Achieving Numerical Accuracy and High Performance using Recursive Tile LU Factorization

Jack J. Dongarra, Mathieu Faverge, Hatem Ltaief, Piotr Luszczek

Achieving Numerical Accuracy and High Performance using Recursive Tile LU Factorization

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
[2] KAUST Supercomputing Laboratory
Thuwal, Saudi Arabia
{dongarra,faverge,luszczek}@eecs.utk.edu
Hatem.Ltaief@kaust.edu.sa

Abstract. The LU factorization is an important numerical algorithm for solving systems of linear equations in science and engineering, and is characteristic of many dense linear algebra computations. It has even become the de facto numerical algorithm implemented within the LINPACK benchmark to rank the most powerful supercomputers in the world, collected bt the TOP500 website. In this context, the challenge in developing new algorithms for the scientific community resides in the combination of two goals: achieving high performance and maintaining the accuracy of the numerical algorithm. This paper proposes a novel 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. In particular, it is up to 40% faster when compared to the equivalent Intel MKL routine and up to 3-fold faster than LAPACK with multithreaded Intel MKL BLAS.

Keywords: 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 massive parallelism as well as the memory hierarchy design brought by this architecture. Indeed, LAPACK [1] has shown some significant 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.

The PLASMA library [3, 4] initiated, with other software packages like FLAME [5], this effort of redesigning standard numerical algorithms to match the hardware requirements of multicore architectures. 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 [6]. Based on tile data layout, which consists of splitting the matrix into small square regions of data contiguous in memory, PLASMA has alleviated the bottlenecks (1) and (2) from LAPACK by rather bringing the parallelism to the fore, minimizing the synchronization overhead, and relying on dynamic scheduling of fine-grained tasks. However, the panel factorization phase has not really been improved for the one-sided factorizations. It is still rich in 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 issues, particularly for the LU factorization. Indeed, the numerical accuracy of the solution has been deteriorated due to the necessary replacement of the standard partial pivoting scheme in the panel factorization by an incremental pivoting strategy [7]. The number of pivoted elements dramatically increases, which may eventually trigger a considerable growth pivot factor and can potentially make the whole numerical scheme unstable [8–10].

This paper presents a novel approach for computing the LU factorization on multicore architectures, which not only improves the overall performance compared to LAPACK and PLASMA, but also sustains 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. 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 the different memory hierarchy levels and to cast memory-bound kernels into Level 3 BLAS operations to increase the computational rate even further. Based on conflict-free partitioning of the data and lockless synchronization mechanisms, our implementation lets the parallel computation flow naturally without contention and reduces synchronization. The dynamic runtime system called QUARK [11, 12] (also used in PLASMA) 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 depicted by a directed acyclic graph (DAG), where nodes represent computational tasks and edges define the dependencies between them. The DAG is actually never built entirely since it would obviously not fit in the main memory for large matrix sizes. As the computation progresses, the DAG is unrolled just enough 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, QUARK can transparently integrate algorithmic lookahead in order to overlap successive computational steps, and to keep all processing units 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 MKL version 10.3 [13]. Finally, Section 7 summarizes the results of this paper and describes the ongoing work.

2 Related Work

This Section presents previous similar works in implementing a recursive and/or parallel panel of the LU factorization.

Recursive formulation of a one-sided factorization (QR) and its parallel implementation has been done on a shared memory machine [14]. Three differences stand out. First, the authors used a sequential panel factorization. This led to the second difference: lack of nested parallelism that we employ. And thirdly, master-worker parallelization was used instead of dynamic DAG scheduling.

Georgiev and Wasniewski [15] presented a recursive version of the LU decomposition. They implemented recursive versions of the main LAPACK and BLAS kernels involved in the factorization i.e., xGETRF and xGEMM, xTRSM, respectively. Their original code is in Fortran 90 and they relied on the compiler technology to achieve the desired recursion.

Recursion was also successfully used in the context of sparse matrix LU factorization [16]. 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 [17]. 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 [18] 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 [19] using Level 1 BLAS; this is a flat parallelism model with fork-join execution (closely related to Bulk Synchronous Processing). The authors refer to their approach as Parallel Cache Assignment (PCA). Our work differs in a few key aspects. We employ recursive formulation [20] and therefore are able to use Level 3 BLAS as opposed to just Level 1 BLAS. 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.

Last but not least, Chan et. al [21] implemented the classical LU factorization with partial pivoting (within the FLAME framework), in which the authors basically 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 we had rather implemented an optimized parallel memory-aware kernel, which performs an *in-place* LU panel factorization with partial pivoting. Both of these lead to high performance.

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

Block algorithms in LAPACK [1] surfaced with the emergence of cache-based architectures. 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. Last but not least, 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 implemented in PLASMA [3] 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 which are contiguous in memory. This is done 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. 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 directed acyclic graph (DAG), where 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 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 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 deteriorate the overall stability of the LU factorization [8–10].

As a matter of fact, 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

This Section describes one of our most unique contributions, which is the parallelization of the LU factorization of a matrix panel using the recursive algorithm [20].

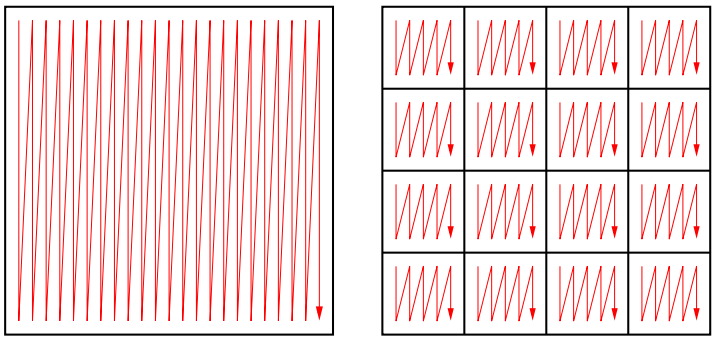


Fig. 1. Translation from LAPACK layout (column-major) to tile data layout

```
function xGETRFR(M, N, column) {
   if N == 1 {                                                    single column, recursion stops
      idx = split_IxAMAX(...)                                     compute local maximum of modulus
      gidx = combine_IxAMAX(idx)                                  combine local results
      split_xSCAL(...)                                            scale local data
   } else {
      xGETRFR(M, N/2, column)                                     recursive call to factor left half
      xLASWP(...)                                                 pivoting forward
      split_xTRSM(...)                                            triangular solve
      split_xGEMM(...)                                            Schur's complement
      xGETRFR(M, N-N/2, column+N/2)                               recursive call to factor right half
      xLASWP(...)                                                 pivoting backward
   }
}
```

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

4.1 Recursive Algorithm and Its Advantages

Figure 2 shows a pseudo-code of our recursive implementation. Even though the panel factorization is a lower order term – $O(N^2)$ – from the computational complexity perspective [22], it still poses a problem in the parallel setting from the theoretical [23] and practical standpoints [19]. To be more precise, the combined panel factorizations' complexity for the entire matrix is

$$O(N^2NB),$$

where N is panel height (and matrix dimension) and NB is panel width. For good performance of BLAS calls, panel width is commonly increased. This creates tension if the panel is a sequential operation because a larger panel width results in larger Amdahl's fraction [24]. Our own experiments revealed this to be a major obstacle to proper scalability of our implementation of tile LU factorization with partial pivoting – a result consistent with related efforts [19].

Aside from gaining high level formulation free of low level tuning parameters, recursive formulation permits to dispense of a higher level tuning parameter commonly called algorithmic blocking. There is already panel width – a tunable value used for merging multiple panel columns together. Non-recursive panel factorizations could potentially establish another level of tuning called *inner-blocking* [2, 4]. This is avoided in our implementation.

4.2 Data Partitioning

The challenging part of the parallelization is the fact that the recursive formulation suffers from inherent sequential control flow that is characteristic of the column-oriented implementation employed by LAPACK and ScaLAPACK. As a first step then, we apply a 1D partitioning technique that has proven successful before [19]. We employed this technique for the recursion-stopping case: single column factorization. The recursive formulation of the LU algorithm poses another problem, namely the use of Level 3 BLAS call for triangular solve – xTRSM() and LAPACK's auxiliary routine for

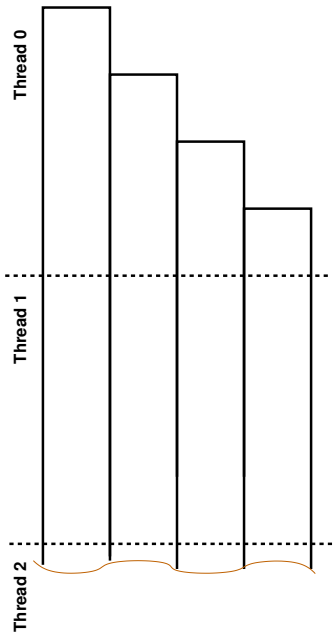


Fig. 3. Fixed partitioning scheme used in the parallel recursive panel factorization.

swapping named xLASWP(). Both of these calls do not readily lend themselves to the 1D partitioning scheme due to two main reasons:

1. each call to these functions occurs with a variable matrix size and
2. 1D partitioning makes the calls dependent upon each other thus creating synchronization overhead.

The latter problem is fairly easy to see as the pivoting requires data accesses across the entire column and memory locations may be considered random. Each pivot element swap would then require coordination between the threads that the column is partitioned amongst. The former issue is more subtle in that the overlapping regions of the matrix create a memory hazard that may be at times masked by the synchronization effects occurring in other portions of the factorization. To deal with both issues at once, we chose to use 1D partitioning across the rows and not across the columns as before. This removes the need for extra synchronization and affords us 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 Level 3 BLAS kernel xGEMM() and this is also a new function that is not present within the panel factorizations from LAPACK and ScaLAPACK. 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 resulting synchronization points.

To summarize the observations that we made throughout the preceding text, we consider data partitioning among the threads to be of paramount importance. Unlike the PCA method [19], we do not perform extra data copy to eliminate memory effects that are detrimental to performance such as TLB misses, false sharing, etc. By choosing the recursive formulation, we rely instead on 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 [20]. What is left to do for our code is the introduction of parallelism that is commonly missing from Level 3 BLAS when narrow rectangular matrices are involved.

Instead of low level memory optimizations, we turned our focus towards avoiding synchronization points and let 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. Regardless of the current column height (within the panel being factored), we always assign the same amount of rows to each thread except for the first thread. Figure 3 shows that this causes a load imbalance as the thread number 0 has progressively smaller amounts of work to perform as the panel factorization progresses from the first to the last column. This is counter-balanced by the fact that the panels are relatively tall compared to the number of threads and the first thread usually has greater responsibility in handling pivot bookkeeping and synchronization tasks.

The tile data layout specific to PLASMA algorithms does not allow such partitioning. In this case, the partition is then following the tile structure and each thread handles a fixed number of tiles. The second problem due to this data

```
function xGETRFR(M, N, column) {
   xGETRFR(M, N/2, column)          recursive call to factor left half
   xLASWP(...)                      pivoting forward
   xTRSM(...)                       triangular solve on the first tile
   foreach local tile {
     xSCAL( N/2-1 )                 scale the last column of the left part
     xGEMM(...)                     Schur's complement
     idx = IxAMAX( N/2 )            update local maximum of the first column in the right part
   }
   gidx = combine_IxAMAX(idx)       combine local results
   xGETRFR(M, N-N/2, column+N/2)    recursive call to factor right half
   xLASWP(...)                      pivoting backward
}
```

Fig. 4. Pseudo-code for the recursive panel factorization on tile layout.

storage is the number of cache misses generated by the algorithm described previously. If each stage is performed one after another (scale the column, compute the Schur complement and search the pivot), they will each require a loop over the tiles owned by the current thread, making it three loops over all the tiles of the panel. All the benefit from this data storage is then lost in memory load. The solution is to reorder the three stages to apply them in one shot to each tile as described by the figure 4).

4.3 Scalability Results of the Parallel Recursive Panel Kernel

Figures 5 and 6 show a scalability study using column-major layout (used in LAPACK) and tile layout (used in PLASMA), respectively, on the four socket, twelve core NUMA *Opteron-48* machine (see Section 6.1 for detailed hardware specifications) of our parallel recursive panel LU factorization with four different panel widths: 32, 64, 128, and 256 against equivalent routines from LAPACK. The idea is to highlight and to understand the impact of the data layout on our recursive parallel panel algorithm. We limit our parallelism level to 12 cores (one socket) because our main factorization needs the remaining cores for updating the trailing submatrix. First of all, the parallel panel implementation based on column-major layout achieves the best performance compared to the tile layout. Indeed, as shown in Figure 3 with column-major layout, thread 0 loads into the local cache memory not only its data partition but also the remaining data from the other thread partitions, which obviously contributes in preventing invalid cache lines. In contrast, the recursive parallel panel LU algorithm with tile layout may engender high bus traffic since each thread needs to acquire its own data partition independently from each other (latency overhead). Secondly, when compared with the panel factorization routine xGETF2() (mostly Level 2 BLAS), we achieve super-linear speedup for a wide range of panel heights with the maximum achieved efficiency exceeding 550%. In an arguably more relevant comparison against the xGETRF() routine, which could be implemented with mostly Level 3 BLAS, we achieve perfect scaling for 2 and 4 threads and easily exceed 50% efficiency for 8 and 16 threads. This is consistent with the results presented in the related work section [19].

4.4 Further Implementation Details and Optimization Techniques

We exclusively use lockless data structures [25] throughout our code. This choice was dictated by 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 using mutex locks was deemed inappropriate at such frequency as it has a potential of incurring system call overhead.

Together with lockless synchronization, we use *busy waiting* on shared-memory locations to exchange information between threads using a coherency protocol of the memory subsystem. While fast in practice [19], this causes extraneous traffic on the shared-memory interconnect, which we aim to avoid. We do so by changing busy waiting for computations on independent data items. Invariably, this leads to reaching the parallel granularity levels that are most likely hampered by spurious memory coherency traffic due to false sharing. Regardless of the drawback, we feel this is a satisfactory solution as we are motivated by avoiding busy waiting, which creates even greater demand for inter-core bandwidth because it has no useful work to interleave with the shared-memory polling. We refer this optimization technique as *delayed waiting*.

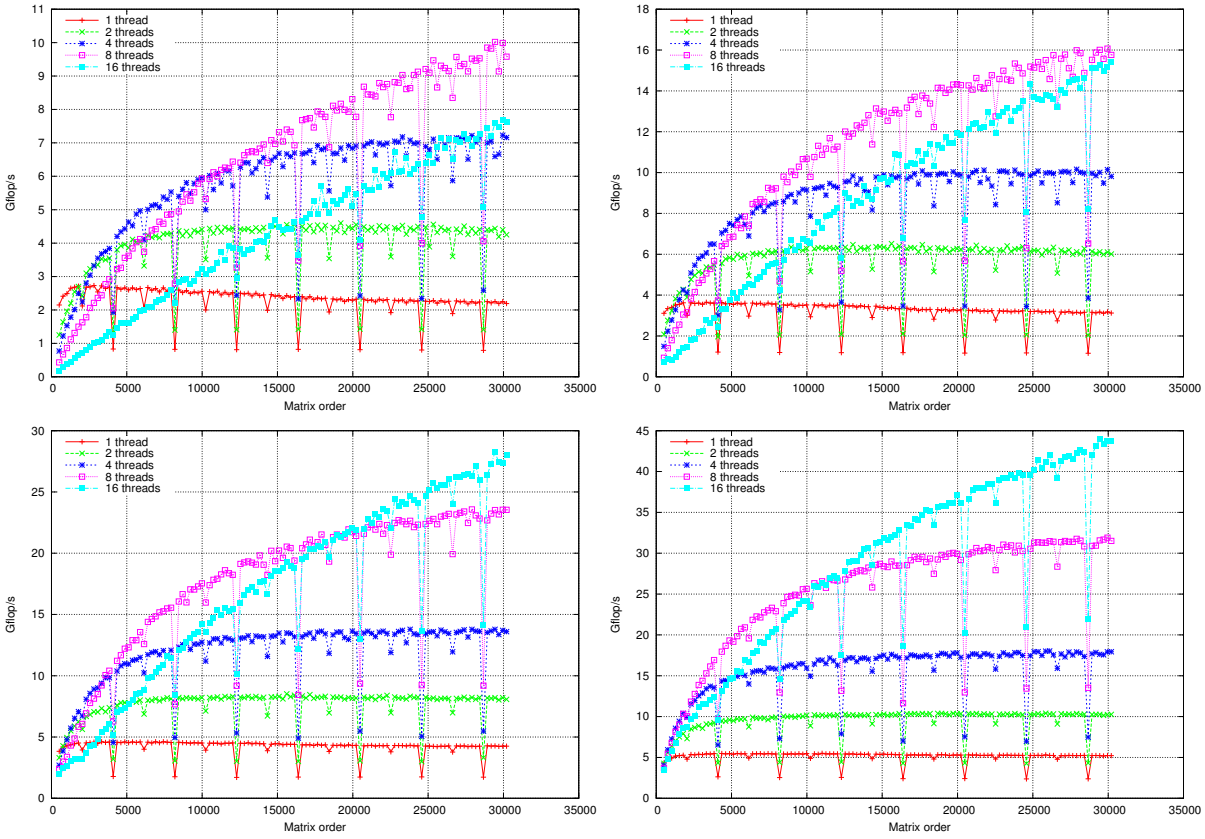


Fig. 5. Scalability study of the recursive parallel panel factorization in double precision on LAPACK layout with various panel widths: 32 (top-left), 64 (top-right), 128 (bottom-left), and 256 (bottom-right).

Another technique we use to optimize the inter-core communication is what we call *synchronization coalescing*. The essence of this method is to conceptually group unrelated pieces of code that require a synchronization into a single aggregate that synchronizes once. The prime candidate for this optimization is the search and the write of the pivot index. Both of these operations require a synchronization point. The former needs a parallel reduction operation while the latter requires global barrier. Neither of these are ever considered to be related to each other in the context of sequential parallelization. But with our synchronization coalescing technique, they are deemed related in the communication realm and, consequently, we implemented them 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 (and potentially a global thread barrier) to ensure global data consistency. Multiple writes are usually considered a hazard and are not guaranteed to occur in a specific order in most of the consistency models for shared memory systems. We completely side step this issue, however, as we guarantee algorithmically that each thread writes exactly the same value to memory. Clearly, this seems as 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, and more precisely, performance portability, was also an important goal in our overall design. In our lock-free synchronization, we heavily rely on shared-memory consistency – a 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 which, to our best knowledge, are satisfied 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 visibility of loads and stores to nearby memory locations. What we need from the compiler tool-chain is an appropriate handling of C's `volatile` keyword. This, combined with the use of primitive data types that are guaranteed to be contained within a single cache line, is sufficient in preventing unintended shared-memory side effects.

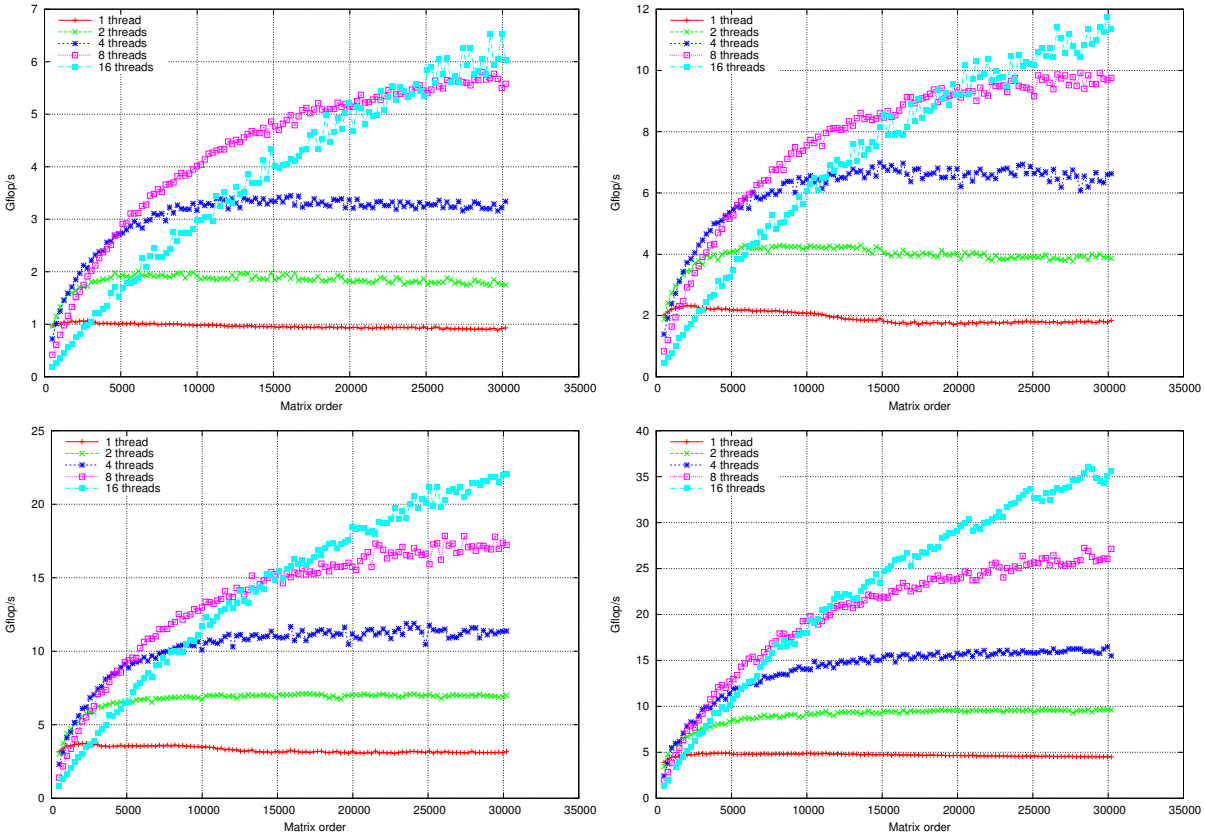


Fig. 6. Scalability study of the recursive parallel panel factorization in double precision on tile layout with various panel widths: 32 (top-left), 64 (top-right), 128 (bottom-left), and 256 (bottom-right).

5 Dynamic Scheduling and Lookahead

This Section provides further details about the dynamic scheduling of the recursive tile LU factorization along with the runtime environment system employed to schedule the various heterogeneous computational tasks.

5.1 Discussion of Implementation Variants

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

Additionally, our code has been originally formulated while having in mind the right-looking variant of LU factorization [26] as it makes it easier to take advantage to the available parallelism. This variant was also chosen for LAPACK [1] and ScaLAPACK [27]. However, the execution flow of our code is driven by the data dependencies that are communicated to the QUARK runtime system. This may result in an asynchronous out-of-order scheduling. The dynamic runtime environment ensures that enough parallelism is available through the entire execution (right looking), while advancing the critical path for lookahead purposes (left looking). Therefore, the strict right-looking variant available in LAPACK [1] and ScaLAPACK [27] cannot be guaranteed anymore. The asynchronous nature of the DAG execution provides sufficient lookahead opportunities for many algorithmic variants to coexist with each other regardless of the visitation order of the DAG [28].

5.2 Snapshot of the Parallel Recursive Tile LU Factorization

Figure 7 shows the initial factorization steps of a matrix subdivided into 9 tiles (a 3-by-3 grid of tiles). The first step is a recursive parallel factorization of the first panel consisting of three leftmost tiles. Only when this finishes, the other tasks may start executing, which creates an implicit synchronization point. To avoid the negative impact on parallelism,

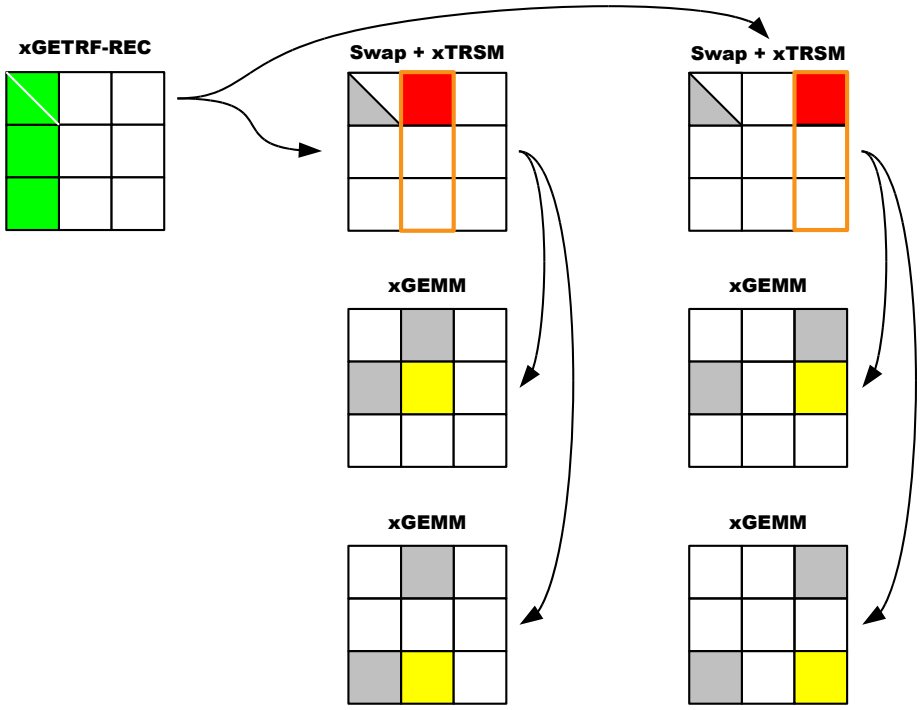


Fig. 7. Execution breakdown for recursive tile LU factorization: factorization of the first panel using the parallel kernel is followed by the corresponding updates to the trailing submatrix.

we execute this step on multiple cores (see Section 4 for further details) to minimize the running time. However, we use nested parallelism model as most of the tasks are handled by a single core and only the panel tasks are assigned to more than one core. Unlike similar implementations [19], we do not use all cores to handle the panel. There are two main reasons for this decision. First, we use dynamic scheduling that enables us to hide the negative influence of the panel factorization behind more efficient work performed by concurrent tasks. And second, we have clearly observed the effect of diminishing returns when using too many cores for the panel. Consequently, we do not use them all and instead we keep the remaining cores busy with other critical tasks.

The next step is pivoting to the right of the panel that has just been factorized. We combine in this step the triangular update (xTRSM in the BLAS parlance) because there is no advantage of scheduling them separately due to cache locality considerations. Just as the panel factorization locks the panel and has a potential to temporarily stall the computation, the pivot interchange has a similar effect. This is indicated by a rectangular outline encompassing the tile updated by xTRSM of the tiles below it. Even though so many tiles are locked by the triangular update, there is still a potential for parallelism because pivot swaps and the triangular update itself for a single column is independent of other columns. We can then easily split the operations along the tile boundaries and schedule them as independent tasks. This observation is depicted in Figure 7 by showing two xTRSM updates 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 7 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 that performs the whole update of the trailing submatrix, we use multiple invocations of the routine because we use a tile-based algorithm. In addition to exposing more parallelism and the ability to alleviate the influence the algorithm's synchronization points (such as the panel factorization), by splitting the Schur update operation we are able to obtain better performance than a single call to a parallelized vendor library [2].

One thing not shown in Figure 7 is pivoting to-the-left because it does not occur in the beginning of the factorization. It is necessary for the second and subsequent panels. The swaps originating from different panels have to be ordered correctly but are independent for each column, which is the basis 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. On the other hand, the memory accesses are done with 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 NUMA hardware. It is

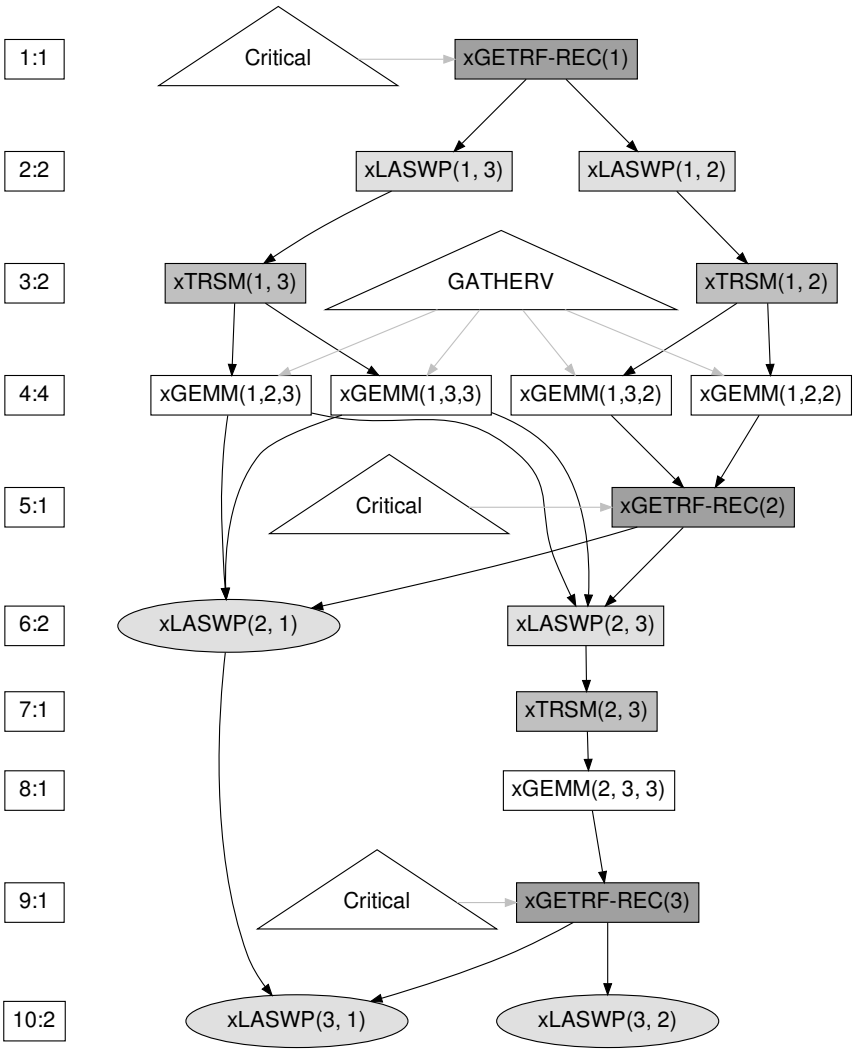


Fig. 8. Annoted DAG for parallel recursive tile LU factorization of a 3-by-3 tile matrix. The annotations are indicated with triangles.

also noteworthy to mention 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 toward the end of the factorization (see the tracing figures in Section 6.4). This is also highlighted in Figure 8, which draws the DAG of the parallel recursive tile LU factorizations of a 3-by-3 tile matrix. The nodes marked as xLASWP are *end nodes* and do not directly participate 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 [29, 30]. We define high performance computational kernels and the effect on their parameters and submit them to the QUARK [11,12] scheduler for parallel execution as dependent tasks. Due to the data dependences between the tasks, the amount of available parallelism is limited and may be increased by decreasing the computational load of each task which results an increase in 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, but rather uses a localized heuristic that works very well in practice [2, 6, 31, 32]. The heuristic is based on generative exploration of the DAG that caps the number of outstanding tasks with a tunable parameter called *task window size*.

To explore more advanced features of QUARK we turn to Figure 8 which shows an annotated DAG of tasks that results from executing our LU factorization on a matrix with 3-by-3 tiles. One feature that we believe makes QUARK stand out is availability of nested parallelism without any constraints on the number of threads executing within a parallel task. The tasks that use these features (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 8. The former 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 other tasks in an optimally scheduled DAG. Even though in practice the schedule is not optimal due to 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 labelled as GATHERV in Figure 8. This feature allows for submission of tasks that write to different portions of the same submatrix. The Schur's complement update is performed with xGEMMs 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 so abundant in the update would have been lost. GATHERV allows for the recovery of 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.

And finally, QUARK can optionally generate DAGs such as the one featured in Figure 8. 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 results on the largest shared memory machines we could access at the time of writing this paper. 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 few cores to few tens of cores; from compute nodes with order $O(10)$ cores to $O(100)$ designs, and from Front Side Bus memory interconnect (Intel's NetBurst and Core Architectures) to NUMA and ccNUMA hardware (AMD's HyperTransport and Intel's QuickPath Interconnect).

6.1 Environment Settings

All the experiments are run on a single system that we will call *MagnyCour-48*. *MagnyCour-48*, is composed of four AMD Opteron Magny Cour 6172 Processors of twelve cores each, running at 2.1 GHz, with 128 GB of memory. The theoretical peak for this architecture in single and double precision arithmetics is 806.4 Gflop/s (16.8 Gflop/s per core) and 403.2 Gflop/s (8.4 Gflop/s per core), respectively.

We compare the results against the latest parallel version of the Intel MKL 10.3.2 library released in January 2011, and against the reference LAPACK 3.2 from Netlib linked against the Intel MKL BLAS multithreaded library. We also link against the sequential version of Intel MKL for our code and PLASMA.

6.2 Performance Results

Figures 9 and 10 present the performance comparisons of the parallel recursive tile LU algorithm against Intel MKL and PLASMA libraries on *MagnyCour-48*, in single and double precision arithmetics, respectively. Each curve is obtained by using the maximum number of cores available on the architecture, and by tuning the parameters to achieve the best asymptotic performance. Five configurations are shown: the LAPACK implementation from Netlib linked against the parallel BLAS from Intel MKL to study the fork-join model, the *Intel MKL* version of DGETRF, the previous algorithm of PLASMA based on incremental pivoting on the two different layouts studied: column-major (or *LAPACK*) and tile layout proper to PLASMA tile algorithms, and finally, our new algorithm *LU rec - Par. Panel* equally on both layouts. Both PLASMA versions correspond to the tile LU algorithm with incremental pivoting and use QUARK as a dynamic scheduling framework. The first version handles the LAPACK interface (native interface), which requires an input matrix in column-major data layout, similar to Intel MKL. It thus implies that PLASMA has to convert the matrix in tile data layout before the factorization can proceed and converts it back to column-major data layout at the end of the computation, as originally given by the user. 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.

We used a tile size $NB = 240$ and an inner-blocking $IB = 20$ for PLASMA algorithms and a panel width $NB = 280$ for our parallel recursive tile LU. Those parameters have been tuned for asymptotic performances. The recursive parallel panel LU algorithm based on column-major and tile data layout obtains roughly a 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, and 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 competitive compared to the column-major data layout version.

Moreover, Figures 9 and 10 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, is significantly better than Intel MKL asymptotically. The curves also show that the recursive LU algorithm is more sensitive for tuning small sizes than PLASMA algorithms, which produces good performance numbers on small matrix sizes even with the selected couple NB/IB for large matrices. For example, the configuration $N = 5000$ with 16 threads, gives the same performance ($\approx$ 75 Gflop/s) on PLASMA's incremental pivoting algorithm as on our new algorithm.

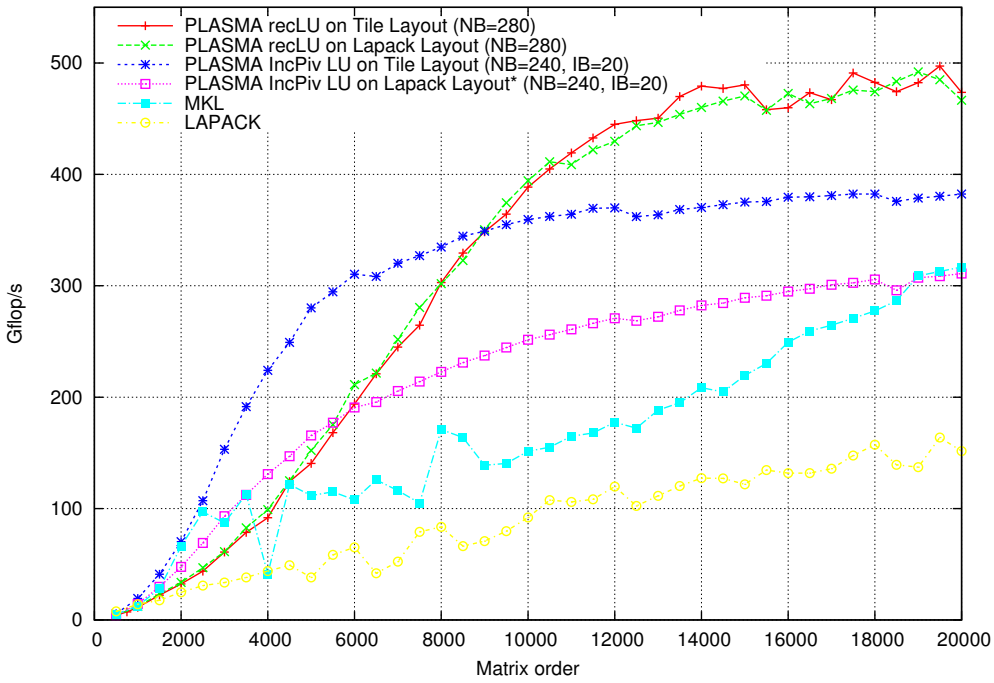


Fig. 9. Performances of SGETRF on *MagnyCour-48*.

The parallel recursive tile LU algorithm thus provides good performances on many-core architectures. It also retains the standard numerical accuracy as opposed to 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 efficiently parallelizing the panel factorization, and by the increase of the level of parallelism during the phase of the trailing submatrix updates, compared to the PLASMA's algorithms. Taking into account the fact that PLASMA's algorithm loses digits of precision, especially when the number of tiles increases [10], our new recursive tile LU factorization clearly appears to be a good alternative, and will eventually replace the current LU algorithm in PLASMA.

6.3 Scalability

Figures 11 and 12 show the scalability of the parallel recursive tile LU algorithm on *MagnyCour-48* in single and double precision arithmetics, respectively. The scaling experiments have been done according to the number of cores per socket

[1] PLASMA incpiv 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.

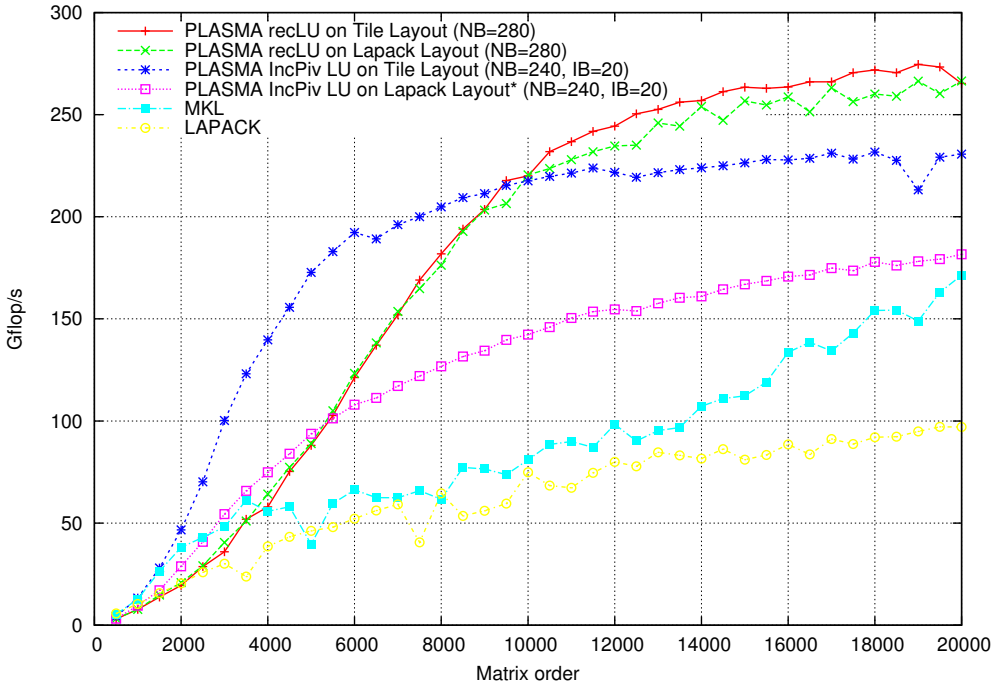


Fig. 10. Performances of `DGETRF` on *MagnyCour-48*.

(i.e., twelve cores), using 6, 12, 24 and 48 threads. For each curve, we used a panel width $NB = 280$, which gives the best performance on 48 threads. Since the data is usually allocated by the user and we do not move it 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 NUMA policy by allocating the memory close to the cores, where the threads are bound. We observe that our algorithm scales almost linearly up to 48 threads

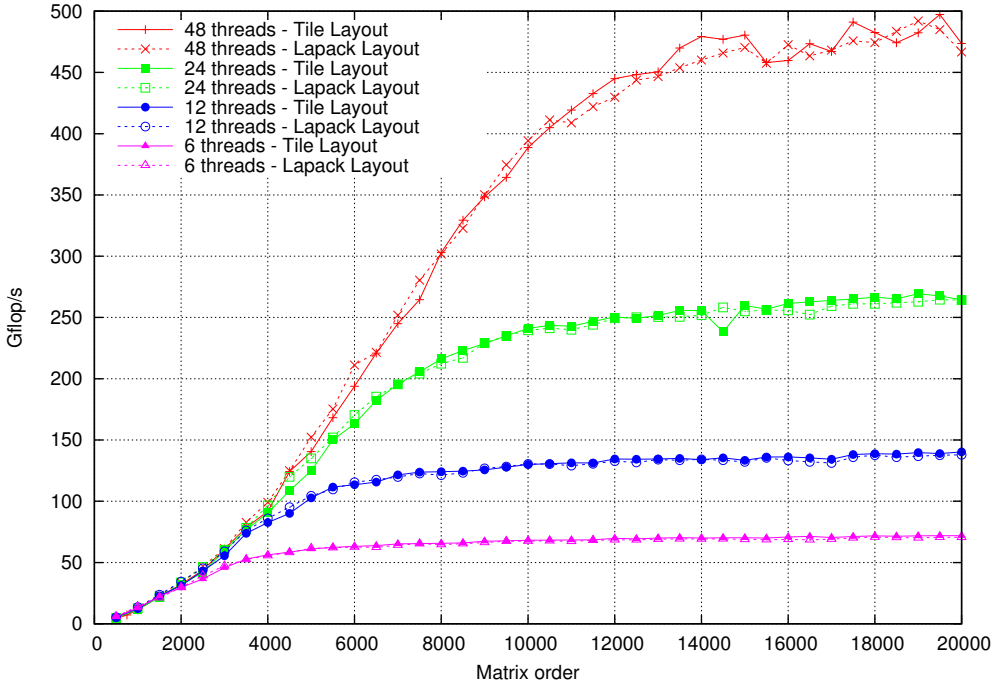


Fig. 11. Scalability of PLASMA recursive LU on *MagnyCour-48* in single precision.

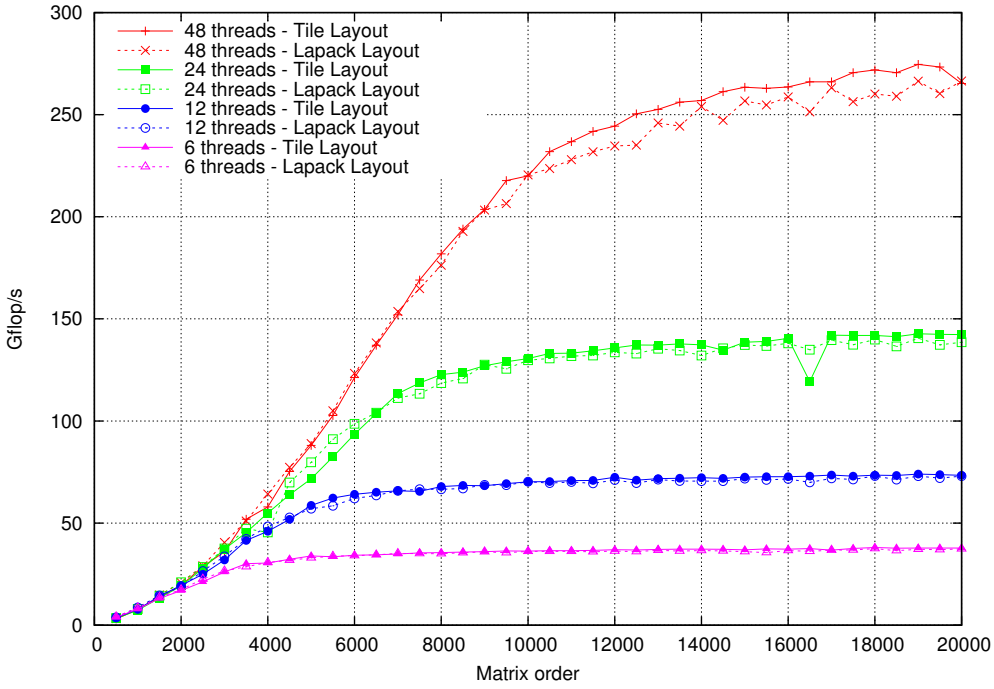


Fig. 12. Scalability of PLASMA recursive LU on *MagnyCour-48* in double precision.

6.4 Execution Trace

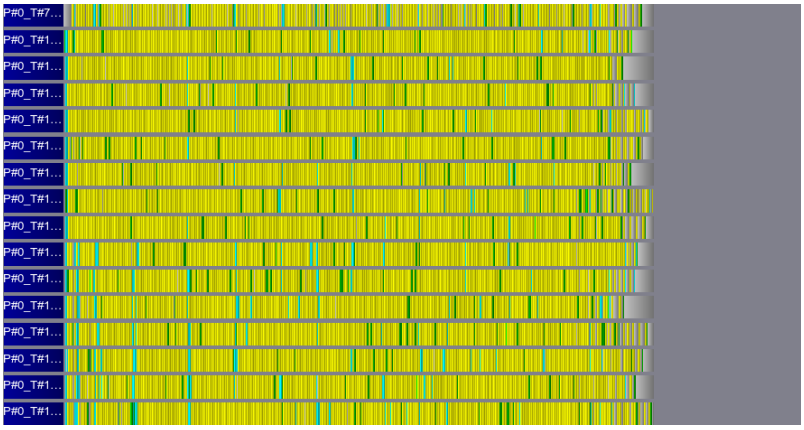
This section shows the execution traces of three different versions of the LU algorithm on *MagnyCour-48* with 16 threads on matrices of size 5000 × 5000. These traces have been generated thanks to the EZTrace library [33] and ViTE software [34]. On each trace, the green color is dedicated to the factorization 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 highlights the idle time.

The first trace 13(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 13(b) depicts the recursive tile LU, where the panel is rather 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 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 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.

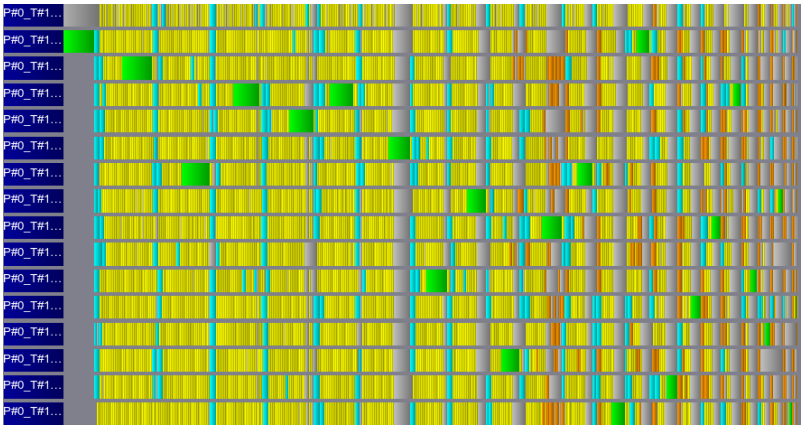
Finally, Figure 13(c) shows the execution trace of the same algorithm but with a parallel panel computation instead. This results in a reduced factorization step, which drastically reduces the overall idle time. It is also noteworthy to mention how the lookahead transparently comes into effect at runtime, thanks to the dynamic scheduler QUARK. Moreover, the non-critical tasks, which perform pivot interchanges behind the panel (xLASWP), are postponed until the end of the factorization in order to stress the pursuit of the critical path.

7 Summary and Future Work

This paper introduced a novel parallel recursive LU factorization with partial pivoting on multicore architecture. Our implementation is characterized by a parallel recursive panel factorizations while the computation on the trailing submatrix is carried on by using a tiled algorithm. Not only does this implementation achieve higher performance than the corresponding routines in LAPACK, (up to 3-fold speed-up) and MKL (up to 40% speed-up), but it also maintains the numerical quality of the standard LU factorization algorithm with partial pivoting which is not the case for PLASMA. Our approach uses a parallel fine-grained *recursive* formulation of the panel factorization step. Based on conflict-free



(a) Incremental pivoting `DGETRF` with $N = 5000$, $NB = 220$ and $IB = 20$



(b) Recursive LU `DGETRF` with sequential panel factorization, $N = 5000$ and $NB = 220$

(c) Recursive LU `DGETRF` with parallel panel factorization, $N = 5000$ and $NB = 220$

Fig. 13. 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`.

partitioning of the data and lockless synchronization mechanisms, our implementation lets the overall computation naturally flow without contention. The dynamic runtime system QUARK is then able to schedule tasks with heterogeneous granularities and to transparently introduce algorithmic lookahead.

The natural extension for this work would be the application of our methodology and implementation techniques to tile QR factorization. Even though, 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, a performance hit has been noticed for asymptotic sizes (also seen for the tile LU from 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 (similarly to the LU panel), the update would be much simpler (especially when targeting distributed systems) and based on single calls to xGEMM. The overall performance will then be guided solely by the performance of the matrix-matrix multiplication kernel, which is crucial when targeting asymptotic performance.

References

1. Anderson E, Bai Z, Bischof C, Blackford SL, Demmel JW, Dongarra JJ, Croz JD, Greenbaum A, Hammarling S, McKenney A, *et al.*. *LAPACK User's Guide*. 3rd edn., 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. University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3* November 2010.
4. 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**.
5. The FLAME project April 2010. http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage.
6. Luszczek P, Ltaief H, Dongarra J. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. *IEEE International Parallel and Distributed Processing Symposium* May 2011; .
7. Sorensen DC. Analysis of pairwise pivoting in gaussian elimination. *IEEE Tranasactions on Computers* March 1985; **C-34**(3).
8. Buttari A, Langou J, Kurzak J, Dongarra JJ. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parellel Comput. Syst. Appl.* 2009; **35**:38–53.
9. Quintana-Ortí G, Quintana-Ortí ES, Geijn RAVD, Zee FGV, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.* July 2009; **36**:14:1–14:26.
10. 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; .
11. Kurzak J, Ltaief H, Dongarra J, Badia R. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience* January 2010; **22**(1):15–44.
12. 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.
13. Intel, Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/.
14. Elmroth E, Gustavson FG. New serial and parallel recursive QR factorization algorithms for SMP systems. *Proceedings of PARA 1998*, 1998.
15. Georgiev K, Wasniewski J. Recursive Version of LU Decomposition. *Revised Papers from the Second International Conference on Numerical Analysis and Its Applications* 2001; :325–332.
16. Dongarra J, Eijkhout V, Luszczek P. Recursive approach in sparse matrix LU factorization. *Sci. Program.* January 2001; **9**:51–60.
17. 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.
18. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience* 2003; **15**:1–18.
19. 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* 2010; :223–232.
20. Gustavson FG. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* November 1997; **41**(6):737–755.
21. 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* 2010; :200–208.
22. 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.
23. 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.
24. Gustafson JL. Reevaluating Amdahl's Law. *Communications of ACM* 1988; **31**(5):532–533.
25. Sundell H. Efficient and practical non-blocking data structures. Department of computer science, Chalmers University of Technology, Göteborg, Sweden November 5 2004. PhD dissertation.
26. Yi Q, Kennedy K, You H, Seymour K, Dongarra J. Automatic blocking of QR and LU factorizations for locality. *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.
27. Blackford LS, Choi J, Cleary A, D'Azevedo EF, Demmel JW, Dhillon IS, Dongarra JJ, Hammarling S, Henry G, Petitet A, *et al.*. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics: Philadelphia, 1997.
28. 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; .

29. Dijkstra EW. On the role of scientific thought. *Selected writings on Computing: A Personal Perspective*, Dijkstra EW (ed.). Springer-Verlag New York, Inc.: New York, NY, USA, 1982; 60âĂŞ66. ISBN 0-387-90652-5.
30. Reade C. *Elements of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1989. ISBN 0201129159.
31. Buttari A, Langou J, Kurzak J, Dongarra JJ. Parallel Tiled QR Factorization for Multicore Architectures. *Concurrency Computat.: Pract. Exper.* 2008; **20**(13):1573–1590. http://dx.doi.org/10.1002/cpe.1301 DOI: 10.1002/cpe.1301.
32. Perez J, Badia R, Labarta J. A dependency-aware task-based programming environment for multi-core architectures. *Cluster Computing, 2008 IEEE International Conference on*, 2008; 142 –151, doi:10.1109/CLUSTR.2008.4663765.
33. Dongarra J, Faverge M, Ishikawa Y, Namyst R, Rue F, Trahay F. Eztrace: a generic framework for performance analysis. *Technical Report*, Innovative Computing Laboratory, University of Tennessee dec 2010.
34. Visual Trace Explorer. http://vite.gforge.inria.fr/.