CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **26**:1408–1431
Published online 18 September 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3110

Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting

Jack Dongarra [1], Mathieu Faverge [1], Hatem Ltaief [2] and Piotr Luszczek [1,*,†]

[1] *Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA*
[2] *KAUST Supercomputing Laboratory, Thuwal, Saudi Arabia*

SUMMARY

The LU factorization is an important numerical algorithm for solving systems of linear equations in science and engineering and is a characteristic of many dense linear algebra computations. For example, it has become the *de facto* numerical algorithm implemented within the LINPACK benchmark to rank the most powerful supercomputers in the world, collected by the TOP500 website. Multicore processors continue to present challenges to the development of fast and robust numerical software due to the increasing levels of hardware parallelism and widening gap between core and memory speeds. In this context, the difficulty in developing new algorithms for the scientific community resides in the combination of two goals: achieving high performance while maintaining the accuracy of the numerical algorithm. This paper proposes a new approach for computing the LU factorization in parallel on multicore architectures, which not only improves the overall performance but also sustains the numerical quality of the standard LU factorization algorithm with partial pivoting. While the update of the trailing submatrix is computationally intensive and highly parallel, the inherently problematic portion of the LU factorization is the panel factorization due to its memory-bound characteristic as well as the atomicity of selecting the appropriate pivots. Our approach uses a parallel fine-grained *recursive* formulation of the panel factorization step and implements the update of the trailing submatrix with the *tile* algorithm. Based on conflict-free partitioning of the data and lockless synchronization mechanisms, our implementation lets the overall computation flow naturally without contention. The dynamic runtime system called QUARK is then able to schedule tasks with heterogeneous granularities and to transparently introduce algorithmic lookahead. The performance results of our implementation are competitive compared to the currently available software packages and libraries. For example, it is up to 40% faster when compared to the equivalent Intel MKL routine and up to threefold faster than LAPACK with multithreaded Intel MKL BLAS. Copyright © 2013 John Wiley & Sons, Ltd.

Received 19 October 2011; Revised 1 July 2013; Accepted 12 July 2013

KEY WORDS: recursion; LU factorization; parallel linear algebra; shared memory synchronization; threaded parallelism

1. INTRODUCTION

The multicore era has forced the scientific software community to reshape their state-of-the-art numerical libraries to be able to address the increasing parallelism of the hardware as well as the complex memory hierarchy design brought by this architecture. Indeed, LAPACK [1] has shown scalability limitations on such platforms and can only achieve a small portion of the theoretical peak performance [2]. The reasons for this are mainly threefold: (1) the overhead of its fork-join model of parallelism; (2) the coarse-grained task granularity; and (3) the memory-bound nature of the panel factorization.

*Correspondence to: Piotr Luszczek, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA.
†E-mail: luszczek@eecs.utk.edu

Successful high-performance results have already been reported for one-sided factorizations (e.g., QR/LQ, LU, and Cholesky factorizations) and, more recently, for the tridiagonal reduction needed to solve the symmetric eigenvalue problems [3] and the bidiagonal reduction required for the singular value decomposition [4–6]. These implementations are based on tile algorithms, which operate on the original matrix using small square regions called tiles; they alleviated the bottlenecks of and block algorithms on column-major storage by bringing the parallelism to the fore, minimizing the synchronization overhead, and relying on dynamic scheduling of fine-grained tasks. The data within the tiles can be contiguously stored in memory (i.e., tile data layout) or left as is, following the standard column-major format, which makes the data layout completely independent of the formulation of the numerical algorithms. However, the panel factorization phase of a block algorithm has not really been improved for the one-sided factorizations, which is mostly critical for LU and QR. It is still constrained by memory-bound operations and runs sequentially. The performance impact of the sequential panel for one-sided factorizations is somewhat minimal, though, and mostly hidden by the large amount of fine-grained parallelism introduced in the update of the trailing submatrix.

However, the performance gain comes at the price of numerical accuracy, particularly for the LU factorization. Indeed, 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 pairwise pivoting [7], which is related to an updating LU for out-of-core computations [8] when blocking of computations has to be performed for better performance. It has resurfaced in the form of what is now called the incremental pivoting [9] strategy 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 [9–11].

This paper presents a new approach for computing the LU factorization on multicore architectures, which not only improves the overall performance compared to the state-of-the-art implementations but also achieves the numerical quality of the standard LU factorization algorithm with partial pivoting. The originality of this work resides in the improvement of the panel factorization with partial pivoting. Involving mostly Level 2 BLAS operations, the parallelization of the panel is very challenging because of the low ratio between the amount of transferred data from memory and the actual computation. The atomicity of selecting the appropriate pivots is yet another issue, which has prevented efficient parallel implementation of the panel.

Our approach uses a parallel fine-grained *recursive* formulation of the panel factorization step while the update of the trailing submatrix follows the *tile* algorithm principles, that is, redesign of block algorithms using efficient numerical kernels operating on small tiles and scheduled through a dynamic scheduler. The fine-grained computation occurs at the level of the small caches associated with the cores, which may potentially engender super-linear speedups. The recursive formulation of the panel allows one to take advantage of multiple levels of the memory hierarchy and to cast memory-bound kernels into Level 3 BLAS operations to increase the computational rate even further. Based on a conflict-free partitioning of the data and lockless synchronization mechanisms, our implementation allows the parallel computation to flow naturally without contention and reduces synchronization. The dynamic runtime system is then able to schedule sequential tasks (from the update of the trailing submatrix) and parallel tasks (from the panel factorization) with heterogeneous granularities. The execution flow can then be modeled with a directed acyclic graph (DAG), where nodes represent computational tasks and edges define the dependencies between them. The DAG is never realized in its entirety because its size grows cubically with the matrix size. As the computation progresses, the DAG is unrolled progressively to initiate lookahead between subsequent steps of the factorization. Only the tasks located within a particular window are therefore instantiated. This window size may be tuned for maximum performance. Moreover, the scheduler can transparently integrate algorithmic lookahead in order to overlap successive computational steps and to keep all processing cores busy during the execution time as much as possible.

The remainder of this paper is organized as follows. Section 2 gives an overview of similar projects in this area. Section 3 recalls the algorithmic aspects and the pivoting schemes of the existing block LU (LAPACK) and tile LU (PLASMA) factorizations. Section 4 describes our new approach to compute the tile LU factorization with partial pivoting using a parallel recursive panel.

Section 5 presents some implementation details. Section 6 presents performance results of the overall algorithm. Also, comparison tests are run on shared memory architectures against the corresponding routines from LAPACK [1] and the vendor library Intel Math Kernel Library (MKL) [12]. Finally, Section 7 summarizes the results of this paper and describes the ongoing work.

2. RELATED WORK

This section presents related work in implementing a recursive and/or parallel panel of the LU factorization as well as tile algorithms.

The PLASMA library [13, 14] initiated with other software packages like FLAME [15], this effort of redesigning standard numerical algorithms to match the hardware requirements of multicore architectures. For runtime scheduling of the kernel tasks, we use an implementation of DAG-based scheduler called QUARK [16, 17], which is also used in PLASMA.

Recursive formulation of the QR factorization and its parallel implementation has been performed on a shared memory machine [18]. In relation to our work presented here, we see three main differences. First, our panel factorization is parallel. This leads to the second difference: our use of nested parallelism. And third, we use dynamic DAG scheduling instead of the master–worker parallelization.

Georgiev and Wasniewski [19] presented a recursive version of the LU decomposition. They implemented recursive versions of the main LAPACK and BLAS kernels involved in the factorization, that is, xGETRF and xGEMM, xTRSM, respectively. Their original code is in Fortran 90, and they relied on the compiler technology to achieve the desired recursion. Unlike in our case, their focus was a sequential implementation.

Recursion was also successfully used in the context of sparse matrix LU factorization [20]. It lacked pivoting code, which is essential to ensure numerical stability of our implementation. In addition, here, we focus on dense matrices only – not the sparse ones.

A distributed memory version of the LU factorization has been attempted and compared against ScaLAPACK's implementation [21]. One problem cited by the authors was excessive, albeit provably optimal, communication requirements inherent in the algorithm. This is not an issue in our implementation because we focus exclusively on the shared memory environment. Similarly, our open source implementation of the high-performance LINPACK benchmark [22] uses recursive panel factorization on local data, thus avoiding the excessive communication cost.

More recently, panel factorization has been successfully parallelized and incorporated into a general LU factorization code [23] using a careful rewrite of Level 1 BLAS that takes into account the intricacies of cache hierarchies of modern multicore chips. It uses a flat parallelism model with fork-join execution that is closely related to Bulk Synchronous Processing [24]. The authors refer to their approach as Parallel Cache Assignment. Our work differs in a few key aspects. We employ recursive formulation [25] and therefore are able to use Level 3 BLAS as opposed to just Level 1 BLAS. The technical reason for this lies in the design decision to maintain clear division between Level 1 BLAS calls of the panel (the un-blocked code) and Level 3 BLAS of the block update. The recursive formulation allows us to do away with the notion of the un-blocked panel code and have all the updates performed through Level 3 BLAS calls. Another important difference is the nested parallelism with which we have the flexibility to allocate only a small set of cores for the panel work while other cores carry on with the remaining tasks such as the Schur complement updates. Finally, we use dynamic scheduling that executes fine-grained tasks asynchronously, which is drastically different from a fork-join parallelism. A more detailed account of the differences is given in Section 4.

Chan *et al.* [26] implemented the classic LU factorization with partial pivoting (within the FLAME framework), in which the authors separate the runtime environment from the programmability issues (i.e., the generation of the corresponding DAG). There are mainly two differences with the work presented in this paper: (1) their lookahead opportunities are determined by sorting the enqueued tasks in a separate stage called an *analyzer phase*, while in our case, the lookahead occurs naturally at runtime during the process of pursuing the critical path of the DAG (and can also be strictly enforced by using priority levels), and (2) we do not require a copy of the panel, called a *macroblock*, in standard column-major layout in order to determine the pivoting sequence, but, rather, we had implemented an optimized parallel memory-aware kernel, which performs an *in-place* LU panel factorization with partial pivoting.

A more recent attempt at parallel LU factorization using the recursive formulation has been attempted but only on a limited number of cores [27] and without the use of DAG and dynamic scheduling of tasks.

3. THE BLOCK AND TILE LU FACTORIZATIONS

This section describes the block and tile LU factorization as implemented in the LAPACK and PLASMA libraries, respectively.

3.1. The block LU from LAPACK

Development of block algorithms was informed with the emergence of cache-based architectures [1]. They are characterized by a sequence of panel-update computational phases. The panel phase calculates all transformations using mostly memory-bound operations and applies them as a block to the trailing submatrix during the update phase. This panel-update sequence introduces unnecessary synchronization points, and lookahead is prevented, while it can be conceptually achieved. Moreover, the parallelism in the block algorithms implemented in LAPACK resides in the BLAS library, which follows the fork-join paradigm. In particular, the block LU factorization is no exception, and the atomicity of the pivot selection has further exacerbated the problem of the lack of parallelism and the synchronization overhead. At the same time, the LU factorization is numerically stable in practice and produces a reasonable growth factor. Finally, the LAPACK library also uses the standard column-major layout from Fortran, which may not be appropriate in the current and next generation of multicore architectures.

3.2. The tile LU from PLASMA

Tile algorithms with tile data layout implemented in PLASMA [13] propose to take advantage of the small caches associated with the multicore architecture. The general idea is to arrange the original dense matrix into small square regions of data that are contiguous in memory. This is performed to allow efficiency by allowing the tiles to fit into the core's caches. Figure 1 shows how the translation proceeds from column-major to tile data layout. In fact, as previously discussed in Section 1, the data layout is completely independent of the formulation of the algorithms, and therefore, tile algorithms can likewise operate on top of LAPACK layout (column-major). Breaking the matrix into tiles may require a redesign of the standard numerical linear algebra algorithms. Furthermore, tile algorithms allow parallelism to be brought to the fore and expose sequential computational fine-grained tasks to benefit from any dynamic runtime system environments, which will eventually schedule the different tasks across the processing units. The actual framework boils down to scheduling a DAG, where

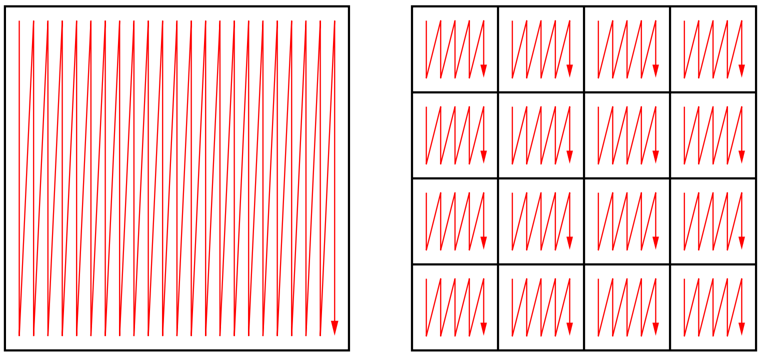


Figure 1. Translation from LAPACK layout (column-major) to tile data layout.

tasks represent nodes and edges define the data dependencies between them. This may produce an out-of-order execution and therefore permits the removal of the unnecessary synchronization points between the panel and the update phases noticed in the LAPACK algorithms. Lookahead opportunities also become practical and engender a tremendous amount of concurrent tasks. Unlike the Cholesky factorization, the original QR and LU factorizations had to be redesigned with new kernel implementations to work on top of a tile data layout. The tile QR factorization is based on orthogonal transformations, and therefore, it did not numerically suffer from the necessary redesign. However, the tile LU factorization has seen its pivoting scheme completely revised. The partial pivoting strategy has been replaced by the incremental pivoting. It consists of performing pivoting in the panel computation between two tiles on top of each other, and this mechanism is reproduced further down the panel in a pairwise fashion. And obviously, this pivoting scheme may considerably degrade the overall stability of the LU factorization [9–11].

In essence then, the goal of our new LU implementation is to achieve high performance, comparable to PLASMA, while sustaining numerical stability of the standard LU implementation in LAPACK.

4. PARALLEL RECURSIVE LU FACTORIZATION OF A PANEL

Our central contribution is a parallel algorithm for LU factorization of a rectangular matrix. It is based on the sequential formulation [25] and offers a speedup on multicore hardware, which we observed missing from existing vendor and open source implementations. We present two different versions of the algorithm selected based on the data layout that is used to perform the factorization: column-major or tile layout.

4.1. Recursive algorithm and its advantages

Figure 2 shows a pseudo-code of our recursive implementation on column-major layout. This single piece of code is executed by all threads simultaneously, and the function calls with the split prefix simply denote the fact that each thread works on its local portion of the data. The combine prefix indicates a reduction operation among the threads. In line 1, we see the parameters to the function: current matrix dimensions M, N and the current column number. Line 2 allows to stop the recursion when a single-column case is reached. That case is then handled by lines 3 through 5. In line 3, each thread selects an element of maximum magnitude in its own portion of the current column of the

```
 1: function xGETRFR(M, N, column) {
 2:    if N == 1 {                                  single column, recursion stops
 3:       idx = split_IxAMAX(...)                   compute local maximum of modulus
 4:       gidx = combine_IxAMAX(idx)                combine local results
 5:       xSCAL(...)                                scale local data
 6:    } else {
 7:       xGETRFR(M, N/2, column)                   recursive call to factor left half
 8:       xLASWP(...)                               pivoting forward
 9:       xTRSM(...)                                triangular solve
10:       xGEMM(...)                                Schur's complement
11:       xGETRFR(M, N-N/2, column+N/2)             recursive call to factor right half
12:       xLASWP(...)                               pivoting backward
13:    }
14: }
```

Figure 2. Pseudo-code for the recursive panel factorization on column-major layout.

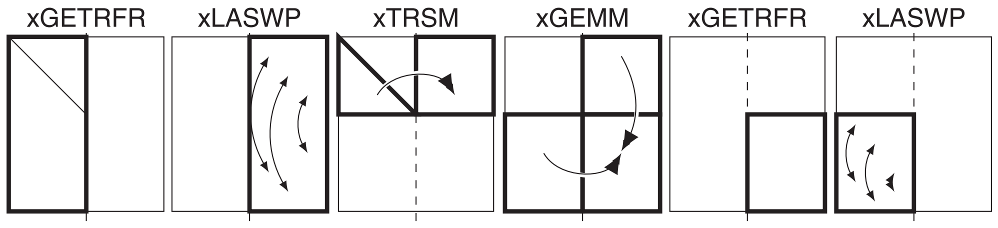


Figure 3. Visual representation of the steps of the recursive panel factorization on column-major layout.

input matrix A:

$$\mathsf{idx} = \max_{\mathsf{column} \leqslant i \leqslant \mathsf{M}} |A_{i,\mathsf{column}}|.$$

In line 4, all the local idx values are combined into a global (across threads) location of the maximum magnitude value: gidx, which is then used in line 5 to scale the matrix entries in the current column below the diagonal. Lines 7 through 12 constitute the code for multiple columns. It is rich in calls to the Level 3 BLAS. Line 7 factorizes the left half of the current matrix. Line 8 applies the pivoting sequence from the recursive call on the left half. Lines 9 and 10 perform the triangular solve and Schur complement update, respectively. Line 11 factorizes the lower right portion of the matrix through a recursive call, and line 12 applies the pivoting sequence from that call to the left half of the matrix. Figure 3 shows a visual representation of the steps of the factorization from lines 7 through 12.

From the computational complexity perspective, the panel factorization contributes only to the lower order term of the overall flop count: $O(\mathsf{N}^2)$ [28]. However, this still poses a problem in the parallel setting from both the theoretical [29] and practical standpoints [23]. To be more precise, the combined panel factorizations' complexity for the entire matrix is

$$O(\mathsf{N}^2\mathsf{NB}),$$

where N is the panel height (and the global matrix dimension) and NB is the panel width. For good performance of calls to BLAS in the Schur complement update, the panel width NB is commonly increased. This creates tension when the panel is a sequential operation because a wider panel creates a larger Amdahl fraction [30] and undermines scaling. Our own experiments revealed that it was a major obstacle to reasonable scalability of our implementation of the tile LU factorization with partial pivoting. This result is also consistent with related efforts [23].

Aside from gaining a high-level formulation that is free of low-level tuning parameters, recursive formulation removes the need for a higher-level tuning parameter commonly called an algorithmic blocking factor. It determines the panel width – a tunable value that is used for merging multiple columns of the panel together. Non-recursive panel factorizations could potentially use another level of tuning called *inner blocking* [2, 14]. This is avoided in our implementation by allowing the recursion to proceed until reaching a single-column of the panel.

4.2. Data and work partitioning

In the case of recursive LU, the challenging part of the parallelization is the fact that the recursive formulation suffers from inherent sequential control flow. This is unlike the column-oriented implementation employed by LAPACK and ScaLAPACK, which may simply be run in parallel by partitioning the loop iterations between threads. Instead, we use data partitioning to guide the parallelization. As the first step, we apply a block 1D partitioning technique that is similar to what has proven successful before [23]. The partitioning decouples the data dependence between threads, which is the key to reducing the overhead of concurrent access to data locations in the main memory.

The data partitioning was applied for the recursion-stopping case: the single-column factorization step in lines 3–5 of Figure 2. The recursive formulation of the LU algorithm poses another problem in the Schur complement update, namely an efficient use of Level 3 BLAS call for triangular solve:

xTRSM() (line 9 in Figure 2) and LAPACK's auxiliary routine for applying the pivot sequence called xLASWP() (lines 8 and 12). Both of these calls do not readily lend themselves to the 1D partitioning scheme because of two main reasons:

1. Each call to these functions occurs with a matrix size that keeps changing across calls during factorization of a single panel, and
2. 1D partitioning makes the calls dependent upon each other and thus creates synchronization overhead because concurrent access from multiple threads has to be orchestrated to preserve the code's correctness.

The latter problem is fairly easy to see as the pivoting requires data accesses across the entire column and the memory locations that are being accessed are unknown until execution time – a condition known as *runtime dependence*. Each swap of the pivot element requires coordination between the threads, among which the column is partitioned. For correctness, we have to assume that the swap involves data blocks assigned to two distinct threads. The former issue is more subtle, in that it involves overlapping regions of the matrix and that it creates a memory hazard – a situation whereby multiple writes and reads from multiple cores might have different results depending on the ordering of memory transactions by the main memory hardware, which arbitrates conflict resolution. That may, at times, be masked by the synchronization effects occurring in other portions of the factorization. The data overlap would occur if each column was divided to equalize the work load because the work is performed on the subdiagonal portion of the column, which keeps changing as the panel factorization progresses. To deal with both issues at once, we chose to use a fixed scheme for block 1D partitioning, with blocks of rows assigned to threads. This removes the need for extra synchronization and affords us a parallel execution, albeit a limited one due to the narrow size of the panel.

The Schur complement update is commonly implemented by a call to the Level 3 BLAS kernel called xGEMM(), and this is also a new function that is not present within the panel factorizations from LAPACK and ScaLAPACK. However, parallelizing this call is much easier than all the other new components of our panel factorization. We chose to reuse the across-columns 1D partitioning to simplify the management of overlapping memory references and to, again, reduce synchronization points.

To emphasize the observations, which we made throughout the preceding text, we consider the data partitioning among the threads to be of paramount importance. Unlike the Parallel Cache Assignment method [23], we do not perform an extra data copy to eliminate the shared memory effects that are detrimental to performance, such as Translation Look-aside Buffer (TLB) misses, false sharing of cache line, and so on. By choosing the recursive formulation, we rely instead on the Level 3 BLAS to perform these optimizations for us. Not surprisingly, this was also the goal of the original recursive algorithm and its sequential implementation [25]. What is left to do for our code is the introduction of parallelism that is commonly missing from the Level 3 BLAS when narrow rectangular matrices are involved – the very shape of matrices that occur in the recursive LU panel factorization.

Instead of low-level memory optimizations, we turned our focus toward avoiding synchronization points and letting the computation proceed asynchronously and independently as long as possible, until it is absolutely necessary to perform communication between threads. One design decision that stands out in this respect is the fixed partitioning scheme for the panel data. Within the panel being factored, regardless of the height of the pivoting-eligible portion of the current column, we always assign the same number of rows to each thread except for the first one, whose row-count decreases. Figure 4 shows that this causes a load imbalance as the thread number 0 has a progressively smaller amount of work to perform, as the panel factorization progresses from the first to the last column. This is counterbalanced by the fact that the panels are relatively tall compared to the number of threads, which makes the imbalance small in relative terms. Also, the first thread usually has greater responsibility in handling pivot bookkeeping and synchronization tasks.

Besides the previously mentioned considerations, the tile layout implementation of our algorithm requires extra care for efficient memory accesses. For one, the tile layout does not allow such flexibility in partitioning. Instead, the data distribution between threads strictly follows the tile structure,

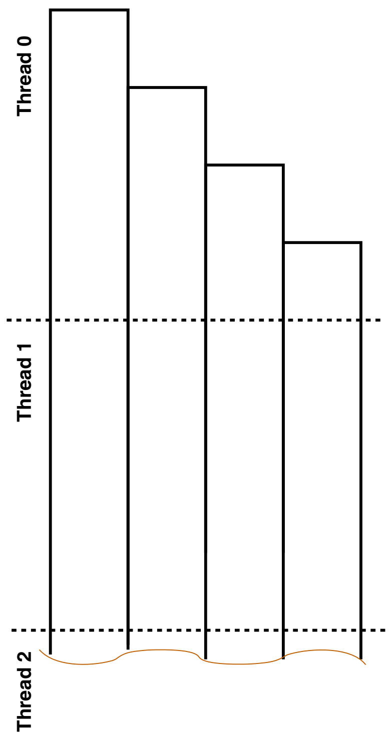


Figure 4. Fixed partitioning scheme used in the parallel recursive panel factorization.

```
 1: function xGETRFR(M, N, column) {
 2:    xGETRFR(M, N/2, column)                  recursive call to factor left half
 3:    xLASWP(...)                              pivoting forward
 4:    xTRSM(...)                               triangular solve on the first tile
 5:    foreach local tile {
 6:       xSCAL( N/2-1 )                        scale the last column of the left part
 7:       xGEMM(...)                            Schur complement
 8:       idx = split_IxAMAX( N/2 )             update local maximum of the first column in the right part
 9:    }
10:    gidx = combine_IxAMAX(idx)               combine local results
11:    xGETRFR(M, N-N/2, column+N/2)            recursive call to factor right half
12:    xLASWP(...)                              pivoting backward
13: }
```

Figure 5. Pseudo-code for the recursive panel factorization on tile layout.

and each thread handles a fixed number of tiles. Secondly, the number of cache misses due to this data storage can increase excessively when compared to the column-major data layout. The classic implementation performs the main stages of the algorithm one after another: scale the column, compute the Schur complement, and search the pivot. Each of these stages requires iteration over the tiles owned by a single thread – this would become three independent loops accessing all the tiles of the panel. Such an implementation loses the benefit of data locality. The solution is to intertwine the three stages to apply them in a single pass to each tile as shown in Figure 5. The pseudo-code in the figure is slightly rearranged from the one shown in Figure 2 for column-major layout. In particular, line 2 in Figure 5 is a recursive call to factor the left half of the columns of the current panel. Line 3 applies the pivoting sequence from the recursive call. Line 4 applies the triangular solve to the right

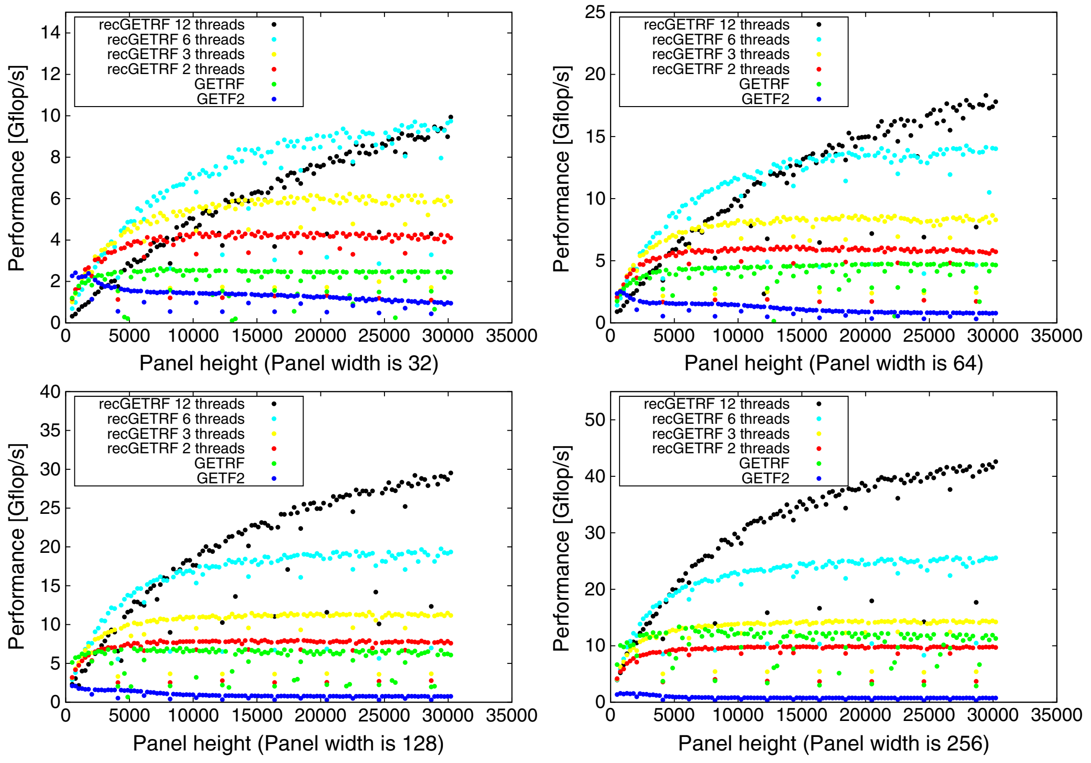


Figure 6. Scalability study on *MagnyCours-48* of the recursive parallel panel factorization in double precision on column-major layout with various panel widths: 32 (top left), 64 (top right), 128 (bottom left), and 256 (bottom right).

side of the panel columns. Lines 6 through 8 handle a single-column case locally, for each thread. And line 10 combines the local results. Line 11 factorizes the lower portion of the panel, and line 12 applies the resulting pivot sequence to the left portion of the panel.

4.3. Scalability results of the parallel recursive panel kernel

In this section, we study the scalability of our implementations of the recursive panel factorization in double precision using column-major layout (used in LAPACK) and tile layout (used in PLASMA) on three different architectures: *MagnyCours-48* composed of four AMD 12-core Opteron CPUs, *Xeon-16* composed of four quad-core Intel Xeon Core 2 Duo CPUs, and *SandyBridge-32* composed of two eight-core Intel Xeon SandyBridge CPUs (see Section 6.1 for detailed hardware specifications). Unless otherwise specified, all of our experiments are performed with the sequential Intel MKL Intel Corporation, Santa Clara, CA 10.3.6 on the three different architectures because it offered the highest performance rate when compared with the alternatives.

Figures 6 and 7 show a scalability study of our parallel recursive panel LU factorization with four increasing panel widths: 32, 64, 128, and 256, respectively. Both the column-major and tile layouts are compared against the equivalent routines from LAPACK: the un-blocked implementation GETF2 and the blocked version GETRF. Both are linked against the multi-threaded version of MKL for a fair comparison. The experiment is performed on the *MagnyCours-48* architecture, which has a theoretical peak of 8.4 Gflop/s per core. One Gflop/s represents the capability of the computer to perform 10^9 floating-point operations per second. For each of the different implementations of the algorithm, the number of operations used to compute the performance rate of each kernel is given by the formula [28]:

$$mn^2 - \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$$

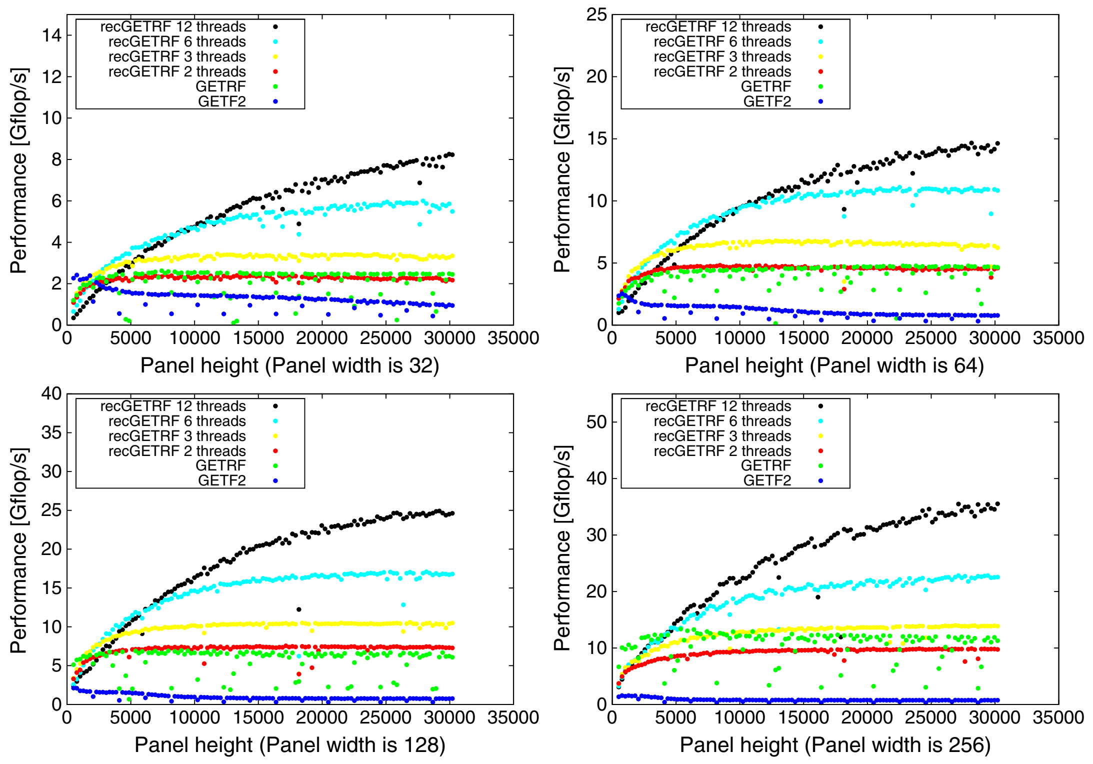


Figure 7. Scalability study on *MagnyCours-48* of the recursive parallel panel factorization in double precision on tile layout with various panel widths: 32 (top left), 64 (top right), 128 (bottom left), and 256 (bottom right).

where m is the number of rows and n is the number of columns. This formula has been used for all the experiments in this paper, even for the incremental pivoting algorithms that require more operations.

The idea is to highlight and understand the impact of the data layout on our recursive parallel panel algorithm. We limit our parallelism level to 12 cores (a single multicore socket) because our main factorization routine needs the remaining cores for updating the trailing submatrix. First, note how the parallel panel implementation, which is based on column-major layout, achieves better performance compared to the tile layout. Indeed, as shown in Figure 4 for the column-major layout, thread 0 loads into its local cache memory some data from the other threads' sections of the column in addition to its own data. This preloading of other threads' data prevents some of the cold cache misses. In contrast, the recursive parallel panel LU on top of tile layout may engender higher bus traffic because each thread needs to acquire its own data independently from the other threads, which increases the memory latency overhead. Second, when compared with the panel factorization routine xGETF2(), which uses Level 2 BLAS, we achieve super-linear speedup for a wide range of panel heights with the maximum efficiency exceeding 550%. In an arguably more relevant comparison against the xGETRF() routine, which could be implemented with mostly Level 3 BLAS, we achieve a perfect scaling for 2 and 4 threads compared to Intel's MKL and easily reach 50% efficiency for 6 and 12 threads. This is consistent with the results presented in the related work section [23].

Figure 8 presents a study of the efficiency of our panel factorization linked with AMD's ACML 5.1 (top) and with OpenBLAS 0.1alpha2 (bottom) in order to observe the influence of linking against the Intel MKL library. Even if the AMD's ACML 5.1 library has a similar performance to that of Intel's MKL on a single core and even if it has better performance for multi-threaded applications, the use of this library for our implementation significantly decreases the efficiency of the code. However, the OpenBLAS library improves, if only by a few flops, the performance of the algorithm for the tile data layout.

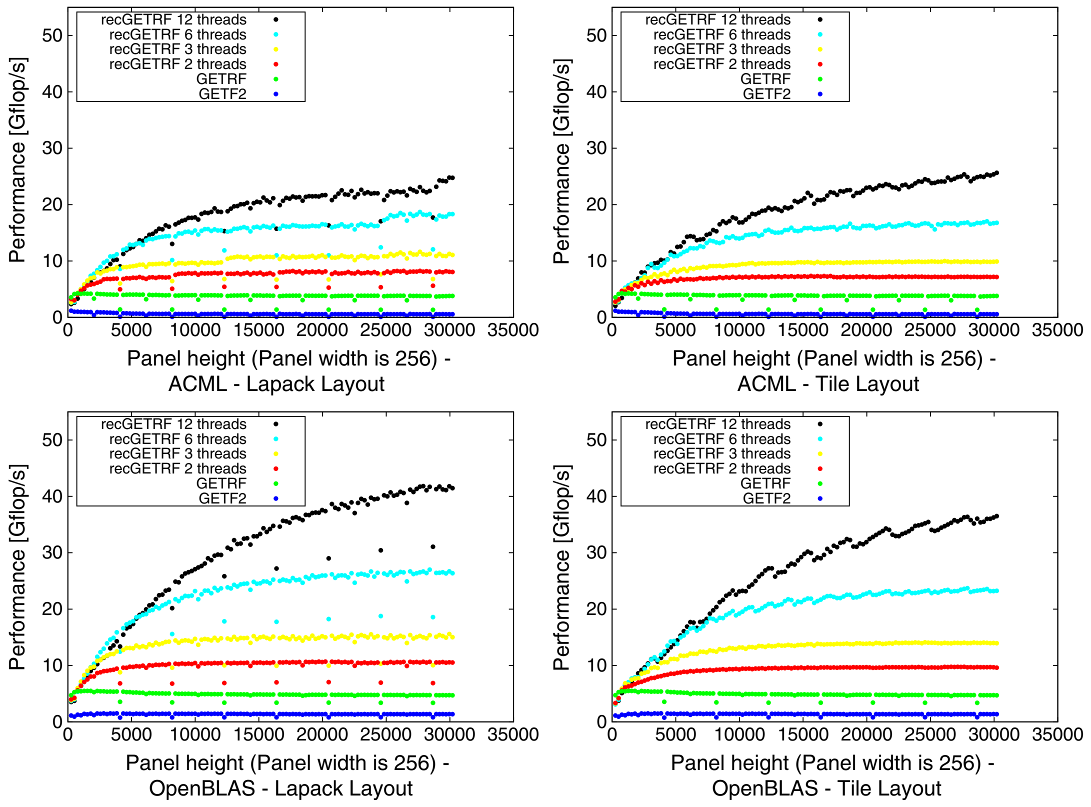


Figure 8. Scalability study on *MagnyCours-48* of the recursive parallel panel factorization in double precision for column-major (left) and tile (right) layouts linked with AMD ACML 5.1 (top) and OpenBLAS 0.1alpha2 (bottom) with a panel width of 256.

From now on, all our experiments will be performed solely with Intel MKL 10.3.6 library because the tested platforms are based on Intel processors and the OpenBLAS library does not support the SandyBridge architecture as of this writing. Such support is crucial to taking advantage of the new vector floating-point instruction set: Advanced Vector Extensions (AVX).

Figure 9 shows the experimental results on the last two architectures: *Xeon-16* in the top row and *SandyBridge-32* in the bottom row. Our algorithm has a good scalability for up to eight cores on *Xeon-16* but shows some limitations on 16 cores. For this reason, we will limit the number of threads involved in the panel factorization to 8 in the full factorization. On the SandyBridge architecture, the recursive panel factorization scales up to 16 cores and reaches its asymptotic value faster than on the other architectures, when the cores from only a single socket are used (eight cores or less). Note that on this architecture, the tile layout is improving the efficiency of the algorithm by more than 15%, thanks to the improved architecture of the memory bus.

4.4. *Further implementation details and optimization techniques*

We use the lockless data structures [31] exclusively throughout our code. This decision was dictated by the need for fine granularity synchronization, which occurs during the pivot selection for every column of the panel, and at the branching points of the recursion tree. Synchronization that uses mutex lock was deemed inappropriate for such frequent calls because contention increases the lock acquisition time drastically, and some implementations have the potential to incur a system call overhead.

Together with the lockless synchronization, we use *busy waiting* on shared memory locations to exchange information between threads by taking advantage of the cache coherency protocol of the memory subsystem. Even though such a scheme is fast in practice [23], it causes

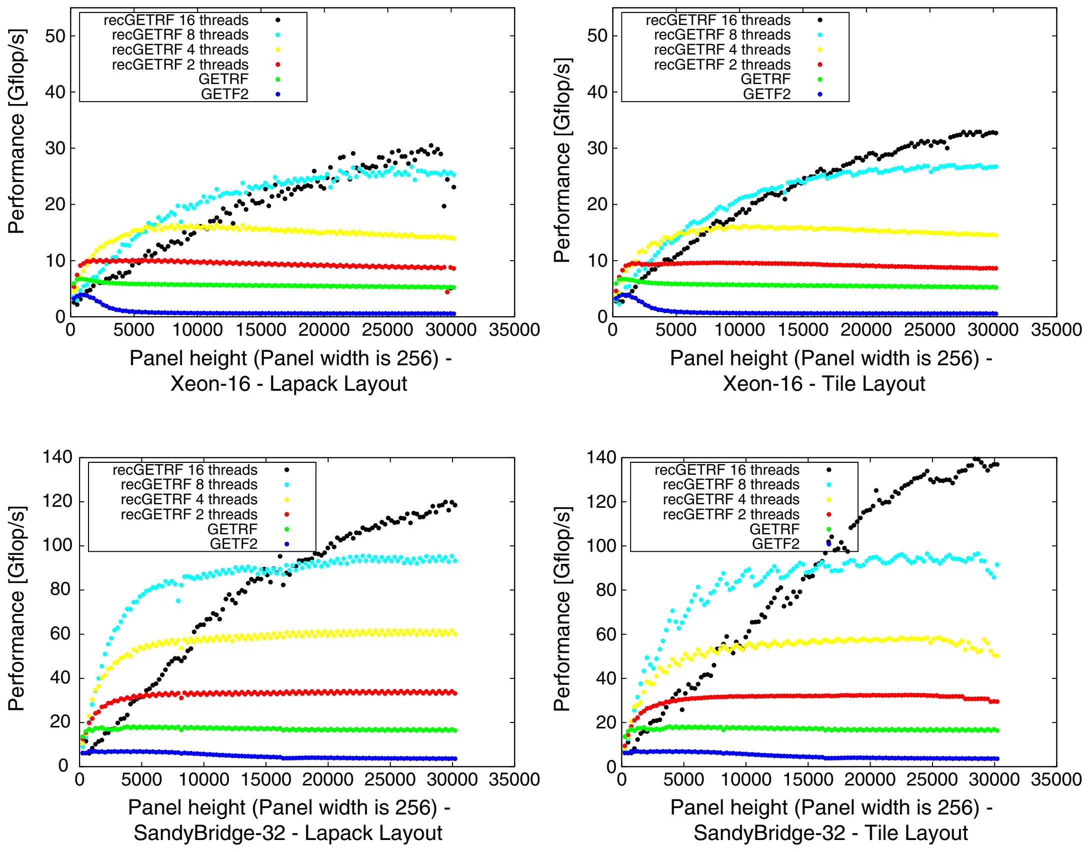


Figure 9. Scalability study of the recursive parallel panel factorization in double precision on column-major (left) and tile (right) layouts with a panel width of 256 on *Xeon-16* (top) and *SandyBridge-32* (bottom).

extraneous traffic on the shared memory interconnect, which we try to minimize. Therefore, we changed the occurrences of busy waiting into computations on independent data items. Invariably, this led to spurious memory–coherency traffic due to false sharing, as we reached such a fine granularity level of the parallelism. Regardless of the drawbacks, we feel that this is a satisfactory solution because our goal is the avoidance of busy waiting, which creates even greater demand for inter-core bandwidth. Principally, busy waiting has no useful work available in order to overlap it with the shared memory polling. We refer to this optimization technique as *delayed waiting*.

Another technique that we use to optimize the inter-core communication is what we call *synchronization coalescing*. The essence of this method is to conceptually group the unrelated pieces of code that require a synchronization into a single aggregate that synchronizes only once. The prime candidate for this optimization is the reduction operation that determines the pivot and the write operation of the pivot's index. Both of these operations require a synchronization primitive to guarantee consistency of information between threads. The former operation requires a parallel reduction operation, while the latter requires a global thread barrier. Neither of these is ever considered to be related to each other in the context of a sequential optimization. But with our synchronization coalescing technique, they are viewed as related in the communication realm and, consequently, are implemented in our code as a single operation.

Finally, we introduced a *synchronization avoidance* paradigm whereby we opt for multiple writes to shared memory locations instead of introducing a memory fence (a hardware instruction that flushes all the buffers used by cores to perform memory transactions to ensure a consistent view of the memory content among all the cores) or, potentially, a global thread barrier to ensure global data consistency.

Multiple writes are usually considered a hazard (the value stored in memory depends on which thread happened to be the last writer based on the conflict resolution hardware of the memory controller) and are not guaranteed to occur in a specific order in most of the consistency models adopted for shared memory systems. We completely sidestep this issue, however, as we guarantee algorithmically that each thread writes exactly the same value to the main memory. Clearly, this seems like an unnecessary overhead in general, but in our tightly coupled parallel implementation, this is a worthy alternative to either explicit (via inter-core messaging) or implicit (via memory coherency protocol) synchronization. In short, this technique is another addition to our contention-free design.

Portability or, more precisely, performance portability was also an important goal in our overall design. In our lock-free synchronization, we rely heavily on shared memory consistency – a seemingly problematic feature from the portability standpoint. To address this issue reliably, we make two basic assumptions about the shared memory hardware and the software tools. Both of the assumptions, to our knowledge, are satisfied by the majority of modern computing platforms. From the hardware perspective, we assume that memory coherency occurs at the cache line granularity. This allows us to rely on global and instantaneous visibility of loads and stores to nearby memory locations. From the software perspective, we assume that the compiler tool chain has an appropriate handling of C's `volatile` keyword. In other words, data structures marked with that keyword should not be cached but always loaded from memory. When combined with the use of primitive data types that are guaranteed to be contained within a single cache line, it is sufficient to prevent the unintended side effects resulting from the peculiarities of a given shared memory design or its implementation.

5. DYNAMIC SCHEDULING AND LOOKAHEAD

This section provides the details about the dynamic scheduling of the recursive tile LU factorization and the information about the runtime environment system that we use to efficiently schedule the computational tasks, regardless of the relative differences between their execution time.

5.1. *Discussion of implementation variants*

Our implementation may be qualified as a *tile algorithm* because of the way it accesses the matrix data and not because of the way the matrix is stored in memory. In fact, our algorithm works equally well on matrices stored using either the column-major or the tile data layouts.

Our code was originally formulated with the right-looking variant of LU factorization [32–34] in mind. This variant makes it easier to take advantage of the available parallelism. It was also chosen for LAPACK [1] and ScaLAPACK [35]. However, the execution flow of our code is driven by the data dependencies that are communicated to the QUARK runtime system [36]. This usually results in an asynchronous and out-of-order scheduling. The dynamic runtime environment ensures that enough parallelism is available throughout the entire execution (the right-looking variant), while a steady progress is made along the critical path for lookahead purposes (left-looking variant). Therefore, the strict right-looking variant available in LAPACK [1] and ScaLAPACK [35] cannot be guaranteed at the user-level code and is, in fact, discouraged. The asynchronous nature of the DAG execution provides sufficient lookahead opportunities for many algorithmic variants to coexist within the execution space regardless of the visitation order of the DAG [37].

5.2. *Overview of the parallel recursive tile LU factorization*

Figure 10 shows the initial factorization steps of a matrix subdivided into nine tiles (a three-by-three grid of tiles). The first step is a recursive parallel factorization of the first panel, marked in green. The panel consists of the three leftmost tiles. When the factorization of the first panel finishes, the other tasks may start executing, which creates an implicit synchronization point. To avoid the negative impact on parallelism, we execute this step on multiple cores (see Section 4 for further details) to minimize the running time. We use a nested parallelism model. Most of the tasks are executed by a single core, and only the panel tasks are assigned to multiple cores. Unlike similar implementations [23], we do not use all the available cores to handle the panel. There are two main reasons for

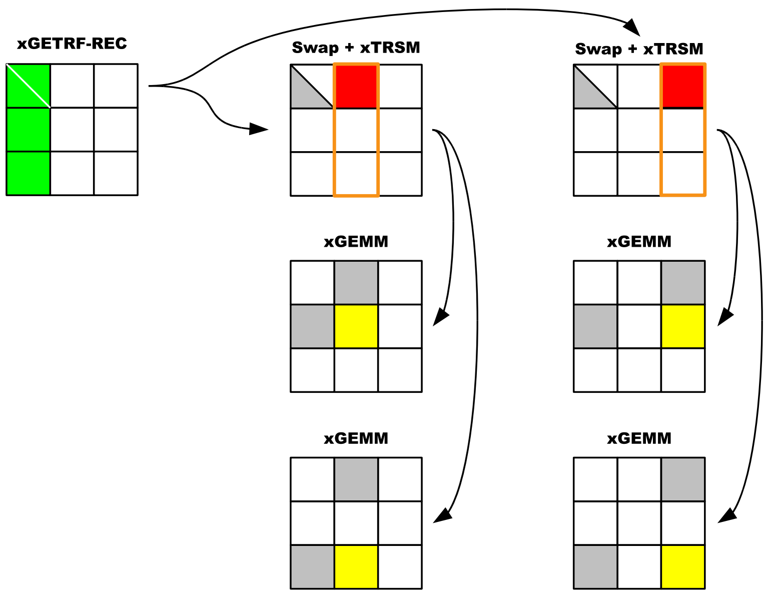


Figure 10. Breakdown of the execution tasks for the recursive tile LU factorization: Factorization of the first panel using the parallel kernel is followed by the corresponding updates to the trailing submatrix.

this decision. First, we use dynamic scheduling, which enables us to counter the negative influence of the panel factorization by hiding it behind more efficient work performed by the update tasks. Second, we have clearly observed the effect of diminishing returns when using too many cores for the panel. Consequently, we do not use all the available cores, and, instead, we keep the remaining cores busy with the update tasks.

The next step is the application of the pivoting sequence to the right of the panel that has just been factorized. In this step, we combine the pivot application with the triangular update (xTRSM in the BLAS parlance), because there is no advantage of scheduling it separately due to the cache locality considerations. Just as the panel factorization locks the panel and has a potential to temporarily stall the computation, the pivot interchanges may have a similar impact. This is indicated by an orange rectangular outline that encloses the tile updated by xTRSM as well as the tiles below it where the row interchange appears. Even though so many tiles are locked by the pivot swaps and the triangular update, there is still a potential for parallelism because both of these operations for a single column of tiles are independent of the same operations for other columns of tiles. We can then easily split the operations along the tile boundaries and schedule them as independent tasks. This observation is depicted in Figure 10 by showing two xTRSM updates in red for two adjacent tiles in the topmost row of tiles instead of one update for both tiles at once.

The last step shown in Figure 10 is an update based on the Schur complement. It is the most computationally intensive operation in the LU factorization and is commonly implemented with a call to a Level 3 BLAS kernel called xGEMM. Instead of a single call, which performs the whole update of the trailing submatrix, we use multiple invocations of the routine because we use a tile-based algorithm. In addition, by splitting the Schur update operation into many small tasks, we expose more parallelism, and we are able to alleviate the influence of the algorithm's synchronization points (such as the panel factorization). This results in better performance than is possible with a single call to a vendor library with parallel implementations of BLAS [2].

One thing not shown in Figure 10 is the pivoting to the left because it does not occur in the beginning of the factorization. Pivoting of the leftmost tiles is necessary for subsequent panels, including the second one. The swaps originating from different panels have to be ordered correctly but are otherwise independent for each column, which is the necessary condition for running them in parallel. The only inhibitor of parallelism, then, is the fact that the swapping operations are inherently memory bound because they do not involve any computation and incur substantial data

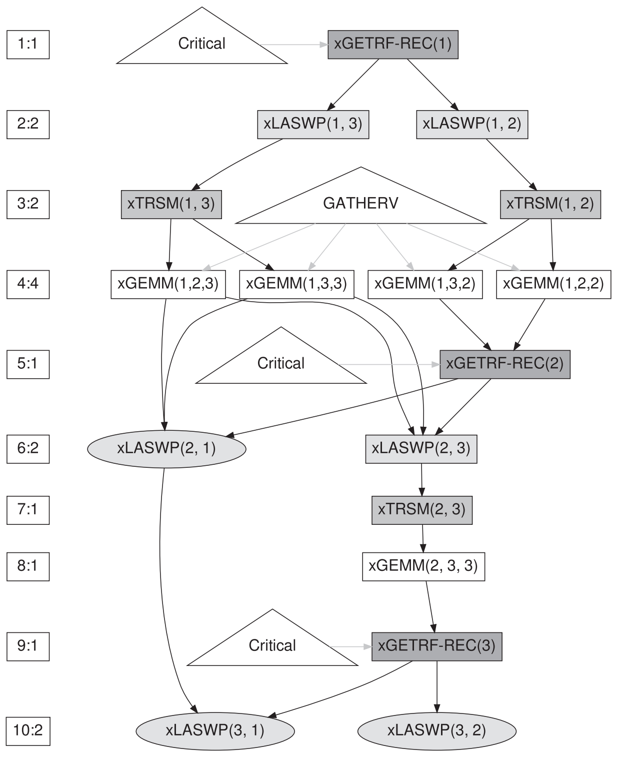


Figure 11. Annotated DAG for the parallel recursive tile LU factorization of a three-by-three tile matrix. The annotations are indicated with triangles.

movement. Furthermore, these memory accesses are performed with only a single level of indirection, which makes them very irregular in practice. Producing such memory traffic from a single core might not take advantage of the main memory's ability to handle multiple outstanding data requests and the parallelism afforded by Non-Uniform Memory Access (NUMA) hardware. It is also worth mentioning that the tasks performing the pivoting behind the panels are not located on the critical path and, therefore, are not essential for the remaining computational steps in the sense that they could potentially be delayed until the end of the factorization (see the tracing figures in Section 6.4). This is also highlighted in Figure 11, which shows the DAG of the parallel recursive tile LU factorizations of a three-by-three tile matrix. The nodes marked as xLASWP are the *end nodes* and do not directly contribute to the completion of the factorization.

5.3. QUARK: parallel runtime system for dynamic scheduling

Our approach to extracting parallelism is based on the principle of separation of concerns [38, 39]. We define high-performance computational kernels together with their interface and submit them to the QUARK scheduler [16, 17] for parallel execution as tasks with well-defined dependencies. These data dependencies between the tasks limit the amount of available parallelism and may decrease the computational load of each task, which results in an increase of the total number of tasks. The optimal schedule for the tasks is the one with the shortest height of the spanning tree of the DAG. But QUARK does not seek to attain the optimal schedule because it is computationally infeasible. Rather, it uses a localized heuristic that works very well in practice [2, 40–42]. The heuristic is based on a generative exploration of the DAG that limits the number of outstanding tasks by a tunable parameter called *task window size*.

To explore more advanced features of QUARK, we turn to Figure 11, which shows an annotated DAG of tasks that results from executing our LU factorization on a matrix divided into three-by-three tiles. One feature that we believe makes QUARK stand out is the availability of nested parallelism without any constraints on the number of threads executing within a parallel task. The tasks that use this feature (and thus are parallel tasks) are the nodes marked as xGETRF-REC(). Each of these tasks may use a variable number of threads to execute, and this is determined at runtime as the panel height decreases with the progress of the factorization.

Another feature of QUARK that we use in our code is the ability to assign priorities to tasks. For our particular situation, we only use two priorities: critical and non-critical. The former is reserved for the panel factorization and is marked with a triangle in Figure 11. The latter is used for the remaining tasks. This choice was made because the xGETRF-REC() tasks are on the critical path and cannot be overlapped with the other tasks in an optimally scheduled DAG. Even though in practice the schedule is not optimal because of a fixed number of cores and the scheduling heuristic, highly prioritized panel factorization is still beneficial.

A feature that is also useful for our code is marked with a triangular node that is labeled as GATHERV in Figure 11. This feature allows for submission of tasks that write to different portions of the same submatrix but are otherwise independent. The Schur complement update is performed with xGEMM's and can either be seen as four independent tasks that update disjoint portions of the trailing matrix, or as a single task that updates the trailing matrix as a whole. In the latter case, the parallelism that is so abundant in the update would have been lost. GATHERV allows the algorithm to recover this parallelism by submitting not one but multiple tasks that update the same portion of memory. The GATHERV annotations inform QUARK that these multiple tasks are independent of each other even though their data dependencies indicate otherwise.

Finally, QUARK can optionally generate DAGs such as the one featured in Figure 11. This is controlled with an environment variable and can be turned on as necessary as a debugging or profiling feature.

6. EXPERIMENTAL RESULTS

In this section, we show the results on the largest shared memory machines we could access at the time of this writing. They are representative of a vast class of servers and workstations commonly used for computationally intensive workloads. They clearly show the industry's transition from chips with a few cores to a few tens of cores, from compute nodes with an order of 10 cores to as large as 100-core designs, and from Front Side Bus memory interconnect (Intel's NetBurst and Core Architectures) to NUMA and ccNUMA hardwares (AMD's HyperTransport and Intel's QuickPath Interconnect).

6.1. Environment settings

All the experiments were run on three systems that we refer to as *MagnyCours-48*, *Xeon-16*, and *SandyBridge-32*. *MagnyCours-48*, is composed of four AMD Opteron Magny-Cours 6172 processors of 12 cores each, running at 2.1 GHz, with a total of 128 GiB of main memory. The theoretical peak for this machine in single-precision and double-precision arithmetic is 806.4 Gflop/s (or 16.8 Gflop/s per core) and 403.2 Gflop/s (or 8.4 Gflop/s per core), respectively.

Xeon-16 is composed of four quad-core Intel Xeon E7340 processors, running at 2.4 GHz, with a total of 32 GiB of memory. The theoretical peak of this machine in single-precision and double-precision arithmetic is 307.2 and 153.6 Gflop/s, respectively. *SandyBridge-32* is based on two Intel Xeon E5-2690 processors with eight cores using HyperThreading and frequency scaling technologies. For our experiments, we fixed the frequency to its maximum and used the Portable Hardware Locality (hwloc) library [43] to bind only one thread per physical core. The theoretical peak of this machine is 742.4 Gflop/s in single precision and 371.2.4 Gflop/s in double precision.

We compare the results against the latest parallel version of the Intel MKL 10.3.6 library released in August 2011. It features optimizations for the AVX instruction set and is essential on the *SandyBridge-32* architecture. We also compare with the reference code base of LAPACK 3.2 from Netlib and linked with the Intel MKL BLAS library with multithreading enabled. We link our code with the sequential version of the Intel MKL library and also use it for PLASMA.

6.2. *Performance results*

Figure 12 shows the performance comparisons of the parallel recursive tile LU algorithm against Intel's MKL library and the old LU routine from the PLASMA library on *MagnyCours-48* in single-precision arithmetic. Figure 13 shows the same comparison but for double-precision arithmetic. Data points on each curve are obtained from runs that used the maximum number of available cores of the machine (48 cores in total). In addition, tuning was performed for the algorithmic parameters in order to achieve the best asymptotic performance. Six implementations are shown:

- The *LAPACK* implementation from Netlib that was linked against the parallel BLAS from Intel MKL to study the fork-join model,
- The *Intel MKL* version of `DGETRF`,
- The old code from PLASMA based on incremental pivoting with column-major (or *LAPACK*) data layout,
- The old code from PLASMA based on incremental pivoting with tile data layout for PLASMA's tile algorithms,
- Our new algorithm *Recursive-LU* with column-major data layout, and
- Our new algorithm *Recursive-LU* with tile data layout.

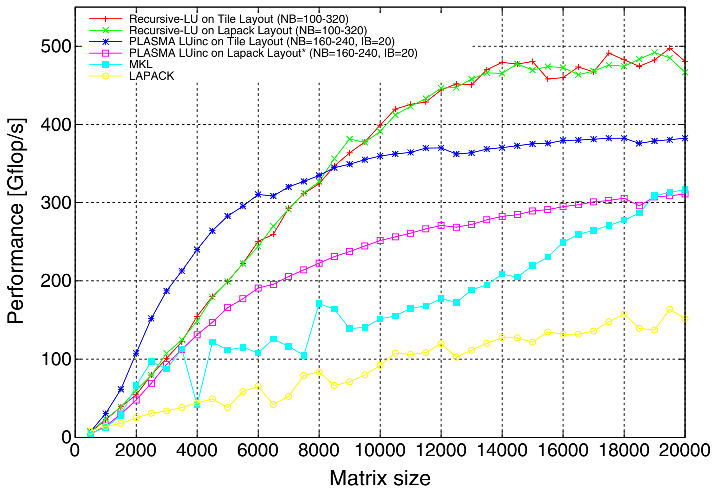


Figure 12. Performances of `SGETRF` on *MagnyCours-48*.

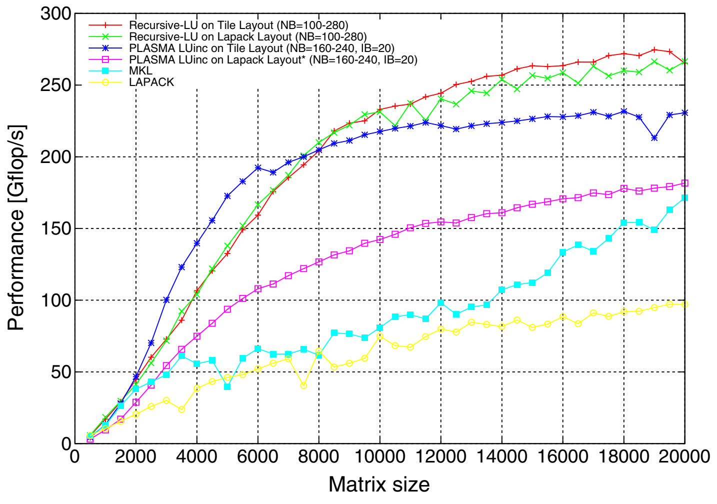


Figure 13. Performances of `DGETRF` on *MagnyCours-48*.

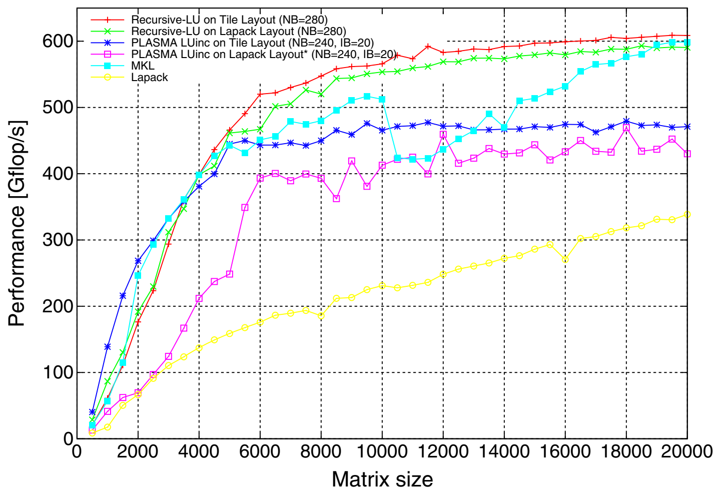


Figure 14. Performances of SGETRF on *SandyBridge-32*.

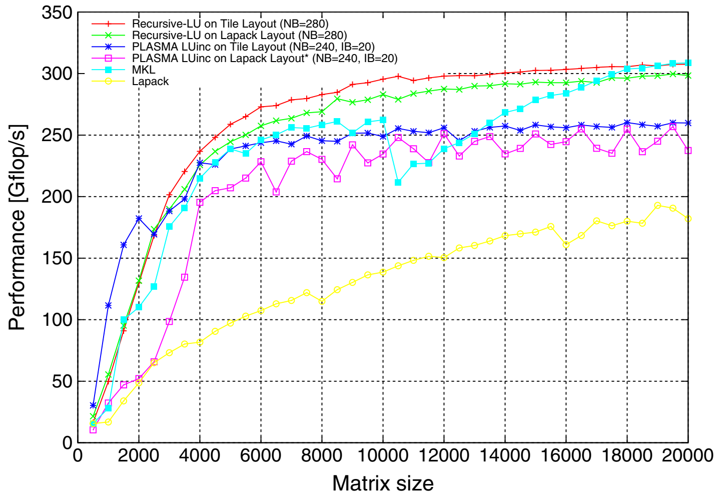


Figure 15. Performances of DGETRF on *SandyBridge-32*.

In a similar fashion, Figures 14 and 15 show the performance comparison of the parallel LU factorization algorithm on SandyBridge-32. Both PLASMA versions correspond to the tile LU algorithm with incremental pivoting and use QUARK as the dynamic scheduler. The first version handles the LAPACK interface (native interface), which requires an input matrix in column-major data layout, similar to Intel's MKL. Thus, it implies that PLASMA has to convert the matrix into tile data layout before the factorization can proceed, and converts it back to column-major data layout at the end of the computation to conform to the user's original layout. The second configuration is the tile interface (expert interface), which accepts matrices already in tile data layout, and therefore avoids both layout conversions. For our algorithm, the kernel used for the panel is chosen according to the data layout, so no conversions are required.

On the *SandyBridge-32* machine, we used a fixed tile size $NB = 240$ and inner-blocking $IB = 20$ for the PLASMA algorithms, as well as tile size $NB = 280$ for our parallel recursive tile LU. These parameters have been chosen to give the best performance on the full range of sizes we used in our experiments. On the *MagnyCours-48* system, the tile size had to be adapted for small problems to provide more parallelism and better performance. For our algorithm, NB selection varied between 100 and 280 in double and 320 in single precision. The PLASMA algorithm required the tile size

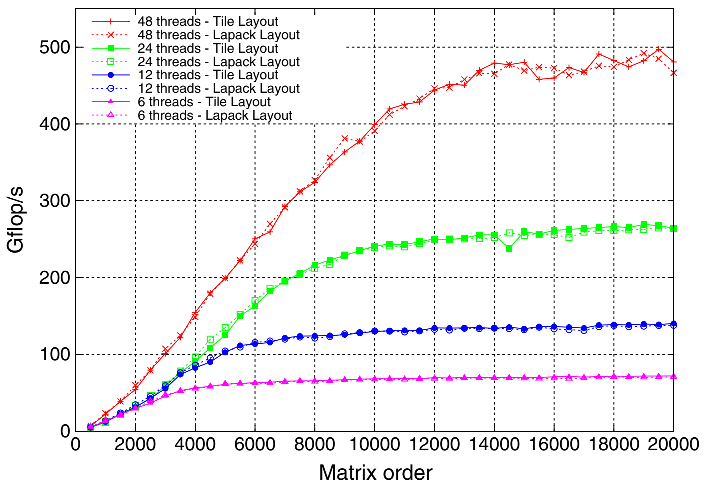


Figure 16. Scalability of PLASMA recursive LU on *MagnyCours-48* in single precision.

between 160 and 240 for both cases. For all the runs with incremental pivoting, we used the internal blocking of 20, which gave us the best results for every (NB, IB) pair we have tried.

The recursive parallel panel LU algorithm based on column-major and tile data layouts obtains similar performance, which at first glance may look surprising after the conclusions drawn previously in Section 4.3. It is important to understand that the step of the trailing submatrix update becomes the leading phase in such factorizations because it is rich in Level 3 BLAS operations. Thus, the overhead of memory accesses in the panel is completely hidden by the efficiency of the compute-intensive kernels on tile layout, which makes the recursive parallel panel LU algorithm based on tile data layout slightly faster than the column-major data layout version.

Moreover, Figures 12 and 13 show that, asymptotically, we take advantage of the monolithic xGEMM kernel by increasing the performance up to 20% compared to both PLASMA versions. Our implementation, however, has higher performance than Intel's MKL, asymptotically at least, on the *MagnyCours-48* system, while it provides more constant performance on *SandyBridge-32*, and saturates faster than Intel's MKL for smaller problems.

For matrices smaller than 8000 on the AMD systems and less than 3000 on the *SandyBridge-32* one, our algorithm suffers from the synchronization points for each panel and the lack of parallelism compared to the incremental pivoting implemented in PLASMA. But the loss of performance is balanced by the better accuracy provided by the partial pivoting algorithm.

We may conclude that the parallel recursive tile LU algorithm provides good performance on multicore architectures. It also retains the standard error control and numerical accuracy unlike the incremental pivoting strategy from PLASMA. This obviously comes at a price of a synchronization point added right after each panel computation. And this synchronization point has been considerably weakened by the efficient parallelization of the panel factorization. Our new implementation also benefits from the increase of the level of parallelism during the phase of the trailing matrix updates when compared to PLASMA's algorithms. Taking into account the fact that PLASMA's algorithm loses a few digits of precision in the solution, especially when the number of tiles increases [11], our new recursive tile LU factorization becomes a worthwhile alternative and will eventually replace the current LU algorithm in PLASMA.

6.3. *Scalability*

Figures 16 and 17 show the scalability of the parallel recursive tile LU algorithm on *MagnyCours-48* in single-precision and double-precision arithmetic, respectively. The scaling experiments have been

[†]PLASMA luinc on column-major data layout includes the translation of the matrix to tile layout for the factorization step, and the translation back to LAPACK layout to return the result to the user.

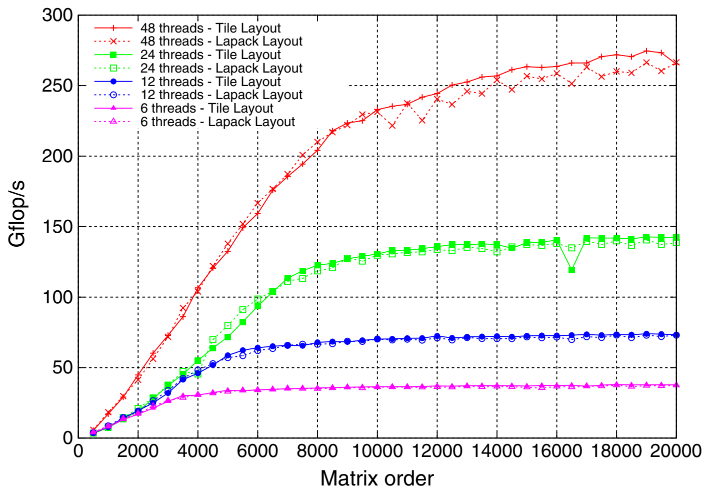


Figure 17. Scalability of PLASMA recursive LU on *MagnyCours-48* in double precision.

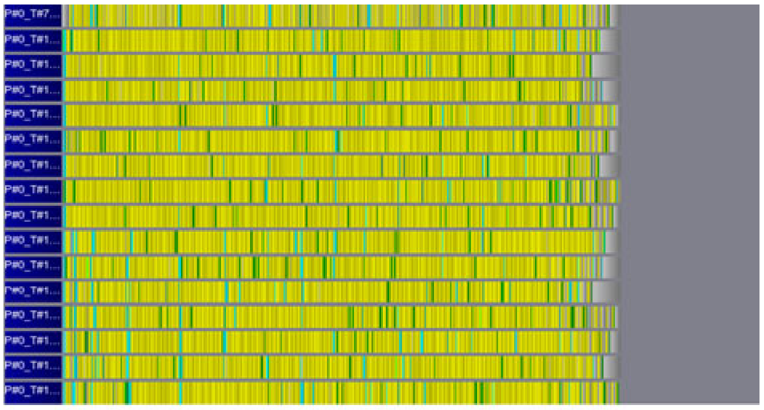
performed according to the number of cores per socket (i.e., 12 cores), using 6, 12, 24, and 48 threads. For each curve, we used the panel width $NB = 280$, which gives the best performance for 48 threads. Because the data are usually allocated by the user and we do not move them around, we used the Linux command `numactl -interleave=0-X`, where `X` is one less than the number of threads. This command allows us to control the NUMA policy by allocating the memory close to the cores, where the threads are bound. We observe that our algorithm scales almost linearly for up to 48 threads

6.4. *Execution traces*

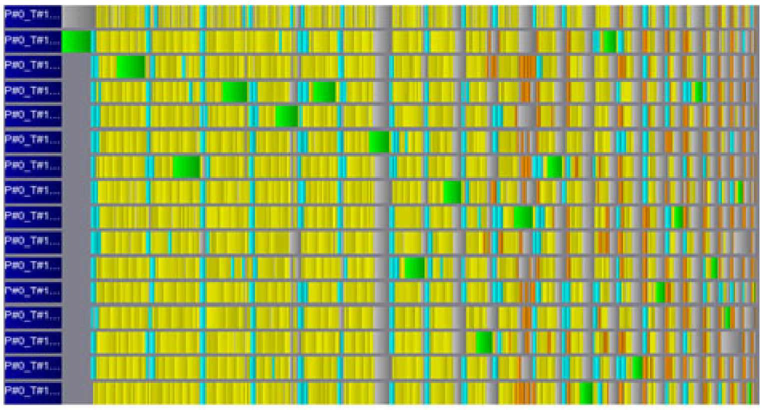
This section shows the execution traces of three different versions of the LU algorithm on *MagnyCours-48* with 16 threads on matrices of size 5000×5000. These traces have been generated, thanks to the EZTrace library [44] and ViTE software [45]. On each trace, the green color is dedicated to the factorization of the panel (light for `dgetrf` and dark for `dtstrf`), the blue color illustrates the row update (`dtrsm+dlaswp` or `dgessm`), the yellow color represents the update kernel (`dgemm` or `dssssm`), the orange color shows the backward swaps, and finally, the gray color indicates the idle time.

The first trace of Figure 18(a) is the result of the execution of the PLASMA algorithm. It shows that the tile algorithm results in increasing the degree of parallelism triggered by the first task. The second trace of Figure 18(b) shows the recursive tile LU, where the panel is performed by a call to the sequential `dgetrf` routine from Intel MKL. This algorithm releases as much parallelism as the previous one after the first task, but we may clearly observe on the execution trace that the time spent to factorize the first panel is longer than the time needed to factorize the block in the tile algorithm. Another concern in this version is the fact that the time spent on the critical path is significant, which leads to substantial idle time intervals, especially after the first half of the factorization.

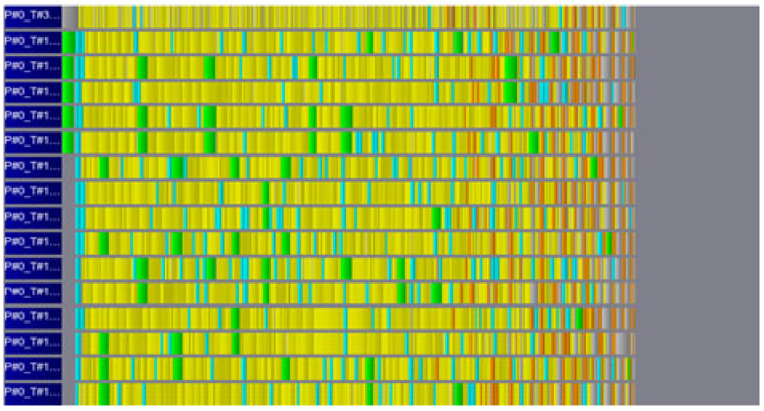
Finally, Figure 18(c) shows the execution trace of the same algorithm but with a parallel panel computation using recursive LU. This results in a reduced time for the factorization step, which drastically reduces the overall idle time. It is also worth mentioning how the lookahead transparently comes into effect at runtime, thanks to the dynamic scheduling from QUARK. This can be seen by the irregularity in the appearance of the `dtrsm` blocks in the trace. More specifically, in the second diagram, one can observe that each portion filled by yellow blocks (`dgemm`) matches one step of the update and overlaps the next panel factorization in green. This overlap is performed, thanks to this natural lookahead provided by the dynamic scheduling. However, in this case, one can notice that for a 5000×5000 matrix, the update stage quickly becomes not that important to overlap with the sequential panel factorization that is appreciably slower. The non-critical tasks,



(a)



(b)



(c)

Figure 18. Execution traces of the different variant of LU factorization using QUARK. Light green: `dgetrf`, dark green: `dtstrf`, light blue: `dtrsm` or `dgessm`, yellow: `dgemm` or `dssssm`, and orange: `dlaswp`. (a) Incremental pivoting with $N = 5000$, $NB = 220$, and $IB = 20$; (b) recursive LU with sequential panel factorization, $N = 5000$, and $NB = 220$; and (c) recursive LU with parallel panel factorization, $N = 5000$, and $NB = 220$.

which perform pivot interchanges behind the panel (xLASWP), are also postponed until the end of the factorization in order to stress the pursuit of the critical path.

7. SUMMARY AND FUTURE WORK

This article described a new algorithm for parallel recursive LU factorization with partial pivoting on multicore architectures. Our implementation is characterized by a parallel recursive panel factorization, while the computation on the trailing submatrix is carried out by following a tile algorithm. Not only does this implementation achieve higher performance than the corresponding routines from LAPACK (up to threefold speedup) and MKL (up to 40% speedup), but it also maintains the numerical quality of the standard LU factorization algorithm with partial pivoting, which is unlike the accuracy of the incremental pivoting method. Our approach uses a parallel fine-grained *recursive* formulation of the panel factorization step. Our implementation is based on conflict-free partitioning of the data and lockless synchronization mechanisms. It allows the overall computation for the panel to flow naturally without contention. QUARK, a dynamic runtime system, is used to schedule tasks with heterogeneous granularities and to transparently introduce algorithmic lookahead.

The natural extension for this work is the application of our methodology and implementation techniques to the tile QR factorization. The tile QR factorization does not suffer from loss of numerical accuracy, when compared to the standard QR factorization, thanks to the use of orthogonal transformations. However, a performance hit may be noticed for asymptotic sizes just as it is the case for the tile LU as implemented in PLASMA. And this is mainly due to the most compute intensive kernel, which is composed of successive calls to Level 3 BLAS kernels. If the QR panel would have been parallelized (similar to the LU panel), the update would be much simpler (especially when targeting distributed systems) and based only on calls to xGEMM. The overall performance will then be determined solely by the performance of the matrix–matrix multiplication kernel, which is crucial when targeting asymptotic performance.

ACKNOWLEDGEMENTS

Research reported here was partially supported by the National Science Foundation, Department of Energy, and Microsoft Research.

REFERENCES

1. Anderson E, Bai Z, Bischof C, Blackford SL, Demmel JW, Dongarra JJ, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen DC. *LAPACK User's Guide*, 3rd ed. Society for Industrial and Applied Mathematics: Philadelphia, 1999.
2. Agullo E, Hadri B, Ltaief H, Dongarrra J. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM: New York, NY, USA, 2009; 1–12, DOI: http://doi.acm.org/10.1145/1654059.1654080.
3. Haidar A, Ltaief H, Dongarra J. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM: New York, NY, USA, 2011; 8:1–8:11, DOI: http://doi.acm.org/10.1145/2063384.2063394.
4. Haidar A, Ltaief H, Luszczek P, Dongarra J. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IEEE Computer Society: Shanghai, China, 2012 May 21-25; 25–35.
5. Ltaief H, Luszczek P, Dongarra J. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In *9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011*, Vol. 7203, Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J (eds), Parallel Processing and Applied Mathematics: Torun, Poland, 2012; 661–670.
6. Ltaief H, Luszczek P, Dongarra J. High performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *ACM TOMS* 2013; **39**(3):16:1–16:22. In publication.
7. Sorensen DC. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Tranasactions on Computers* 1985; **C-34**(3):274–278.
8. Yip EL. FORTRAN subroutines for out-of-core solutions of large complex linear systems. *Techical Report CR-159142*, NASA, 1979.

9. Buttari A, Langou J, Kurzak J, Dongarra JJ. A class of parallel tiled Linear algebra algorithms for multicore architectures. *Parellel Computer and System Applications* 2009; **35**:38–53.
10. Quintana-Ortí G, Quintana-Ortí ES, Van De Geijn RA, Van Zee FG, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transaction on Mathematical Software* 2009; **36**: 14:1–14:26.
11. Agullo E, Augonnet C, Dongarra J, Faverge M, Langou J, Ltaief H, Tomov S. LU factorization for accelerator-based systems. *ICL Technical Report ICL-UT-10-05, Submitted to AICCSA 2011*, December 2010.
12. Intel, Math Kernel Library (MKL). (Available from: http://www.intel.com/software/products/mkl/) [Accessed 31 August 2013].
13. University of Tennessee. PLASMA users' guide, parallel linear algebra software for multicore architectures, Version 2.3, November 2010.
14. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 2009; **180**:1–5.
15. The FLAME project, April 2010. (Available from: http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage) [Accessed 31 August 2013].
16. Kurzak J, Ltaief H, Dongarra J, Badia R. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience* 2010; **22**(1):15–44.
17. Ltaief H, Kurzak J, Dongarra J, Badia R. Scheduling two-sided transformations using tile algorithms on multicore architectures. *Journal of Scientific Computing* 2010; **18**:33–50.
18. Elmroth E, Gustavson FG. New serial and parallel recursive QR factorization algorithms for SMP systems. *Proceedings of PARA 1998*, Umea, Sweden, 1998; 120–128.
19. Georgiev K, Wasniewski J. Recursive version of LU Decomposition. *Revised Papers from the Second International Conference on Numerical Analysis and Its Applications*, London, UK, 2001; 325–332.
20. Dongarra J, Eijkhout V, Luszczek P. Recursive approach in sparse matrix LU factorization. *Science Programming* 2001; **9**:51–60.
21. Irony D, Toledo S. Communication-efficient parallel dense LU using a 3-dimensional approach. *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, USA, 2001; 1–10.
22. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: past, present, and future. *Concurrency and Computation: Practice and Experience* 2003; **15**:1–18.
23. Castaldo AM, Whaley RC. Scaling LAPACK panel operations using parallel cache assignment. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, 2010; 223–232.
24. Valiant LG. A bridging model for parallel computation. *Communications of ACM* 1990; **33**(8):103–111.
25. Gustavson FG. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* 1997; **41**(6):737–755.
26. Chan E, van de Geijn R, Chapman A. Managing the complexity of lookahead for LU factorization with pivoting. *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, 2010; 200–208.
27. 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, 2012; 1–6. IEEE Catalog Number: CFP12HPE-CDR, ISBN: 978-1-4673-1574-6.
28. Anderson E, Dongarra J. Implementation guide for LAPACK. *Technical Report UT-CS-90-101*, University of Tennessee, Computer Science Department, April 1990. LAPACK Working Note 18.
29. Amdahl GM. Validity of the single-processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, vol, 30, AFIPS Press, Reston, VA: Atlantic City, N.J., 1967; 483–485.
30. Gustafson JL. Reevaluating Amdahl's law. *Communications of ACM* 1988; **31**(5):532–533.
31. Sundell H. Efficient and practical non-blocking data structures. *Ph.D. Dissertation*, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 5 2004.
32. Anderson E, Dongarra JJ. Evaluating block algorithm variants in LAPACK. In *Parallel Processing for Scientific Computing*, Dongarra J, Messina P, Sorensen D, Voight R (eds). SIAM Publications: Philadelphia, PA, 1990; 3–8. (Available from:http://www.netlib.org/lapack/lawnspdf/lawn19.pdf).
33. Dackland K, Elmroth E, Køagström B, Van Loan CV. Design and evaluation of parallel block algorithms: LU factorization on an IBM 3090 VF/600J. *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Publications, Philadelphia, PA, 1992; 3–10.
34. Yi Q, Kennedy K, You H, Seymour K, Dongarra J. Automatic blocking of QR and LU factorizations for locality. *Proceedings of 2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, Washington, DC, 2004; 2–22.
35. Blackford LS, Choi J, Cleary A, D'Azevedo EF, Demmel JW, Dhillon IS, Dongarra JJ, Hammarling S, Henry G, Petitet A, Stanley K, Walker DW, Whaley RC. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics: Philadelphia, 1997.
36. YarKhan A, Kurzak J, Dongarra J. QUARK users' guide: queueing and runtime for kernels. *Technical Report ICL-UT-11-02*, University of Tennessee, Innovative Computing Laboratory, 2011.

37. Haidar A, Ltaief H, YarKhan A, Dongarra JJ. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *ICL Technical Report UT-CS-11-666, LAPACK working note #243, Submitted to Concurrency and Computations*, 2010.
38. Dijkstra EW. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, Dijkstra EW (ed.). Springer-Verlag New York, Inc.: New York, NY, USA, 1982; 60–66. ISBN 0-387-90652-5.
39. Reade C. *Elements of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1989. ISBN 0201129159.
40. Buttari A, Langou J, Kurzak J, Dongarra JJ. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience* 2008; **20**(13):1573–1590. DOI: 10.1002/cpe.1301. (Available from:http://dx.doi.org/10.1002/cpe.1301).
41. Perez J, Badia R, Labarta J. A dependency-aware task-based programming environment for multi-core architectures. *2008 IEEE International Conference on Cluster Computing*, New York, NY, 2008; 142 –151, DOI: 10.1109/CLUSTR.2008.4663765.
42. Luszczek P, Ltaief H, Dongarra J. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. *Proceedings of IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, USA, 2011; 944–955.
43. 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. *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, IEEE Computer Society Press: Pisa, Italia, 2010; 180–186, DOI: 10.1109/PDP.2010.67. (Available from:http://hal.inria.fr/inria-00429889).
44. Dongarra J, Faverge M, Ishikawa Y, Namyst R, Rue F, Trahay F. EZTrace: a generic framework for performance analysis. *ICL Technical Report, Submitted to CCGrid 2011*, Innovative Computing Laboratory, University of Tennessee, December 2010.
45. Visual Trace Explorer. (Available from:http://vite.gforge.inria.fr/) [Accessed 31 August 2013].