CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3306

A survey of recent developments in parallel implementations of Gaussian elimination

Simplice Donfack[1], Jack Dongarra[1], Mathieu Faverge[2], Mark Gates[1], Jakub Kurzak[1,*,†], Piotr Luszczek[1] and Ichitaro Yamazaki[1]

[1]*EECS Department, University of Tennessee, Knoxville, TN, USA*
[2]*IPB ENSEIRB-Matmeca, Inria Bordeaux Sud-Ouest, Bordeaux, France*

SUMMARY

Gaussian elimination is a canonical linear algebra procedure for solving linear systems of equations. In the last few years, the algorithm has received a lot of attention in an attempt to improve its parallel performance. This article surveys recent developments in parallel implementations of Gaussian elimination for shared memory architecture. 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 Partial 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 multisocket multicore systems are presented. Performance and numerical accuracy is analyzed. Copyright © 2014 John Wiley & Sons, Ltd.

Received 15 July 2013; Revised 26 April 2014; Accepted 2 May 2014

KEY WORDS: Gaussian elimination; LU factorization; parallel; shared memory; multicore

1. INTRODUCTION

Gaussian elimination has a long history that can be traced back some 2000 years [1]. Today, dense systems of linear equations have become a critical cornerstone for some of the most compute intensive applications. A sampling of domains using dense linear equations are fusion reactor modeling [2], aircraft design [3], acoustic scattering [4], antenna design, and radar cross-section studies [5]. For instance, simulating fusion reactors generates dense systems that exceed half a million unknowns, solved using LU factorization [6]. Many dense linear systems arise from the solution of boundary integral equations via boundary element methods [7], variously called the method of moments in electromagnetics [8], and the panel method in fluid dynamics [9]. These methods replace a sparse three-dimensional problem of $O(n^3)$ unknowns with a dense two-dimensional problem of $O(n^2)$ unknowns. Any improvement in the time to solution for dense linear systems has a direct impact on the execution time of these applications.

1.1. Motivation

The aim of this article is to give in-depth 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

*Correspondence to: Jakub Kurzak, EECS Department, University of Tennessee, Knoxville, TN, USA.
†E-mail: kurzak@eecs.utk.edu

[10]. Rather than focusing on multiple factorizations and performance across multiple hardware architectures [11], we switch our focus to multiple pivoting strategies and provide uniform treatment for each, while maintaining sufficient hardware variability over shared-memory systems to increase the meaningful impact of our results. No distributed environment is considered in this study.

The stability of LU is strongly tied to the *growth factor* [12], $\rho = \max_{i,j} |u_{ij}| / \max_{i,j} |a_{ij}|$. Ever since the probabilistic properties of partial pivoting were established [13], the community has fully embraced the method, despite the high upper bound on the growth factor that it can theoretically incur: 2^{n-1} growth versus a much more acceptable $O(n^{1/2+1/4 \log n})$ growth provided by complete pivoting [12]. Probabilistically, partial pivoting achieves an average growth factor of $n^{2/3}$. No-pivoting LU has an unbounded growth factor, but probabilistically, an average growth of $n^{3/2}$ has been proven [14], which led to including it in this survey. However, complete lack of any pivoting strategy is discouraged for practical applications.

1.2. Related work

In this paper, we evaluate five LU pivoting schemes: partial pivoting, incremental pivoting, tournament pivoting, the Partial Random Butterfly Transformation, and no-pivoting. A summary of each is given here; further details are given in Section 2.

The partial pivoting code that we evaluate is based on a parallel panel factorization that uses a recursive formulation of LU [15, 16]. This recent implementation was introduced to address the bottleneck of the panel computation [17, 18] and has been successfully applied to matrix inversion [19]. However, it should not be confused with a globally recursive implementation based on column-cyclic distribution [20]. Neither is it similar to a nonrecursive parallelization effort [21] that focuses only on a cache efficient implementation of the existing level 1 basic linear algebra subroutines (BLAS) kernels.

Incremental pivoting [22, 23] has its origins in pairwise pivoting [24], which is related to an updating LU algorithm for out-of-core computations [25], where blocking of computations is performed for better performance. Incremental pivoting uses a pairwise treatment of tiles and, as a consequence, reduces dependencies between tasks, which aids the parallelism that has become so important with the proliferation of multicore hardware. However, it causes the magnitude of pivot elements to increase substantially, rendering the factorization numerically less stable [22, 26, 27].

The tournament pivoting technique originated in communication-avoiding LU, which was initially introduced for distributed memory [28, 29] and later adapted to shared memory [30]. The main design goal for this pivoting scheme is to attain the minimum bounds on the amount of data communicated and the number of messages exchanged between the processors. To achieve these bounds, it minimizes the communication that occurs during the panel factorization by performing $O(n^2)$ extra computations, leaving the highest order computational complexity term, $\frac{2}{3}n^3$, unchanged. In terms of stability, the best known bound for communication-avoiding LU's growth factor increases exponentially with the amount of parallelism, but in experience, it behaves similarly to partial pivoting.

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 [31] and then made practical by Baboulin *et al.* [32] with adaptations that limited the recursion depth without compromising the numerical properties of the method.

Another pivoting strategy recently introduced by Khabou *et al.* [33] uses rank revealing QR to choose pivot rows, which are then factored using Gaussian elimination. This method has been shown to have a smaller growth factor than partial pivoting. Unfortunately, we do not yet have an implementation to test.

A similar study was published before [11], but it was mostly focused on performance across a wide range of hardware architectures. It featured results for the main three one-sided factorization schemes: Cholesky, QR, and LU. No algorithmic variants for a particular method were considered.

DOI: 10.1002/cpe

1.3. Original contribution

The unique contribution of this survey is in implementing and comparing all the algorithms 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, as described in Section 3. Our results in Section 4 offer 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.

2. ALGORITHMS

We start with an overview of each of the LU pivoting schemes that we evaluate. Familiarity with the BLAS is assumed. Briefly, level 1 BLAS have $O(n)$ work and data, level 2 BLAS have $O(n^2)$ work and data, while level 3 BLAS have $O(n^3)$ work on $O(n^2)$ data, making level 3 BLAS operations significantly more efficient because of the increased computation-to-memory-access ratio.

A summary of the floating point operations (flops) and growth factors for each algorithm is given in Table I. For comparison, we also include complete pivoting in Table I, though it is not implemented in our tests. Note that because of differences in dynamic parallelism and kernel efficiency, flops is only a gross indicator of performance. The experiments given in Section 4 yield more meaningful comparisons.

2.1. Partial pivoting

The LAPACK block LU factorization is the main point of reference here, and the LAPACK naming convention is followed. (The initial '`D`' in names denotes double precision routines.) 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 n_b 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 up to n_b rows of the trailing submatrix (`DLASWP`), according to the pivoting pattern, application of a triangular solve with multiple right-hand sides to the top n_b 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 n_b rows, A_{kj} is the top n_b rows of the trailing submatrix, and A_{ij} is the

Table I. Summary of floating point operations (add, multiply, divide), floating point comparisons, best known growth factor bound, and average case growth factor.

	Flops	Comparisons	Growth factor bound	Avg. growth factor
Complete pivoting [13]	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$	$\mathbf{O(n^3)}$	$< Cn^{1/2+1/4 \log n}$	$n^{1/2}$
Partial pivoting [13]	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$	2^{n-1}	$n^{2/3}$
Incremental pivoting [13, 22, 34]	$\frac{2}{3}n^3 \left(\mathbf{1} + \frac{\mathbf{i_b}}{\mathbf{2n_b}}\right) + O(n^2)$	$\frac{1}{2}n^2 - \frac{1}{2}n$	4^{n-1}	n or $> n$
Tournament pivoting [34]	$\frac{2}{3}n^3 + \mathbf{n_b n^2} + O(n)$	$\frac{1}{2}n^2 + \mathbf{O(p_r n_b n)}$	$< 2^{nH}$	$1.5n^{2/3}$
PRBT [32]	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n + \mathbf{5dn^2}$	None	Unbounded	$n^{3/2}$
No pivoting [14]	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$	None	Unbounded	$n^{3/2}$

Differences in flops compared with partial pivoting are in bold.
PRBT, Partial Random Butterfly Transformation.

DOI: 10.1002/cpe

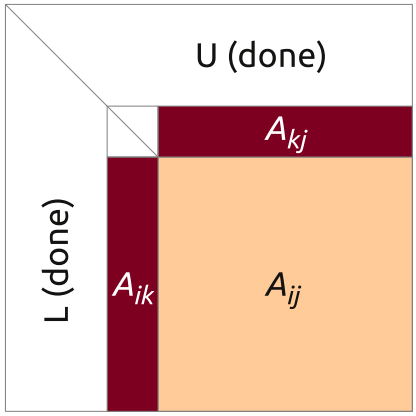


Figure 1. The block LU factorization (level 3 basic linear algebra subroutines algorithm of LAPACK).

trailing submatrix without the top n_b 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 [35].

Instead of the level 2 BLAS panel factorization (`DGETF2`) used in LAPACK, we use a recursive partial pivoting panel factorization, which uses level 3 BLAS operations. For a panel of k columns, the algorithm recursively factors the left $k/2$ columns, updates the right $k/2$ columns with `DLASWP`, `DTRSM`, and `DGEMM`, then recursively factors the right $k/2$ columns, and finally, applies swaps to the left $k/2$ columns with `DLASWP`. The recursion terminates at a single column, where it simply searches for the maximum pivot, swaps elements, and scales the column by the pivot. Parallelism is introduced by splitting the panel into p block rows, which are assigned to different processors.

2.2. *Incremental pivoting*

The most 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 an 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 the 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 refactored 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 (Figure 2). This way of pivoting is basically the idea of pairwise pivoting applied at the level of tiles, rather than individual elements. 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.

A straight-forward implementation of incremental pivoting incurs more floating point operations than partial pivoting. To reduce this additional cost, a second level of blocking, called *inner blocking*, is used within each tile [22, 23]. The total cost to factor an $n \times n$ matrix is

$$\frac{2}{3}n^3\left(1+\frac{i_b}{2n_b}\right)+O(n^2),$$

DOI: 10.1002/cpe

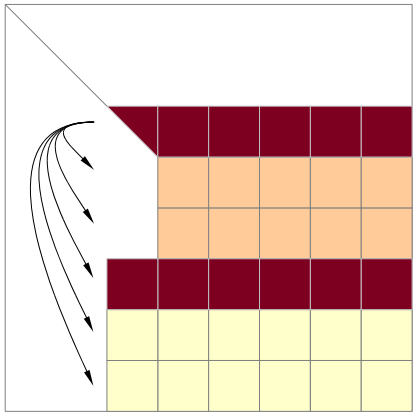


Figure 2. Incremental LU factorization.

where n_b is the tile size and i_b is the inner blocking size. A small value for i_b decreases the flops but may hurt the efficiency of level 3 BLAS operations, so n_b and i_b should be chosen to offer the best compromise between extra operations and BLAS efficiency.

With $n_b = 1$, incremental pivoting reduces to pairwise pivoting; hence, the stability of incremental pivoting is assumed to be the same as pairwise pivoting [23]. For pairwise pivoting, the growth factor bound is 4^{n-1}. Trefethen and Schreiber [13] observed an average growth factor of n for $n \leqslant 1024$, but more recently, Grigori *et al.* [34] observed an average growth factor $> n$ for $n \geqslant 4096$, leaving the question of stability open for large n.

2.3. Tournament pivoting

Classic approaches to the panel factorization with partial pivoting communicate asymptotically more than the established lower bounds [28]. The basic idea of communication avoiding LU is to minimize communication by replacing the pivot search performed at each column with a block reduction of the all the pivots together. This is carried out thanks to a new pivoting strategy referred to as *tournament pivoting*, which performs extra computations and is shown to be stable in practice. It factors the panel in two steps. First, using a tournament selection, it identifies rows that can be used as good pivots for the factorization of the whole panel. Second, it swaps the selected pivot rows to the top of the panel and then factors the entire panel without pivoting. With this strategy, the panel is efficiently parallelized, and the communication is probably minimized.

Figure 3 presents the first step of tournament pivoting for a panel W using a binary tree for the reduction operation. First, the panel is partitioned into p_r blocks, that is, $W = [W_{00}, W_{10}, \ldots, W_{p_r-1,0}]$, where W_{ij} represents the block owned by thread i at step j of the reduction operation. Figure 3 shows an example with $p_r = 4$.

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, unfactored block W_{i0}, and the first n_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. At each node of the tree, the pivot candidates of two child nodes are merged on top of each other, and Gaussian elimination with partial pivoting is applied on the merged block. The resulting permutation matrix is then applied on the original, unfactored merged block, and the first n_b rows are selected as new pivot candidates. By using a binary tree, this step is repeated $\log_2 p_r$ times. The pivots obtained at the root of the tree are then considered to be good pivots for the whole panel. Once these pivots are permuted to the top of the panel, each thread 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 tournament pivoting implementation in PLASMA, used for experiments in this paper, reduces four tiles each

DOI: 10.1002/cpe

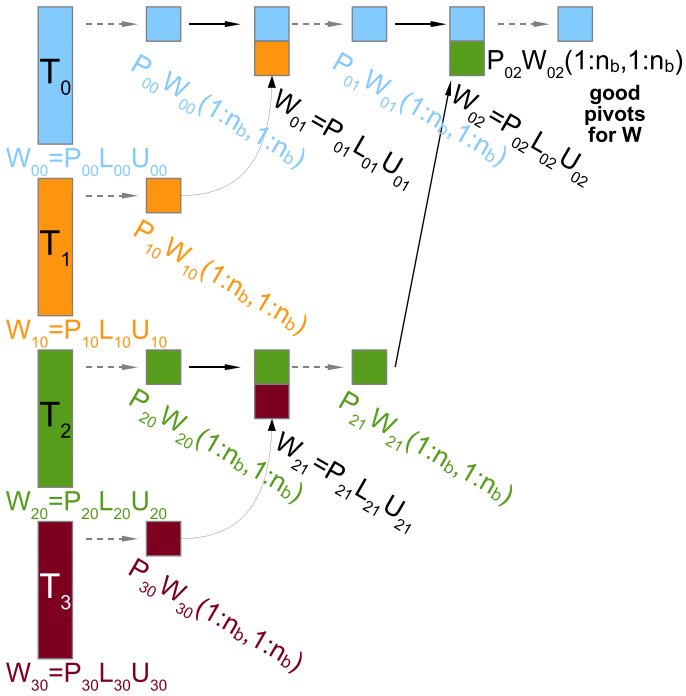


Figure 3. Example of tournament pivoting with $p_r = 4$ processors, using a binary tree for the reduction operation.

level. This number of tiles is chosen because it gives a good ratio of kernel efficiency over one single core, relative to the time spent to perform the factorization of the subset.

For tournament pivoting, the best growth factor bound proved so far is 2^{nH}, where H is the height of the reduction tree. With a binary tree, $H = \log_2 p_r$, so the bound is p_r^n. However, Grigori *et al.* [34] have never observed a growth factor as large as 2^{n-1}, even for pathological cases such as the Wilkinson matrix, and they conjecture that the upper bound is the same as partial pivoting, 2^{n-1}. They observed an average growth factor of $1.5n^{2/3}$, independent of p. Our experiments in Section 4 confirm that tournament pivoting is as stable as partial pivoting in practice.

2.4. *Random Butterfly Transform*

As an alternative to pivoting, the 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 [31] and later adapted by Baboulin *et al.* [32] to reduce the computational cost of the transformation. 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

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

We use a depth $d = 2$, previously found to be sufficient in most cases [32]. Because each R and S is diagonal, W and V can be stored as $n \times d$ arrays. Because of the regular sparsity pattern, multiplying an $m \times n$ matrix by W and V is an efficient, $O(dmn)$ operation.

DOI: 10.1002/cpe

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 PRBT.

1: $A_r = W^\top AV$
2: factor $A_r = LU$ without pivoting
3: $y = U\backslash(L\backslash(W^\top b))$
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. As with no-pivoting LU, the growth factor remains unbounded. This can easily be seen by letting $A = W^{-\top}BV^{-1}$, where the first pivot, b_{00}, is zero. However, probabilistically no-pivoting LU (and hence PRBT) has been shown to have an average growth factor of $n^{3/2}$ [14].

2.5. No pivoting

This implementation of Gaussian elimination completely abandons pivoting. This can be carried out very rarely in practice without risking serious numerical consequences or even a complete breakdown 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, because the panel operations now become parallel, and can also be pipelined with the updates to the trailing submatrix.

2.6. Iterative refinement

Iterative refinement is an iterative method proposed by Wilkinson [36] to improve the accuracy of numerical solutions to systems of linear equations. When solving a linear system $Ax = b$, the computed solution may deviate from the exact solution because of the presence of roundoff errors. 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: $\quad r = b - Ax$
3: $\quad z = U\backslash(L\backslash Pr)$
4: $\quad x = x + z$
5: **until** x is 'accurate enough'

As Demmel points out [35, p. 60], the iterative refinement process is equivalent to Newton's method applied to $f(x) = b - Ax$. If the computation was performed exactly, the exact solution would be produced in one step.

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, in the refinement process, the original matrix is required to compute the residual. Therefore, application of iterative refinement doubles the memory requirement of the algorithm.

3. IMPLEMENTATION

We now turn to the specifics of our implementation, including the data layout, parallel layout transformation routines, and dynamic runtime scheduling system.

DOI: 10.1002/cpe

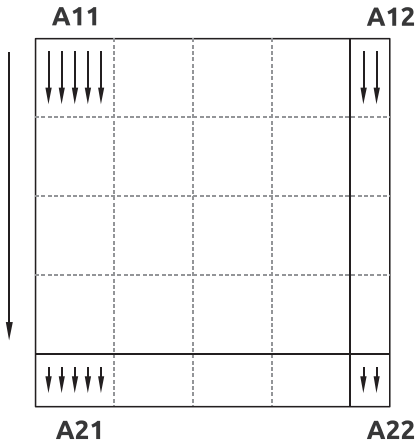


Figure 4. The column-column rectangular block (CCRB) matrix layout.

3.1. Tile layout

For best performance, it is beneficial 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 [37] 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}.

From 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* 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 assemble their matrices in the standard column-major layout, common to Fortran and LAPACK, rather than in our tiled layout. Therefore, for our experiments, the matrix always starts in column-major layout and is then translated into tile layout, and the results translated back to column-major layout. The overhead of translating the matrix from column-major layout to the CCRB layout and back is always included in the timing. This also provides a fair comparison with MKL's LU factorization (`DGETRF`), which takes the matrix in column-major layout. Because the entire matrix occupies a contiguous region of memory, translation between tile layout and column-major layout can be done in place, without changing the memory footprint. Gustavson *et al.* [38] 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.

3.2. Dynamic scheduling

In order to exploit fine-grained parallelism to its fullest, efficient multithreading mechanisms have to be designed where data dependencies are preserved, that is, data hazards are prevented. This has been performed for both the simpler one-sided factorizations, such as Cholesky, LU, and QR [11, 17, 22, 39, 40], as well as the more complicated two-sided factorizations, such as the reductions to band bidiagonal and band tridiagonal form [41–44]. The process of constructing such schedules through manipulation of loop indexes and enforcing them by progress tables is tedious and

DOI: 10.1002/cpe

error-prone. Using a runtime dataflow scheduler is a good alternative. A superscalar scheduler is used here.

Superscalar schedulers, such as SMPSs [45], StarPU [46], SuperMatrix [47], and *QUeuing And Runtime for Kernels* (QUARK) [48] exploit multithreaded parallelism in a similar way as superscalar processors exploit *instruction level parallelism*. Scheduling proceeds under the constraints of data hazards: *Read after Write*, *Write after Read*, and *Write after Write*. 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, and input-and-output), and the scheduling is left to the runtime. For example, Algorithm 3 demonstrates how incremental pivoting inserts each task with its dependencies into the scheduler. The only difference compared with a serial implementation is inserting tasks into a scheduler rather than directly executing tasks. Scheduling is carried out by conceptually exploring the *directed acyclic graph* (DAG), or task graph, of the problem. An example DAG is shown in Figure 5 for a 3×3 tile matrix. The scheduler may execute tasks in any order that respects the dependencies shown in the DAG; for instance, the three tasks in the second level may be executed simultaneously. This eliminates the artificial synchronization after each BLAS operation, common to traditional fork-join parallelism as in LAPACK. In practice, the DAG is never built entirely but instead is explored in a *sliding window* fashion. The superscalar scheduler used here is the QUARK [48] system, developed at the University of Tennessee.

Algorithm 3 Incremental pivoting. A_{ij} refers to tile i, j. Directionality of each parameter is indicated by (in) or (in, out).

Require: matrix A with $m_t \times n_t$ tiles
1: **for** $k = 0$ to $m_t - 1$ **do**
2: insert task: factor A_{kk} (in, out)
3: **for** $j = k + 1$ to $n_t - 1$ **do**
4: insert task: update A_{kj} (in, out) using A_{kk} (in)
5: **end for**
6: **for** $i = k + 1$ to $m_t - 1$ **do**
7: insert task: factor $\begin{bmatrix} A_{kk} \\ A_{ik} \end{bmatrix}$ (in, out)
8: **for** $j = k + 1$ to $n_t - 1$ **do**
9: insert task: update A_{ij} (in, out) using A_{ik} (in) and A_{kj} (in)
10: **end for**
11: **end for**
12: **end for**

4. EXPERIMENTAL RESULTS

4.1. 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 with 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 with 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 the QUARK dynamic scheduler. The GCC 4.1.2 compiler was used for compiling the software stack, and Intel MKL (Composer XE 2013) was used to provide an optimized implementation of serial BLAS. On both systems, to avoid important variation in NUMA effects, the PLASMA

DOI: 10.1002/cpe

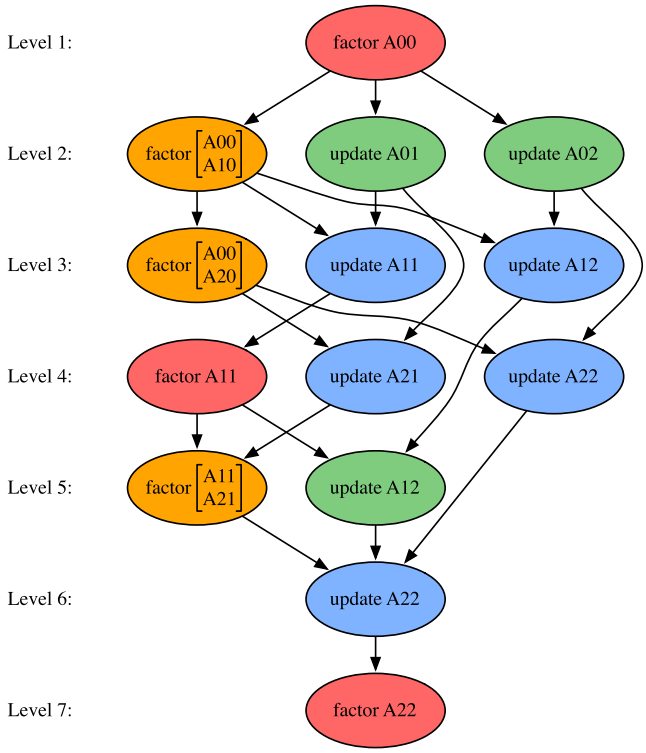


Figure 5. Directed acyclic graph for incremental pivoting of matrix with 3×3 tiles. Arrows show dependencies between tasks. Colors match those in Algorithm 3.

framework relies on the HwLoc library [49] to bind efficiently the threads over the multiple sockets, and all experiments are run with the *numactl* command to distribute the data in a round-robin fashion over the memory banks of the machine. When the number of threads is smaller than the machine size, the memory is only dispatched on the associated memory banks and not on all the machine.

4.2. Performance

We now study the performance of our implementations on square random matrices in double real precision, and compare their performance with that of the LU factorization in MKL (`DGETRF`). While the absolute performance is different for single, single-complex, and double-complex precisions, we have observed trends similar to double precision for all precisions. In all cases, the matrix is initially in column-major layout, and we include the conversion to and from tiled layout in the time. Because each of our implementations uses a different pivoting strategy, for a fair performance comparison, we ran iterative refinement with all the algorithms to achieve the same level of accuracy. 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`, that is, 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 $r = A\widehat{x} - b$ is the residual vector and $\widehat{x}$ is the computed solution, **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.

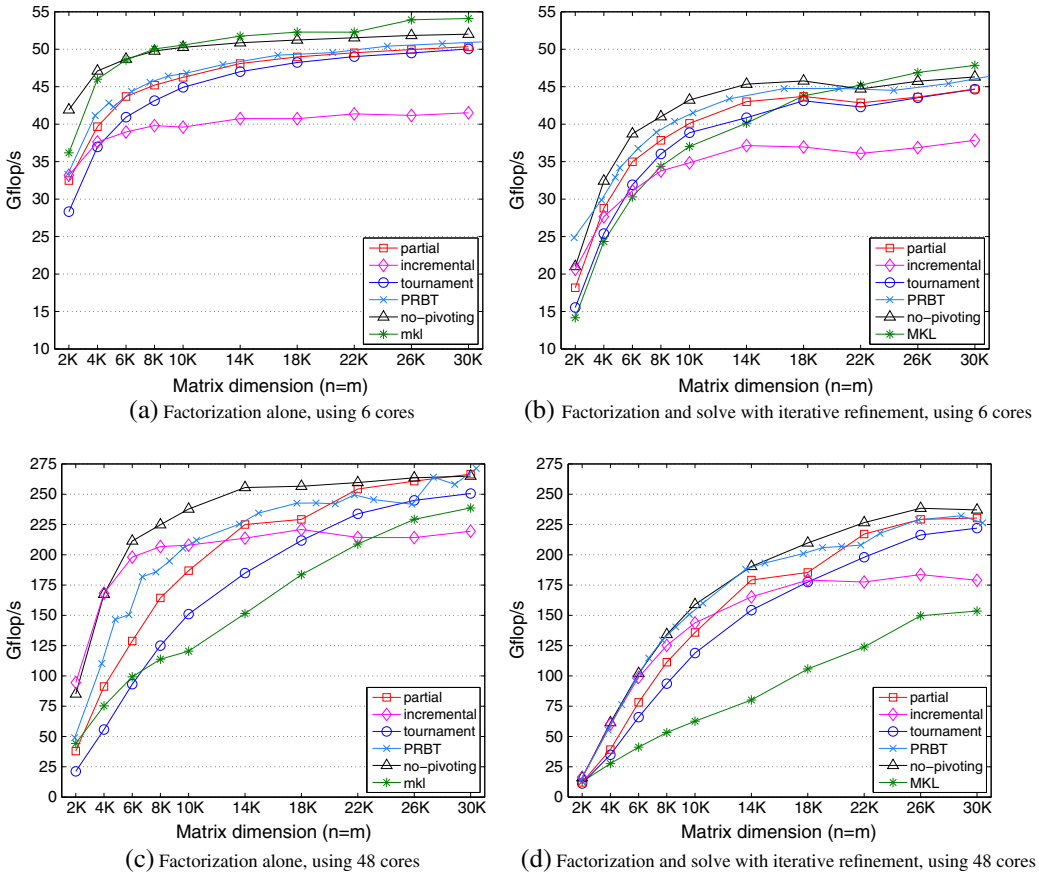
For all of our experiments, iterative refinement converged in less than 10 iterations. We observed that even with partial pivoting, it requires a couple of iterations to satisfy this stopping crite-

DOI: 10.1002/cpe

ria (Section 4.3). 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 n_b. Hence, for each matrix dimension n on a different number of cores, we studied the performance of each algorithm with the tile sizes of $n_b = 80$, 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 $n_b = 340$, 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 $i_b = 10$, 20, and 40 for both $n_b = 80$ and 160; $i_b = 10$, 20, 30, 40, 60, and 80 for $n_b = 240$; and $i_b = 10$, 20, 40, and 80 for both $n_b = 320$ and 400. Figures 6 and 7 show the performance obtained using the tile and block sizes that obtained the highest performance. For all algorithms, we compute the effective Gflop/s using the flops for partial pivoting given in Table I. For the solve with iterative refinement, we also include a pair of forward and backward substitutions, $2n^2$. For the tile algorithms, we included the data layout conversion time as a part of the solution time. We summarize our findings in the succeeding texts:

- PRBT with the default transformation depth of two added only a small overhead over no-pivoting in all the test cases.
- In comparison with 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, because of the extra computation and special kernels required by incremental pivoting to update the trailing



(a) Factorization alone, using 6 cores

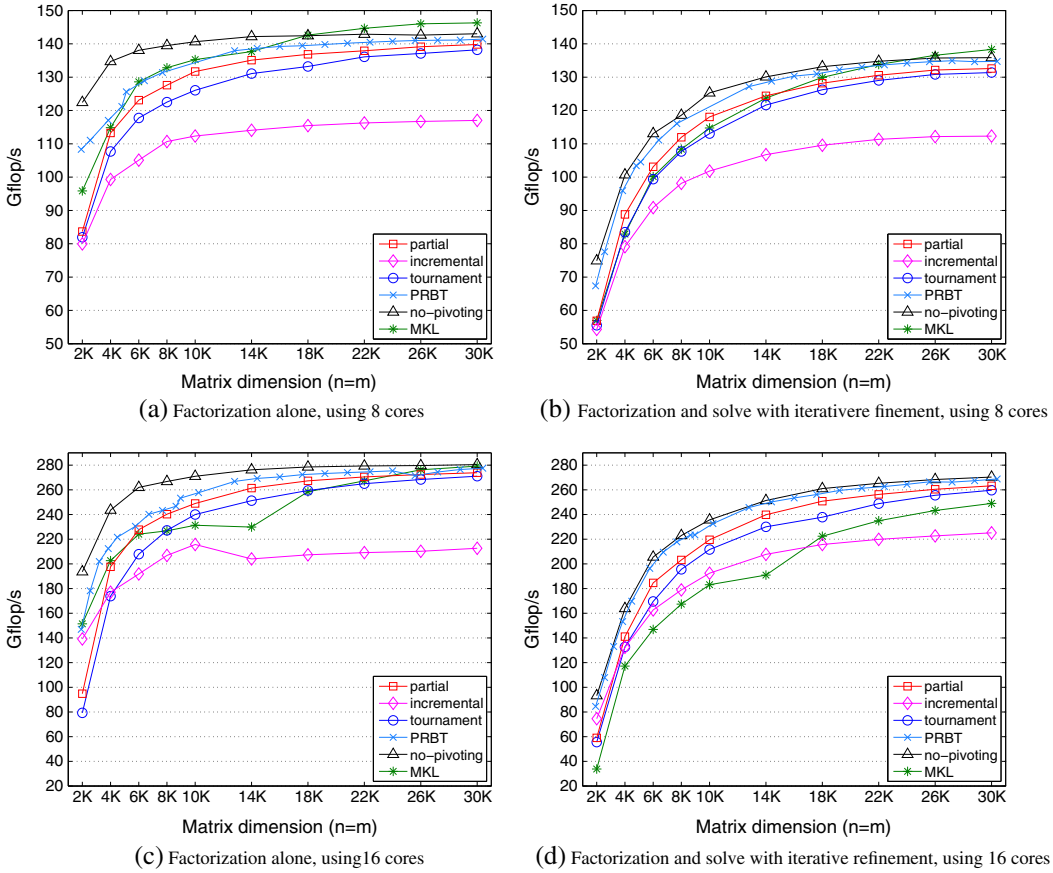
(b) Factorization and solve with iterative refinement, using 6 cores

(c) Factorization alone, using 48 cores

(d) Factorization and solve with iterative refinement, using 48 cores

Figure 6. (a) Factorization alone, using six cores, (b) factorization and solve with iterative refinement, using six cores, (c) factorization alone, using 48 cores, and (d) factorization and solve with iterative refinement, using 48 cores. Asymptotic performance comparison of LU factorization algorithms on AMD Opteron.

DOI: 10.1002/cpe



(a) Factorization alone, using 8 cores

(b) Factorization and solve with iterativere finement, using 8 cores

(c) Factorization alone, using16 cores

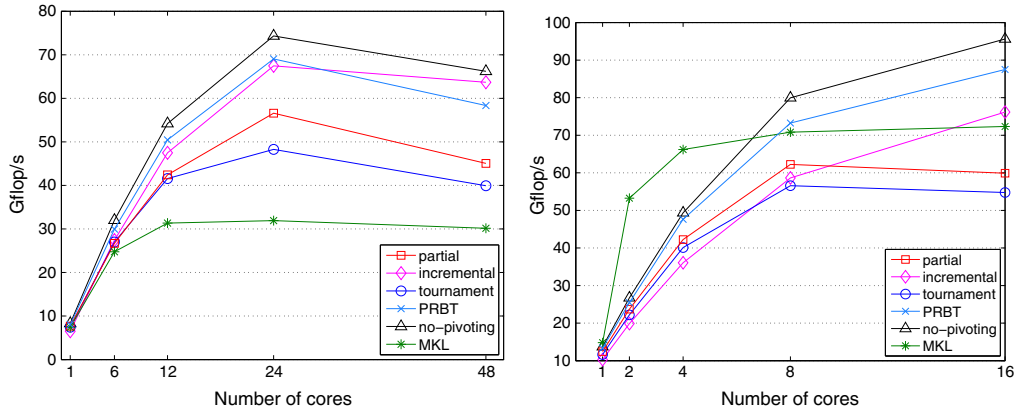
(d) Factorization and solve with iterative refinement, using 16 cores

Figure 7. (a) Factorization alone, using eight cores, (b) factorization and solve with iterative refinement, using eight cores, (c) factorization alone, using 16 cores, and (d) factorization and solve with iterative refinement, using 16 cores. Asymptotic performance comparison of LU factorization algorithms on Intel Sandy Bridge.

submatrix, its performance was lower than that of the partial or tournament pivoting LU that uses level 3 BLAS `DGEMM` for their trailing submatrix updates.

- In comparison with MKL, our partial pivoting and tournament pivoting LU could effectively utilize a larger number of cores, as seen for medium-sized matrices on multiple sockets. This is probably a result of the extra parallelism attained by using a superscalar scheduler, as well as parallel panel factorizations.
- In this shared-memory environment, partial pivoting outperformed tournament pivoting in all cases. However, this observation should not be extrapolated to distributed-memory machines, where the communication latency becomes a significant part of the performance. There, the reduced communication in tournament pivoting may be more favorable.
- MKL performed well for large matrices, where the trailing submatrix update dominates the performance. Note the combined cost for of all the panels is $O(n_b n^2)$, compared with $O(n^3)$ for the trailing submatrix update. Moreover, on a single socket, MKL outperformed no-pivoting for a large enough matrix dimension. This could be because a tiled implementation loses efficiency because of the smaller BLAS kernels used during its trailing submatrix updates.
- For small matrices, iterative refinement incurred a significant overhead, which diminishes for large matrices as the $O(n^3)$ factorization cost dominates the $O(n^2)$ solve cost. The refinement overhead was particularly large for MKL, which we suspect is due to $|A|\,|\widehat{x}| + |b|$ not being effectively parallelized in MKL.

DOI: 10.1002/cpe



(a) AMD Opteron ($m = n = 4000$). (b) Intel Sandy Bridge ($m = n = 2000$).

Figure 8. (a) AMD Opteron ($m = n = 4000$) and (b) Intel Sandy Bridge ($m = n = 2000$). Strong scaling comparison of LU factorization algorithms.

Finally, Figure 8 shows the strong scaling of our implementations.[‡] 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. Because the matrix dimensions were relatively small in this study, in order to obtain high performance, it becomes imperative to exploit parallelism as much as possible. In particular, PRBT and incremental pivoting obtained excellent parallel efficiency.

4.3. Accuracy

To study the numerical behavior of our implementations, we used the synthetic matrices from two recent papers [32, 34]. We tested all these matrices, but here present only a representative subset that demonstrate the numerical performance of the pivoting strategies. We also conducted the numerical experiments using all the matrix dimensions used in Section 4.2, but here show only the results of $n = 30000$, which represent the numerical performance trends of the pivoting strategies for all other matrix dimensions. Table II 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.* [34], 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.* [32], where the last test matrix `gfpp` is one of the pathological matrices that exhibits an exponential growth factor using partial pivoting. Because 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, 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 Sandy Bridge. All the results are in double real precision.

Figure 9(a) shows the component-wise backward errors, $\max_i |r_i|/(|A||\widehat{x}|+|b|)_i$, at each step of iterative refinement. 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 $[-0.5, 0.5]$. In the succeeding texts, we summarize our findings:

- For all the test matrices, tournament pivoting obtained initial backward errors comparable with 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

[‡] Using the default transformation depth of two, our current implementation of PRBT 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 PRBT.

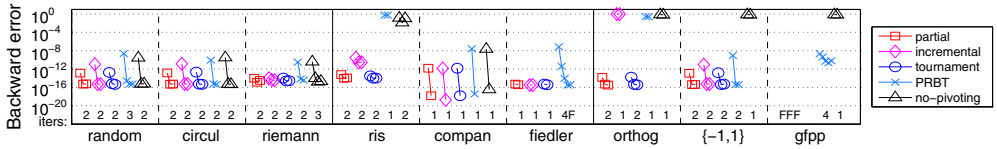
DOI: 10.1002/cpe

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

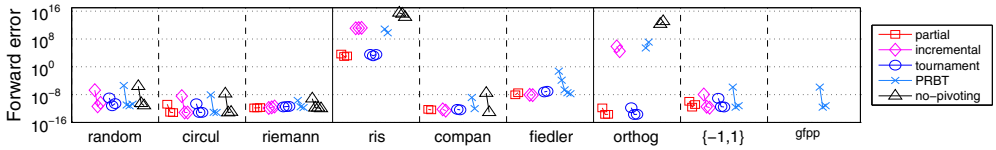
Name	Description	$\|A\|_1$	Cond$(A, 2)$	$\|L\|_1$	$\|L^{-1}\|_1$	max $\|U(i, j)\|$	max $\|U(i, i)\|$	$\|U\|_1$	Cond$(U, 1)$
`random`	dlarnv(2)	7.59e + 03	4.78e + 05	1.50e + 04	8.60e + 03	1.54e + 02	1.19e + 02	2.96e + 05	2.43e + 09
`circul`	gallery('circul', $1 : n$)	2.43e + 04	6.97e + 02	6.66e + 03	8.64e + 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}	$a_{ij} = -1$ 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`, $n = 1000$.

DOI: 10.1002/cpe



(a) Component-wise relative backward error.



(b) Norm-wise relative forward error.

Figure 9. (a) Component-wise relative backward error and (b) norm-wise relative forward error. Numerical accuracy of LU factorization algorithms, showing error before refinement (top point of each line) and after each refinement iteration (subsequent points). Number of iterations is given at bottom of (a); 'F' indicates failure (e.g., overflow).

significantly greater than those of partial pivoting but were improved after a few refinement iterations.

- 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 of the same order as those of partial pivoting after a few refinement iterations. The only exception was with the `ris` matrix, where the refinement stagnated before reaching an accuracy similar to that of partial pivoting. In each column of the `ris` matrix, entries with smaller magnitudes are closer to the diagonal (i.e., $a_{ij} = 0.5/(n - i - j + 1.5)$). As a result, the permutation matrix P of partial pivoting has ones on the antidiagonal.
- When no-pivoting was successful, its backward errors were similar to those of PRBT. On the other hand, PRBT 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, PRBT randomizes the original structure of the matrix and was able to compute the solution to a reasonable accuracy. It is also possible to construct pathological test matrices where partial pivoting is unstable while tournament pivoting is stable and vice versa [34].

Figure 9(b) shows the relative forward error norms of our implementations, which were computed as $\|x - \widehat{x}\|_\infty / \|x\|_\infty$. 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 refinement could not adequately improve the forward errors of incremental pivoting and PRBT. Also, even though the backward errors of the `ris` test matrix were on the order of machine epsilon with partial and tournament pivoting, their relative forward errors were $O(1)$ because of the large condition number.

5. CONCLUSIONS

When implemented well – using a recursive panel factorization, tile data layout, and dynamic scheduling – the canonical LU factorization with partial (row) pivoting is a numerically robust method that attains near the peak achievable performance given by LU factorization without pivoting.

In our experiments on synthetic matrices, tournament pivoting turned out to be as stable as partial pivoting, confirming results by its inventors. It also proved to be fairly fast. However, it failed to outperform partial pivoting, which can be attributed to the low communication cost in a shared-memory environment. The method has much more potential for distributed memory systems, where communication matters much more.

DOI: 10.1002/cpe

Incremental pivoting showed the worst asymptotic performance because of the use of exotic kernels, instead of the GEMM kernel. On the other hand, it showed strong scaling properties on small matrices almost as good as PRBT 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.

Partial Random Butterfly Transformation is the fastest method, both asymptotically and in terms of strong scaling, because it adds only a small overhead of preprocessing and postprocessing to the time of factorization without pivoting. Similar to incremental pivoting, it produces a high initial residual, but the accuracy can be recovered via iterative refinement. It can also fail in some 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.

6. FUTURE DIRECTIONS

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

While the comparison given here gives good insight into the performance properties of the different LU factorization algorithms using two relatively large shared memory systems, by today's standards, 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.

ACKNOWLEDGEMENTS

This work was supported in part by the US Department of Energy, the National Science Foundation, and the Intel Corporation.

REFERENCES

1. Grcar JF. Mathematicians of Gaussian elimination. *Notices of the AMS* June/July 2011; **58**(6):782–792.
2. Jaeger EF, Harvey RW, Berry LA, Myra JR, Dumont RJ, Philips CK, Smithe DN, Barrett RF, Batchelor DB, Bonoli PT, Carter MD, D'Azevedo EF, D'ippolito DA, Moore RD, Wright JC. Global-wave solutions with self-consistent velocity distributions in ion cyclotron heated plasmas. *Nuclear Fusion* 2006; **46**(7):S397–S408.
3. Quaranta E, Drikakis D. Noise radiation from a ducted rotor in a swirling-translating flow. *Journal of Fluid Mechanics* 2009; **641**:463.
4. Bendali A, Boubendir Y, Fares M. A feti-like domain decomposition method for coupling finite elements and boundary elements in large-size problems of acoustic scattering. *Computers & Structures* 2007; **85**(9):526–535.
5. Zhang Y, Taylor M, Sarkar T, Moon H, Yuan M. Solving large complex problems using a higher-order basis: parallel in-core and out-of-core integral-equation solvers. *IEEE Antennas and Propagation Magazine* 2008; **50**(4):13–30.
6. Barrett RF, Chan THF, D'Azevedo EF, Jaeger EF, Wong K, Wong RY. Complex version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Experience* April 10 2010; **22**(5):573–587.
7. Edelman A. Large dense numerical linear algebra in 1993: the parallel computing influence. *International Journal of High Performance Computing Applications* 1993; **7**(2):113–128.
8. Harrington R. Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine* June 1990; **32**:31–35.
9. Hess JL. Panel methods in computational fluid dynamics. *Annual Reviews of Fluid Mechanics* 1990; **22**:255–274.
10. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: a view from Berkeley. *Technical Report UCB/EECS-2006-183*, EECS Department, University of California: Berkeley, 2006.
11. Agullo E, Hadri B, Ltaief H, Dongarrra J. 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*. ACM: New York, NY, USA, 2009; 1–12.
12. Wilkinson JH. Error analysis of direct methods of matrix inversion. *Journal of the ACM* 1961; **8**(3):281–330.

DOI: 10.1002/cpe

13. Trefethen L, Schreiber R. Average case analysis of Gaussian elimination. *SIAM Journal on Matrix Analysis and Applicatiions* 1990; **11**(3):335–360.
14. Yeung MC, Chan TF. Probabilistic analysis of Gaussian elimination without pivoting. *Technical Report CAM95-29*, Department of Mathematics, University of California: Los Angeles, 1995.
15. Dongarra J, Eijkhout V, Luszczek P. Recursive approach in sparse matrix LU factorization. *Science Program* January 2001; **9**:51–60.
16. Gustavson FG. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* 1997; **41**(6):737–756.
17. Dongarra J, Faverge M, Ltaief H, Luszczek P. Exploiting fine-grain parallelism in recursive LU factorization. *Parco 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.
18. Dongarra J, Faverge M, Ltaief H, Luszczek P. Exploiting fine-grain parallelism in recursive LU factorization. *Advances in Parallel Computing, Special Issue* 2012; **22**:429–436.
19. Dongarra J, Faverge M, Ltaief H, Luszczek P. 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. ACM: New York, NY, USA, 2011; 33–42.
20. Luszczek P, Dongarra J. Anatomy of a globally recursive embedded LINPACK benchmark. *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.
21. Castaldo AM, Whaley RC. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'10*. ACM: Bangalore, India, January 2010. (submitted to ACM TOMS).
22. Buttari A, Langou J, Kurzak J, Dongarra JJ. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel in Computing, Systems and Application* 2009; **35**:38–53.
23. Joffrain T, Quintana-Orti ES, van de Geijn RA. Rapid development of high-performance out-of-core solvers. In *Applied Parallel Computing. State of the Art in Scientific Computing*, vol. 3732, Dongarra J, Madsen K, Waśniewski J (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2006; 413–422.
24. Sorensen DC. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers* March 1985; **C-34**(3):274–278.
25. Yip EL. FORTRAN subroutines for out-of-core solutions of large complex linear systems. *Technical Report CR-159142*, NASA, 1979.
26. Agullo E, Augonnet C, Dongarra J, Faverge M, Langou J, Ltaief H, Tomov S. LU factorization for accelerator-based systems. *Proceedings of the 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA'11)*, December 2010; 217–224. Best Paper award.
27. Quintana-Ortí G, Quintana-Ortí ES, Geijn Robert AVD, Zee FGV, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software* July 2009; **36**:14:1–14:26.
28. Demmel J, Grigori L, Hoemmen M, Langou J. Implementing communication-optimal parallel and sequential QR factorizations. *Arxiv preprint arXiv:0809.2407* 2008.
29. Grigori L, Demmel JW, Xiang H. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 2008; 29.
30. Donfack S, Grigori L, Gupta AK. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010; 1–10.
31. Parker DS. A randomizing butterfly transformation useful in block matrix computations. *Technical Report CSD-950024*, Computer Science Department, University of California, 1995.
32. Baboulin M, Dongarra J, Herrmann J, Tomov S. Accelerating linear system solutions using randomization techniques. *ACM Transactions on Mathematical Software* 2013; **39**:8:1–8:13.
33. Khabou A, Demmel J, Grigori L, Gu M. Lu factorization with panel rank revealing pivoting and its communication avoiding version. *Technical Report UCB/EECS-2012-15*, EECS Department, University of California: Berkeley, 2012.
34. Grigori L, Demmel JW, Xiang H. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications* 2011; **32**:1317–1350.
35. Demmel JW. *Applied Numerical Linear Algebra*. SIAM, 1997.
36. Wilkinson JH. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.
37. Gustavson FG. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications*. Kluwer Academic Publishers: Ottawa, Canada, October 2-4 2000; 211–234.
38. Gustavson FG, Karlsson L, Kågström B. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software* 2012; **38**(3):article 17.
39. Buttari A, Langou J, Kurzak J, Dongarra JJ. Parallel tiled QR factorization for multicore architectures. *Concurrency Computation: Practice and Experience* 2008; **20**(13):1573–1590.
40. Kurzak J, Ltaief H, Dongarra JJ, Badia RM. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computation: Practice and Experience* 2009; **21**(1):15–44.
41. Haidar A, Ltaief H, Dongarra J. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of SC '11*. ACM: New York, NY, USA, 2011; 8:1–8:11.

DOI: 10.1002/cpe

42. Haidar A, Ltaief H, Luszczek P, Dongarra J. 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*. IEEE Computer Society: Shanghai, China, May 21-25 2012; 25–35.
43. Ltaief H, Kurzak J, Dongarra J. Parallel band two-sided matrix bidiagonalization for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems* April 2010; **21**(4).
44. Luszczek P, Ltaief H, Dongarra J. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society: Anchorage, Alaska, USA, Unknown Month May 16; 944–955.
45. Badia RM, Herrero JR, Labarta Js, Pérez JM, Quintana-Ortí ES, Quintana-Ortí G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience* 2009; **21**(18):2438–2456.
46. Augonnet C, Thibault S, Namyst R, Wacrenier P. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computation: Practice and Experience* 2011; **23**(2):187–198.
47. Chan E, Van Zee FG, Quintana-Orti ES, Quintana-Orti G, Van De Geijn R. Satisfying your dependencies with SuperMatrix. In *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007; 91–99.
48. YarKhan A, Kurzak J, Dongarra J. QUARK users' guide: QUeueing And Runtime for Kernels. *Technical Report ICL-UT-11-02*, Innovative Computing Laboratory, University of Tennessee, 2011.
49. Broquedis F, Clet-Ortega J, Moreaud S, Furmento N, Goglin B, Mercier G, Thibault S, Namyst R. hwloc: a generic framework for managing hardware affinities in HPC applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. IEEE: Pisa, Italie, February 2010.

DOI: 10.1002/cpe