$B_i^T A_{ij} B_j, \tag{12}$$

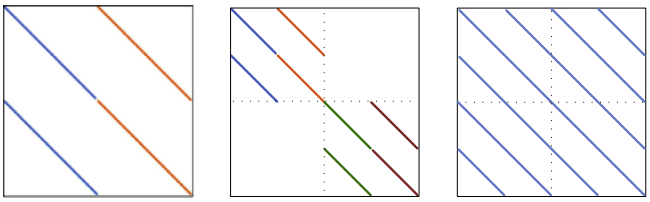
where A_{ij} is a block of A and B_* is a random butterfly matrix, both of size $m \times m$. The block A_{ij} can either be symmetric (diagonal block, *i.e.* $i = j$) or non-symmetric (off-diagonal block, *i.e.* $i \neq j$).

Recalling that B has a well defined structure

$$B = \begin{bmatrix} R & S \\ R & -S \end{bmatrix} \quad \text{and} \quad B^T = \begin{bmatrix} R & R \\ S & -S \end{bmatrix},$$

where R and S are diagonal matrices, and given that A_{ij} is divided into four submatrices of same size, such as

$$A_{ij} = \begin{bmatrix} TL & TR \\ BL & BR \end{bmatrix}$$



(a) U_1 (b) U_2 (c) $U_2 \times U_1$

Fig. 1. Sparsity pattern of matrix U.

and that

$$W_{TL} = (TL + BL) + (TR + BR),$$
$$W_{BL} = (TL - BL) + (TR - BR),$$
$$W_{TR} = (TL + BL) - (TR + BR),$$
$$W_{BR} = (TL - BL) - (TR - BR).$$

Eq. (12) can be written as

$$B_i^T A_{ij} B_j = \begin{bmatrix} R \cdot W_{TL} \cdot R & R \cdot W_{TR} \cdot S \\ S \cdot W_{BL} \cdot R & S \cdot W_{BR} \cdot S \end{bmatrix}.$$

Note that only the signs differ in calculating each W_*. Hence, all four cases can be generalized as

$$W = (TL \circ BL) \circ (TR \circ BR), \tag{13}$$

where the operator $\circ$ denotes an addition or a subtraction. Eq. (13) shows that each matrix W_* depends on all four submatrices of A_{ij}. More specifically, any given element of W depends on four elements of A. Therefore, the data dependencies among elements could be depicted as:

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 2 & 4 & 5 & 2 & 4 & 5 \\ 3 & 5 & 6 & 3 & 5 & 6 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 2 & 4 & 5 & 2 & 4 & 5 \\ 3 & 5 & 6 & 3 & 5 & 6 \end{bmatrix}}_{\text{Symmetric}} \quad \underbrace{\begin{bmatrix} 1 & 4 & 7 & 1 & 4 & 7 \\ 2 & 5 & 8 & 2 & 5 & 8 \\ 3 & 6 & 9 & 3 & 6 & 9 \\ 1 & 4 & 7 & 1 & 4 & 7 \\ 2 & 5 & 8 & 2 & 5 & 8 \\ 3 & 6 & 9 & 3 & 6 & 9 \end{bmatrix}}_{\text{General}}$$

where same numbers means these elements depend on each other. For the symmetric case, the strictly upper triangular part is not calculated, and for this work, not stored either. Hence, elements above the diagonal ($i < j$) must be computed as their transpose, *i.e.* A_{ij} is read and written as A_{ji}.

This can be extended to blocks, or tiles, to comply with the tiled LDL^T algorithm presented in [21]. If each number above is taken as a tile (or block), the data dependencies can be sketched as in Fig. 2. The actual computation can be done in different ways; details of the approach adopted in this work are given in Section 2.3.

2.3. Implementing SRBT randomization using PaRSEC

We use the *PaRSEC* [12] runtime system to implement the SRBT transformation and the subsequent LDL^T factorization with no pivoting. *PaRSEC* employs a dataflow programming and execution model to provide a dynamic platform that can address the challenges posed by distributed heterogeneous many-core resources. The central component of the system, the runtime, combines the task and dataflow information of the source program with supplementary information provided by the user – such as the distribution of data, or hints about the importance of different tasks – and orchestrates the execution of the tasks on the available hardware. We suggest to the interested reader to delve into the details of the *PaRSEC*

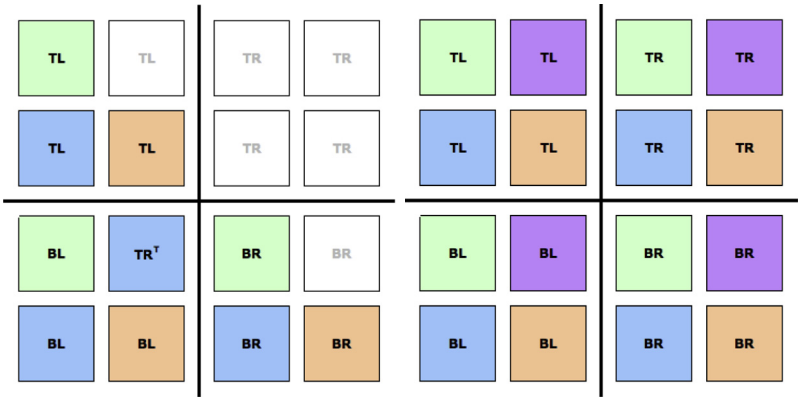


Fig. 2. SRBT core kernel ($B_i^T A_{ij} B_j$), data dependencies among tiles of A_{ij}, given by matching colors.

programming environment through [23]. *PaRSEC* enables the algorithm developer to ignore all details that do not relate to the algorithm being implemented, including system and hardware properties. *PaRSEC* takes as input:

1. a set of kernels that will operate on the user data, in the form of compiled functions inside a library or object files,
2. an algebraic description of the data-flow edges between the tasks that will execute the kernels, in a specific format that we refer to as the JDF (for Job Data Flow).

Given this input, *PaRSEC* will automatically handle the communication, task placement, memory affinity, load balancing, work stealing, and all other aspects of high performance parallel programming that are critical for achieving good performance, but constitute a nuisance to an algorithm developer.

In our implementation over *PaRSEC*, the flow of data is maintained with the most basic unit of data being not a single element, but a tile ($nb * nb$ elements). A tile can be a block of contiguous memory, or more generally a block of memory containing data separated by strides in any way that an MPI datatype can describe. Moreover, we assume that the input matrix, given to us by the calling application, is organized according to the format used in the PLASMA [24] library. That is, tiles of contiguous data are stored in a column major order. As discussed in Section 2.2 the data dependencies of the SRBT are those dictated by Eq. (13) and exhibit a symmetry. For SRBT at depth 1, the symmetry is about the horizontal and vertical lines that pass through the center of the matrix. For SRBT at depth 2, each quarter of the matrix is independent, and the symmetry is about the center of each quarter. Clearly, if the tile size is such that the number of tiles is not divisible by 2^d, then the symmetries will cause data dependencies that will cross the tile boundaries.

As an example, Fig. 4 depicts the case where a matrix is divided in 3×3 tiles. Eq. (13) demands that each element of each quarter of the matrix is processed with the corresponding elements of the other three quarters of the matrix. However, here, the number of tiles is not divisible by the number of blocks for a butterfly level 1, and therefore processing any tile of the top left quarter requires elements from the other quarters that belong to multiple tiles. The largest blocks of memory that do not cross tile boundaries and have matching blocks are those depicted with the different colors in Fig. 4. Clearly, for higher levels of the SRBT transformation, the situation can become even more complicated, as is depicted in Fig. 5, where the matrix is organized in 5×5 tiles which is neither divisible by 2^1, nor 2^2.

While choosing a tile size that creates this situation will unequivocally lead to performance degradation (since there will be four times as many messages of significantly smaller size), our implementation of the SRBT transformation supports arbitrary tile sizes and number of tiles. There are two ways to implement this with *PaRSEC*. The first is to transfer, for every tile, all tiles that contain matching data, perform the core kernels on the consolidated data, and then transfer the results back to the original location of the tiles. The second way is to logically subdivide the tiles into segments that do not cross tile boundaries (as depicted in Figs. 4 and 5 with different colors), and operate on this logical segment space. We chose to implement the latter approach, as it does not perform unnecessary data transfers, since it does not transfer the parts of the tiles that will not be used. We made this choice because the computation performed by SRBT is relatively light weight ($O(n^2)$) and thus it would be difficult to hide the additional overhead of the unnecessary communication.

Besides the performance concerns, there is a correctness concern. As discussed in Section 2.2, regardless of the size of a contiguous block, all four W_{TL}, W_{BL}, W_{TR}, and W_{BL} require the same data (TL, BL, TR and BR) and also overwrite this data. Clearly, the computation of all four W_* must be performed before any original data is overwritten, using temporary memory, in a double buffering fashion. *PaRSEC* provides mechanisms for handling the allocation and deallocation of temporary buffers of size and shape that match a given MPI datatype, simplifying the process.

Fig. 3 depicts a snippet of the JDF file that *PaRSEC* will use as input to guide the execution of the different tasks and the necessary communication between them. As is implied by its name, task *Diag* will process the segments that lie on the diagonal of the matrix. The execution space of the task goes up only to the middle of the matrix, since, due to the translation symmetry we mentioned earlier, the four quarters of the matrix need to be processed together to generate the result. For this reason, the *Diag* task communicates with the Reader_$*$ tasks to receive the corresponding segments from the three quarters of the matrix, top left (TL), bottom left (BL) and bottom right (BR) and sends back the resulting segments to the Writer_$*$ tasks. Clearly, since the matrix is symmetric and this task is processing diagonal blocks, there is no need to fetch the top right block, since it contains the same data as the bottom left block, transposed.

In addition to specifying which tasks constitute communication peers, the JDF specifies the type of data to be exchanged. This can be done with a predetermined, static type, or – as is the case here – using C code that will compute the appropriate type, at runtime, based on parameters of the task, such as "`i`" that identify the segment that is being processed. In this example we make call to a C function called `seg2type()`.

The reason for calculating the datatype at runtime becomes clearer if we consider an example from Fig. 5. In particular, let us look at the top left tile and consider that we are processing the (blue colored) segment that is in the bottom left of the tile and is short and wide. Fig. 7 shows a magnified version of this tile (with fainter colors for readability). Because of the data dependencies of SRBT, the data of the blue segment have to be considered independently of the rest of the tile. However, as shown in Fig. 7, the segment itself is not stored contiguously in memory, but it is stored within a tile. Nevertheless, given the start and end points of the segment, the length of a contiguous block of data, and the stride between blocks of data, we can process the data of the segment in a systematic way and transfer them over the network without wasting bandwidth. The need for dynamic discovery of the type arises because, as can be derived by the multitude of segment shapes and sizes in

```
Diag(i)
  /* Execution space */
  i = 0 .. mt/2-1

  : A(i,i)

  /* Atl: A from Top Left */
  RW  Atl <- A Reader_TL(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]
          -> A Writer_TL(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]

  /* Abl: A from Bottom Left */
  RW  Abl <- A Reader_BL(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]
          -> A Writer_BL(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]

  /* Abr: A from Bottom Right */
  RW  Abr <- A Reader_BR(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]
          -> A Writer_BR(i,i) [ inline_c %{ return seg2type(descA, i, i); %}]

BODY
...
END
```

Fig. 3. Part of the *PaRSEC* input file, JDF, containing the description of the *Diag* task.

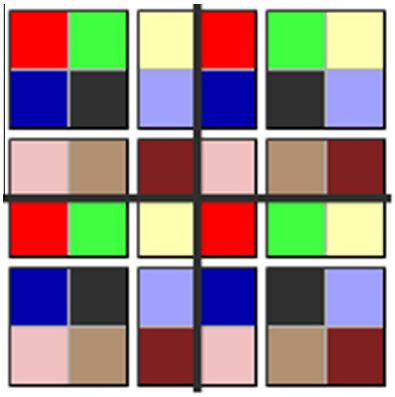


Fig. 4. Tile count (3×3) non divisible by 2^1.

Fig. 5, each segment can potentially have a different type. Since the types depend on the matrix size and the size (or count) of tiles in the matrix, it is impossible to assign those types statically.

Fig. 6 shows the Direct Acyclic Graph (DAG) that describes the execution of a level 1 SRBT for a matrix with 4×4 tiles. We can see that the three tasks (*Diag (0)*, *Diag (1)* and *Lower (1,0)*) that process the top left quarter of the matrix do not depend on each other and can therefore run independently. Also, the DAG clearly shows the communication edges between the processing tasks and the corresponding *Reader* and *Writer* tasks that will fetch the data from the other quarters of the matrix. Interestingly, the DAG of Lower (1,0) shows that data is read from the bottom left quarter twice (*Reader_BL (1,0)* and *Reader_BL (0,1)*). This is because the matrix is symmetric and lower triangular, so all data that would belong to the top right quarter are instead found at the transpose location in the bottom left quarter.

Unlike the SRBT transformation, the LDL^T factorization was rather straightforward to implement in *PaRSEC*. In particular, the operation exists as a C code in the PLASMA library, similar to the pseudocode shown in [21]. Because of that, we were able to use the kernels verbatim from the PLASMA library and generate the JDF automatically using the serial code compiler of *PaRSEC* (followed by hand-optimizations). The DAG of this factorization on a 4×4 tile matrix is depicted in Fig. 8. The mapping of tasks onto nodes obeys the "owner computes" rule. Each task will modify some data. It is thus preferable if it executes on the node where this data resides. The data distribution follows a two-dimensional block cyclic pattern. The actual data placement depends on the number of nodes and the process grid – both user configurable parameters. Therefore, the mapping of tasks onto nodes is decided after the run-time has established the data placement for a given execution; so each task is placed such that it has local access to the data it modifies.

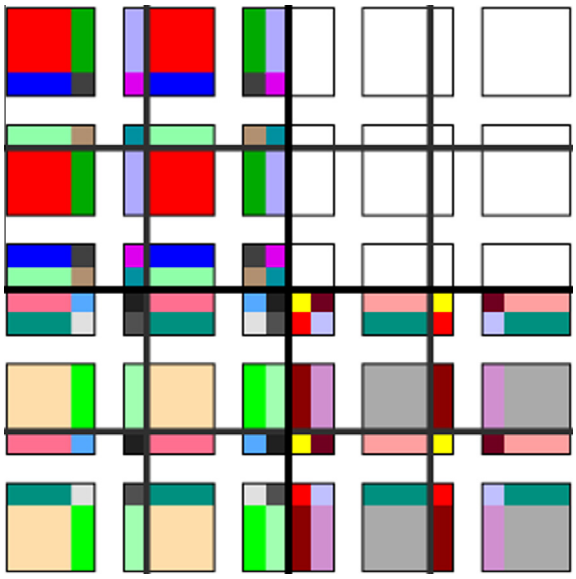


Fig. 5. Tile count (5×5) non divisible by 2^2, or 2^1.

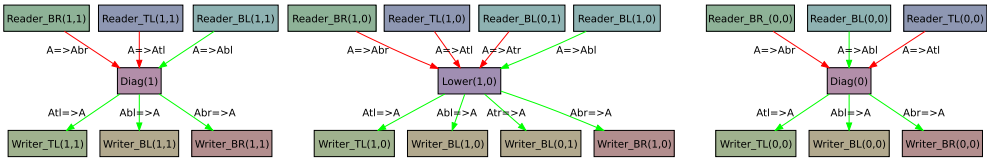


Fig. 6. SRBT DAG for 4×4 tile matrix.

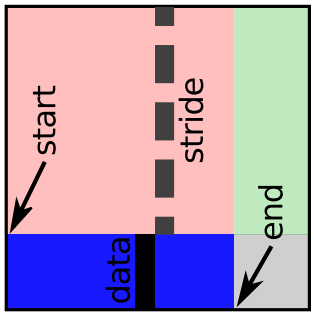


Fig. 7. Segment data inside a tile.

3. Performance results

The performance experiments were performed on a 16 node cluster with Infiniband 20G. Each node features two NUMA Nehalem Xeon E5520 processors clocked at 2.27 GHz, with four cores each and 2 GB of memory (4 GB total per node). The Infiniband NICs are connected through PCI-E and the operating system is Linux 2.6.31.6-2. All experiments presented in this paper were performed using double precision arithmetics.

Fig. 9 shows the results of problem scaling when all $16 \times 8 = 128$ cores of the cluster are used. We plot the LDL^T with and without the SRBT transformation and we separate, into two different curves, the cases where the number of tiles is divisible by 2^2 ($d = 2$ since we perform two levels of SRBT) and the cases where it is non-divisible. Also, the figure includes the execution time of Cholesky and LU on a matrix of the same size. These curves help give some context to our measurements. Cholesky can only operate on a subset of the matrices that SRBT + LDL^T can operate on, but it consists of a lower bound in the execution time of SRBT + LDL^T, since it performs slightly less operations and communication (because there is no randomization). On the other hand, LU is a general factorization algorithm, but in terms of execution time it provides us with an

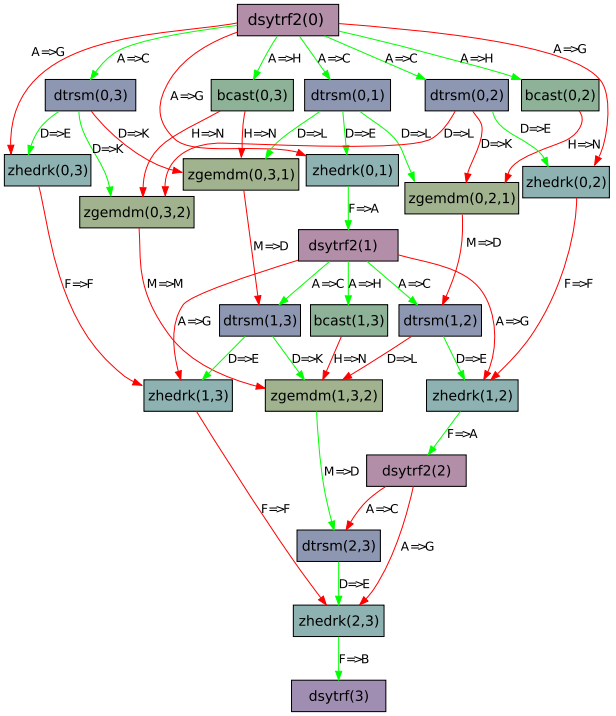
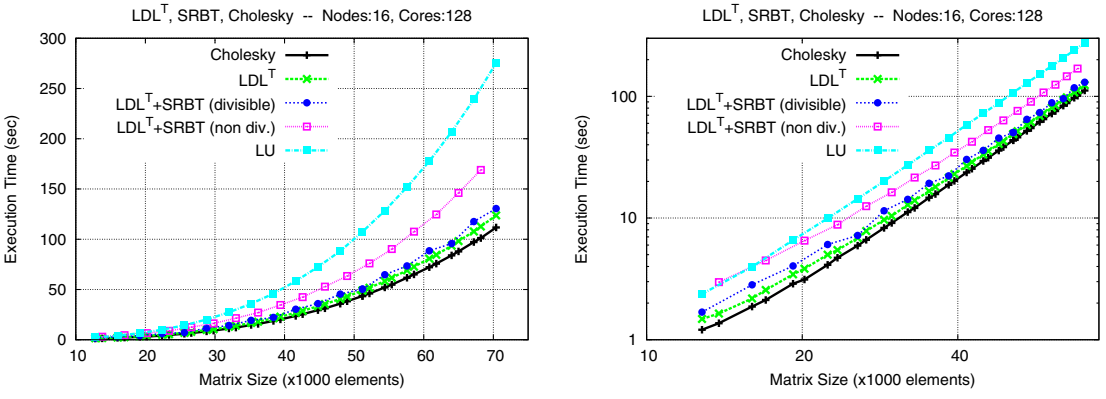


Fig. 8. LDL^T DAG for 4×4 tile matrix.



(a) Execution Time (linear) (b) Execution Time (logarithmic)

Fig. 9. Problem scaling comparison of LDL^T+SRBT against Cholesky on a $16 \times 8 = 128$ core cluster.

upper bound, since it performs significantly more operations. We provide both a linear and a logarithmic plot of the data, since the former tends to exaggerate the larger values at the expense of the small ones and vice versa.

In addition, we show, in Fig. 10, the performance of LDL^T+SRBT in Gflops per second. In this figure we have excluded LU, since it algorithmically performs a different number of operations, but we provide the value of the "practical peak". We estimate this peak by measuring the performance of DGEMM (double precision, general, matrix–matrix multiply) operations on the CPU cores without any communication, or synchronization between them. From these graphs we can make the following observations:

1. LDL^T, even without the SRBT, is slightly slower than Cholesky because of (a) additional synchronization (due to the scaling of the diagonal at the end of the factorization) and (b) the broadcasting of the diagonal elements during the factorization.
2. When the number of tiles is divisible by 2^d, SRBT adds a small overhead, which is especially small when the number of tiles is divisible by 32 ($2*$number of processors) since that helps with the process-to-tile mapping.

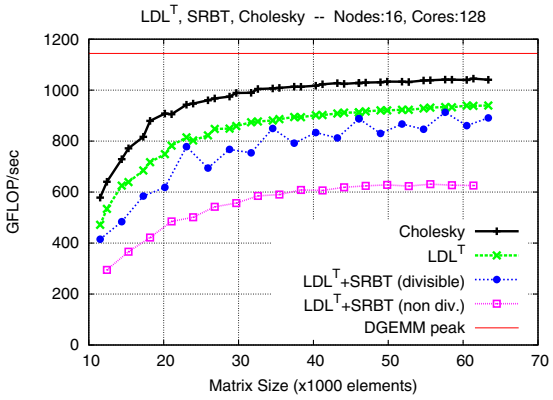


Fig. 10. Problem scaling (Flops) on a $16 \times 8 = 128$ core cluster.

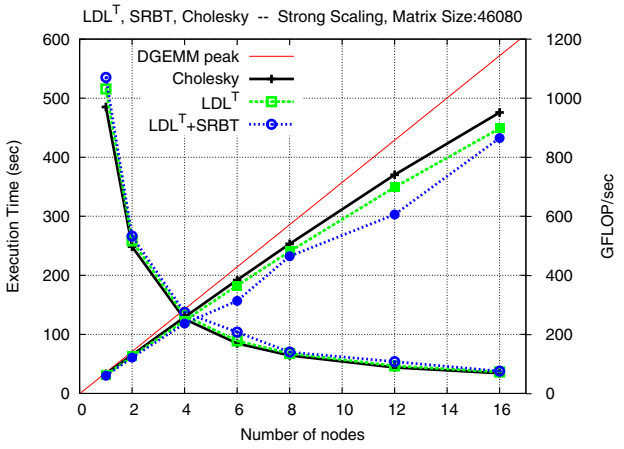


Fig. 11. Strong Scaling using a 46080×46080 matrix.

3. When the number of tiles is not divisible by 2^d, SRBT adds significant overhead.
4. Even in the worst case, SRBT + LDL^T is significantly faster than LU, and the difference in execution time grows with the problem size.

The first conclusion that can be drawn from these observations is that for matrices that are symmetric, but not positive definite, there is a clear advantage in using SRBT+LDL^T instead of LU in terms of performance. Second, it is important, for performance, to have a number of tiles that is divisible by 2^d, *i.e.*, 4, and it is even better if the number of tiles is divisible by the number of nodes that participate in the solution. Since the matrix tile size (and thus the number of tiles in the matrix) is a parameter chosen by the application that calls the *PaRSEC* implementation of the algorithm, we provide this insight as a suggestion to the application developers who might wish to use our code.

Fig. 11 presents strong scaling results for a moderate problem size ($N = 46$ K). Both execution time and operations per second are depicted in the graph. Clearly, the algorithm exhibits good performance, similar to that of Cholesky and rather close to the practical peak of the machine. Interestingly, the performance drops for the cases when 6 and 12 nodes were used. The reason for this behavior is that in these cases, the number of nodes does not divide the number of tiles, and thus the (2D block-cyclic) mapping of tiles to nodes creates additional communication.

4. Conclusion

We described an innovative implementation of a randomized algorithm for solving large dense symmetric indefinite systems on clusters of multicores. Such solvers have immediate application in several domains, such as the Radar Cross Section problems in Electromagnetism. Existing direct methods, generally based on LU solvers, have a computational and storage cost increased by a factor of two compared to the approach proposed in this paper. The randomization step, used in our approach, allows us to eliminate the pivoting completely and therefore reduce the critical path. Instead, a

communication pattern with higher parallelism, which takes place prior to the factorization, is used to apply a preconditioner to the original matrix. The cost of this randomization step is significantly lower than the overall cost of pivoting. The randomization and the LDL^T algorithms have been implemented using the *PaRSEC* runtime system. Due to the tile data layout used by the proposed algorithm, the runtime is capable of enforcing a strict data locality scheduling strategy, resulting in high cache locality. As a result the performance obtained is similar to the Cholesky solver, and close to the practical peak of the platform. We showed that for symmetric indefinite systems, the proposed algorithm, together with an efficient runtime, is a better performing approach than LU on these systems, and therefore a well suited replacement for it. More generally this illustrates how randomized algorithms can enhance performance by decreasing the amount of communication in linear algebra algorithms.

Acknowledgments

The research conducted in this project has been supported by the National Science Foundation through the grant #1244905 and the Université Paris-Sud – Inria research chair DRH/SA/2011/92.

References

[1] M.W. Mahoney, Randomized algorithms for matrices and data, Found. Trends Mach. Learn. 3 (2) (2011) 123–224.
[2] A. Avron, P. Maymounkov, S. Toledo, Blendenpick: Supercharging LAPACK's least-squares solvers, SIAM J. Sci. Comput. 32 (2010) 1217–1236.
[3] M. Baboulin, J. Dongarra, J. Herrmann, S. Tomov, Accelerating linear system solutions using randomization techniques, ACM Trans. Math. Softw. 39 (2) (2013) 8.
[4] 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, R. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, 1997.
[5] A. Buttari, J.J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, S. Tomov, The impact of multicore on math software, in: Eighth International Workshop, PARA, Applied Parallel Computing. State of the Art in Scientific Computing, Lecture Notes in Computer Science, vol. 4699, Springer, 2006, pp. 1–10.
[6] E. Chan, F.G. Van Zee, P. Bientinesi, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks, in: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2008, pp. 123–132.
[7] E. Agullo, J. Demmel, J.J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects, J. Phys.: Conf. Ser. 180 (2009).
[8] R. Dolbeau, S. Bihan, F. Bodin, HMPP: a hybrid multi-core parallel programming environment, in: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), 2007.
[9] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency Comput.: Pract. Experience 23 (2) (2011) 187–198.
[10] P. Husbands, K.A. Yelick, Multi-threading and one-sided communication in parallel LU factorization, in: B. Verastegui (Ed.), Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10–16, 2007, ACM Press, Reno, Nevada, USA, 2007.
[11] F.G. Gustavson, L. Karlsson, B. Kågström, Distributed SBP Cholesky factorization algorithms with near-optimal scheduling, ACM Trans. Math. Softw. 36 (2) (2009) 1–25.
[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J.J. Dongarra, DAGuE: A generic distributed DAG engine for high performance computing, Parallel Computing 38 (1–2) (2012) 37–51.
[13] M. Cosnard, E. Jeannot, Automatic parallelization techniques based on compact DAG extraction and symbolic scheduling, Parallel Process. Lett. 11 (2001) 151–168.
[14] A. Björck, Numerical Methods for Least Squares Problems, SIAM, Philadelphia, 1996.
[15] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, 2000.
[16] J.-C. Nédélec, Acoustic and electromagnetic equations, Integral Representations for Harmonic Problems, vol. 144, Springer-Verlag, New-York, 2001.
[17] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, third ed., SIAM, Philadelphia, 1999.
[18] J.R. Bunch, L. Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Math. Comput. 31 (1977) 163–179.
[19] D.S. Parker, Random butterfly transformations with applications in computational linear algebra, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.
[20] D. Becker, M. Baboulin, J. Dongarra, Reducing the amount of pivoting in symmetric indefinite systems, in: R. Wyrzykowski et al. (Eds.), Ninth International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), Lecture Notes in Computer Science, vol. 7203, Springer-Verlag, Heidelberg, 2012, pp. 133–142.
[21] M. Baboulin, D. Becker, J. Dongarra, A parallel tiled solver for dense symmetric indefinite systems on multicore architectures, in: Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012), 2012, pp. 14–24.
[22] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM, Philadelphia, 2002.
[23] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, J. Dongarra, Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach, in: Scalable Computing and Communications: Theory and Practice, John Wiley & Sons, 2013, pp. 699–733.
[24] U. of Tennessee, PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3, 2010.