On Algorithmic Variants of Parallel Gaussian Elimination: Comparison of Implementations in Terms of Performance and Numerical Properties [*]

Simplice Donfack
EECS Department
University of Tennessee
Knoxville, TN, USA
sdonfack@eecs.utk.edu

Jack Dongarra
EECS Department
University of Tennessee
Knoxville, TN, USA
dongarra@eecs.utk.edu

Mathieu Faverge
IPB ENSEIRB-Matmeca
Inria Bordeaux Sud-Ouest
Bordeaux, France
mathieu.faverge@inria.fr

Mark Gates
EECS Department
University of Tennessee
Knoxville, TN, USA
mgates3@eecs.utk.edu

Jakub Kurzak
EECS Department
University of Tennessee
Knoxville, TN, USA
kurzak@eecs.utk.edu

Piotr Luszczek
EECS Department
University of Tennessee
Knoxville, TN, USA
luszczek@eecs.utk.edu

Ichitaro Yamazaki
EECS Department
University of Tennessee
Knoxville, TN, USA
iyamazak@eecs.utk.edu

ABSTRACT

Gaussian elimination is a canonical linear algebra procedure for solving linear systems of equations. In the last few years, the algorithm received a lot of attention in an attempt to improve its parallel performance. This article surveys recent developments in parallel implementations of the Gaussian elimination. Five different flavors are investigated. Three of them are based on different strategies for pivoting: partial pivoting, incremental pivoting, and tournament pivoting. The fourth one replaces pivoting with the Random Butterfly Transformation, and finally, an implementation without pivoting is used as a performance baseline. The technique of iterative refinement is applied to recover numerical accuracy when necessary. All parallel implementations are produced using dynamic, superscalar, runtime scheduling and tile matrix layout. Results on two multi-socket multicore systems are presented. Performance and numerical accuracy is analyzed.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices; G.4 [**MATHEMATICAL SOFTWARE**]: Parallel and vector implementations

[*]This work was supported in part by the U. S. Department of Energy, the National Science Foundation, and the Intel Corporation.

General Terms

Algorithms, Implementation

Keywords

Gaussian elimination, LU factorization, parallel, shared memory, multicore

1. INTRODUCTION

Gaussian elimination has a long history that can be traced some 2000 years back [22]. Ever since, dense systems of linear equations have been a critical cornerstone for some of the most compute intensive applications. Any improvement in the time to solution for these dense linear systems has a direct impact on the execution time of numerous applications. A short list of domains directly using dense linear equations to solve some of the most challenging problems our society faces are: airplane wing design, radar cross-section studies, flow around ships and other off-shore constructions, diffusion of solid bodies in a liquid, noise reduction, and diffusion of light by small particles.

Computational experiments of self-sustaining fusion reactions could give us an informed perspective on how to build a device capable of producing and controlling the high performance [4]. Modeling the heating response of plasma due to radio frequency waves in the fast wave time scale leads to solving the generalized Helmholtz equation. The time harmonic terms of effective approximations of the electric field, magnetic field, and distribution function as a time-averaged equilibrium satisfy the equation. The Scientific Discovery through Advanced Computing project (SciDAC) Numerical Computation of Wave Plasma-Interactions in Multi-dimensional Systems developed and implemented a simulation code that gives insight into how electromagnetic waves can be used for driving current flow, heating and controlling instabilities in the plasma. The code is called AORSA [34–36] and stands for All ORders Spectral Algorithm. The resulting computation requires a solution of a system of linear equations exceeding half a million unknowns and the fastest method is LU factorization through Gaussian elimination with partial pivoting [6].

The electromagnetic community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem – a signal of fixed frequency bounces off an object; the goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the method of moments [30, 49]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the panel methods [31, 32], so named from the quadrilaterals that discretize and approximate a structure such as an airplane. Generally, these methods are called boundary element methods. Use of these methods produces a dense linear system of size $\mathcal{O}(N)$ by $\mathcal{O}(N)$, where N is the number of boundary points (or panels) being used. It is not unusual to see size $3N$ by $3N$, because of three physical quantities of interest at every boundary element. A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution. The builders of stealth technology who are interested in radar cross-sections are using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian. Another major source of large dense linear systems is problems involving the solution of boundary integral equations [19]. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in $\mathcal{O}(n^3)$ variables is replaced by a dense problem in $\mathcal{O}(n^2)$.

This article is organized as follows: at the beginning the motivation for this survey is given in Section 2; then the related work is discussed in Section 3; the original contribution of this work is described in Section 4; Section 5 presents the algorithms evaluated here and Section 6 gives implementation details; Section 7 contains the performance and numerical results of the experiments. Finally, Section 8 concludes the article and Section 9 shows potential future extensions of this work.

2. MOTIVATION

The aim of this article is to give exhaustive treatment to both performance and numerical stability of various pivoting strategies that have emerged over the past few years to cope with the need for increased parallelism in the face of the paradigm shifting switch to multicore hardware [7]. Rather than focusing on multiple factorizations and performance across multiple hardware architectures [3], we switch our focus to multiple pivoting strategies and provide uniform treatment for each, while maintaining sufficient hardware variability to increase the meaningful impact of our results

Ever since the probabilistic properties of partial pivoting [48] were established, the community has fully embraced the method despite the high upper bound on pivot growth that it theoretically can incur [20]: $\mathcal{O}(2^n)$ versus a much more acceptable $\mathcal{O}(n^3)$ provided by complete pivoting [21]. Probabilistically, similar results have been proven for no-pivoting LU [52], which lead to include it in this survey, but complete lack of any pivoting strategy is discouraged for practical applications.

3. RELATED WORK

Incremental pivoting [9] has its origins in pair-wise pivoting [46], both of which have numerical issues relating to the pivot growth factor [1, 9, 45]. But they offer a reduction of dependencies between tasks that aids parallelism that has become so important with the introduction and current proliferation of multicore hardware.

The partial pivoting code that we use for evaluation in this article is based on the parallel panel factorization that uses a recursive formulation of LU [15, 25]. This recent implementation was introduced to address the bottleneck of the panel computation [16, 18], and has been successfully applied to matrix inversion [17]. However, it should not be confused with a globally recursive implementation based on column-cyclic distribution [41]. Neither it is similar to a non-recursive parallelization effort [10] that focuses only on cache-efficient implementation of the existing Level 1 BLAS kernels rather than using the recursive formulation that was used in this article's experiments.

With incremental pivoting, the numerical quality of the solution does not match that of the standard partial pivoting scheme in the panel factorization because it has to be replaced by a form of pair-wise pivoting [46], which is related to an updating-LU for out-of-core computations [53] when blocking of computations has to be done for better performance. It has resurfaced in the form of what is now called the pivoting incremental strategy [9] that allows pairwise treatment of tiles and, as a consequence, reduces dependencies between tasks and aids parallelism. This causes the magnitude of pivot elements to increase substantially, which is called the pivot growth factor, and results in rendering the factorization numerically unstable [1, 9, 45].

A probabilistic technique called the Partial Random Butterfly Transformation (PRBT) is an alternative to pivoting and may, in a sense, be considered a preconditioning step that renders pivoting unnecessary. It was originally proposed by Parker [44] and then applied in practice with adaptations by Baboulin et al [5], that limited the recursion depth without compromising the numerical properties of the method.

The tournament pivoting originated in CALU [11, 14, 23] – Communication-Avoiding LU. The main design goal for this new pivoting scheme was to attain the minimum bounds on the amount of data communicated and the number of messages exchanged between the computing processors. The way to achieve these bounds was to minimize the communication that occurs during the factorization of the panel by performing redundant computations. The extra operations do not hurt the scheme in the global sense because they only account for an increase in the constant for a lower term: $\mathcal{O}(n^2)$, while the highest order term – $\mathcal{O}(n^3)$ – remains unchanged. In terms of stability, CALU could potentially cause a greater pivot growth, which increases exponentially with the amount of parallelism. Unlike the partial pivoting, however, the bounds might not be attainable in practice.

Similar study was published before [3] but it was mostly focused on performance across a wide range of hardware architectures. It featured results for the main three of the one-sided factorization schemes: Cholesky, QR, and LU. No algorithmic variants for a particular method were considered.

4. ORIGINAL CONTRIBUTION

The unique contribution of this survey is in implementing all the algorithms, being compared using the same framework, the same data layout, and the same set of parallel layout translation routines, as well as the same runtime scheduling system. This allows for gaining a level of insight into the trade-offs of the different methods that one could not reach by comparing published data for different implementations in different environments.

5. ALGORITHMS

5.1 Partial Pivoting

The LAPACK block LU factorization is the main point of reference here, and LAPACK naming convention is followed. The LU factorization of a matrix A has the form

$$PA = LU,$$

where L is a unit lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix. The LAPACK algorithm proceeds in the following steps: Initially, a set of NB columns (*the panel*) is factored and a pivoting pattern is produced (implemented by the `DGETF2` routine). Then the elementary transformations, resulting from the panel factorization, are applied in a block fashion to the remaining part of the matrix (*the trailing submatrix*). This involves swapping of up to NB rows of the trailing submatrix (`DLASWP`), according to the pivoting pattern, application of a triangular solve with multiple right-hand-sides to the top NB rows of the trailing submatrix (`DTRSM`), and finally application of matrix multiplication of the form $A_{ij} \leftarrow A_{ij} - A_{ik} \times A_{kj}$ (`DGEMM`), where A_{ik} is the panel without the top NB rows, A_{kj} is the top NB rows of the trailing submatrix and A_{ij} is the trailing submatrix without the top NB rows. Then the procedure is applied repeatedly, descending down the diagonal of the matrix (Figure 1). The block algorithm is described in detail in section 2.6.3 of the book by Demmel [13]

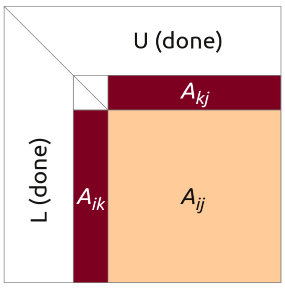


Figure 1: The block LU factorization (Level 3 BLAS algorithm of LAPACK).

5.2 Incremental Pivoting

The worst performance-limiting aspect of Gaussian elimination with partial pivoting is the panel factorization operation. First, it is an inefficient operation, usually based on a sequence of calls to Level 2 BLAS. Second, it introduces synchronization, by locking the entire panel of the matrix at a time. Therefore, it is desirable to split the panel factorization into a number of smaller, finer-granularity operations, which is the basic premise of the *incremental pivoting* implementation, also known in literature as the *tile LU* factorization.

In this algorithm, instead of factoring the panel one column at a time, the panel is factored one tile at a time. The operation proceeds as follows: First the diagonal tile is factored, using the standard LU factorization procedure. Then the factored tile is combined with the tile directly below it, and factored. Then the re-factored diagonal tile is combined with the next tile, and factored again. The algorithm descends down the panel until the bottom of the matrix is reached. At each step, the standard partial pivoting procedure is applied to the tiles being factored. Also, at each step, all the tiles to the right of the panel are updated with the elementary transformations resulting from the panel operations. This way of pivoting is basically the idea of pairwise pivoting applied at the level of tiles, rather than individual elements (Figure 2). The main benefit comes from the fact that updates of the trailing submatrix can proceed alongside panel factorizations, leading to a very efficient parallel execution, where multiple steps of the algorithm are smoothly pipelined.

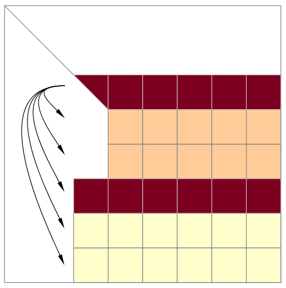


Figure 2: Incremental LU factorization.

5.3 Tournament Pivoting

The panel factorization is one of the most important tasks, because it creates parallelism for the update of the trailing submatrices. Hence, its ineffective execution suffices to reduce considerably the performance of the overall algorithm. Classic approaches that implement partial pivoting algorithm spend more time to perform communication during the panel factorization and hence are not optimal. This is because pivoting forces the algorithm to factor the panel column by column, and then this leads to an algorithm which communicates asymptotically more than the established lower bounds [11].

The basic idea of communication avoiding algorithms, initially introduced for distributed memories [11, 23], and later adapted to shared memories [14], is to replace the search for maximum, performed at each column, by a single reduction of the maximums altogether. This is done thanks to a new pivoting strategy referred to as *tournament pivoting* (TSLU), which performs redundant computations and is shown to be stable in practice. TSLU reduces the bottleneck introduced by the pivoting operation through a block reduction operation to factor the panel. It factors the panel in two steps. The first one identifies rows, which can be used as good pivots for the factorization of the whole panel, with a tournament selection. The second one swaps the selected pivot to the top of the panel, and then factors the entire panel without pivoting in a tiled Cholesky-like operation. With this strategy, the panel is efficiently parallelized and the communication is provably minimized.

Figure 3 presents the first step of TSLU for a panel W using a binary tree for the reduction operation. First, the panel is partitioned into $T_r = 4$ blocks, that is, $W = [W_{00}, W_{10}, \ldots, W_{T_r-1,0}]$, where T_r represents the number of threads participating in the operation.

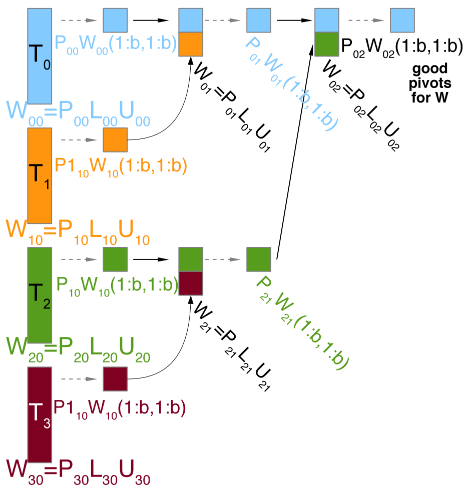


Figure 3: Example of panel factorization using TSLU with $P = 4$ processors. A binary tree is used for the reduction operation

At the first step of the reduction operation, each thread I applies Gaussian elimination with partial pivoting to its block W_{I0}, then the resulting permutation matrix P_{I0} is applied to the original block W_{I0}, and the first b rows of the permuted block $P_{I0}W_{I0}$ are selected as pivot candidates. These first pivot candidates represent the leaves of the reduction tree. In the next steps, W_{IJ} represents the block owned by thread I at step J of the reduction operation. At each node of the tree, the pivot candidates of its descending children are merged one on top of another, and then Gaussian elimination with partial pivoting is applied on the merged block, the resulting permutation matrix is again applied on the original merged block and then, the first b rows are selected as a new pivot candidate. By using a binary tree, this step is repeated $\log T_r$ times. The pivots obtained at the root of the tree are then considered as the good pivots for the whole panel. Once these pivots are permuted at the top of the panel, each thread I applies Gaussian elimination without partial pivoting to its block W_{I0}.

The example presented in Figure 3 uses a binary tree with two tiles reduced together at each level, but any reduction tree can be used depending on the underlying architecture. The TSLU implementation in PLASMA, used for experiments in this paper, reduces tiles four by four at each level. The number of 4 tiles has been chosen because it gave a good ratio of kernel efficiency over one single core relative to the time spent to perform the factorization of the subset.

5.4 Random Butterfly Transform

As an alternative to pivoting, the Partial Random Butterfly Transformation (PRBT) preconditions the matrix as $A_r = W^\top AV$, such that, with probability close to 1, pivoting is unnecessary. This technique was proposed by Parker [44] and later adapted by Baboulin et al [5]. An $n \times n$ butterfly matrix is defined as

$$B^{(n)} = \frac{1}{\sqrt{2}} \begin{bmatrix} R & S \\ R & -S \end{bmatrix},$$

where R and S are random diagonal, nonsingular matrices. W and V are recursive butterfly matrices of depth d, defined by

$$W^{(n,d)} = \begin{bmatrix} B_1^{(n/2^{d-1})} & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & B_{2^{d-1}}^{(n/2^{d-1})} \end{bmatrix} \times \cdots \times B^{(n)}.$$

We use a depth $d = 2$, previously found to be sufficient in most cases [5]. Since each R and S is diagonal, W and V can be stored as $n \times d$ arrays. Due to the regular sparsity pattern, multiplying an $m \times n$ matrix by W and V is an efficient, $O(mn)$ operation.

After applying the PRBT, Gaussian elimination without pivoting is used to obtain the solution, as indicated in Algorithm 1.

Algorithm 1 Solving $Ax = b$ using RBT.

```
1: $A_r = W^\top AV$
2: factor $A_r = LU$ without pivoting
3: solve $LUy = W^\top b$ for $y$
4: $x = Vy$
```

While the randomization reduces the need for pivoting, the lack of pivoting can still be unstable, so we use iterative refinement to reduce the potential for instability. The cost of pivoting is thus avoided, at the expense of applying the PRBT and iterative refinement.

5.5 No Pivoting

This implementation of Gaussian elimination completely abandons pivoting. This can be done very rarely in practice without risking serious numerical consequences, or even a complete break-down of the algorithm if a zero is encountered on the diagonal. Here the implementation serves only as a performance baseline. Dropping pivoting increases performance for two reasons. First, the overhead of swapping matrix rows disappears. Second, the level of parallelism dramatically increases, since the panel operations now become parallel, and can also be pipelined with the updates to the trailing submatrix.

5.6 Iterative Refinement

Iterative refinement is an iterative method proposed by James Wilkinson to improve the accuracy of numerical solutions to systems of linear equations. When solving a linear system $Ax = b$, due to the presence of roundoff errors, the computed solution may deviate from the exact solution. Starting with the initial solution, iterative refinement computes a sequence of solutions that converges to the exact solution when certain assumptions are met (Algorithm 2).

Algorithm 2 Iterative refinement using MATLAB™ backslash notation.

```
1: repeat
2:     $r = b - Ax$
3:     $z = L\backslash(U\backslash Pr)$
4:     $x = x + z$
5: until x is "accurate enough"
```

As Demmel points out [13, pp.60], the iterative refinement process is equivalent to Newton's method applied to $f(x) = b - Ax$. If the computation was done exactly, the exact solution would be produced in one step. Iterative refinement was studied by Wilkinson [50], Moler [43], and Demmel [12], and is covered in the books by Higham [33] and Stewart [47].

Iterative refinement introduces a memory overhead. Normally, in the process of factorization, the original matrix A is overwritten with the L and U factors. However, the original matrix is required, in the refinement process, to compute the residual. Therefore, application of iterative refinement doubles the memory requirement of the algorithm.

6. IMPLEMENTATION

6.1 Tile Layout

It is always beneficial for performance to couple the algorithm with a data layout that matches the processing pattern. For tile algorithms, the corresponding layout is the *tile layout*, developed by Gustavson et al. [26] and shown in Figure 4. The matrix is arranged in square submatrices, called tiles, where each tile occupies a continuous region of memory. The particular type of layout used here is referred to as *Column-Column Rectangular Block* (CCRB). In this flavor of the tile layout, tiles follow the column-major order and elements within tiles follow the column-major order. The same applies to the blocks A_{11}, A_{21}, A_{12}, and A_{22}.

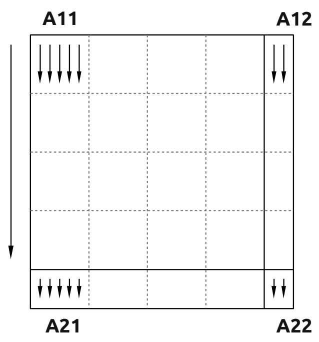


Figure 4: The Column-Column Rectangular Block (CCRB) matrix layout.

Form the standpoint of serial execution, tile layout minimizes *conflict* cache misses, because two different memory locations within the same tile cannot be mapped to the same set of a set-associative cache. The same applies to the *Translation Look-aside Buffer* (TLB) misses. In the context of parallel execution, tile layout minimizes the probability of *false sharing*, which is only possible at the beginning and end of the continuous memory region occupied by each tile, and can easily be eliminated altogether, if the matrix is aligned to cache lines and tiles are divisible by the cache line size. Tile layout is also beneficial for prefetching, which in the case of strided memory access is likely to generate useless memory traffic.

It is only fair to assume that most users of shared memory systems assemble their matrices in the standard column-major layout, common to FORTRAN 77 and LAPACK, Therefore, the overhead of translating the matrix from the column-major layout to the CCRB layout and back is always included in the timing. Because the entire matrix occupies a contiguous region of memory, translation between the tile layout and the legacy FORTRAN 77 layout can be done in place, without changing the memory footprint. Gustavson et al. [26] devised a collection of routines for performing this translation in a parallel and cache efficient manner. It is important to observe that the layout translation routines have a broader impact in forming the basis for a fast transposition operation. The codes are distributed as part of the PLASMA library.

6.2 Dynamic Scheduling

In order to exploit the fine-grained parallelism to its fullest, efficient multithreading mechanisms have to be designed, where data dependencies are preserved, i.e., data hazards are prevented. This has been done for both the simpler single-sided factorizations, such as Cholesky, LU and QR [2, 3, 8, 9, 16–18, 29, 37], as well as the more complicated two-sided factorizations, such as the reductions to band bi-diagonal and band tri-diagonal form [27, 28, 38–40, 42]. The process of constructing such schedules through manipulation of loop indexes and enforcing them by progress tables is tedious and error-prone. Using a runtime dataflow scheduler is a good alternative. A superscalar scheduler is used here.

Superscalar schedulers exploit multithreaded parallelism in a similar way as superscalar processors exploit *Instruction Level Parallelism* (ILP). Scheduling proceeds under the constraints of data hazards: *Read after Write* (RaW), *Write after Read* (RaW) and *Write after Write* (RaW). In the context of multithreading, superscalar scheduling is a way of automatically parallelizing serial code. The programmer is responsible for encapsulating the work in side-effect-free functions (parallel tasks) and providing directionality of their parameters (input, output, input-and-output), and the scheduling is left to the runtime. Scheduling is done by conceptually exploring the *Directed Acyclic Graph* (DAG), or task graph, of the problem. In practice the DAG is never built entirely, and instead explored in a *sliding window* fashion. The superscalar scheduler used here is the *QUeuing And Runtime for Kernels* (QUARK) [51] system, developed at the University of Tennessee.

7. EXPERIMENTAL RESULTS

7.1 Hardware and Software

The experiments were run on an Intel system with 16 cores and an AMD system with 48 cores. The Intel system has two sockets of eight-core Intel Sandy Bridge CPUs clocked at 2.6 GHz, with a theoretical peak of $16\ cores \times 2.6\ GHz \times 8\ ops\ per\ cycle \simeq 333\ Gflop/s$ in double precision arithmetic. The AMD system has eight sockets of six-core AMD Istanbul CPUs clocked at 2.8 GHz, with a theoretical peak of $48\ cores \times 2.8\ GHz \times 4\ ops\ per\ cycle \simeq 538\ Gflop/s$ in double precision arithmetic.

All presented LU codes were built using the PLASMA framework, relying on the CCRB tile layout and QUARK dynamic scheduler. The GCC compiler was used for compiling the software stack and Intel MKL (Composer XE 2013) was used to provide an optimized implementation of serial BLAS.

7.2 Performance

We now study the performance of our implementations on square random matrices in double real precision, and compare their performance with that of MKL. Since each of our implementations uses a different pivoting strategy, for a fair performance comparison, we ran iterative refinement with all the algorithms. Namely, for MKL, we used the MKL iterative refinement routine DGERFS, while for the tile algorithms, we implemented a tiled iterative refinement with the same stopping criteria as that in DGERFS, i.e., the iteration terminates when one of the following three criteria is satisfied: (1) the component-wise backward error, $\max_i |r_i|/(|A||\widehat{x}| + |b|)_i$, is less than or equal to $((n+1) * \mathbf{sfmin})/\mathbf{eps}$, where r is the residual vector (i.e, $r = A\widehat{x} - b$ with the computed solution $\widehat{x}$), **eps** is the relative machine precision, and **sfmin** is the smallest value such that $1/\mathbf{sfmin}$ does not overflow, (2) the component-wise backward error is not reduced by half, or (3) the number of iterations is equal to ten. For

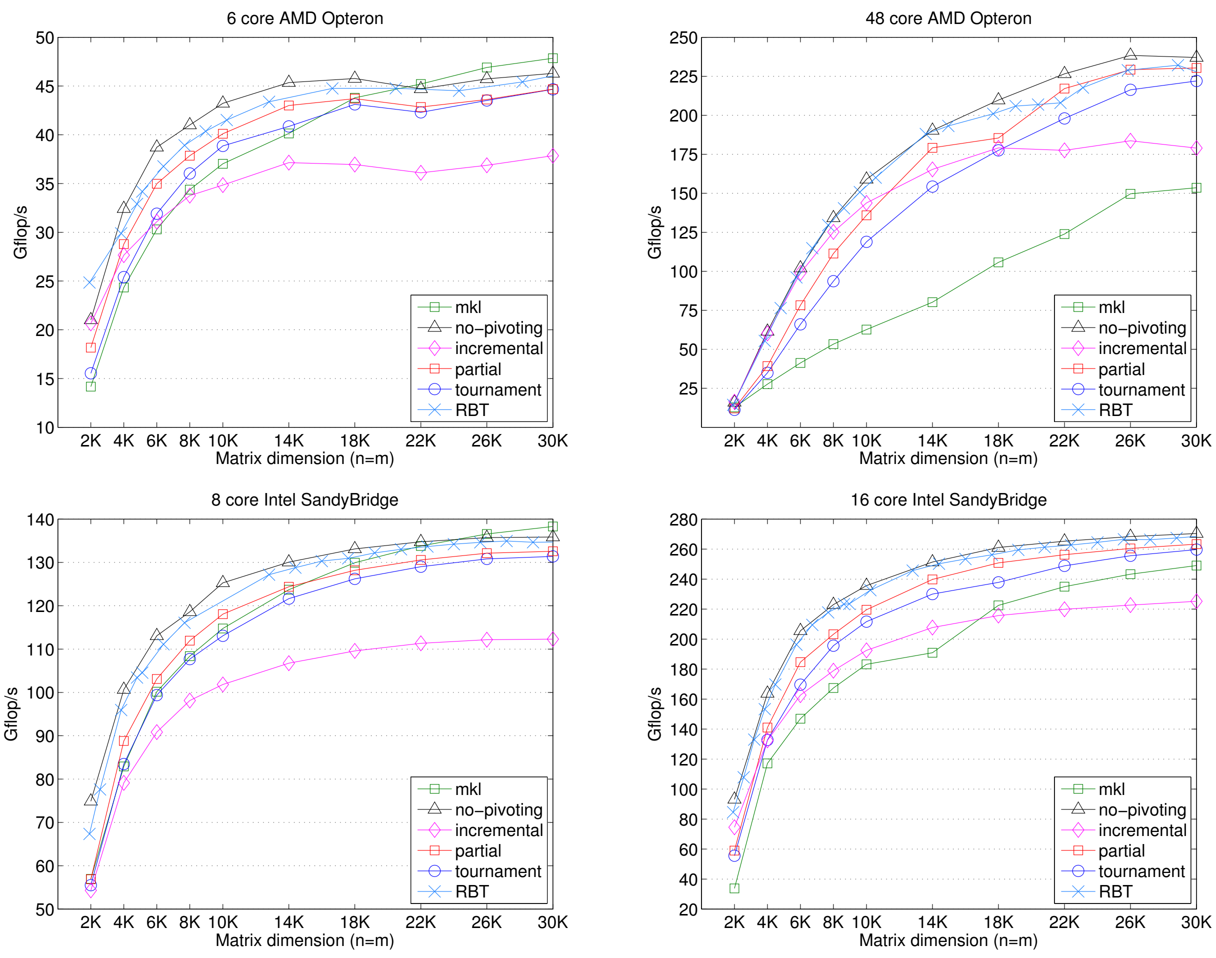


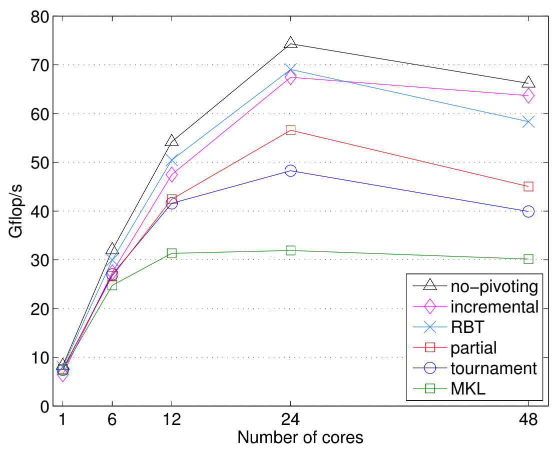
Figure 5: Asymptotic performance comparison of LU factorization algorithms.

all of our experiments, the iterative refinements converged in less than ten iterations. We observed that even with partial pivoting, it requires a couple of iterations to satisfy this stopping criteria (see Section 7.3). Furthermore, in many cases, DGERFS of MKL did not scale as well as our implementation. We suspect this is due to the need to compute $|A||\widehat{x}|+|b|$ at each iteration.

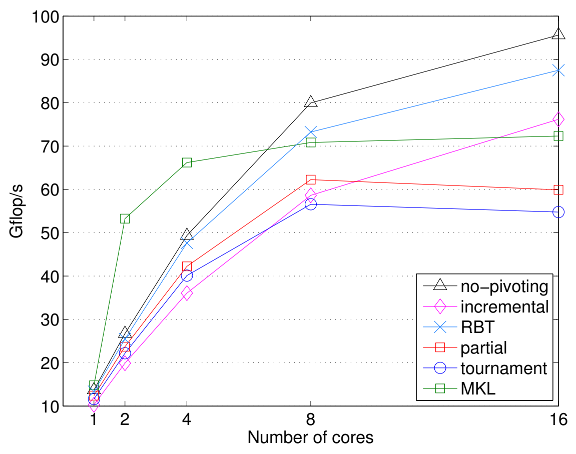
The performance of our implementations is sensitive to the tile size *nb*. Hence, for each matrix dimension n on a different number of cores, we studied the performance of each algorithm with the tile sizes of $nb = 80, 160, 240, 320$, and 400. We observed that on 48 cores of our AMD machine, the performance is especially sensitive to the tile size, and we tried the additional tile sizes of $nb = 340, 360$ and 380. In addition, the performance of our incremental pivoting is sensitive to the inner blocking size, and we tried using the block sizes of $ib = 10, 20$, and 40 for both $nb = 80$ and 160, $ib = 10, 20, 30, 40, 60$, and 80 for $nb = 240$, and $ib = 10, 20, 40$, and 80 for both $nb = 320$ and 400. Figure 5 shows the performance obtained using the tile and block sizes that obtained the highest performance of factorization.[1] For the tile algorithms, we included the data layout conversion time as a part of the solution time. We summarize our finding below:

- RBT used the default transformation depth of two and added only a small overhead over no-pivoting in all the test cases.
- In comparison to other pivoting strategies, incremental pivoting could exploit a large number of cores more effectively. As a result, when the performance is not dominated by the trailing submatrix updates (e.g., for a small matrix dimension on 48 cores), it obtained performance that is close to that of no-pivoting. However, for a large matrix dimension, due to the special kernels required for the incremental pivoting to update the trailing submatrix, its performance was lower than that of the partial or tournament pivoting LU that uses BLAS-3 DGEMM of MKL for their trailing submatrix updates.
- In comparison to MKL, partial and tournament pivoting LU may reduce the communication and could effectively utilize a larger number of cores. As a result, when the communication impacts the performance (e.g., for medium sized matrices on multiple sockets), they outperformed MKL. We can clearly see this on 48 cores of our AMD machine.
- MKL performed well especially when the trailing submatrix update dominated the performance. Moreover, on the single

[1] We computed the Gflop/s as the ratio of the number of flops required for LU factorization and a pair of forward and backward substitutions, $\frac{2}{3}n^3 + 2n^2$, over the total time required for the factorization and iterative refinement.



(a) AMD Opteron ($m = n = 4000$).



(b) Intel SandyBridge ($m = n = 2000$).

Figure 6: Strong scaling comparison of LU factorization algorithms.

socket, MKL outperformed no-pivoting for a large enough matrix dimension. This could be because a tiled implementation looses efficiency due to its smaller BLAS kernels used during the trailing submatrix updates.

Clearly, for a large enough matrix size, the computational kernels used for the trailing submatrix update (e.g., DGEMM) dominate the performance. On the other hand, when the communication has a significant impact on the performance (e.g., on a distributed-memory machine), the algorithms that reduce the communication (e.g., RBT and tournament-pivoting) can outperform the standard algorithm. This is implied by our results on 48 cores, and the difference could be greater when the communication becomes more dominant. In our experiments on a shared-memory machine, partial pivoting LU outperformed tournament pivoting in all the cases. However, when the communication latency becomes a significant part of the performance, the tournament pivoting may be favored.

Finally, Figure 6 shows the strong scaling of our implementations.[2] In these experiments, for each algorithm, we used the tile size that obtained the best performance on the 48 or 16 cores of the AMD or Intel machine, respectively. Since the matrix dimensions were relatively small in this study, in order to obtain high performance, it becomes imperative to reduce communication and exploit parallelism as much as possible. In particular, RBT and incremental pivoting obtained excellent parallel efficiency.

7.3 Accuracy

To study the numerical behavior of our implementations, we used the synthetic matrices from two recent papers [5, 24]. In our experiments, all the five pivoting strategies performed similarly on most of the test matrices. In this section, we only present the results of the test matrices which demonstrate the performance characteristics of the pivoting strategies. We also conducted the numerical experiments using all the matrix dimensions used in Section 7.2, but here, we only show the results of $n = 30000$, which represent the performance trends of the pivoting strategies for all other matrix dimensions. Table 1 shows some properties of these test matrices and the stability results of using partial pivoting. In the table, the second through the sixth test matrices are from the paper by Grigori et al. [24], where the first two have relatively small condition numbers, while the rest are more ill-conditioned. The last three matrices are from the paper by Baboulin et al. [5], where the last test matrix `gfpp` is one of the pathological matrices that exhibit exponential growth factor using partial pivoting. Since the condition number of the `gfpp` matrix increases rapidly with the matrix dimension, we used the matrix dimension of $m = 1000$ for our study. Finally, our incremental and tournament pivoting exhibit different numerical behavior using different tile sizes. For the numerical results presented here, we used the tile and block sizes that obtain the best performance on the 16 core Intel SandyBridge. All the results are in double real precision.

Figure 7(a) shows the component-wise backward errors, $\max_i |r_i|/(|A||\hat{x}|+|b|)_i$, at each step of the iterative refinements. For these experiments, the right-hand-side b is chosen such that the entries of the exact solution x are uniformly distributed random numbers in the range of $[-0.5, 0.5]$. Below, we summarize our findings:

- For all the test matrices, tournament pivoting obtained initial backward errors comparable to those of partial pivoting.
- No-pivoting was unstable for five of the test matrices (i.e., `ris`, `fiedler`, `orthog`, {-1 1}, and `gfpp`). For the rest of the test matrices, the initial backward errors of no-pivoting were significantly greater than those of partial pivoting, but were improved after a few iterative refinements.
- Our incremental pivoting failed for the `fiedler`, `orthog`, and `gfpp` matrices. For other test matrices, its backward errors were greater than those of partial pivoting, but were improved to be in the same order as those of partial pivoting after a few iterative refinements. The only exception was

[2]Using the default transformation depth of two, our current implementation of RBT assumes the matrix dimension to be a multiple of four times the tile size. Hence, for these experiments, in order to use the same block sizes as those used for no-pivoting, we set the matrix dimensions to be $m = 1920$ and $m = 3840$ on the AMD and Intel machines, respectively, for RBT.

name	description	$\|A\|_1$	cond$(A,2)$	$\|L\|_1$	$\|L^{-1}\|_1$	max $|U(i,j)|$	max $|U(i,i)|$	$\|U\|_1$	cond$(U,1)$
`random`	dlarnv(2)	7.59e+03	4.78e+05	1.50e+04	8.60e+03	1.54e+02	1.19e+02	2.96e+05	2.43e+09
`circul`	gallery('circul', 1 : n)	2.43e+04	6.97e+02	6.66e+03	8.64e+03	5.08e+03	3.87e+03	1.50e+06	4.23e+07
`riemann`	gallery('riemann', n)	1.42e+05	3.15e+05	3.00e+04	3.50e+00	3.00e+04	3.00e+04	2.24e+05	1.25e+08
`ris`	gallery('ris', n)	1.16e+01	3.34e+15	2.09e+04	3.43e+02	7.34e+00	3.30e+00	3.46e+02	1.42e+21
`compan`	compan(dlarnv(3))	4.39e+00	1.98e+04	2.00e+00	1.01e+01	3.39e+00	1.85e+00	1.90e+01	8.60e+01
`fiedler`	gallery('fiedler', 1 : n)	1.50e+04	1.92e+09	1.50e+04	1.37e+04	2.00e+00	1.99e+00	2.71e+04	9.33e+09
`orthog`	gallery('orthog', n)	1.56e+02	1.00e+00	1.91e+04	1.70e+03	1.57e+03	1.57e+02	2.81e+03	3.84e+08
{-1,1}	$A(i,j) = -1$ or 1	3.00e+04	1.81e+05	3.00e+04	8.67e+03	5.47e+03	3.78e+02	1.00e+06	8.35e+08
`gfpp`[†]	gfpp(triu(rand(n)),1e-4)	1.00e+03	1.42e+19	9.02e+02	2.10e+02	4.98e+00	2.55e+00	4.28e+02	5.96e+90

[†] For `gfpp`, $n = 1000$.

Table 1: Properties of test matrices and stability results of using partial pivoting $(n = m = 30000)$**.**

with the `ris` matrix, where the refinements stagnated before reaching a similar accuracy as that of partial pivoting. In each column of the `ris` matrix, entries with smaller magnitudes are closer to the diagonal (i.e., $A(i,j) = 0.5/(n-i-j+1.5)$). As a result, the permutation matrix P of partial pivoting has ones on the anti-diagonal.

- When no-pivoting was successful, its backward errors were similar to those of RBT. On the other hand, RBT was more stable than no-pivoting, being able to obtain small backward errors for the `fiedler`, {-1,1}, and `gfpp` matrices.
- Partial pivoting was not stable for the pathological matrix `gfpp`. On the other hand, RBT randomizes the original structure of the matrix and was able to compute the solution of reasonable accuracy. It is also possible to construct pathological test matrices, where partial pivoting is unstable while tournament pivoting is stable, and vice versa [24].

Figure 7(b) shows the relative forward error norms of our implementations, which were computed as $\|x-\hat{x}\|_\infty/\|x\|_\infty$. We observed similar trends in the convergence of the forward error norms as in that of the backward errors. One difference was with the `orthog` test matrix, where iterative refinements could not adequately improve the forward errors of incremental pivoting and RBT. Also, even though the backward errors of the `ris` test matrix were in the order of machine epsilon with partial and tournament pivoting, their relative forward errors were in the order of $O(1)$ due to the large condition number.

8. CONCLUSIONS

When implemented well, using a fast, recursive panel factorization, tile data layout, and dynamic scheduling, the canonical LU factorization with partial (row) pivoting, is a fast and numerically robust method for solving dense linear systems of equations. On a shared-memory multicore system, its asymptotic performance is very close to the performance of LU factorization without pivoting.

In our experiments on synthetic matrices, tournament pivoting turned out to be as stable as partial pivoting, which has been theoretically proven by its inventors in the first place. It also proved to be fairly fast. However, it failed to deliver on its promise of outperforming partial pivoting, which can be attributed to the shared-memory environment. The method has much more potential for distributed memory systems, where communication matters much more.

Incremental pivoting showed the worst asymptotic performance due to the use of exotic kernels, instead of the GEMM kernel. On the other hand, it showed strong scaling properties almost as good as RBT and no-pivoting. It is harder to make strong claims about its numerical properties. Its initial residual is usually worse than that of partial and tournament pivoting, but in most cases the accuracy is quickly recovered in iterative refinement. It can fail in some situations, when partial and tournament pivoting prevail.

RBT is the fastest method, both asymptotically, and in terms of strong scaling, because it only adds a small overhead of preprocessing and postprocessing to the time of factorization without pivoting. Similarly to incremental pivoting, it produces a high initial residual, but the accuracy can be recovered in iterative refinement. Similarly to incremental pivoting, it can fail in situations, when partial pivoting and tournament pivoting prevail.

And finally, it can be observed that iterative refinement is a powerful mechanism of minimizing the backward error, which in most cases translates to minimizing the forward error.

9. FUTURE DIRECTIONS

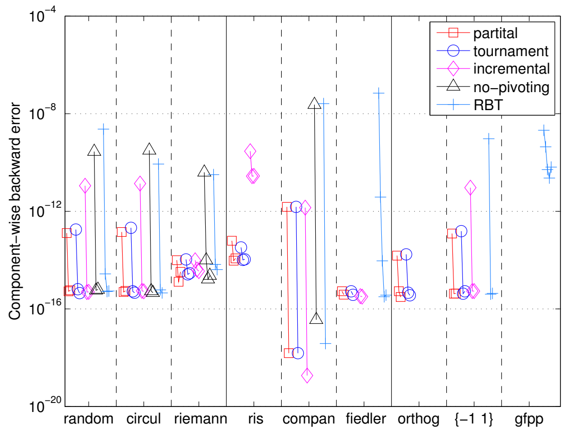
Although we believe that the wide range of synthetic matrices with different properties gives a good understanding of numerical properties of the different flavors of the LU factorization, ultimately it would be invaluable to make such comparison using matrices coming from real world applications, such as plasma burning or radar cross section.

Although we believe that the comparison using relatively large shared memory systems, by todays standards, gives a good insight into the performance properties of the different LU factorization algorithms, we acknowledge that the picture can be very different in a distributed memory environment. Ultimately we would like to produce a similar comparison using distributed memory systems.

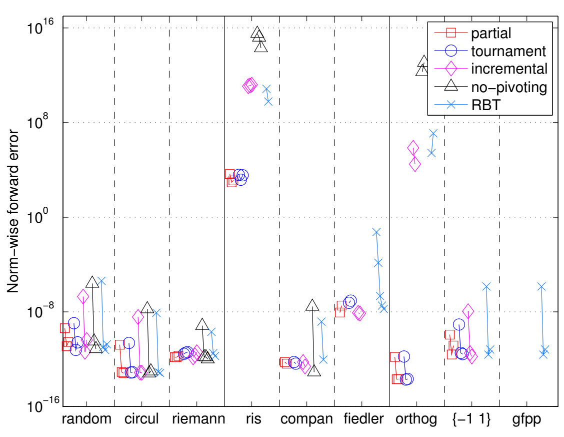
References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Proceedings of the 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA'11)*, pages 217–224, dec 2010. Best Paper award.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009. DOI: 10.1088/1742-6596/180/1/012037.



(a) Relative backward error.



(b) Relative forward error.

Figure 7: Numerical accuracy of LU factorization algorithms, showing error before refinement (top point of each line) and after each refinement iteration (subsequent points).

[3] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[4] R. Aymar, V. Chuyanov, M. Huguet, and Y. Shimomura. Overview of ITER-FEAT - the future international burning plasma experiment. *Nuclear Fusion*, 41(10), 2001.

[5] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39:8:1–8:13, 2013.

[6] R. F. Barrett, T. H. F. Chan, E. F. D'Azevedo, E. F. Jaeger, K. Wong, and R. Y. Wong. Complex version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Experience*, 22(5):573–587, April 10 2010.

[7] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The Impact of Multicore on Math Software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2006.

[8] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[9] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

[10] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10*, Bangalore, India, January 2010. ACM. DOI: 10.1145/1693453.1693484 (submitted to ACM TOMS).

[11] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Implementing communication-optimal parallel and sequential QR factorizations. *Arxiv preprint arXiv:0809.2407*, 2008.

[12] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Soft.*, 32(2):325–351, 2006. DOI: 10.1145/1141885.1141894.

[13] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713897.

[14] S. Donfack, L. Grigori, and A. Gupta. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[15] J. Dongarra, V. Eijkhout, and P. Luszczek. Recursive approach in sparse matrix LU factorization. *Sci. Program.*, 9:51–60, January 2001.

[16] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.

[17] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on lu factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 33–42, New York, NY, USA, 2011. ACM.

[18] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. *Advances in Parallel Computing, Special Issue*, 22:429–436, 2012. ISBN 978-1-61499-040-6 (print); ISBN 978-1-61499-041-3 (online).

[19] A. Edelman. Large dense numerical linear algebra in 1993: the parallel computing influence. *International Journal of High Performance Computing Applications*, 7(2):113–128, 1993.

[20] L. V. Foster. Gaussian elimination with partial pivoting can fail in practice. *SIAM J.Matrix Anal. Appl.*, 15:1354–1362, 1994.

[21] G. H. Golub and C. F. C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. ISBN: 0801854148.

[22] J. F. Grcar. Mathematicians of Gaussian elimination. *Notices of the AMS*, 58(6):782–792, June/July 2011.

[23] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008.

[24] L. Grigori, J. Demmel, and H. Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. Appl.*, 32:1317–1350, 2011.

[25] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997. DOI: 10.1147/rd.416.0737.

[26] F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Trans. Math. Soft.*, 38(3):article 17, 2012. DOI: 10.1145/2168773.2168775.

[27] A. Haidar, H. Ltaief, and J. Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of SC '11*, pages 8:1–8:11, New York, NY, USA, 2011. ACM.

[28] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 21-25 2012. ISBN 978-1-4673-0975-2.

[29] A. Haidar, H. Ltaief, A. YarKhan, and J. J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency Computat.: Pract. Exper.*, 2011. DOI: 10.1002/cpe.1829.

[30] R. Harrington. Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine*, 32:31-35, June 1990.

[31] J. L. Hess. Panel methods in computational fluid dynamics. *Annual Reviews of Fluid Mechanics*, 22:255–274, 1990.

[32] L. Hess and M. O. Smith. Calculation of potential flows about arbitrary bodies. In D. Kuchemann, editor, *Progress in Aeronautical Sciences*, volume 8:1-138. Pergamon Press, 1967.

[33] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM: Society for Industrial and Applied Mathematics, 2002. ISBN: 0898715210.

[34] E. Jaeger, L. Berry, E. DâĂŹAzevedo, D. Batchelor, M. C. MD, K. White, and H. Weitzner. Advances in full-wave modeling of radio frequency heated multidimensional plasmas. *Physics of Plasmas*, 9(5):1873–1881, 2002.

[35] E. Jaeger, L. Berry, J. Myra, D. Batchelor, E. DâĂŹAzevedo, P. Bonoli, C. Philips, D. Smithe, D. DâĂŹIppolito, M. Carter, R. Dumont, J. Wright, and R. Harvey. Sheared poloidal flow driven by mode conversion in Tokamak plasmas. *Phys. Rev. Lett.*, 90(19), 2003.

[36] E. Jaeger, R. Harvey, L. Berry, J. Myra, R. Dumont, C. Philips, D. Smithe, R. Barrett, D. Batchelor, P. Bonoli, M. Carter, E. DâĂŹazevedo, D. DâĂŹippolito, R. Moore, and J. Wright. Global-wave solutions with self-consistent velocity distributions in ion cyclotron heated plasmas. *Nuclear Fusion*, 46(7):S397âĂŞS408, 2006.

[37] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009. DOI: 10.1002/cpe.1467.

[38] H. Ltaief, J. Kurzak, and J. Dongarra. Parallel band two-sided matrix bidiagonalization for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), April 2010.

[39] H. Ltaief, P. Luszczek, and J. Dongarra. High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. *ACM TOMS*, 39(3), 2013. In publication.

[40] H. Ltaief, P. Luszczek, A. Haidar, and J. Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011*, volume 7203 of *Parallel Processing and Applied Mathematics*, pages 661–670, Torun, Poland, 2012.

[41] P. Luszczek and J. Dongarra. Anatomy of a globally recursive embedded LINPACK benchmark. In *Proceedings of 2012 IEEE High Performance Extreme Computing Conference (HPEC 2012)*, Westin Hotel, Waltham, Massachusetts, September 10-12 2012. IEEE Catalog Number: CFP12HPE-CDR, ISBN: 978-1-4673-1574-6.

[42] P. Luszczek, H. Ltaief, and J. Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, USA, May 16-20 2011.

[43] C. B. Moler. Iterative refinement in floating point. *J. ACM*, 14(2):316–321, 1967. DOI: 10.1145/321386.321394.

[44] D. S. Parker. A randomizing butterfly transformation useful in block matrix computations. Technical Report CSD-950024, Computer Science Department, University of California, 1995.

[45] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36:14:1–14:26, July 2009.

[46] D. C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Tranasactions on Computers*, C-34(3), March 1985.

[47] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973. ISBN: 0126703507.

[48] L. Trefethen and R. Schreiber. Average case analysis of Gaussian elimination. *SIAM J. Mat. Anal. Appl.*, 11(3):335–360, 1990.

[49] J. J. H. Wang. *Generalized Moment Methods in Electromagnetics*. John Wiley & Sons, New York, 1991.

[50] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965. ISBN: 0198534183.

[51] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011. http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf.

[52] M.-C. Yeung and T. F. Chan. Probabilistic analysis of Gaussian elimination without pivoting. Technical Report CAM95-29, Department of Mathematics, University of California, Los Angeles, June 1995.

[53] E. L. Yip. FORTRAN Subroutines for Out-of-Core Solutions of Large Complex Linear Systems. Technical Report CR-159142, NASA, November 1979.