IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 21, 2018, accepted January 14, 2019, date of publication January 31, 2019, date of current version February 14, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2895541

A Case for Malleable Thread-Level Linear Algebra Libraries: The LU Factorization With Partial Pivoting

SANDRA CATALÁN[1], JOSÉ R. HERRERO[2], ENRIQUE S. QUINTANA-ORTÍ[1], RAFAEL RODRÍGUEZ-SÁNCHEZ[3], AND ROBERT VAN DE GEIJN[4]

[1]Departarmento Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12071 Castellón de la Plana, Spain
[2]Departamento d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain
[3]Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, 28040 Madrid, Spain
[4]Department of Computer Science, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, USA

Corresponding author: José R. Herrero (josepr@ac.upc.edu)

This work was supported in part by the Spanish *Ministerio de Economía y Competitividad* under Project TIN2014-53495-R, Project TIN2015-65316-P, and Project TIN2017-82972-R, in part by the H2020 EU FETHPC "INTERTWinE" under Project 671602, in part by the Generalitat de Catalunya under Project 2017-SGR-1414, and in part by the NSF under Grant ACI-1550493.

ABSTRACT We propose two novel techniques for overcoming load-imbalance encountered when implementing so-called look-ahead mechanisms in relevant dense matrix factorizations for the solution of linear systems. Both techniques target the scenario where two thread teams are created/activated during the factorization, with each team in charge of performing an independent task/branch of execution. The first technique promotes *worker sharing* (WS) between the two tasks, allowing the threads of the task that completes first to be reallocated for use by the costlier task. The second technique allows a fast task to alert the slower task of completion, enforcing the *early termination* (ET) of the second task, and a smooth transition of the factorization procedure into the next iteration. The two mechanisms are instantiated via a new *malleable* thread-level implementation of the *basic linear algebra subprograms*, and their benefits are illustrated via an implementation of the LU factorization with partial pivoting enhanced with look-ahead. Concretely, our experimental results on an Intel-Xeon system with 12 cores show the benefits of combining WS+ET, reporting competitive performance in comparison with a task-parallel runtime-based solution.

INDEX TERMS Solution of linear systems, multi-threading, workload balancing, thread malleability, basic linear algebra subprograms (BLAS), linear algebra package (LAPACK).

I. INTRODUCTION

In the 1970s and 80s, the scientific community recognized the value of defining standard interfaces for dense linear algebra (DLA) operations with the introduction of the *Basic Linear Algebra Subprograms* (BLAS) [1]–[3]. Ever since, the key to performance portability in this domain has been the development of highly-optimized, architecture-specific implementations of the BLAS, either by hardware vendors (e.g., Intel MKL [4], AMD ACML [5], IBM ESSL [6], and NVIDIA CUBLAS [7]) or independent developers (e.g., GotoBLAS [8], [9], OpenBLAS [10], ATLAS [11], and BLIS [12]).

Multi-threaded instances of the BLAS for current multi-core processor architectures take advantage of the simple data dependencies featured by these operations to exploit loop/data-parallelism at the block level (hereafter referred to as *block-data parallelism*, or BDP). For more complex DLA operations, like those supported by LAPACK [13] and `libflame` [14], exploiting *task-parallelism* with dependencies[1] (TP) is especially efficient when performed by a runtime that semi-automatically decomposes the computation into tasks and orchestrates their dependency-aware scheduling [15]–[18]. For the BLAS kernels though,

[1]We view TP as the type of concurrency present in an operation that can be decomposed into a collection of suboperations (tasks) connected by a rich set of data dependencies. Compared with this, BDP is present when the operation basically consists of a number of independent (inherently parallel) suboperations, each acting on a disjoint block of data.

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

exploiting BDP is still the preferred choice, because it allows tighter control on the data movements across the memory hierarchy and avoids the overhead of a runtime that is unnecessary due to the (mostly) nonexistent data dependencies in the BLAS kernels.

Exploiting both BDP and TP, in a sort of nested parallelism can yield more efficient solutions as the number of cores in processor architectures continues to grow. For example, consider an application composed of two independent tasks, $\mathsf{T_A}$ and $\mathsf{T_B}$, both of which are inherently parallel and preceded/followed by synchronization points. In this scenario, exploiting TP only is inefficient, because it can keep at most 2 threads busy. To address this, we could take advantage of the BDP inside $\mathsf{T_A}$ and $\mathsf{T_B}$ via TP but, given their inherent parallelism, this is likely to incur certain overhead compared with a direct exploitation of BDP. Finally, extracting BDP only is surely possible, but it may be less scalable than a nested TP+BDP solution that splits the threads into two teams, and puts them to work on $\mathsf{T_A}$ and $\mathsf{T_B}$ concurrently.

Let us consider now that the previous scenario occurs during the execution of a complex DLA operation where both tasks, $\mathsf{T_A}$ and $\mathsf{T_B}$, can be computed via simple calls to a multi-threaded instance of BLAS. Although the solution seems obvious, exploiting nested TP+BDP parallelism here can still be suboptimal. In particular, *all multi-threaded instances of BLAS offer a rigid interface to control the threading execution of a routine, which only allows one to set the number of threads that will participate before the routine is invoked. Importantly, this number cannot be changed during its execution*. Thus, in case $\mathsf{T_A}$ is completed earlier, the team of threads in charge of its execution will remain idle waiting for the completion of $\mathsf{T_B}$, producing a suboptimal execution from a performance perspective.

The scenario that we have described is far from being rare in DLA. To illustrate this, we will employ the LU factorization with partial pivoting for the solution of linear systems [19]. High-performance algorithms for this decomposition consist of a loop-body that processes the matrix from its top-left corner to the bottom-right one, at each iteration computing a panel[2] factorization and updating a trailing submatrix via calls to BLAS. We will review this factorization as a prototypical example to make the following contributions in our paper:

- *Malleable DLA libraries*: We introduce a malleable thread-level implementation of BLIS [12] that allows the number of threads that participate in the execution of a BLAS kernel to dynamically change at execution time.
- *Worker Sharing (WS):* In case the panel factorization is less expensive than the update of the trailing submatrix, we leverage the malleable instance of the BLAS to improve workload balancing and performance, by allowing the thread team in charge of the panel factorization to be reallocated to the execution of the trailing update.
- *Early Termination (ET)*: To tackle the opposite case, where panel factorization is more expensive than the update of the trailing submatrix, we design an ET mechanism that allows the thread team in charge of the trailing update to communicate the alternative team of this event. This alert forces an ET of the panel factorization, and the advance of the factorization into the next iteration.
- We perform a comprehensive experimental evaluation on a 6-core Intel Xeon E5-2603 v3 processor, using execution traces to illustrate actual benefits of our approach, and comparing its performance to those obtained with a runtime-based solution using OmpSs [15]. Our study includes results using a dual-socket (i.e., twelve-core) configuration.

The key to our approach is that we depart from conventional instances of BLAS to instead view the cores/threads as a pool of computational resources that, upon completing the execution of a BLAS/LAPACK routine, can be tapped to participate in the execution of another BLAS/LAPACK routine that is already in progress. This WS supports a dynamic choice of the algorithmic block size as the operation progresses. Furthermore, the same idea carries over to all other major matrix decompositions for the solution of linear systems, such as the QR, Cholesky and LDL^T factorizations [19].

In [20] we leverage the techniques presented in this paper, focussing on the exploitation of hardware/frequency-asymmetric multicore architectures, studying the optimal mapping of the different types of tasks to the existing heterogenous resources in seach of faster global execution and lower energy consumption.

II. THE BLIS IMPLEMENTATION OF BASIC LINEAR ALGEBRA KERNELS

BLIS is a framework that allows developers to rapidly deploy new high-performance implementations of BLAS and BLAS-like operations on current and future architectures [12]. A key property of the BLIS open source effort is that it exposes the internal implementation of the BLAS kernels at a finer-grain level than OpenBLAS or commercial libraries while offering performance that is competitive with GotoBLAS, OpenBLAS, Intel MKL, and ATLAS [21], [22]. We start by reviewing the design principles that underlie BLIS, using the implementation of gemm as a particular case study.

Consider the matrices $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, and $C \in \mathbb{R}^{m\times n}$. BLIS mimics GotoBLAS to implement the gemm kernel[3]

$$C \mathrel{+}= A \cdot B \tag{1}$$

[2]In the following, we will refer to a block with more rows than columns as a panel.

[3]Actually, the kernel in the BLAS interface/BLIS implementation for gemm computes $C = \alpha C + \beta op(A) \cdot op(B)$, where α, β are scalars, $op(\cdot)$ performs an optional transposition/Hermitian-conjugation, and $op(A)$ is $m \times k$, $op(B)$ is $k \times n$, C is $m \times n$. For simplicity, in the description we address the case where $\alpha = \beta = 1$ and the operator $op(\cdot)$ does not perform any transformation on the input matrix.

Loop 1 **for** $j_c = 0, \ldots, n-1$ **in steps of** n_c
Loop 2 **for** $p_c = 0, \ldots, k-1$ **in steps of** k_c
$B_c := B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1)$ // Pack into B_c
Loop 3 **for** $i_c = 0, \ldots, m-1$ **in steps of** m_c
$A_c := A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1)$ // Pack into A_c
Loop 4 **for** $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r // Macro-kernel
Loop 5 **for** $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
$C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ // Micro-kernel
$+= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1)$
$\cdot B_c(0 : k_c - 1, j_r : j_r + n_r - 1)$
endfor
endfor
endfor
endfor
endfor

FIGURE 1. High performance implementation of gemm in BLIS. In the code, $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is just a notation artifact, introduced to ease the presentation of the algorithm. In contrast, A_c, B_c correspond to actual buffers that are involved in data copies.

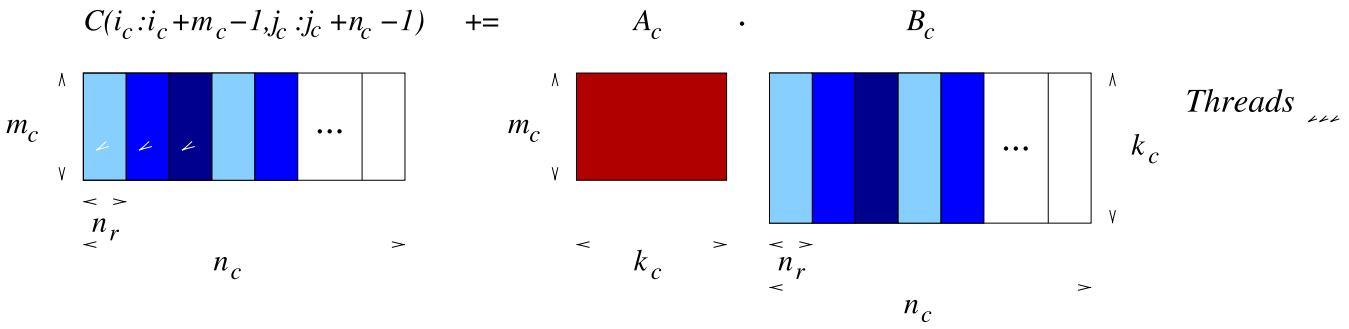

FIGURE 2. Distribution of the workload among $t = 3$ threads when Loop 4 of BLIS gemm is parallelized. Different colors in the output C distinguish the panels of this matrix that are computed by each thread as the product of A_c and corresponding panels of the input B_c.

as three nested loops around a macro-kernel plus two packing routines (see Loops 1–3 in FIGURE 1). The macro-kernel is then implemented in terms of two additional loops around a *micro-kernel* (Loops 4 and 5 in that figure). The loop ordering embedded in BLIS, together with the packing routines and an appropriate choice of the BLIS cache configuration parameters (n_c, k_c, m_c, n_r and m_r), orchestrate a regular pattern of data transfers across the levels of the memory hierarchy, and amortize the cost of these transfers with enough computation from within the micro-kernel [12] to attain near-peak performance. In most architectures, m_r, n_r are in the range 4–16; m_c, k_c are in the order of a few hundreds; and n_c can be up to a few thousands [12], [21].

The parallelization of BLIS's gemm for multi-threaded architectures has been analyzed for conventional symmetric multicore processors [21], modern many-threaded architectures [22], and asymmetric multicore processors [23]. In all these cases, the parallel implementation exploits the BDP exposed by the nested five-loop organization of gemm, at one or more levels (i.e., loops), using OpenMP or POSIX threads.

A convenient option in most single-socket systems is to parallelize either Loop 3 (indexed by i_c), Loop 4 (indexed by j_r), or a combination of both [21]–[23]. For example, when Loop 3 is parallelized, each thread packs a different macro-panel A_c into the L2 cache and executes a different instance of the macro-kernel. In contrast, when Loop 4 is parallelized, different threads will operate on independent instances of the micro-kernel, but access the same macro-panel A_c in the L2 cache.

Consider, for example, a version of BLIS gemm that extracts BDP from Loop 4 only, to be executed on a multicore architecture with t (physical) cores and one thread mapped per core. The iteration space of Loop 4 is then statically distributed among the t threads in a round-robin fashion, equivalent to the effect attained by adding an OpenMP directive `#pragma omp parallel for`, with a `static` schedule, around that loop; see Figure 2. To improve performance, the packing is also performed in parallel so that, at each iteration of Loop 3, all t threads collaborate to copy and re-organize the entries of $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1)$ into the buffer A_c.

III. ALGORITHMS FOR THE LU FACTORIZATION ON MULTI-THREADED ARCHITECTURES

Given a matrix $A \in \mathbb{R}^{m \times n}$, its LU factorization produces lower and upper triangular factors, $L \in \mathbb{R}^{m \times n}$ and $U \in \mathbb{R}^{n \times n}$ respectively, such that $PA = LU$, where $P \in \mathbb{R}^{m \times m}$ defines a permutation that is introduced for numerical stability [19]. In this section we first review the conventional unblocked

and blocked algorithms for the LU factorization, and then describe how BDP is exploited from within them. Next, we introduce a re-organized version of the algorithm that integrates look-ahead in order to enhance performance in a nested TP+BDP execution.

The experiments in the remainder of the paper were obtained using ieee double-precision arithmetic on a server equipped with Intel Xeon E5-2603 v3 technology. One socket (six cores) at a nominal frequency 1.6 GHz were used in all experiments except those in section V-C1, which use a dual-socket configuration. The implementations were linked with BLIS version 0.1.8 or a tailored version of this library especially developed for this work. Unless otherwise stated, BDP parallelism is extracted only from Loop 4 of the BLIS kernels. All traces in this paper were obtained using `Extrae` version 3.3.0 [24].

A. BASIC ALGORITHMS AND BDP

There exist a number of algorithmic variants of the LU factorization that can accommodate partial pivoting [19]. Among these, Figure 3 (top) shows an unblocked algorithm for the so-called *right-looking* (RL) variant, expressed using the FLAME notation [25]. For simplicity, we do not include pivoting in the following description of the algorithms, though all our actual implementations, (and in particular those employed in our experimental evaluation,) integrate standard partial pivoting. The cost of computing the LU factorization of an $m \times n$ matrix, via any of the algorithms presented in this paper, is $mn^2 - n^3/3$ floating-point arithmetic operations (flops). Hereafter, we will consider square matrices of order n for which, the cost boils down to $2n^3/3$ flops. For the RL variants, the major part of these operations are concentrated in the initial iterations of the algorithm(s). For example, the first 25% iterations account for almost 58% of the flops; the first half for 87.5%; and the first 75% for more than 98%. Thus, the key to high performance mostly lies in the initial stages of the factorization.

For performance reasons, dense linear algebra libraries compute the LU factorization via a blocked algorithm that casts most computations in terms of gemm. Figure 3 (bottom) presents the blocked RL algorithm. For each iteration, the algorithm processes panels of b columns, where b is the algorithmic block size. The three operations in the loop body factorize the "current" panel $A_p = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$, via the unblocked algorithm (LU_unb, **RL1**); and next update the trailing submatrix, consisting of A_{12} and A_{22}, via a triangular system solve (trsm, **RL2**) and a matrix multiplication (gemm, **RL3**), respectively. In practice, the block size b is chosen so that the successive invocations to the gemm kernel deliver high FLOPS (flops per second) rates. If b is too small, the performance of gemm will suffer, and so will that of the LU factorization. On the other hand, reducing b is appealing as this choice decreases the number of flops that are performed in terms of the panel factorization, an operation that can be expected to offer significantly lower throughput (FLOPS) than gemm. (Concretely, provided $n \gg b$, the cost required for all panel factorizations is about $n^2 b/2$ flops.) Thus, there is the tension between these two requisites.

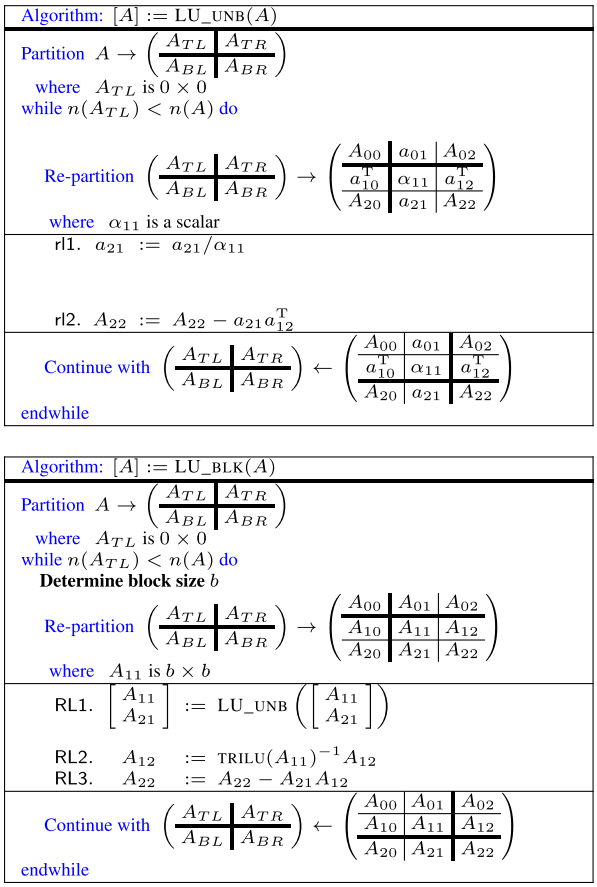

FIGURE 3. Unblocked and blocked RL algorithms for the LU factorization (top and bottom, respectively). In the notation, $n(\cdot)$ returns the number of columns of its argument, and trilu$(\cdot)$ returns the strictly lower triangular part of its matrix argument, setting the diagonal entries of the result to ones; furthermore, the arrows "$\leftarrow$" and "$\rightarrow$" are simply used to denote partitioning/repartitionings of the operands. These algorithms correspond to the well-known right-looking version of the factorization and are included for reference.

When the target platform is a multicore processor, the conventional parallelization of the LU factorization simply relies on multi-threaded instances of trsm and gemm to exploit BDP only. Compared with this, the panel factorization of A_p, which lies in the critical path of the blocked RL factorization algorithm, exhibits a reduced degree of concurrency. Thus, depending on the selected block size b and certain hardware features of the target architecture (number of cores, floating-point performance, memory bandwidth, etc.), this operation may easily become a performance bottleneck; see Figure 4.

To illustrate the performance relevance of the panel factorization, Figure 5 displays a fragment of a trace corresponding to the LU factorization of a $10,000 \times 10,000$ matrix, using the blocked RL algorithm in Figure 3, with partial pivoting and "outer" block size $b = b_o = 256$. The code is linked with multi-threaded versions of the BLIS kernels for gemm and

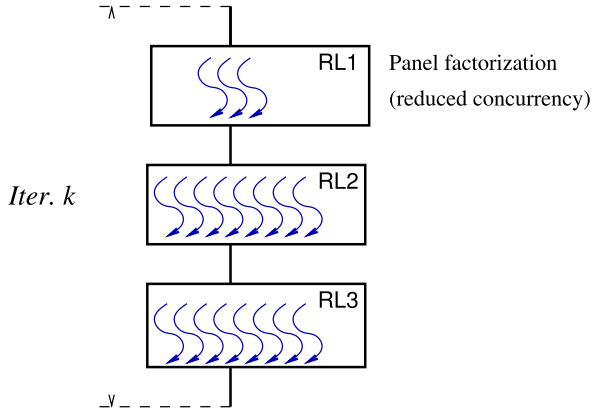

FIGURE 4. Exploitation of BDP in the blocked RL LU parallelization. A single thread team executes all the operations, with less active threads for RL1 due to the reduced concurrency of this kernel. In this algorithm, RL1 stands in the critical path.

trsm. The panel factorization (panel) is performed via a call to the same blocked algorithm, with "inner" block size $b = b_i = 32$, and also extracts BDP from the same two kernels. With this configuration, the panel factorization represents less than 2% of the flops performed by the algorithm. However, the trace of the first four iterations reveals that its practical cost is much higher than could be expected. (The cost of factorizing a panel relative to the cost of an iteration becomes even larger as the iteration progresses.) Here we note also the significant cost of the row permutations, which are performed via the sequential legacy code for this routine in LAPACK (laswp). However, this second operation is embarrassingly parallel and its execution time can be expected to decrease linearly with the number of cores.

At this point, we note that the operations inside the loop body of the blocked algorithm in Figure 3 (bottom) present strict dependencies (denoted heafter with the symbol "⇒") that enforce their computation in the order RL1 ⇒ RL2 ⇒ RL3. Therefore, there seems to be no efficient manner to formulate a TP version of the blocked algorithm in that figure.

B. STATIC LOOK-AHEAD AND NESTED TP+BDP

A strategy to tackle the hurdle represented by the panel factorization in a parallel execution consists in the introduction of look-ahead [26] into the algorithm. Concretely, during each iteration of the decomposition this technique aims to overlap the factorization of the "next" panel with the update of the "current" trailing submatrix, in practice enabling a TP version of the algorithm with two separate branches of execution, as discussed next.

Figure 6 illustrates a version of the blocked RL algorithm for the LU factorization re-organized to expose look-ahead. The key is to partition the trailing submatrix into two block column panels:

$$\left(\frac{A_{12}}{A_{22}}\right) \rightarrow \left(\begin{array}{c|c} A_{12}^P & A_{12}^R \\ \hline A_{22}^P & A_{22}^R \end{array}\right) \quad (2)$$

where A_{22}^P corresponds to the block that, in the conventional version of the algorithm (i.e., without look-ahead,) would be factorized during the next iteration. This effectively separates the blocks that are modified as part of the next panel factorization from the the remainder updates, left and right of the 2×2 partitioning in (2), respectively. Proceeding in this manner creates two coarse-grain independent tasks (groups of operations in separate branches of execution): $\mathsf{T_{PF}}$, consisting of PF1, PF2, PF3; and $\mathsf{T_{RU}}$, composed of RU1 and RU2; see Figure 6. The "decoupling" of these block panels thus facilitate that, in a TP execution of an iteration of the loop body of the look-ahead version, the updates on A_{12}^P, A_{22}^P and the factorization of the latter (operations on the next panel, in $\mathsf{T_{PF}}$) proceed concurrently with the updates of A_{12}^R, A_{22}^R (remainder operations, in $\mathsf{T_{RU}}$), as there are no dependencies between $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$.

By carefully tuning the block size b and adjusting the amount of computational resources (threads) dedicated to each of the two independent tasks, $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$, a nested TP+BDP execution of the algorithm enhanced with this static look-ahead can partially or totally overcome the bottleneck represented by the panel factorization; see Figure 7.

Figure 8 illustrates a complete overlap of $\mathsf{T_{RU}}$ with $\mathsf{T_{PF}}$ attained by the look-ahead technique. The results in that figure correspond to a fragment of a trace obtained for the LU factorization of a $10,000 \times 10,000$ matrix, using the blocked RL algorithm in Figure 6, with partial pivoting, and outer block size $b = b_o = 256$. For this experiment, the $t = 6$ threads are partitioned into two teams: PF with $t_{\mathsf{pf}} = 1$ thread in charge of $\mathsf{T_{PF}}$, and RU with $t_{\mathsf{ru}} = 5$ threads responsible for $\mathsf{T_{RU}}$. The panel factorization (panel) is performed via a call to the same algorithm, with $b = b_i = 32$, and this operation proceeds sequentially (as PF consists of a single

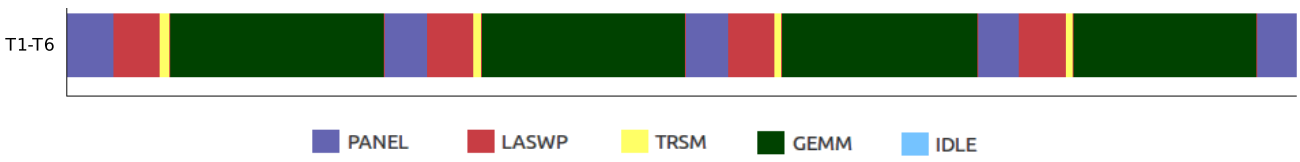

FIGURE 5. Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$. The x-axis represents the execution timeline while the different colors correspond to the fraction of the time that the threads spend in each of the different kernels appearing in the LU factorization: from left to right, panel factorization (PANEL), row permutations (LASWP), triangular system solve (TRSM), matrix-matrix multiplication (GEMM), and idle.

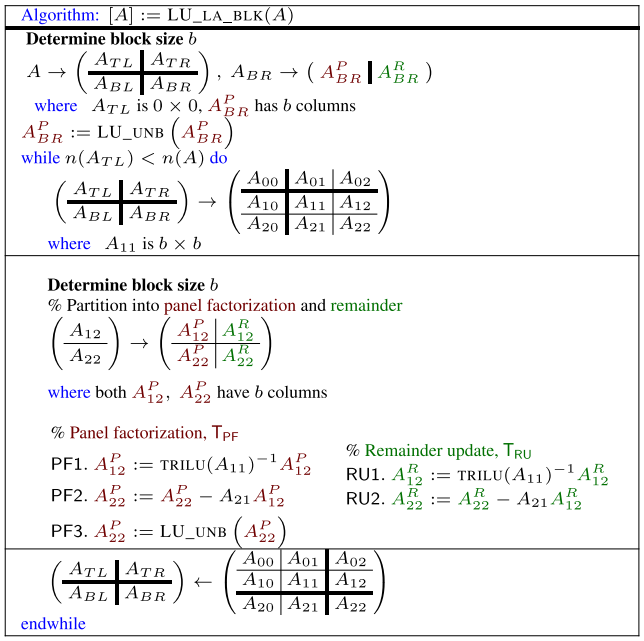

FIGURE 6. Blocked RL algorithm enhanced with look-ahead for the LU factorization. This algorithm is included for reference. Note that there is an implicit schedule in the formulation of this algorithm, as the operations corresponding to the panel factorization can be computed in parallel with those for the remainder update.

thread). The application of the row permutations is distributed between all 6 cores. As argued earlier, the net effect of the look-ahead is that the cost of the panel factorization no longer has a practical impact on the execution time of the (first four iterations of) the factorization algorithm, which is now basically determined by the cost of the remaining operations.

Given a static mapping of threads to tasks, b should balance the time spent in the two tasks as, if the operations in $\mathsf{T_{PF}}$ take longer than those in $\mathsf{T_{RU}}$, or vice-versa, the threads in charge of the less expensive part will become idle, causing a performance degradation. This was already visible in Figure 8, which shows that, during the first four iterations, the operations in $\mathsf{T_{PF}}$ are considerably less expensive than the updates performed as part of the remainder $\mathsf{T_{RU}}$. The complementary case, where $\mathsf{T_{PF}}$ requires longer than $\mathsf{T_{RU}}$, is illustrated using the same configuration, for a matrix of dimension $2,000 \times 2,000$, in Figure 9. Unfortunately, as the factorization proceeds, the theoretical costs and execution times of $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$ vary, making it difficult to determine the optimal value of b, which will need to be adapted during the factorization process.

To close this section, note that there exist strict dependencies that serialize the operations within each task: PF1 $\Rightarrow$ PF2 $\Rightarrow$ PF3 and RU1 $\Rightarrow$ RU2. Therefore, there is no further TP in the loop-body of this re-organized version. However, the basic look-ahead mechanism of level/depth 1 described in this subsection can be refined to accommodate further levels of TP, by "advancing" to the current iteration the panel factorization of the following d iterations, in a look-ahead of level/depth d. This considerably complicates the code of the algorithm, but can be seamlessly achieved by a runtime system enhanced with priorities.

IV. ADVOCATING FOR MALLEABLE THREAD-LEVEL LA LIBRARIES

For simplicity, in the following discussion we will assume that $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$ consist only of the panel factorization involving A_{22}^P (PF3) and the update of A_{22}^R (RU2), respectively. Furthermore, we will consider a nested TP+BDP execution

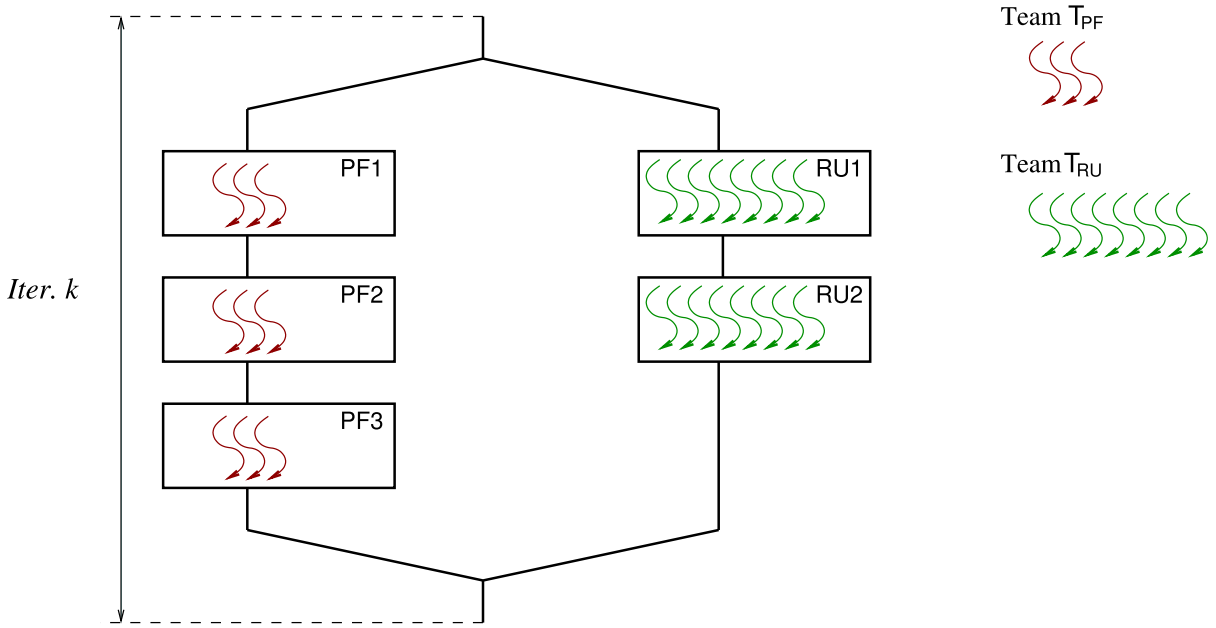

FIGURE 7. Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead. The execution is performed by teams $\mathsf{T_{PF}}$ and $\mathsf{T_{RU}}$, consisting of $t_{pf} = 3$ and $t_{ru} = 8$ threads, respectively. In this algorithm, the operations on the $(k+1)$-th panel, including its factorization (PF3), are overlapped with the updates on the remainder of the trailing submatrix (RU1 and RU2).

FIGURE 8. Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$.

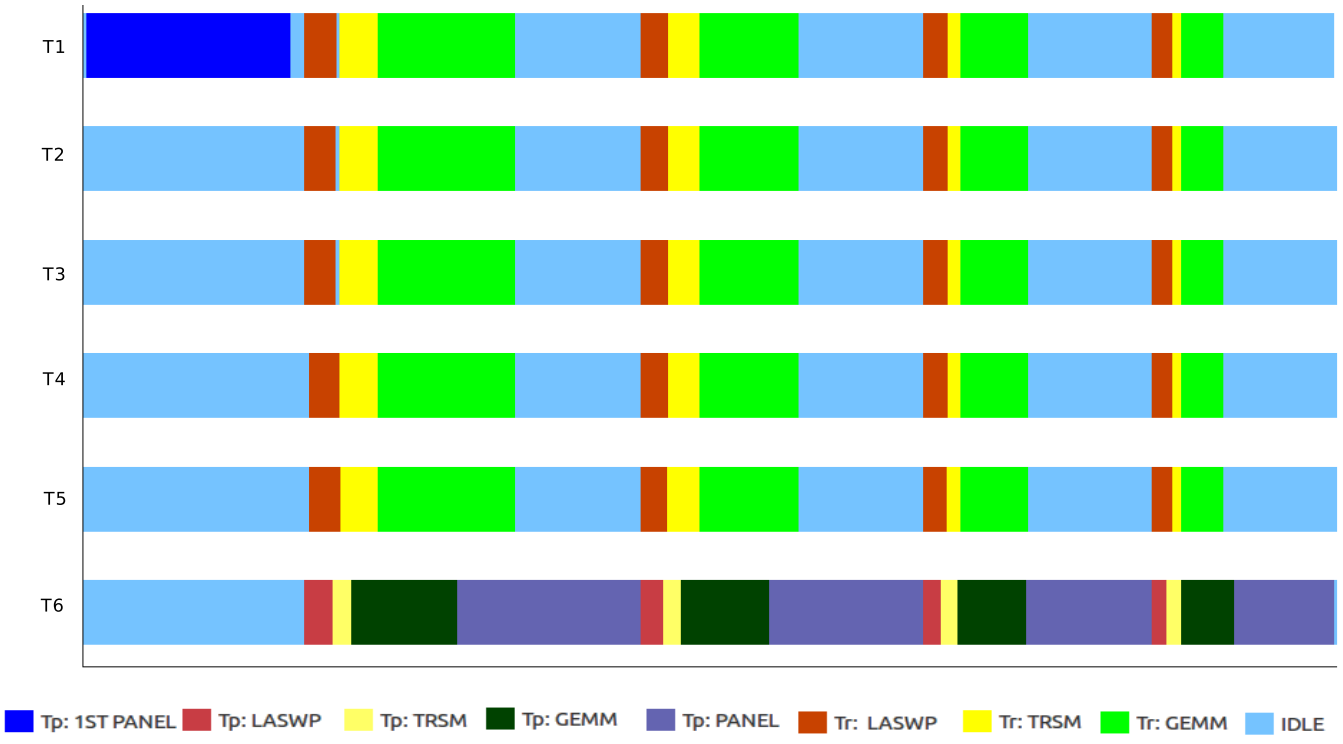

FIGURE 9. Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead, using 6 threads, applied to a square matrix of order 2,000, with $b_o = 256$, $b_i = 32$.

using $t = t_{\mathsf{pf}} + t_{\mathsf{ru}}$ threads, *initially* with a team PF of t_{pf} threads mapped to the execution of PF3 and a team RU of t_{ru} threads computing RU2.

Ideally, for the LU factorization with look-ahead, we would like to perform a *flexible sharing* of the computational resources so that, as soon as the threads in team PF complete

PF3, they join team RU to help in the execution of RU2 or vice-versa. We next discuss these two cases in detail.

A. WORKER SHARING: PANEL FACTORIZATION LESS EXPENSIVE THAN UPDATE

Our goal is to enable, at each iteration of the algorithm for the LU factorization with look-ahead, that the threads in team PF that complete the panel factorization join the thread team RU working on the update. The problem is that, if the multiplication to update A_{22}^R was initiated via an invocation to a traditional gemm, this is not possible as none of the existing high performance implementations of BLAS allow the number of threads working on a kernel that is already in execution to be modified.

1) SUBOPTIMAL SOLUTION: STATIC RE-PARTITIONING

A simple workaround for this problem is to split A_{22}^R into multiple column blocks, for example, $A_{22}^R \rightarrow \left(A_1|A_2|\ldots|A_q\right)$, and to perform a separate call to BLAS gemm in order to exploit BDP during the update of each block. Then, just before each invocation, the kernel's code queries whether the execution of the panel factorization is completed and, if that is the case, executes the suboperation with the threads from both teams (or only those of RU otherwise). Unfortunately, this approach presents several drawbacks:

- Replacing a single invocation to a coarse gemm by multiple calls to smaller gemm may offer lower throughput because the operands passed to gemm are smaller and/or suboptimally ''shaped''. The consequence is that calling gemm multiple times will internally incur re-packing and data movement overheads, which are more difficult to amortize because of the smaller problem dimensions.
- The burden of which loop to partition for parallelism (note that A_{22}^R could have alternatively been split by rows, or into blocks), and the granularity of this partitioning is then placed upon the programmer's shoulders, who may lack the information that is necessary to make a good choice. For example, if the granularity is too coarse, this will have negative effect because the integration of the single thread in the update will likely be delayed. A granularity that is too fine, on the other hand, may reduce the parallelism within the BLAS operation or result in the use of cache blocking parameters that are too small.

2) OUR SOLUTION: MALLEABLE THREAD-LEVEL BLAS

The alternative that we propose in this work exploits BDP inside RU2, *but allows to change the number of threads that participate in this computation even if the task is already in execution!* In other words, we view the threads as a resource pool of workers that can be shared between different tasks and reassigned to the execution of a (BLAS) kernel that is already in progress.

The key to our approach lies in the explicit exposure of the gemm internals (and other BLAS-3 kernels) in BLIS. Concretely, assume that RU2 is computed via a single invocation to BLIS gemm, and consider that this operation is parallelized by distributing the iteration space of Loop 4 among the threads in team PF; (see Figures 1 and 2). Then, just before Loop 4, we force the system to check if the execution of the panel factorization is completed and, based on this information, decides whether this loop is executed using either the union of the threads from both teams or only those in RU; see Figure 10.

Let us re-analyze the problems listed in Subsection IV-A1 for the work-around that statically partitioned the update of A_{22}^R, and compare them with our solution that implicitly embeds this partitioning inside BLIS:

- The partitioning of gemm into multiple calls to smaller matrix multiplications does not occur. Our solution performs a single call to gemm only, so that there is no additional re-packing nor data movements. For example, in the case just discussed, B_c is already packed and re-used independently of whether t or t_{RU} threads participate in the gemm. The buffer A_c is packed only once per iteration of Loop 3 (in parallel by both teams or only RU).
- The decision of the best partitioning/granularity is left in the hands of BLIS, which likely has more information to do a better job than the programmer.

Importantly, the partitioning happens dynamically and is transparent to the programmer.

Figure 11 validates the effect of integrating a malleable version[4] of BLIS into the same configuration that produced the results in Figure 8. A comparison of both figures shows that, with a malleable version of BLIS, the thread executing the operations in $\mathsf{T_{PF}}$, after completing this task, rapidly joins the team that computes the remainder updates, thus avoiding the idle wait.

Compared with BLIS, the same approach cannot be integrated into GotoBLAS because the implementation of gemm in this library only exposes the three outermost loops of FIGURE 1, while the remaining loops are encoded in assembly. The BLAS available as part commercial libraries is not an option either because hardware vendors offer black-box implementations which do not permit the migration of threads.

B. EARLY TERMINATION: PANEL FACTORIZATION MORE EXPENSIVE THAN UPDATE

The analysis of this case will reveal some important insights. In order to discuss them, let us consider that, in the LU factorization with look-ahead, the panel factorization (PF3) is performed via a call to the blocked routine in Figure 3 (right). We assume two blocking parameters: $b = b_o$ for the outer routine that computes the LU factorization of the complete matrix using look-ahead, and $b = b_i$ for the inner routine that factorizes each panel. (Note that, if $b_i = b_o$ or $b_i = 1$,

[4]The discussion on how to create this ''malleable'' instance of BLIS is an implementation detail, which is not relevant for the discussion of the paper.

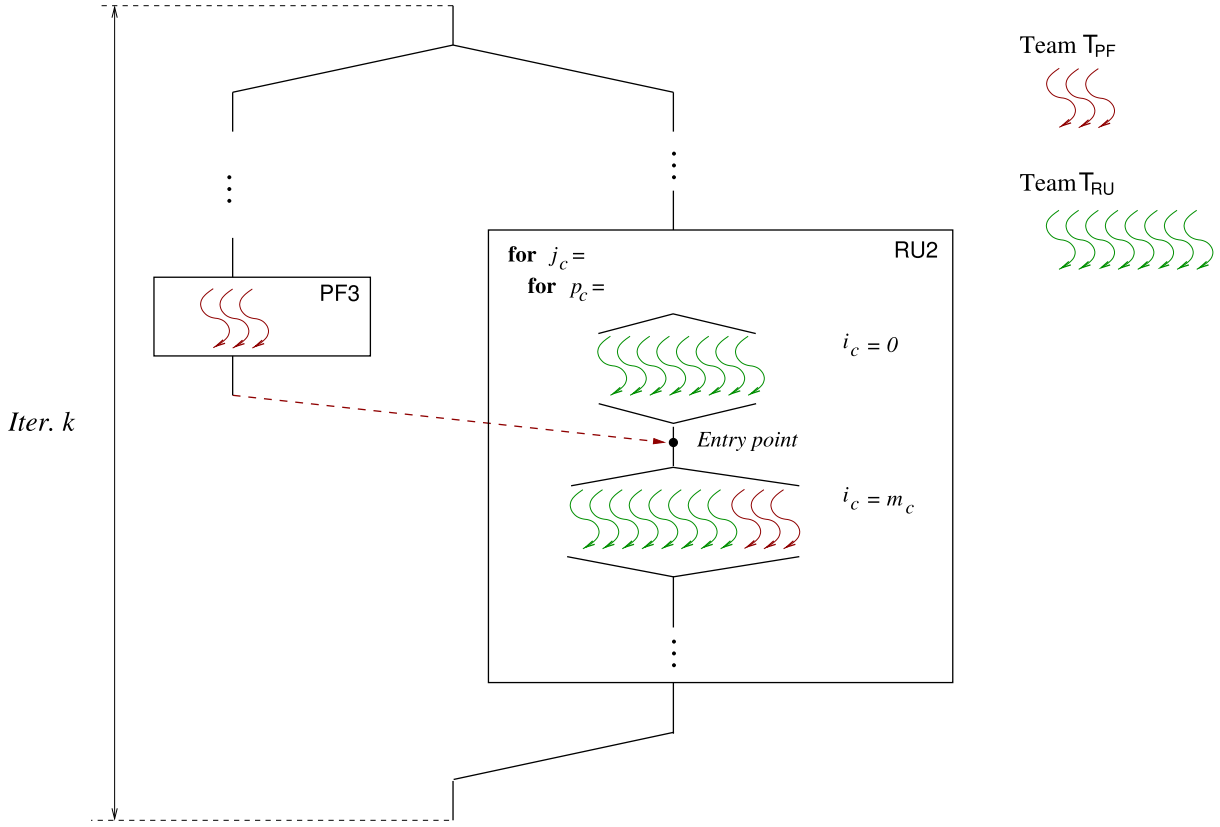

FIGURE 10. Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead and WS. The execution is performed by teams T_{PF} and T_{RU}, consisting of $t_{pf} = 3$ and $t_{ru} = 8$ threads, respectively. In this example, team T_{PF} completes the factorization PF3 while team T_{RU} is executing the first iteration of Loop 3 that corresponds to RU2/gemm ($i_c = 0$). Both teams then merge and jointly continue the update of the remaining iterations of that loop ($i_c = m_c, 2m_c, \ldots$). With the parallelization of gemm Loop 4, one such "entry point" enables the merge at the beginning of each iteration of loop i_c.

the panel factorization is then simply done via the unblocked algorithm.) Furthermore, we will distinguish these two levels by referring to them as the *outer LU* (factorization with look-ahead) and the *inner LU* (factorization via the blocked algorithm without look-ahead). Thus, at each iteration of the outer LU, a panel of b_o columns is factorized via a call to LU_blk (inner LU), and this second decomposition proceeds to factorize the panel using a blocked algorithm with block size b_i; see Figure 12.

From Figure 3 (right), the loop body for the inner LU consists of a call to the unblocked version of the algorithm (**RL1**), followed by the invocations to trsm and gemm that update A_{12} and A_{22}, respectively (**RL2** and **RL3**). Now, let us assume that the update **RU2** by the thread team **RU** is completed while the threads of team **PF** are in the middle of the computations corresponding to an iteration of the loop body of the inner LU. Then, provided the versions of the BDP versions trsm and gemm kernels that are invoked from the inner LU are malleable (see subsection IV-A2), inside them the system will perform the actions that are necessary to integrate the thread team **RU**, which is now idle, into the corresponding (and subsequent) computation(s). Unfortunately, the updates in the loop body of the inner LU involve small-grained computations (A_{12} and A_{22} have at most $b_o - b_i$ columns, decreasing by b_i columns at each iteration), and little parallel performance can be expected from it especially because of partial pivoting.

In order to deal with this scenario, a different option is to force the inner LU to stop at the end of the current iteration, to then rapidly proceed to the next iteration of the outer LU. We refer to this strategy as the *early termination* (ET). In order to do this though, the transformations computed to factorize the current inner-panel must be propagated first to the remaining columns outside this panel, introducing a certain delay in this version of the ET strategy.

A third possibility is to rely on a left-looking (LL) version of the LU factorization for the inner LU, as discussed next. The blocked LL algorithm for the LU factorization differs from the blocked RL variant (see the algorithm in the right-hand side of Figure 3) in the operations performed inside the loop-body, which are replaced by

$$\textbf{LL1.} \quad A_{01} := \text{trilu}(A_{00})^{-1} A_{01}$$

$$\textbf{LL2.} \quad \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} := \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} - \begin{bmatrix} A_{10} \\ A_{20} \end{bmatrix} A_{01}$$

$$\textbf{LL3.} \quad \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} := \text{LU\_unb}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \right)$$

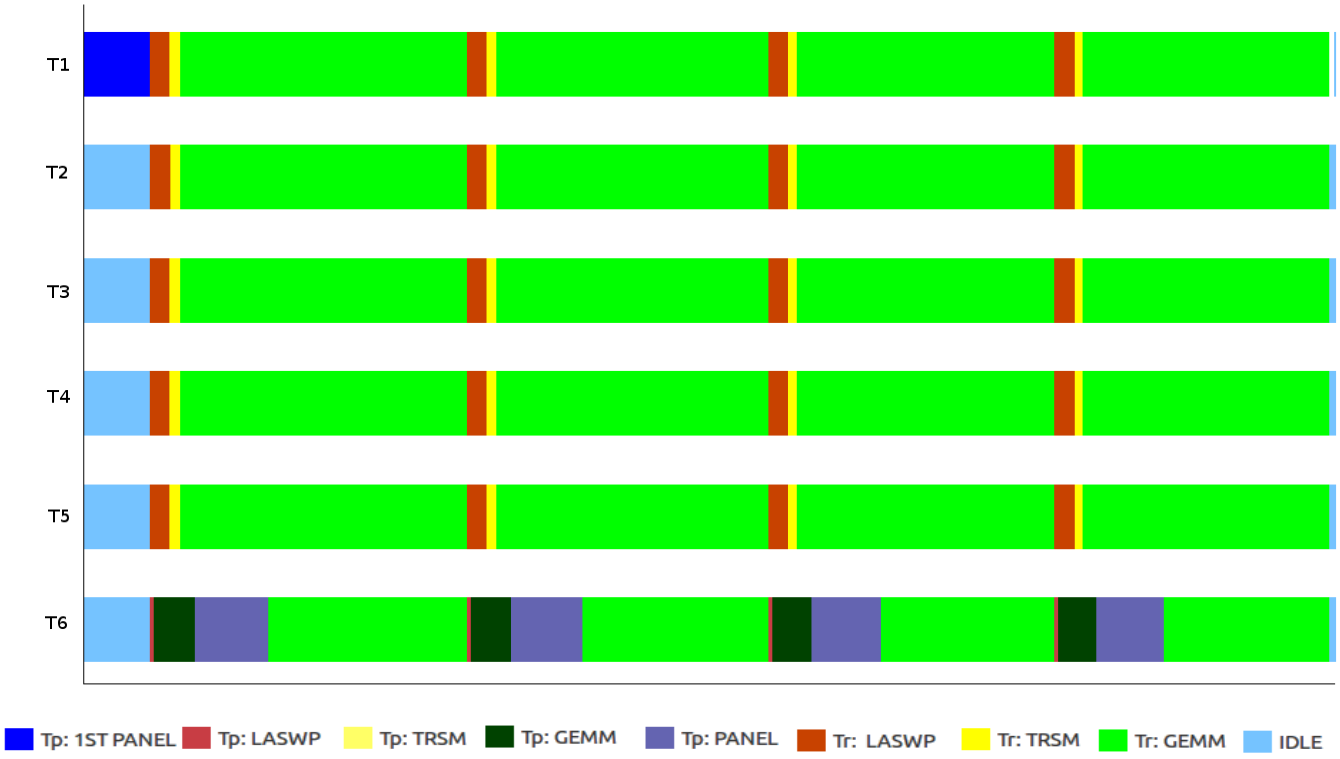

FIGURE 11. Execution trace of the first four iterations of the blocked RL LU factorization with partial pivoting, enhanced with look-ahead and *malleable BLIS*, using 6 threads, applied to a square matrix of order 10,000, with $b_o = 256$, $b_i = 32$.

Thus, at the end of a certain iteration, this variant has only updated the current column of the inner-panel and those to its left. In other words, no transformations are propagated beyond that point (i.e., to the right of the current column/inner-panel), and ET can be implemented in a straight-forward manner, with no delay compared with an inner LU factorization via the RL variant.

A definitive advantage of the LL variant compared with its RL counterpart is that the former implements a lazy algorithm, which delays the operations towards the end of the panel factorization, while the second corresponds to an eager algorithm that advances as much computations as possible to the initial iterations. Therefore, in case the panel factorization has to be stopped early, it is more likely that the LL variant has progressed in the factorization further.[5] The appealing consequence is that this enables the use of larger block sizes for the following updates in the LL variant.

From an implementation point of view, the synchronization between the two teams of threads is easy to handle. For example, at the beginning of each iteration of the outer LU, a boolean flag is set to indicate that the remainder update is incomplete. The thread team RU then changes this value as soon as this task is complete. In the mean time, the flag is queried by the thread team PF, at the end of every iteration of the inner LU, aborting its execution when a change is detected. With this operation mode, there is no need to protect the flag from race conditions. This solution also provides an adaptive (automatic) configuration of the block size as, if chosen too large, it will be adjusted for the current (and, possibly, subsequent) iterations by the early termination of the inner LU. The process is illustrated in Figure 13.

C. RELATION TO ADAPTIVE LOOK-AHEAD VIA A RUNTIME

Compared with our approach, which only applies look-ahead at one level, a TP execution that relies on a run-time for adaptive-depth look-ahead exposes a higher degree of parallelism from "future iterations", which can amortize the cost of the panel factorization over a considerably larger number of flops. This can be beneficial for architectures with a large number of cores, but can be partially compensated by increasing the number of threads dedicated to the panel factorization, combined with a careful fine-grain exploitation of the concurrency [27], in our approach. On the other hand, adaptive-depth look-ahead via a runtime suffers from re-packing and data movement overheads due to multiple calls to gemm. Moreover, it couples the algorithmic block size that fixes the granularity of the tasks to that of the suboperands in gemm. Finally, runtime-based solutions rarely exploit nested TP+BDP parallelism and, even if they do

[5]Consider the factorization of an $m \times n$ matrix that is stopped at iteration $k < n$. The LL algorithm will have performed $m^2k - m^3/3$ flops at that point while, for the RL algorithm, the flop count raises to that of the LL algorithm plus $2(n-k)(mk - k^2/2)$.

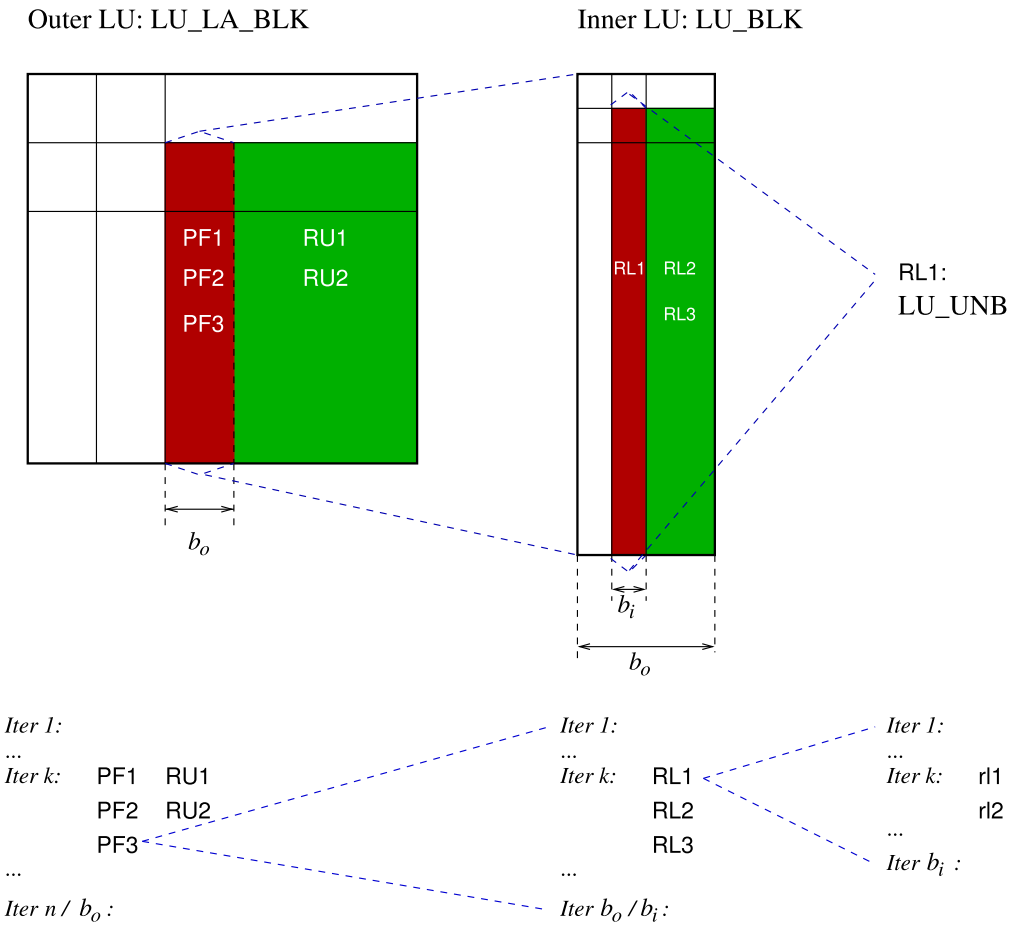

FIGURE 12. Outer vs inner LU and use of algorithmic block sizes.

so, taking advantage of a malleable thread-level BLAS from within them may be difficult.

V. EXPERIMENTAL EVALUATION

In this section we analyze in detail the performance behavior of several multi-threaded implementations of the algorithms for the LU factorization:

- LU: Blocked RL (Figure 3). This code only exploits BDP, via calls to the (non-malleable) multi-threaded BLIS (version 0.1.8). This routine corresponds to a direct translation into C of the legacy Fortran code for routine dgetrf in LAPACK (available at `http://www.netlib.org/lapack`), linked with our multi-threaded implementation of BLIS. The only difference is in the panel factorization which, for performance reasons, in our LU routine is performed via a call to the same routine, with a smaller block size, in order to cast most of the flops in terms of level-3 BLAS. In contrast, the legacy code in LAPACK realizes the panel factorization via routine dgetf2, thus casting most of the flops as level-2 BLAS. In consequence, the performance of the conventional implementation in LAPACK can be expected to stay below that of our LU routine. Note that LAPACK does not implement look-ahead.
- Variants enhanced with look-ahead (Figure 6). The following three implementations take advantage of nested TP+BDP, with 1 thread dedicated to the operations on the panel (team $\mathsf{T_{PF}}$) and $t-1$ to the remainder updates (team $\mathsf{T_{RU}}$).
  - LU_LA (subsection III-B): Blocked RL with look-ahead.
  - LU_MB (subsection IV-A2): Blocked RL with look-ahead and malleable BLIS.
  - LU_ET (subsection IV-B): Blocked RL with look-ahead, malleable BLIS, and early termination of the panel factorization.
- LU_OS: Blocked RL with adaptive look-ahead extracted via the OmpSs runtime (version 16.06). LU_OS decomposes the factorization into a large collection of tasks connected via data dependencies, and then exploits TP only, via calls to a sequential instance of BLIS. In more detail, the OmpSs parallel version divides the matrix into a collection of panels of fixed width b_o. All operations performed during an iteration of the algorithm on the same panel (row permutation, triangular system solve, matrix multiplication and, possibly, panel factorization) are then part of the same task. This implementation

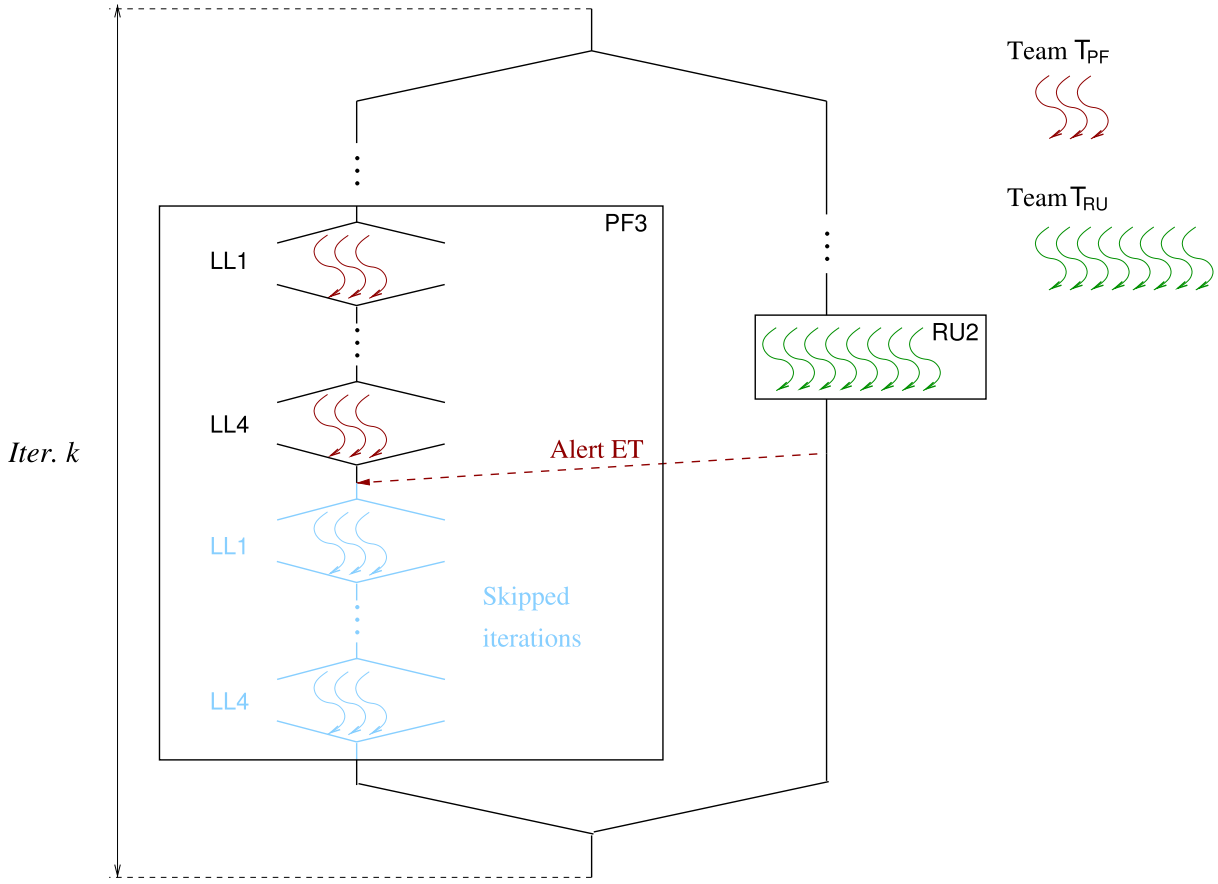

FIGURE 13. Exploitation of TP+BDP in the blocked RL LU parallelization with look-ahead and ET. The execution is performed by teams T_{PF} and T_{RU}, consisting of $t_{pf} = 3$ and $t_{ru} = 8$ threads, respectively. In this example, team T_{RU} completes the update RU2 while team T_{PF} is executing an iteration of the panel factorization PF3. T_{RU} then notifies of this event to T_{PF}, which then skips the remaining iterations of the loop that processes the panel.

includes priorities to advance the schedule of tasks involving panel factorizations.

All codes include standard partial pivoting and compute the same factorization. Also, all solutions perform the panel factorization via the blocked RL algorithm, except for LU_ET and LU_OS, which employ the blocked LL variant. The performance differences between the LL and RL variants, when applied solely to the panel factorization, were small. Nonetheless, for LU_ET, employing the LL variant improves the ET mechanism and unleashes a faster execution of the global factorization. For LU_OS we integrated the LL variant as well to favor a fair comparison between this implementation and our LU_ET. The block size is fixed to b_o during the complete iteration in all cases, except for LU_ET which initially employs b_o, but then adjusts this value during the factorization as part of the ET mechanism.

In the experiments, we considered the factorization of square matrices, with random entries uniformly distributed in (0, 1), and dimension $n = 500$ to 12,000 in steps of 500. The block size for the outer LU was tested for values $b_o = 32$ to 512 in steps of 32. The block size for the inner LU was evaluated for $b_i = 16$ and 32. We employed one thread per core (i.e., $t = 6$) in all executions.

A. OPTIMAL BLOCK SIZE

The performance of the blocked LU algorithms is strongly influenced by the outer block size b_o. As discussed in subsection III-A, this parameter should balance two criteria:

- Deliver high performance for the gemm kernel. Concretely, in the algorithms in Figures 3 and 6, a value of b_o that is too small turns A_{21} and A_{12}/A_{12}^R into narrow column and row panels respectively, transforming the matrix multiplication involving these blocks (RL3 in Figure 3 and RU2 in Figure 6) into a memory-bound kernel that will generally deliver low performance. In the following, we will refer to a gemm (1) with dimensions $m \approx n \gg k$, as a panel-panel multiplication (gepp) [8]. Note that, for the gepp arising in the LU factorizations, $k = b_o$.
- Reduce the amount of operations performed in the panel factorization (about $n^2 b_o/2$ flops, provided $n \gg b_o$), in order to avoid the negative impact of this intrinsically sequential stage.

Figure 14 sheds further light on the roles played by these two factors. The plot in the left-hand side reports the performance of gepp, in terms of GFLOPS (billions of FLOPS), showing that the implementation of this kernel in BLIS achieves

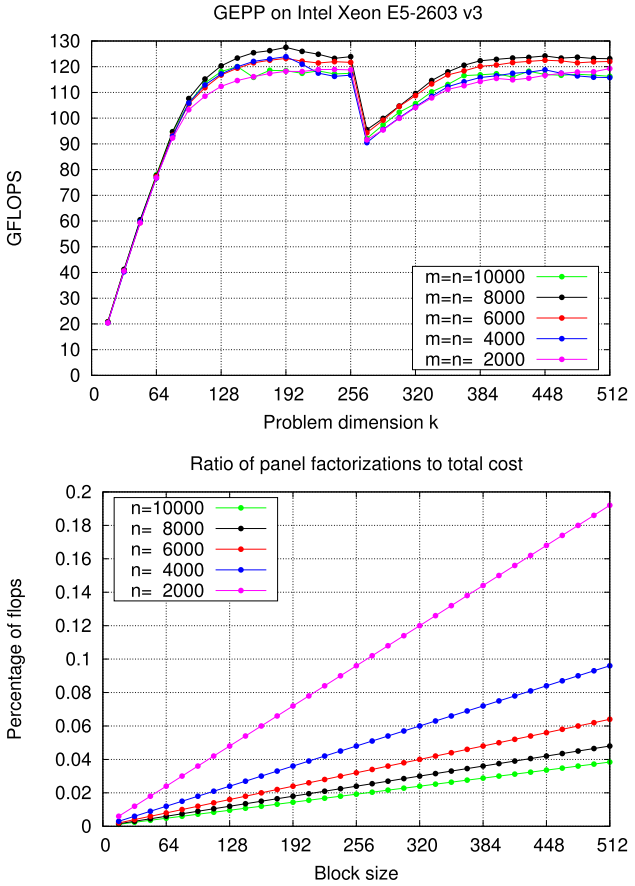

FIGURE 14. GFLOPS attained with gepp (left) and ratio of flops performed in the panel factorizations normalized to the total cost (right).

an asymptotic performance peak for $k(=b_o)$ around 144.[6] The right-hand side plot reports the ratio of flops performed in the panel factorizations with respect to those of the LU factorization.

The combined effect of these criteria seems to point in the direction of choosing the smallest b_o that attains the asymptotic GFLOPS rate for gepp. However, Figure 15 illustrates the experimental optimal block size b_o for the distinct LU factorization algorithms, exposing that this is not the case. We next discuss the behavior for LU, LU_LA and LU_MB, which show different trends. (LU_ET and LU_OS are analyzed latter.) In particular, LU benefits from the use of larger values of b_o than the other two codes for all problem dimensions. The reason is that a large block size operates on wide panels, which turns their factorization into a BLAS-3 operation with a mild degree of parallelism, and reduces the impact of this computation on the critical path of the factorization. LU_LA exhibits a similar behavior for large problems, but favors smaller block sizes for small to moderate problems. The reason is that, for LU_LA, it is important to balance the panel factorization ($\mathsf{T_{PF}}$) and remainder update ($\mathsf{T_{RU}}$) so that their execution approximately requires the same time.

[6]The performance drop observed for k slightly above 256 is due to the optimal value of k_c being equal to that number in this architecture.

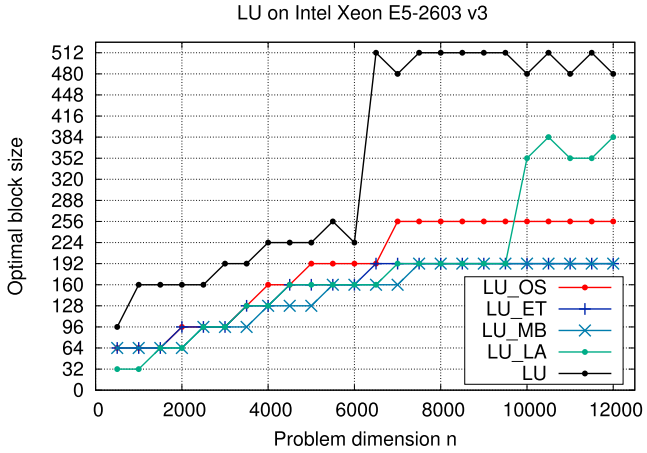

FIGURE 15. Optimal block size of the blocked RL algorithms for the LU factorization.

Compared with the previous two implementations, LU_MB promotes the use of small block sizes, up to $b_o = 192$, for the largest problems. (Interestingly, this corresponds to the optimal value of k for gepp.) One reason for this behavior is that, when the malleable version of BLIS is integrated into LU_MB, the practical costs of the two branches/tasks do not need to be balanced. Let us elaborate this case further, by considering the effect of reducing the block size, for example, from b_o to $b'_o = b_o/2$. For simplicity, in the following discussion we will use approximations for the block dimensions and their costs; furthermore, we will assume that $n \gg b_o$. The first and most straight-forward consequence of halving the block size is that the number of iterations is doubled. Inside each iteration with the original block size b_o, the loop body invokes, among others kernels, a gepp of dimensions $m \times (m - b_o) \times b_o$ (with m the number of rows in the trailing submatrix A_{22}^R), for a cost of $2m^2b_o$ flops; in parallel, the factorization involves a panel of dimension $m \times b_o$, for a cost of $mb_o^2 - b_o^3/3 \approx mb_o^2$ flops. When the block size is halved to b'_o, the same work is basically computed in two consecutive iterations. However, this reduces the amount of flops performed in terms of panel factorizations to about $2m(b'_o)^2 = mb_o^2/2$ while it has a minor impact on the number of flops that are cast as gepp (two of these products, at a cost of $2m^2b'_o = 2m^2b_o/2$ flops each). The conclusion is that, by reducing the block size, we decrease the time that the single thread spends in the panel factorization $\mathsf{T_{PF}}$, favoring its rapid merge with the thread team that performs the remainder update $\mathsf{T_{RU}}$. Thus, in case the execution time of the LU is dominated by $\mathsf{T_{RU}}$, adding one more thread to perform this task (in this scenario, in the critical path) as soon as possible will reduce the global execution time of the algorithm.

B. PERFORMANCE COMPARISON OF THE VARIANTS WITH STATIC LOOK-AHEAD

The previous analysis on the effect of the block size exposes that choosing the optimal block size is a difficult task. Either we need a model that can accurately predict the performance of each building block appearing in the LU factorization,

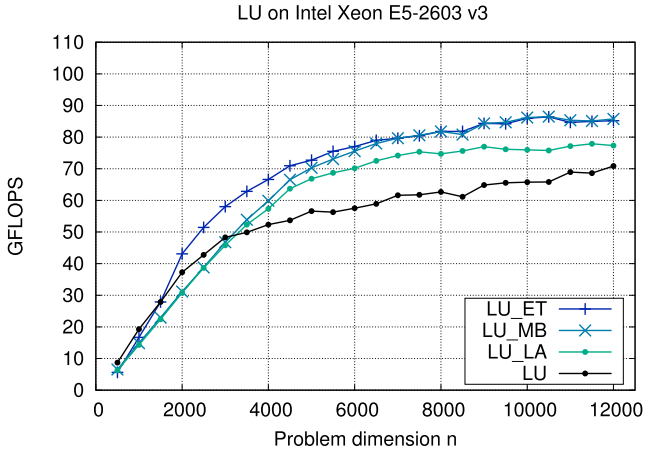

FIGURE 16. Performance comparison of the blocked RL algorithms for the LU factorization (except LU_OS) with a fixed block size $b_o = 256$.

or we perform an extensive experimental analysis to select the best value. The problem is even more complex if we consider that, in practice, an optimal selection would have to vary the block size as the factorization progresses. Concretely, for the factorization of a square matrix of order n via a blocked algorithm, the problem is decomposed into multiple subproblems that involve the factorization of matrices of orders $n - b_o$, $n - 2 \cdot b_o$, $n - 3 \cdot b_o$, etc. From Figure 15, it is clear that the optimal value of b_o will be different for several of these subproblems. In the end, the value that we show in Figure 15 for each problem has to be considered as a compromise that attains fair performance for a wide range of the subproblems appearing in that case.

Figure 16 reports the GFLOPS rates attained by the distinct implementations to compute the plain LU factorization and the variants equipped with static look-ahead (i.e., all except LU_OS), using $b_o = 256$ as a compromise value for all of them. Although this value is optimal for only a few cases, the purpose of this experiment is to show the improvements attained by gradually introducing the techniques enhancing look-ahead. The figure reveals some relevant trends:

- Except for the smallest problems, integrating the look-ahead techniques clearly improves the performance of the plain LU factorization implemented in LU.
- The version with malleable BLAS (LU_MB) improves the performance of the basic version of look-ahead (LU_LA) for the larger problems. This is a consequence of the cost of the panel factorization relative to that of the global factorization. Concretely, for fixed b_o, as the problem size grows, the global flop-cost varies cubically in n, as $2n^3/3$, while the flop-cost of the panel factorizations grows quadratically, with $n^2 b_o/2$. Thus, we can expect that, for large n, the remainder update $\mathsf{T_{RU}}$ becomes more expensive than the panel factorization $\mathsf{T_{PF}}$. This represents the actual scenario that was targeted by the variant with malleable BLIS.
- The version that combines the malleable BLAS with ET (LU_ET) delivers the same performance of LU_MB for large problems, but outperforms all other variants with static look-ahead for the smaller problems. Again, this could be expected by considering the relative cost of the panel factorization for small n.

C. PERFORMANCE COMPARISON WITH OMPSS

We conclude the experimental analysis by providing a comparison of the best variant with static look-ahead, LU_ET, with the implementation that extracts parallelism via the OmpSs runtime, LU_OS. In this last experiment we depart from the previous case, performing an extensive evaluation in order report the performance for the optimal block size for each problem dimension and algorithm. (See Figure 15 for the actual optimal values employed in the experiment.) For LU_OS, we select a value for b_o that is then fixed for the complete factorization. As this variant overlaps the execution of tasks from different iterations in time, it is difficult to vary the block size as the factorization progresses. For LU_ET, the selected value of b_o applies to the first panel factorization only. After that, the ET mechanism automatically adjusts this value during the iteration.

Figure 17 shows the results for this comparison in the lines labelled as "(b_opt)". LU_ET is very competitive, clearly outperforming the runtime-based solution for most problems and offering competitive performance for the largest three.

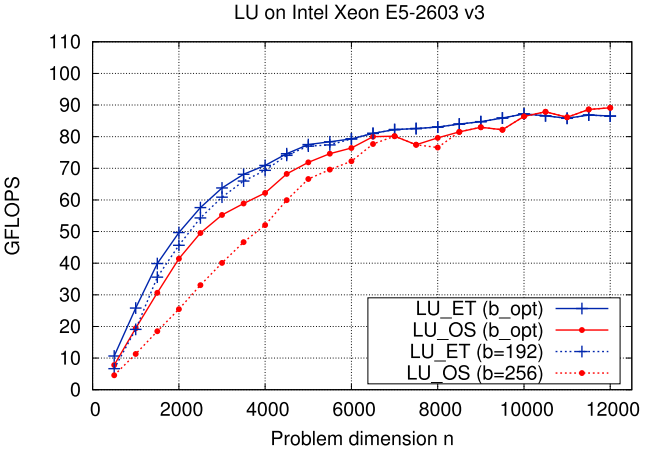

FIGURE 17. Performance comparison between the OmpSs implementation and the blocked RL algorithm for the LU factorization with look-ahead, malleable BLIS and ET. Two configurations are chosen for each algorithm: optimal block size for each problem size; and fixed block sizes $b_o = 192$ for LU_ET and $b_o = 256$ for LU_OS.

Manually tuning the block size to each problem dimension is in general impractical. For this reason, the figure also shows the performance curves when the block size is fixed to $b_o = 192$ for LU_ET and $b_o = 256$ for LU_OS. These values were selected because they offered high performance for a wide range of problem dimensions (especially, the largest ones; see Figure 15). Interestingly, the performance lines corresponding to this configuration, labelled with "(b = 192)"/"(b = 256)", show that choosing a suboptimal value for b_o has a minor impact on the performance of our solution LU_ET, because the ET mechanism adjusts this value on-the-fly (for the smaller problem sizes). Compared with this, the negative effect of a suboptimal selection on LU_OS is clearly more visible.

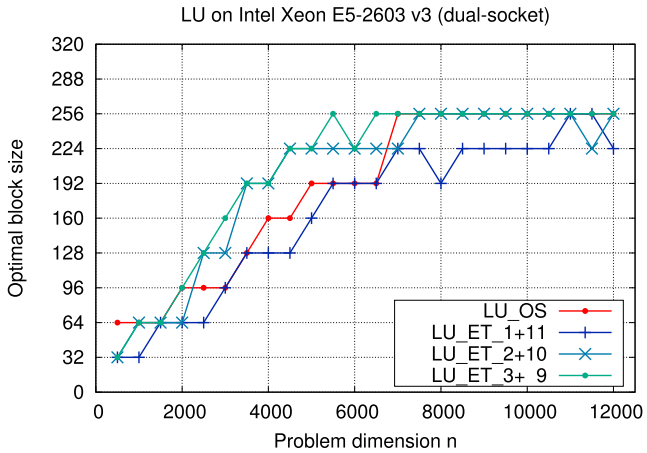

FIGURE 18. Optimal block size of the blocked ET and OmpSs algorithms for the LU factorization.

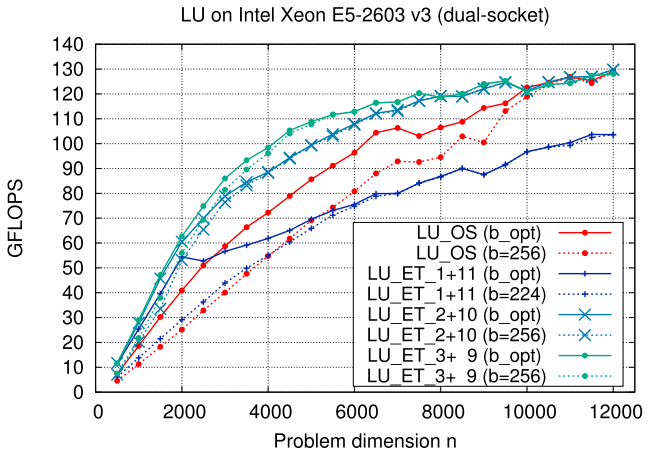

FIGURE 19. Performance comparison between the OmpSs implementation and the blocked RL algorithm for the LU factorization with look-ahead, malleable BLIS and ET. Two configurations are chosen for each algorithm: optimal block size for each problem size; and fixed block size $b_o = 256$ for all cases, except for LU_ET when mapping one thread to $\mathsf{T_{PF}}$ and eleven to $\mathsf{T_{RU}}$.

A comparison with other parallel runtime-based versions of the LU factorization with partial pivoting is possible, but we do not expect the results change the message of our paper. In particular, Intel MKL includes a highly-tuned routine for this factorization that relies in their own implementation of the BLAS and some type of look-ahead. Therefore, whether the advantages of one implementation over the other come simply from the use of a different version of the BLAS, or from the positive effects of our WS and ET mechanism, will be really difficult to infer. The PLASMA library [17] also provides a routine for the LU factorization with partial pivoting supported by a runtime that implements dynamic look-ahead. The techniques integrated in PLASMA's routine are not different from those in the OmpSs implementation evaluated in our paper. Therefore, when linked with BLIS, we do not expect a different behaviour between PLASMA's routine and LU_OS. Similar conclusions apply to `libflame` [14].

1) MULTI-SOCKET PERFORMANCE COMPARISON WITH OMPSS

We conclude the experimental analysis by including a multi-socket experiment that compares different configurations of the best variant with static look-ahead, LU_ET, with the runtime approach via OmpSs, LU_OS.

In this last experiment we consider the two sockets present in the platform, using 12 threads in our tests, and report, as in the previous section, the performance for the optimal block size for each problem dimension and algorithm. The optimal values employed in this case are displayed in Figure 18.

Figure 19 shows the results for this comparison in the lines labelled as "(b_opt)". LU_ET is very competitive, clearly outperforming the runtime-based solution for most problems and offering competitive performance for the largest five, except for the case that maps one thread in $\mathsf{T_{PF}}$ and the rest of resources in $\mathsf{T_{RU}}$.

As in the case where only one socket is employed, the performance curves are obtained for a fixed block size and the optimal block size for each problem dimension. The block size is fixed to $b_o = 256$ for all cases except for LU_ET when mapping one thread to $\mathsf{T_{PF}}$ and eleven to $\mathsf{T_{RU}}$. For LU_ET, according to Figure 18, the value that offers high performance for a wide range of problem dimensions is $b_o = 224$.

As in the previous study where only one socket was considered, the performance lines corresponding to the fixed block size configuration, labelled with "(b = 256)"/"(b = 224)", show how the ET mechanism is less affected by the use of a suboptimal block size value. Note that for the case where only one thread is in charge of $\mathsf{T_{PF}}$, the difference between the optimal block size and the fixed block size is larger than in the other cases. This behavior is due to the reduced number of threads in charge of $\mathsf{T_{PF}}$ which makes the first iteration of the factorization costly. Consequently, adjusting the block size for the next iterations is not enough to overcome the effects of the suboptimal initial block size election.

In addition, the impact of different thread mapping is shown in Figure 19. While in the previous section using one thread in the $\mathsf{T_{PF}}$ and the rest in the $\mathsf{T_{RU}}$ was enough, here we observe the benefits of adding more threads to $\mathsf{T_{PF}}$. Since more resources are available (2 socket vs. 1 sockets), increasing the number of threads in $\mathsf{T_{PF}}$ makes this task finish earlier and, consequently, all the threads can join faster to the execution of $\mathsf{T_{RU}}$. Interestingly, when all cores are in use, employing only one thread in $\mathsf{T_{PF}}$ harms performance, since the time spent in the execution of $\mathsf{T_{PF}}$ is increased (if compared to the other cases) and we are missing resources in $\mathsf{T_{RU}}$ for longer time.

Again, a comparison with other parallel versions of the LU factorization is possible but substantial changes in our results are not expected for the same reason stated in the previous section.

VI. CONCLUDING REMARKS AND FUTURE WORK

We have introduced WS and ET as two novel techniques to avoid workload imbalance during the execution of matrix factorizations, enhanced with look-ahead, for the solution

of linear systems. The WS mechanism especially benefits from the adoption of a malleable thread-level instance of BLIS, which allows the thread team in charge of the panel factorization, upon completion of this task, to be reallocated to the execution of the trailing update. The ET mechanism tackles the opposite situation, with a panel factorization that is costlier than the trailing update. In such scenario, the team that performed the update communicates to the second team that it should terminate the panel factorization, advancing the factorization process into the next iteration.

Our results on a platform equipped with Intel Xeon E5-2603 v3 (12 cores) show the performance benefits of our version enhanced with malleable BLIS and ET compared with a plain LU factorization as well as a version with look-ahead. The experiments also report competitive performance compared with an LU factorization that is parallelized by means of a sophisticated runtime, such as OmpSs, that introduces look-ahead of dynamic (variable) depth. Compared with the OmpSs solution, our approach offers higher performance for most problem dimensions, seamlessly tunes the algorithmic block size, and features a considerably smaller memory footprint as it does not require a sophisticated runtime support. We believe that, with the proper scaling of the problem dimension, these advantages carry over to architectures with larger number of cores.

To conclude, our paper does not intend to propose an alternative to runtime-based solutions. Instead, the message implicitly carried in our experiments aims to emphasize the benefits of malleable thread-level libraries, which we expect to be crucial in order to exploit the massive thread parallelism of future architectures. This work opens a plethora of interesting questions for future research. In particular, how to generalize the ideas to a multi-task scenario, what kind of interfaces may ease thread-level malleability, and what kind of support is necessary in the runtime for this purpose.

ACKNOWLEDGEMENTS

The authors would like to thank the other members of the FLAME team for their support.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

REFERENCES

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Soft.*, vol. 5, pp. 308–323, Sep. 1979.
[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, pp. 1–17, Mar. 1988.
[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, pp. 1–17, Mar. 1990.
[4] Intel. (2015). *Math Kernel Library*. [Online]. Available: https://software.intel.com/en-us/intel-mkl
[5] AMD. (2015). *AMD Core Math Library*. [Online]. Available: http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/
[6] (2015). *Engineering and Scientific Subroutine Library*. [Online]. Available: http://www-03.ibm.com/systems/power/software/essl/
[7] NVIDIA. (2016). *cublas*. [Online]. Available: https://developer.nvidia.com/cublas
[8] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, p. 12, May 2008.
[9] K. Goto and R. van de Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, p. 4, Jul. 2008.
[10] *OpenBLAS. An Optimized BLAS Library*. Accessed: Jan. 2019. [Online]. Available: http://www.openblas.net
[11] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. SC*, Nov. 1998, p. 38.
[12] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, p. 14, 2015.
[13] E. Anderson *et al.*, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, USA: SIAM, 1999.
[14] F. G. V. Zee. (2009). *Libflame: The Complete Reference*. [Online]. Available: www.lulu.com
[15] *OmpSs Project Home Page*. Accessed: Jan. 2019. [Online]. Available: http://pm.bsc.es/ompss
[16] *StarPU Project*. Accessed: Jan. 2019. [Online]. Available: http://runtime.bordeaux.inria.fr/StarPU/
[17] *PLASMA Project Home Page*. Accessed: Jan. 2019. [Online]. Available: http://icl.cs.utk.edu/plasma
[18] *FLAME Project Home Page*. Accessed: Jan. 2019. [Online]. Available: http://www.cs.utexas.edu/users/flame/
[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: The Johns Hopkins University Press, 1996.
[20] S. Catalán, R. Rodríguez-Sánchez, E. S. Quintana-Ortí, and J. R. Herrero, "Static versus dynamic task scheduling of the lu factorization on ARM big. LITTLE architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May/Jun. 2017, pp. 733–742.
[21] F. G. Van Zee *et al.*, "The BLIS framework: Experiments in portability," *ACM Trans. Math. Softw.*, vol. 42, p. 12, Jun. 2016.
[22] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2014, pp. 1049–1059.
[23] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, and E. S. Quintana-Ortí, "Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors," *Cluster Comput.*, vol. 19, no. 3, pp. 1037–1051, 2016.
[24] *Extrae User Guide Manual for Version 2.5.1*, Barcelona Supercomputing Center, Barcelona, Spain, 2016.
[25] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal linear algebra methods environment," *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 422–455, 2001.
[26] P. Strazdins, "A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization," Dept. Comput. Sci., The Austral. Nat. Univ., Canberra, ACT, Australia, Tech. Rep. TR-CS-98-07, 1998.
[27] A. M. Castaldo, R. C. Whaley, and S. Samuel, "Scaling LAPACK panel operations using parallel cache assignment," *ACM Trans. Math. Soft.*, vol. 39, p. 22, Jul. 2013.

SANDRA CATALÁN received the B.Sc. and M.Sc. degrees in intelligent systems and the Ph.D. degree in computer science from Universidad Jaume I, Castelló de la Plana, Spain, in 2012, 2013, and 2018, respectively. She joined the Exa2Green Project, in 2012, and the INTER-TWinE Project, in 2018. In 2018, she was a Postdoctoral Researcher, where she joined the Project Fujitsu-BSC: Porting and Optimization of Math Libraries, Barcelona Supercomputing Center (BSC). She has also been a Visiting Researcher with the Foundations of Cognitive Computing Group, IBM Research–Zurich, Switzerland, also with The University of Texas in Austin, USA, and also with the Parallel Programming Models Group, Barcelona Supercomputing Center, Barcelona, Spain. Her current research interests include moderate-scale clusters and low-power processors, and parallel algorithms for numerical linear algebra.

JOSÉ R. HERRERO has been teaching with the Barcelona School of Informatics (FIB), since 1994, where he has taught 15 different courses corresponding to several areas (computer architecture, operating systems, and parallel programming). He has carried out the duties of the Vice-Dean Head of academic studies and the Vice Dean for Institutional and International Relations with FIB. He has combined these management and teaching tasks with research in HPC, mainly in high performance scientific computing. This research work is made tangible in dozens of articles published in scientific magazines and international conferences. He has done research stays in several research centers (UCI, USA; IST, Portugal; NTUA, Greece; BSC, Spain; QUB, UK; ECM, France; and UJI, Spain). He regularly takes part in program committees and acts as an Associate Editor and a Reviewer for scientific publications. He is currently an Associate Professor with the Computer Architecture Department, Universitat Politècnica de Catalunya, BarcelonaTech.

ENRIQUE S. QUINTANA-ORTÍ received the bachelor's and Ph.D. degrees in computer sciences from the Universidad Politecnica de Valencia, Spain, in 1992 and 1996, respectively. He is currently a Professor in computer architecture with Universidad Jaume I, Castellón de la Plana, Spain. He has published more than 200 papers in international conferences and journals, and has contributed to software libraries like PLiC/SLICOT, MAGMA, FLARE, BLIS, and libflame for control theory and parallel linear algebra. Recently, he has participated/participates in EU projects on parallel programming, such as TEXT, INTERTWinE, and energy efficiency such as EXA2GREEN and OPRECOMP. His current research interests include parallel programming, linear algebra, energy consumption, transprecision computing and bioinformatics, and advanced architectures and hardware accelerators. He has also been a member of the programme committe for around 100 international conferences. In 2008, he was a recipient of an NVIDIA Professor Partnership Award for his contributions to the acceleration of dense linear algebra kernels on graphics processors, and was also a recipient of two awards from NASA for his contributions to fault-tolerant dense linear algebra libraries for space vehicles.

RAFAEL RODRÍGUEZ-SÁNCHEZ received the M.S. and Ph.D. degrees in computer science from the University of Castilla-La Mancha, Spain, in 2010 and 2013, respectively. From 2008 to 2013, he was with the Albacete Research Institute of Informatics, University of Castilla-La Mancha, Spain. In 2013, he joined the Department of Engineering and Computer Sciences, University Jaume I, Castellón de la Plana, Spain. In 2017, he joined the Computer Architecture and Automatics Department, Universidad Complutense de Madrid, Spain. He has also been a Visiting Researcher with the Multimedia Lab, Ghent University, Belgium, and at ARM Ltd., Cambridge, U.K. He has more than 40 publications in these areas in international refereed journals and conference proceedings. His research interests include video coding, parallel programming, heterogeneous computing, optimization and adaptation of numerical libraries, and power and energy consumption.

ROBERT VAN DE GEIJN received the Ph.D. degree in applied mathematics from the University of Maryland at College Park, College Park. He is currently a Professor of computer science and a Core Member of the Institute for Computational Engineering and Sciences, The University of Texas (UT) at Austin.

He heads the FLAME Project, a collaboration between UT Austin, Universidad Jaume I, Spain, RWTH Aachen University, Germany, and Carnegie Mellon University, USA. This project pursues foundational research in the field of linear algebra libraries and has led to the development of the libflame and BLIS libraries, modern, and high-performance dense linear algebra libraries. One of the benefits of these libraries lies with their impact on the teaching of numerical linear algebra, for which he received the UT President's Associates Teaching Excellence Award. His research interests include linear algebra, high-performance computing, parallel computing, and formal derivation of algorithms.