Check for updates

Received: 18 June 2021 | Revised: 2 February 2022 | Accepted: 9 March 2022

DOI: 10.1002/cpe.6999

SPECIAL ISSUE PAPER

WILEY

Fine-grain task-parallel algorithms for matrix factorizations and inversion on many-threaded CPUs

Sandra Catalán[1] | José R. Herrero[2] | Francisco D. Igual[1] | Enrique S. Quintana-Ortí[3] | Rafael Rodríguez-Sánchez[1]

[1]Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Madrid, Spain

[2]Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain

[3]Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain

Correspondence
Francisco D. Igual, Universidad Complutense de Madrid, Madrid, Spain.
figual@ucm.es and rafaelrs@ucm.es

Funding information
Comunidad de Madrid, Grant/Award Numbers: 2017-SGR-1414 o, S2018/TCS-4423; Ministerio de Ciencia e Innovación, Grant/Award Numbers: RTI2018-093684-B-I00, PID2019-107255GB, TIN2017; v PRICIT, Grant/Award Number: PR65/19-22445

Abstract

We extend a two-level task partitioning previously applied to the inversion of dense matrices via Gauss–Jordan elimination to the more challenging QR factorization as well as the initial orthogonal reduction to band form found in the singular value decomposition. Our new task-parallel algorithms leverage the tasking mechanism currently available in OpenMP to exploit "nested" task parallelism, with a first outer level that operates on matrix panels and a second inner level that processes the matrix either by μ-panels or by tiles, in order to expose a large number of independent tasks. We present a detailed performance analysis, including execution traces, which shows that the two-level refinement into fine grain tasks allows for an improved load balancing and delivers high performance on current general-purpose many-core processors (CPUs) from Intel and AMD.

KEYWORDS

CPUs, high performance, matrix factorizations, matrix inversion, OpenMP, task parallelism

1 | INTRODUCTION

In response to the design of general-purpose processors (or CPUs) with a moderate number of cores, a series of efforts have demonstrated the benefits of extracting task parallelism for dense linear algebra operations: PLASMA,[1,2] libFLAME,[3,4] StarPU,[5,6] and OmpSs.[7,8] Following this trend, processor architectures for high performance computing (HPC) have evolved over the past few years to integrate a very large number of cores so that, nowadays, CPUs (e.g., from Intel, AMD and ARM) with 16–64 are not uncommon in HPC servers. The challenge for dense linear algebra operations thus becomes how to extract sufficient task parallelism to feed this amount of cores while maintaining desirable properties, such as numerical stability, for the target methods.

In Reference 9, we introduced a family of advanced task-parallel algorithms for the inversion of dense matrices via Gauss–Jordan elimination (GJE)[10,11] that proposed a partitioning of the matrix operand into two levels of tasks, in order to extract sufficient task parallelism to feed a many-threaded CPU. Concretely, the workload is first partitioned by column blocks (or panels), to accommodate the standard partial pivoting scheme for numerical stability; and then, within the panels, either by thinner column blocks (μ-panels) or by row blocks (tiles). The experimental results for these schemes demonstrated the performance advantage of such fine-grain task-parallel decompositions of the algorithm for matrix inversion via GJE, parallelized with OpenMP, over a standard single-level partitioning on a 20-core Intel processor.

In this article, we extend the two-level task partitioning schemes to a pair of more challenging matrix algebra operations: the QR factorization and the (initial) orthogonal reduction to band form for the two-stage computation of the singular value decomposition (SVD).[12] Compared

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.
© 2022 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

with matrix inversion via GJE, the standard (right-looking) blocked algorithm for the QR factorization[12] presents a variation of the computational cost across the iteration space that turns more difficult (and crucial) balancing the distribution of the work among the computational resources. In addition, while the panel update for the GJE-based matrix inversion algorithm boils down to a simple matrix multiplication, which is easy to decompose into fine-grain tasks, the update for the QR factorization involves some data dependencies that reduce the degree of parallelism for this type of task. These observations for the QR factorization carry over to the band reduction algorithm, which requires a two-sided application of the orthogonal transforms that, a priori, imposes additional constraints on the degree of task parallelism for this particular algorithm.

Concretely, in this article, we make the following contributions:

- We introduce and evaluate advanced task-parallel algorithms for matrix inversion via GJE, the QR factorization, and the orthogonal reduction to band form in the two-stage computation of the SVD.
- We leverage the advanced tasking features in OpenMP to implement a two-level task partitioning, where a first level of tasks works on panels, and a second level of nested tasks operates either on μ-panels or on tiles. In all algorithms, the exploitation of look-ahead[13] is left in the hands of the runtime system.
- We conduct a complete performance analysis of the algorithms, using traces to expose the performance bottlenecks and the impact of the two-level task partitioning applied to different parts of the computation.
- We perform a complete experimental evaluation on two many-core processors with 20 and 64 cores, from Intel and AMD, against state-of-the-art libraries.

At this point, we note that there is a different, tile-based approach, embraced by libraries such as PLASMA and libFLAME, to extract additional fine-grain task parallelism for the QR factorization.[1,3] Concretely, this approach follows the ideas of the incremental QR factorization[14] to initially divide the matrix into "squarish" tiles which then induce a task-parallel workload. The advantage is that, proceeding in this manner, the "sequential" panel factorization, which stands in the critical path of the algorithm, is decomposed into finer grain tasks, and task parallelism is exposed earlier. However, while this scheme is possible for the QR factorization, because of the resilience of orthogonal transformations to rounding error, the same technique cannot be applied, for example, in the LU factorization or matrix inversion via GJE without abandoning the standard partial pivoting scheme.[15] Furthermore, the utilization of a single-level tile-based task-parallel algorithm requires specialized kernels that operate with small blocks within the tiles, which may reduce performance, especially in a massively data-parallel processor such as a GPU (graphics processing unit). Once more, similar comments carry over to the band reduction algorithm.

After introducing our notation in the next subsection, the rest of the article is structured as follows. In Section 2 we review task-parallel algorithms for matrix inversion via GJE. In Section 3 we introduce task-parallel algorithms on matrix factorizations computed via orthogonal transformations. Next, in Section 4 we present and analyze the resulting performance. Finally, we draw a few conclusions in Section 5.

1.1 | Notation

During the presentation of the algorithms, we will consider an $m \times n$ matrix A where, in our notation, $A(i_1 : i_2, j_1 : j_2)$ denotes the submatrix of A that spans the intersection between rows $i_1, i_1 + 1, \ldots, i_2$ and columns $j_1, j_1 + 1, \ldots, j_2$ of the matrix. Also, in our notation matrix indices start at zero.

When necessary we will partition A into $n_p = n/b_p$ panels, where b_p denotes the algorithmic panel size. Each panel may be then partitioned either by column blocks into $n_m = b_p/b_m$ thinner μ-panels, consisting of b_m columns each (the algorithmic μ-panel size); or by row blocks into $m_t = m/b_t$ tiles of b_t rows each (the algorithmic tile size). For simplicity, we will assume that m, n are respectively integer multiples of b_p, b_t and that b_p is an integer multiple of b_m. Finally, we will consider that m is also an integer multiple of b_p and define $m_p = m/b_p$.

2 | REVIEW OF TASK-PARALLEL MATRIX INVERSION VIA GJE

Compared with the conventional approach to compute the matrix inverse, based on the LU factorization, the procedure relying on GJE offers an alluring alternative with the same numerical stability but a more balanced distribution of the computational workload across the inversion process.[11] In this section, we first describe matrix inversion via GJE, to then re-visit the task-parallel algorithms for this operation introduced in Reference 9.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
#define A_p(i,j)  A(i, j*b_p : (j+1)*b_p - 1)
#define p_p(j)    p(j*b_p : (j+1)*b_p - 1)

void GJE_BSB( matrix A, vector p, int n_p, int b_p )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize panel k
    PF( A_p(:, k), p_p(k) );

    // Permute and update panels 0:k-1 w.r.t. panel k
    PP( p_p(k), A_p(:, 0:k-1) );
    PU( A_p(:, k), A_p(:, 0:k-1) );

    // Permute and update panels k+1:n_p-1 w.r.t. panel k
    PP( p_p(k), A_p(:, k+1:n_p-1) );
    PU( A_p(:, k), A_p(:, k+1:n_p-1) );
} }
```

FIGURE 1 BSB algorithm for matrix inversion via GJE. For clarity, the matrix and vector arguments to the kernels specify the region of memory that is "accessed" by the operation using C preprocessor macros

2.1 | Baseline GJE algorithm

The baseline blocked (BSB) algorithm in Figure 1 computes the inverse of a square nonsingular matrix A of order n, using the GJE procedure. The algorithm there is presented with a high-level notation that accommodates the following discussion in the article. The matrix is processed by panels of b_p columns, carrying out three types of operations at each iteration of the loop body:

1. "Factorize" panel k of the matrix, using routine PF (for panel factorization). This includes partial pivoting to ensure the practical numerical stability of the inversion procedure. Upon completion, this operation returns the row permutations stored in the output array p, overwriting the panel with the corresponding "factorization".
2. Permute the submatrices to the left and right of panel k, via two calls to routine PP (for panel permutation).
3. Update the same submatrices to the left and right of panel k via two calls to routine PU (for panel update).

In practice, the algorithmic panel size b_p is chosen to be moderately large, in order to hide the memory access costs with sufficient floating point operations (flops).

2.2 | Basic task-parallel GJE algorithm

The basic task-parallel (BTP) algorithm for matrix inversion via GJE mimics the task-parallel realizations of the LU and QR factorizations[1,3,7,8] to expose concurrency in terms of tasks with annotations that help the runtime to control the task execution schedule taking into account intertask data dependencies.

The GJE-based BTP algorithm for matrix inversion using OpenMP is formulated in Figure 2. Note that, at iteration k, the leading permutations/updates (for panels $j = 0, 1, \ldots, k-1$) plus all the trailing permutations/updates involving panels $j = k+2, k+3, \ldots, n/b_p - 1$ (i.e., all except $j = k+1$), can proceed in parallel with the factorization of panel $k+1$. This technique is often referred to as *look-ahead*[13] and allows to overlap the execution of the mostly sequential PF with the highly parallel PP and PU, breaking the strict dependency between the tasks in the same iteration. The specific task schedule is nondeterministic, as it is orchestrated by the OpenMP runtime during the execution of the algorithm and may result in an overlapped execution of future panel factorizations (not only that for panel $k+1$) with the permutations/updates that are present at iteration k of the routine.

2.3 | Advanced task-parallel GJE algorithm

In Reference 9, we proposed two types of advanced task-parallel (ATP) algorithms, which expose further task-parallelism by partitioning either the kernels for PF, PP, and PU into μ-panels or the kernels for PU into tiles.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
#define A_p(i,j)  A(i, j*b_p : (j+1)*b_p - 1)
#define p_p(j)    p(j*b_p : (j+1)*b_p - 1)

void GJE_BTP( matrix A, vector p, int n_p, int b_p )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize panel k
    #pragma omp task depend( inout: A_p(:,k), out: p_p(k) )
    PF( A_p(:,k), p_p(k) );

    // Permute and update panels 0:k-1 w.r.t. panel k
    for ( j = 0; j < k; j++ ) {
      #pragma omp task depend( in: p_p(k), inout: A_p(:,j) )
      PP( p_p(k), A_p(:,j) );
      #pragma omp task depend( in: A_p(:,k), inout: A_p(:,j) )
      PU( A_p(:,k), A_p(:,j) );
    }

    // Permute and update panels k+1:n_p-1 w.r.t. panel k
    for ( j = k+1; j < n_p; j++ ) {
      #pragma omp task depend( in: p_p(k), inout: A_p(:,j) )
      PP( p_p(k), A_p(:,j) );
      #pragma omp task depend( in: A_p(:,k), inout: A_p(:,j) )
      PU( A_p(:,k), A_p(:,j) );
} } }
```

FIGURE 2 `BTP` routine for matrix inversion via GJE with task parallelism extracted using OpenMP

```
#define A_p(i,j)       A(i, j*b_p : (j+1)*b_p - 1)
#define A_m(i,j_p,j_m) A(i, j_p*b_p + j_m*b_m : j_p*b_p + (j_m+1)*b_m - 1)
#define p_p(j)         p(j*b_p : (j+1)*b_p - 1)

void GJE_ATP_mpanels( matrix A, vector p, int n_p, int n_m, int b_p, int b_m )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize panel k
    #pragma omp task depend( inout: A_p(:,k), out: p_p(k) )
    GJE_BTP( A_p(:,k), p_p(k), n_m, b_m );

    // Permute and update panels 0:k-1 w.r.t. panel k
    // Omitted for brevity

    // Permute and update panels k+1:n_p-1 w.r.t. panel k
    for ( j_p = k+1; j_p < n_p; j_p++ ) {
      // Permute and update μ-panels in panel j_p w.r.t. panel k
      #pragma omp task depend( in: A_p(:,k), in: p_p(k), inout: A_p(:,j_p) )
      {
        for ( j_m = 0; j_m < n_m; j_m++ ) {
          #pragma omp task depend( in: p_p(k), inout: A_m(:,j_p,j_m) )
          PP( p_p(k), A_m(:,j_p,j_m) );
          #pragma omp task depend( in: A_p(:,k), inout: A_m(:,j_p,j_m) )
          PU( A_p(:,k), A_m(:,j_p,j_m) );
        }
        #pragma omp taskwait
} } } }
```

FIGURE 3 `ATP` routine for matrix inversion via GJE with task parallelism extracted using OpenMP and the panel updates divided *into μ-panels* to expose additional tasks

The two-level task-parallel algorithm that embeds a partitioning of the inversion kernels into μ-panels is illustrated in Figure 3. Concretely, the algorithm there firstly performs a division of the matrix into panels of b_p columns; followed by a second partitioning inside these into μ-panels of b_m columns. The `PF` kernel is parallelized by invoking the `BTP` routine with block size b_m, while the fine-grain permutations and updates are simply realized via the corresponding routines (`PP` and `PU`, respectively).

The alternative algorithm that performs an inner partitioning of the panels by tiles of b_t rows is given in Figure 4. In this case, the panel updates (`PU`) are further divided into finer-grain tasks while, due to the usage of partial pivoting, the application of the row permutations within a panel (`PP`)

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
#define A_p(i,j)  A(i, j*b_p : (j+1)*b_p - 1)
#define A_t(i,j)  A(i*b_t : (i+1)*b_t - 1, j*b_p : (j+1)*b_p - 1)
#define p_p(j)    p(j*b_p : (j+1)*b_p - 1)

void GJE_ATP_tiles( matrix A, vector p, int n_p, int n_m, int m_t, int b_p, int b_m )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize panel k
    #pragma omp task depend( inout: A_p(:,k), out: p_p(k) )
    GJE_BTP( A_p(:,k), p_p(k), n_m, b_m );

    // Permute and update panels 0:k-1 w.r.t. panel k
    // Omitted for brevity

    // Permute and update panels k+1:n_p-1 w.r.t. panel k
    for ( j = k+1; j < n_p; j++ ) {
      // Permute panel j w.r.t. panel k
      #pragma omp task depend( in: p_p(k), inout: A_p(:,j) )
      PP( p_p(k), A_p(:,j) );
      // Update tiles in panel j w.r.t. those in panel k
      #pragma omp task depend( in: A_p(:,k), inout: A_p(:,j) )
      {
        for ( i = 0; i < m_t; i++ ) {
          #pragma omp task depend( in: A_t(i,k), inout: A_t(i,j) )
          PU( A_t(i,k), A_t(i,j) );
        }
        #pragma omp taskwait
} } } }
```

FIGURE 4 `ATP` routine for matrix inversion via GJE with task parallelism extracted using OpenMP and the panel updates divided *into tiles* to expose additional tasks

remains as a single task. Also, in this particular realization, `PF` is partitioned into μ-panels of columns via the `BTP` algorithm with an inner block size b_m.

3 | MATRIX FACTORIZATIONS VIA ORTHOGONAL TRANSFORMS

The QR factorization decomposes a (full-rank) $m \times n$ matrix A into the product of an $m \times m$ orthogonal matrix and an $m \times n$ upper triangular factor, yielding a direct method to solve linear systems as well as (full-rank) linear least-squares problems.[12] The SVD can be viewed as a generalization of the QR factorization that decomposes the input matrix A into the product of two orthogonal matrices, containing the left and right singular vectors of A, and a diagonal matrix with its singular values.[12] In this section, we extend the two-level task-parallel schemes, proposed earlier for matrix inversion, to these types of orthogonal factorizations.

3.1 | Baseline QR algorithm

Figure 5 offers the `BSB` algorithm for the QR factorization, showing two relevant differences with respect to the GJE matrix inversion counterpart, from the point of view of exploiting task parallelism:

1. At each iteration of the loop body, the dimensions of the submatrices that are factorized and updated, respectively, using kernels `PF` and `PU`, diminishes by b rows. In contrast, in the `BSB` algorithm for GJE matrix inversion, the panels comprise the full matrix rows. This implies that attaining a balanced distribution of the workload among the processor cores is more difficult for the QR factorization. (Note that this is also the case for the LU factorization.)
2. The `PF` and `PU` kernels involve considerably more complex operations for the QR factorization than is the case for the GJE matrix inversion routine. For example, in the QR factorization, the `PF` kernel decomposes panel k of the matrix, denoted as $A_p(k : m_p - 1, k)$ and with dimension $m' \times b_p = (m - kb_p) \times b_p$, as follows:

$$A_p(k : m_p - 1, k) = Q_k R_k = (I_{m'} - V_k T_k V_k^T) R_k,$$

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

where $I_{m'}$ denotes the square identity matrix of order m' and R_k is an $m' \times b$ upper triangular matrix, while the $m' \times m'$ orthogonal matrix Q_k is stored implicitly in compact form[12] as the product of an $m' \times b$ unit lower triangular factor V_k and a square upper triangular matrix T_k of order b_p.

Therefore, the update of panels $j = k+1, k+2, \ldots n_p - 1$, each performed with an invocation to the PU kernel, requires the following computations:

$$(I_{m'} - V_k T_k V_k^T)\, A_p(k : m_p - 1, j) = A_p(k : m_p - 1, j) - V_k\, (T_k\, (V_k^T A_p(k : m_p - 1, j))),$$

which involves multiplications with the "triangular-shaped" factors V_k, T_k. Compared with this, the PU kernel for GJE matrix inversion only requires a single general (dense) matrix multiplication.

Note that, in contrast with the LAPACK convention, all our routines employing orthogonal transforms return the collection of computed triangular factors T as part of an $b_p \times n$ matrix, allowing their reuse, for example, during the application of the same orthogonal transforms to the independent terms for the solution of a linear system or a linear least-squares problem.

3.2 | Task-parallel QR algorithms

The basic and advanced versions of the task-parallel algorithms for the QR factorization are obtained by extending the corresponding ideas presented for GJE matrix inversion in the previous section, to the BSB algorithm in Figure 5. In the case of the ATP algorithm with an inner division of the panels into μ-panels, the pseudocode for the resulting routine is presented in Figure 6. Its simpler BTP counterpart can be roughly obtained by setting $b_m = b_p$ in the previous ATP algorithm, so that each panel consists of a single μ-panel only.

The variant of the ATP algorithm that performs an inner division into tiles is more elaborate to derive due to the complex computations that are involved in the application of the orthogonal transforms. In particular, consider the trailing submatrix $A_p(k : m_p - 1, k+1 : n_p - 1)$ to be updated with respect to the factorization of panel k. Taking as reference the BTP algorithm for the derivation of the ATP algorithm with a partitioning of panels into tiles, the update of the trailing submatrix is decomposed into panels of b_p columns each:

$$(I_{m'} - V_k T_k V_k^T)\, A_j = A_j - V_k\, (T_k\, (V_k^T A_j)) = A_j - V_k\, (T_k\, W_k^T),$$

where $A_j = A_p(k : m_p - 1, j), j = k+1, k+2, \ldots, n_p - 1$, denotes each one of the b_p-column panels to be updated. This computation thus involves the following sequence of operations:

$$\begin{aligned} &\text{PU1)}\ W_k^T = V_k^T A_j, \\ &\text{PU2)}\ \overline{W}_k^T = T_k W_k^T, \\ &\text{PU3)}\ A_j = A_j - V_k \overline{W}_k^T, \end{aligned}$$

and the identification of additional task parallelism within these three operations can be done by considering a row partitioning of both A_j and V_k into $s = (m - k * b_p)/b_t$ tiles, consisting of b_t rows each:

```
1  #define A_p(i,j)  A(i*b_p : (i+1)*b_p - 1, j*b_p : (j+1)*b_p - 1)
2  #define T_p(j)    T(j*b_p : (j+1)*b_p - 1)
3
4  void QR_BSB( matrix A, matrix T, int m_p, int n_p, int b_p )
5  {
6    for ( k = 0; k < n_p; k++ ) {
7      // Factorize panel k
8      PF( A_p(k:m_p-1, k), T_p(k) );
9
10     // Update panels k+1:n/b_p-1 w.r.t. panel k
11     PU( A_p(k:m_p-1, k), T_p(k), A_p(k:m_p-1, k+1:n_p-1) );
12 } }
```

FIGURE 5 BSB algorithm for the QR factorization

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
#define A_p(i,j)        A(i*b_p : (i+1)*b_p - 1, j*b_p : (j+1)*b_p - 1)
#define A_m(i,jp,jm)    A(i*b_p : (i+1)*b_p - 1, jp*b_p + jm*b_m : jp*b_p + (jm+1)*b_m - 1)
#define T_p(j)          T(j*b_p : (j+1)*b_p - 1)

void QR_ATP_mpanels( matrix A, matrix T, int m_p, int n_p, int n_m, int b_p, int b_m )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize panel k
    #pragma omp task depend( inout: A_p(k:m_p-1, k), out: T_p(k) )
    QR_BTP( A_p(k:m_p-1, k), T_p(k), n_m, b_m );

    // Update panels k+1:-1 w.r.t. panel k
    for ( jp = k+1; jp < n_p; jp++ ) {
      // Update μ-panels jp w.r.t. panel k
      #pragma omp task depend( in:    A_p(k:m_p-1, k), in: T_p(k), \
                               inout: A_m(k:m_p-1, jp, jm) )
      {
        for ( jm = 0; jm < n_m; jm++ ) {
          #pragma omp task depend( in:    A_p(k:m_p-1, k), in: T_p(k), \
                                   inout: A_m(k:m_p-1, jp, jm) )
          PU( A_p(k:m_p-1, k), T_p(k), A_m(k:m_p-1, jp, jm) );
        }
        #pragma omp taskwait
} } } }
```

FIGURE 6 `ATP` routine for the QR factorization with task parallelism extracted using OpenMP and the panel updates divided into μ-panels to expose additional tasks

$$A_j = \begin{bmatrix} A_t(0,j) \\ A_t(1,j) \\ \vdots \\ A_t(s-1,j) \end{bmatrix}, \quad V_k = \begin{bmatrix} V(0) \\ V(1) \\ \vdots \\ V(s-1) \end{bmatrix},$$

Thus, in the first operation we need to perform the block reduction:

$$\text{PU1)}\ W_k^T = V_k^T A_j = V(0)^T A_t(0,j) + V(1)^T A_t(1,j) + \cdots + V(s-1)^T A_t(s-1,j),$$

which corresponds to the fine-grain tasks of the `ATP` algorithm with a partitioning of the panels into tiles. Note that this involves a *task-parallel reduction*. In the second operation, we find only one task:

$$\text{PU2)}\ \overline{W}_k^T = T_k W_k^T;$$

and in the third operation, we have

$$\begin{aligned} \text{PU3)}\ A_j = A_j - V_k \overline{W}_k^T \equiv \\ \begin{bmatrix} A_t(0,j) \\ A_t(1,j) \\ \vdots \\ A_t(s-1,j) \end{bmatrix} &= \begin{bmatrix} A_t(0,j) \\ A_t(1,j) \\ \vdots \\ A_t(s-1,j) \end{bmatrix} - \begin{bmatrix} V(0) \\ V(1) \\ \vdots \\ V(s-1) \end{bmatrix} \overline{W}_k^T \\ &= \begin{bmatrix} A_t(0,j) - V(0)\overline{W}_k^T \\ A_t(1,j) - V(1)\overline{W}_k^T \\ \vdots \\ A(s-1,j) - V(s-1)\overline{W}_k^T \end{bmatrix}, \end{aligned}$$

which exposes the tasks for this particular computation in the `ATP` algorithm for the QR factorization with a division of the panels into tiles. In comparison with the first operation, the last one boils down to a collection of independent tasks.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
#define  A_p(i,j)   A(i*b_p : (i+1)*b_p - 1, j*b_p : (j+1)*b_p - 1)
#define  T_p^L(j)   T^L(j*b_p : (j+1)*b_p - 1)
#define  T_p^R(i)   T^R(i*b_p : (i+1)*b_p - 1)

void BRD_BSB( matrix A, matrix T^L, matrix T^R, int m_p, int n_p, int b_p )
{
  for ( k = 0; k < n_p; k++ ) {
    // Factorize column panel k
    PF( A_p(k+1 : m_p - 1, k), T_p^L(k) );

    // Update column panels k+1 : n_p - 1 w.r.t. column panel k
    PU( A_p(k+1 : m_p - 1, k), T_p^L(k), A_p(k+1 : m_p - 1, k+1 : n_p - 1) );

    // Factorize row panel k
    PF( A_p(k, k+1 : n_p - 1)^T, T_p^R(k)^T );

    // Update row panels k+1 : m_p - 1 w.r.t. row panel k
    PU( A_p(k, k+1 : n_p - 1)^T, T_p^R(k)^T, A_p(k+1 : m_p - 1, k+1 : n_p - 1)^T );
} }
```

FIGURE 7 BSB algorithm for the initial reduction to band form in the SVD

3.3 | Orthogonal reduction to band form

For performance reasons, the computation of the SVD of a "highly rectangular" input matrix A commences with the reduction of A to triangular form, via the QR factorization or one of its variants (QL, RQ, and LQ). Alternatively, in the case of a "squarish" matrix A, it is more efficient to perform an initial (orthogonal) reduction to some sort of band form. In both cases, this first stage comprises a considerable part of the computational cost of obtaining the SVD and, therefore, it is the key to attaining high performance for the complete process.

In Reference 16, we proposed to specialize the initial reduction in order to obtain a band matrix with upper/lower bandwidth w. The BSB algorithm is presented in Figure 7, assuming for simplicity that $b_p = w$. Note that an actual implementation does not transpose any of the blocks involved in the second calls to PF and PU inside the loop body. Instead, such realization computes an LQ factorization of the "current" panel $A_p(k, k+1 : n_p - 1)$, and then applies the corresponding orthogonal transforms to the appropriate trailing submatrix. This formulation of the BSB algorithm easily accommodates the advanced task parallelizations already described for the QR factorization. To avoid duplication we do not include the resulting pseudocodes here.

4 | EXPERIMENTAL RESULTS

In the following, we provide experimental evidence of the potential performance bottlenecks that emerge when mapping the basic task-parallel implementations (BTP) on many-core architectures, and how the fine-grained schemes enabled by the two-level ATP algorithms help to exploit the hardware core concurrency more efficiently, by exposing increasing volumes of tasks.

4.1 | Experimental setup

The experiments in this article were performed using double precision (DP) arithmetic on two platforms:

SKYLAKE: A server equipped with a 20-core Intel Xeon Gold 6138 processor (Skylake microarchitecture) with a nominal frequency of 2.0 GHz. Each core features two AVX-512 FMA (512-bit wide) units, yielding a theoretical peak performance of 51.4 DP GFLOPS (billions of flops per second) per core and a total of 1028 DP GFLOPS for the complete socket. The practical performance peak observed for the realization of in Intel MKL is 950 GFLOPS for the complete socket (when the frequency of the socket cores is set to 1.7 GHz). This processor includes a 32-Kbyte L1 data cache per core, a 1-Mbyte L2 per core, and a 1.375-Mbyte L3 cache per core. The server also includes 96 Gbytes of DDR4 RAM memory. Intel compilers, OpenMP runtime and MKL version 18 were employed as the underlying compiling/execution infrastructure and BLAS/LAPACK library, respectively. PLASMA version 19.8.1 was used in our evaluation.

ROME: A server equipped with a 64-core AMD EPYC 7742 processor (Zen2 microarchitecture) running at 2.25 GHz. Each core features two AVX-2 FMA (256-bit wide) units, yielding a theoretical peak performance of 41.6 DP GFLOPS per core, and a total of 2714 DP GFLOPS

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

for the complete socket. The practical performance peak observed for the realization of in AMD AOCL is around 1800 GFLOPS for the complete socket. The cores are grouped into 16 CCX (Core Complex), each one comprising 4 cores. This processor includes a 32-Kbyte L1 data cache per core, a 512-Kbyte L2 per core, and a 16-Mbyte L3 cache shared across cores in the same CCX. The server also includes 512 Gbytes of DDR4 RAM memory. GNU compilers and OpenMP version 8.3.0 were employed as the underlying compilation/execution infrastructure, and AMD AOCL version 3.0 was employed as the underlying BLAS/LAPACK library. PLASMA version 19.8.1 was used in our evaluation.

In order to avoid the performance distortions caused by the utilization of the power modes (and associated processor core frequencies) automatically applied by the processor under heavy-load circumstances, the operating frequencies were set to 1.7 GHz and 2.25 GHz in SKYLAKE and ROME, respectively, for all cores. These thresholds were experimentally established as *safe* frequency, not being affected by this reduction. SMT capabilities were disabled, one single thread was mapped per physical core, and thread migration was prevented via the appropriate Linux configuration commands.

Unless otherwise stated, the algorithmic panel size b_p was selected individually for each algorithm, matrix dimension, and number of cores in order to optimize performance. All variants realize the `PF` kernel using a blocked procedure, with the inner block size b_{in} (which corresponds to the μ-panel width or the tile height, depending on the algorithm) selected via extensive experimentation with values ranging between eight and b_p. All experiments are repeated to span up to a minimum pre-established time, in order to avoid time variations due to noise, while keeping experimentation time under an affordable limit for large-scale problems. No significant variations were observed in any case.

4.2 | Matrix inversion

The trace* in Figure 8 displays the parallel behavior of the `BTP` algorithm, when executed using all 20 cores of the SKYLAKE platform for a particular problem instance, showing two clear performance bottlenecks:

1. The execution of each `PF` is rather costly compared with that of the parallel (i.e., multitask) collection of `PU`.
2. For `PU`, there is a lack of sufficient tasks, which becomes visible as an unbalanced workload distribution in the last stages of each update, showing a significant number of idle cores.

The obvious solution to the high cost of the execution of the `PF` kernel as a single task consists in dividing this operation into multiple tasks, exposing additional task parallelism which can be leveraged by the system cores to accelerate the execution of this type of kernel. This can be achieved by realizing the `PF` kernel using, for example, the `BTP` routine with an inner μ-panel size b_m, which partitions this kernel into μ-panels. For the second bottleneck, we can refine the panels into either μ-panels or tiles, as described in Section 2.3. The effect of these alternative task-parallel schemes in the behavior of the inversion algorithm is illustrated via a couple of traces in Figure 9, demonstrating a considerable reduction of the idle periods.

We complete the analysis of the matrix inversion with a global comparison of the following four implementations:

- A standard matrix inversion algorithm based on the LU factorization using the appropriate calls to the multithreaded (`MT`) LAPACK routines in Intel `MT` MKL for SKYLAKE and AMD AOCL for ROME (`dgetrf` + `dgetri`).
- The `BSB` GJE routine in Section 1, with parallelism extracted only from the `MT` BLAS in Intel MKL/AMD AOCL.
- The `TP` routines from the PLASMA library to compute matrix inversion via the LU factorization (analogous to `dgetrf` + `dgetri`).
- The `ATP` GJE routine that parallelizes the `PF` kernel by μ-panels and the `PU` kernel by tiles.

This evaluation involves two different types of algorithms for matrix inversion: one based on the LU factorization and our alternative based on GJE. We note that the two types of algorithms perform exactly the same arithmetic operations, though in a different order and, therefore,

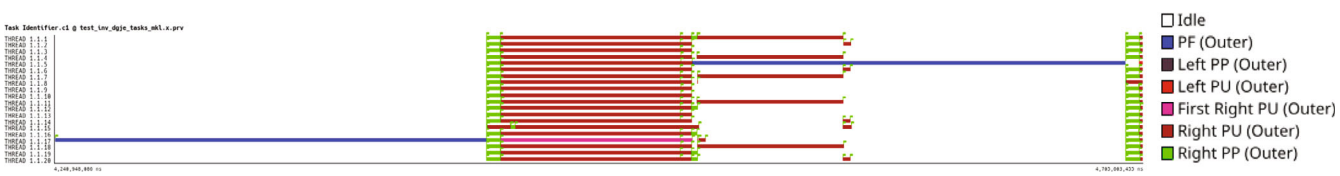
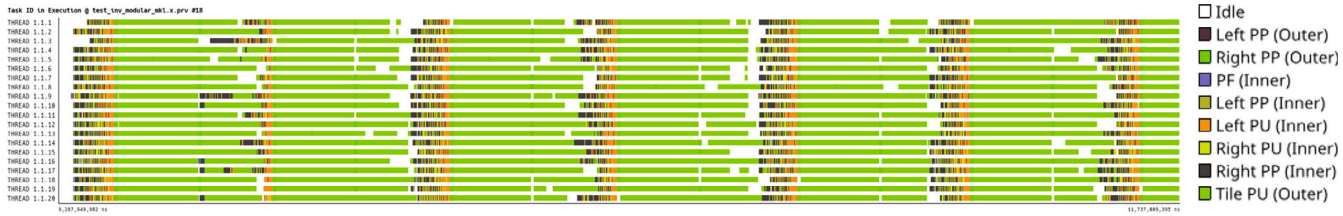
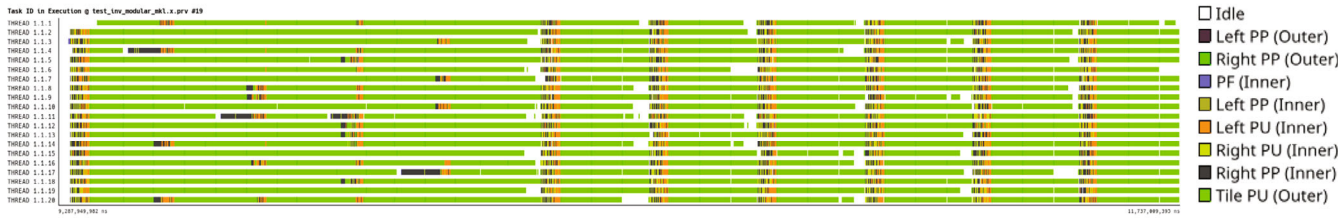


FIGURE 8 Trace of the first two iterations of the execution of the `BTP` GJE algorithm, with $n = 10{,}000$, $b_p = 384$, and 20 cores on SKYLAKE

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Partitionings of PF into μ-panels and PU into tiles, with $b_m = 12, b_t = 4,000$.



(B) Partitionings of both PF and PU into μ-panels, with $b_m = 12$.

FIGURE 9 Traces of the first 0.5 s of the execution of the ATP GJE algorithms, with $n = 16{,}000$, $b_p = 480$, and 20 cores on SKYLAKE

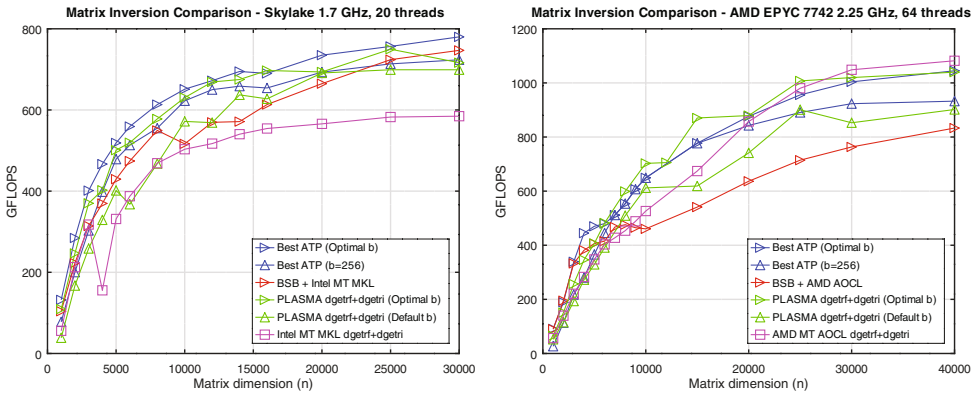


FIGURE 10 Performance of the algorithms for matrix inversion on SKYLAKE and ROME using 20 and 64 cores (left and right, respectively)

present the same computational cost. Furthermore, they integrate the same pivoting strategy, exhibiting the same numerical stability and producing the same numerical results (within rounding error). This evaluation is performed employing the optimal block size(s) for each algorithm, matrix dimension, and number of cores. For the last two inversion procedures (that in PLASMA and our ATP routine), we also include a configuration that employs a fixed value for the outer block size (concretely, set to $b_p = 256$) in order to assess the sensitivity of performance to this parameter.

PLASMA employs the conventional, LAPACK-approach to invert a matrix consisting of three steps: (1) compute the LU factorization; (2) invert one of the triangular factors; and (3) finally solve a linear system. The amount of task parallelism exposed by PLASMA and our GJE-based method is similar as they both proceed by column blocks. However, PLASMA may suffer from working with the triangular factors, which may lead to workload imbalance.

The results in Figure 10 show the benefits of the two-level task partitioning of the workload integrated in our ATP routine in SKYLAKE. The line labeled as *"Best ATP"* corresponds to the best performing ATP implementation on each machine, which corresponds to a division in μ-panels at two levels in both cases. In all cases, the ATP routine consistently delivers a much higher GFLOPS rate compared with the Intel MKL counterpart, and a considerably larger one when compared with the BSB routine for those cases where the ratio between problem size and number of cores is small. In comparison with the PLASMA alternative, the new TP routine is a clear winner, both when selecting the best block size and fixing a default value (again, set to $b = 256$). For ROME, the benefits of the two-level task partitioning are relevant compared with the BSB implementation. In this machine, for midsize matrices, our implementation also outperforms that of AMD AOCL. In all cases, a comparison with PLASMA (using optimal block sizes) reveals similar performance results for all matrix sizes.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.3 | QR factorization

Figure 11 offers a graphical representation, using execution traces, of the potential benefits of the ATP schemes applied to the task-based algorithm for the QR factorization. For brevity, this detailed qualitative study is carried out on ROME only (using 64 cores). The goal is to illustrate the differences in terms of caveats of the BTP/ATP implementations in computer platforms with a high number of cores, and the differences in behavior when compared with the qualitative experimental analysis presented earlier for matrix inversion.

Consider first the trace for the BTP realization of the QR factorization in Figure 11A. In this case, the selected algorithmic panel size b_p dictates both the granularity of the PF and PU kernels. Similarly to the BTP realization for GJE matrix inversion, two main problems arise: (1) the panel factorization (marked in dark blue in the trace) is a major bottleneck in the overall factorization that stands in the critical path of the execution; and (2) punctual workload imbalance arises also for the last stage of the panel updates, mainly due to the large granularity in this case. Tackling both problems is the goal of the ATP realization, with the effects analyzed next.

4.3.1 | Increasing parallelism in PF

Figure 11B captures the effect of the strategy to increase task parallelism by partitioning PF into μ-panels, showing a significant reduction of the relative time devoted to PF compared with the previous BTP implementation. Note, however, that the high number of cores in ROME yields an additional problem: the algorithmic μ-panel size needs to be small enough to expose enough μ-tasks to maintain a high occupancy of all the available cores. This block size reduction is negative for many BLAS and LAPACK implementations as, from our experiments, is the case for AMD AOCL, penalizing the individual performance of the μ-tasks. Our experimental results reveal that the optimal μ-panel size is not small enough to unleash enough task parallelism and thus eliminate work imbalance, as can be observed in the trace.

4.3.2 | Increasing parallelism in PU via μ-panels

A strategy to reduce workload imbalance in the last stages of the PU kernels consists in dividing this type of kernels into narrower μ-panels, effectively decoupling the block size used for PF and each micropanel of PU. Figure 11C shows the impact of this strategy on performance. Compared with Figure 11B, the time devoted to PU per iteration is reduced, and so its work imbalance; Figure 11D illustrates a strategy with a simultaneous partitioning of both PF and PU into μ-panels.

4.3.3 | Additional task parallelism in PU via tiles

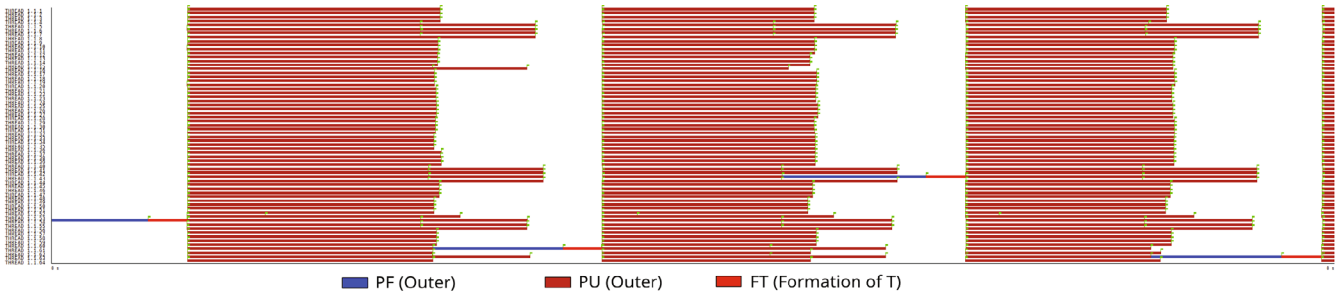
A final ATP implementation dividing PU into tiles is explored in Figure 11E. Even though the tile size in this case is suboptimal (selected to ease the comparison with previous traces), the amount of task parallelism exposed for the PU stage is dramatically higher than that unveiled with previous ATP strategies, yielding even better work balance than in previous cases for the PU stage. The results combining tiling for PU with a division of PF into μ-tiles have been omitted for brevity, even though their combined benefits in terms of performance should be straight forward to derive.

4.3.4 | General performance overview for the QR factorization

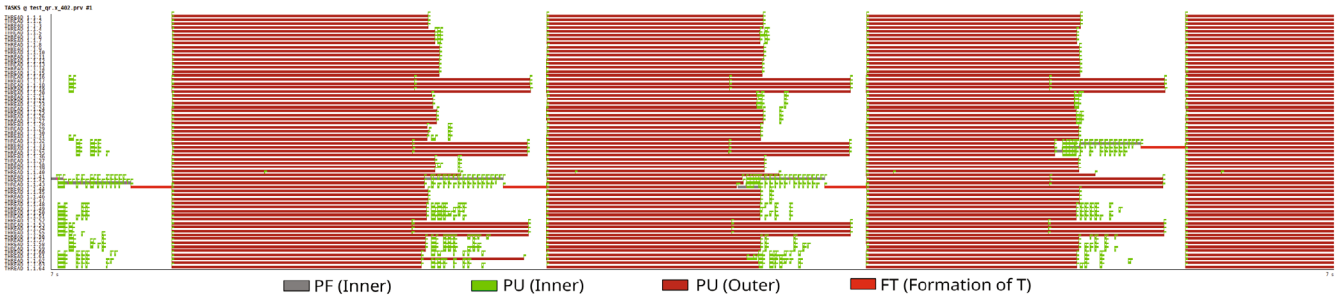
Figure 12 compares the performance attained by the different ATP implementations on both SKYLAKE (20 cores) and ROME (64 cores) for square matrices of order n. Note that, in the former platform the performance benefits of all ATP implementations are similar, while the much larger number of cores in ROME makes it necessary to expose an additional task parallelism. In this case, the partitioning of PU into tiles yields a significant performance improvement compared with that attained for partitioning schemes into μ-panels.

Finally, Figure 13 offers a general comparison of the advantages of ATP implementations compared with state-of-the-art multithreaded libraries (Intel MKL and AMD AOCL for SKYLAKE and ROME, respectively), task-parallel implementations (PLASMA linked with the sequential versions of the aforementioned proprietary libraries), and the baseline blocked implementations. The line labeled as *"best ATP"* corresponds to the best performing ATP implementation on each machine: division in μ-panels at both levels on SKYLAKE, and division of PU into tiles for ROME. In all cases, the parallelization opportunities offered by the ATP algorithms yield higher performance rates than those offered by the BSB counterparts.

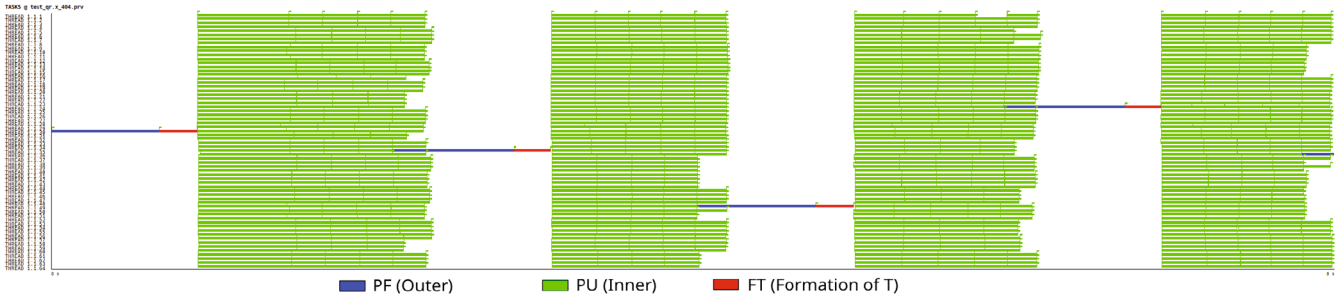
15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



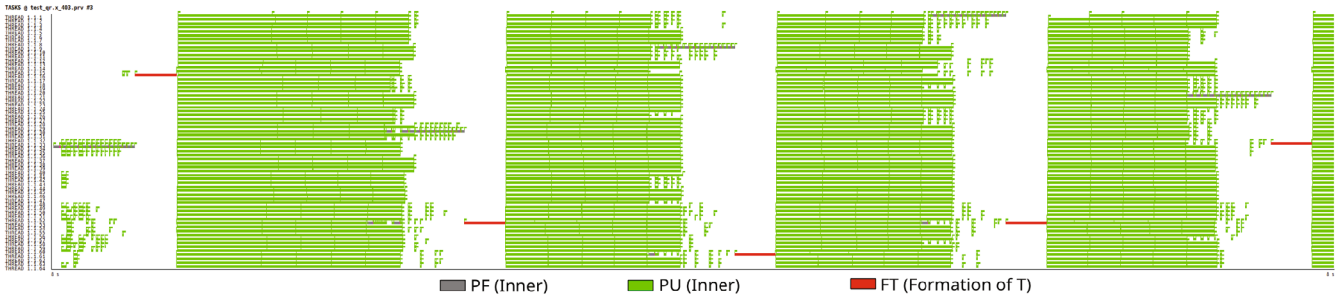
(A) BTP QR algorithm.



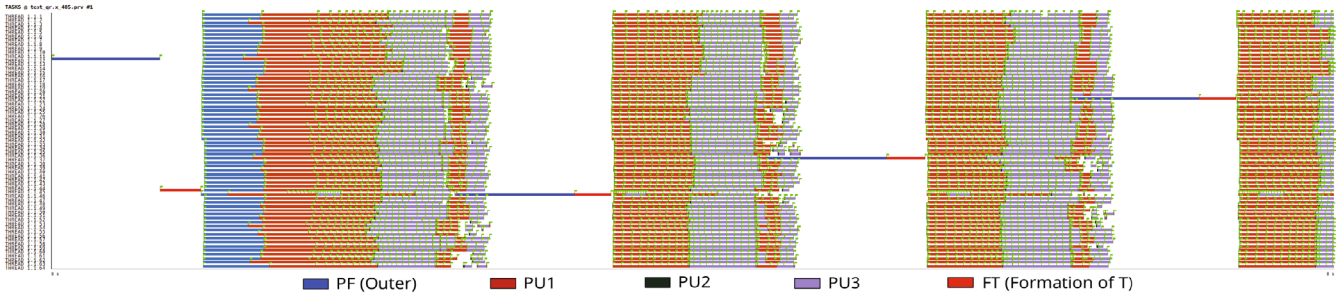
(B) ATP QR algorithm with partitioning of PF into μ-panels.



(C) ATP QR algorithm with partitioning of PU into μ-panels.



(D) ATP QR algorithm with partitionings of both PU and PF into μ-panels.



(E) ATP QR algorithm with partitioning of PU into tiles.

FIGURE 11 Traces of the first 0.3 s of execution of the task-based QR algorithms with $m = n = 10{,}000$, $b_p = 128$, and 64 threads on ROME

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

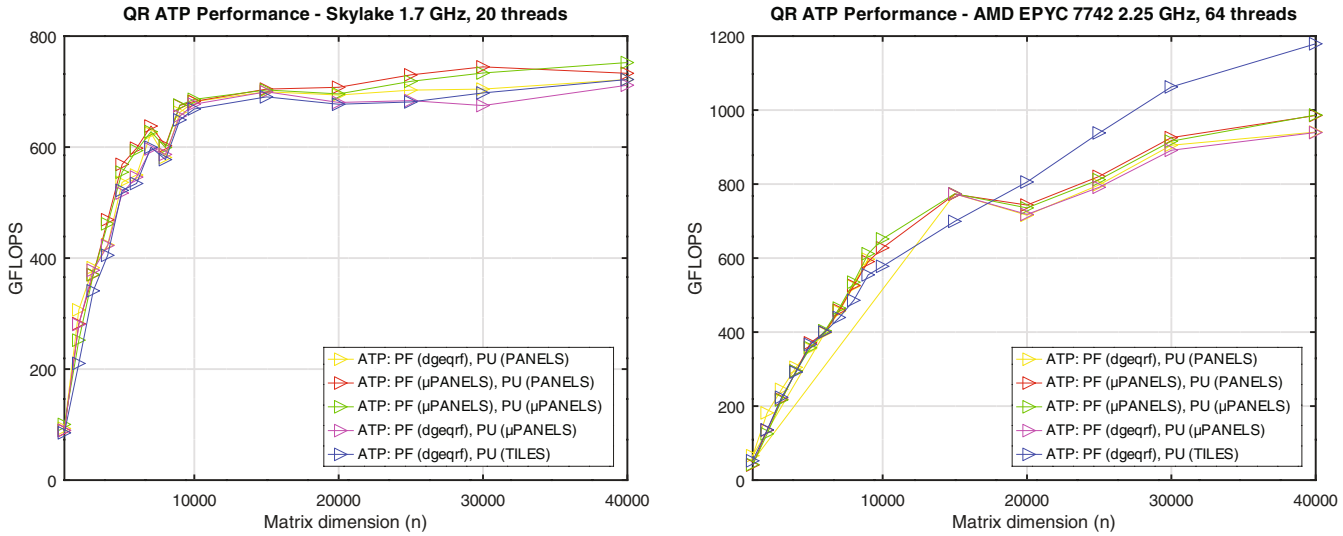


FIGURE 12 Performance of the ATP implementations for the QR factorization on SKYLAKE and ROME using 20 and 64 cores (left and right, respectively)

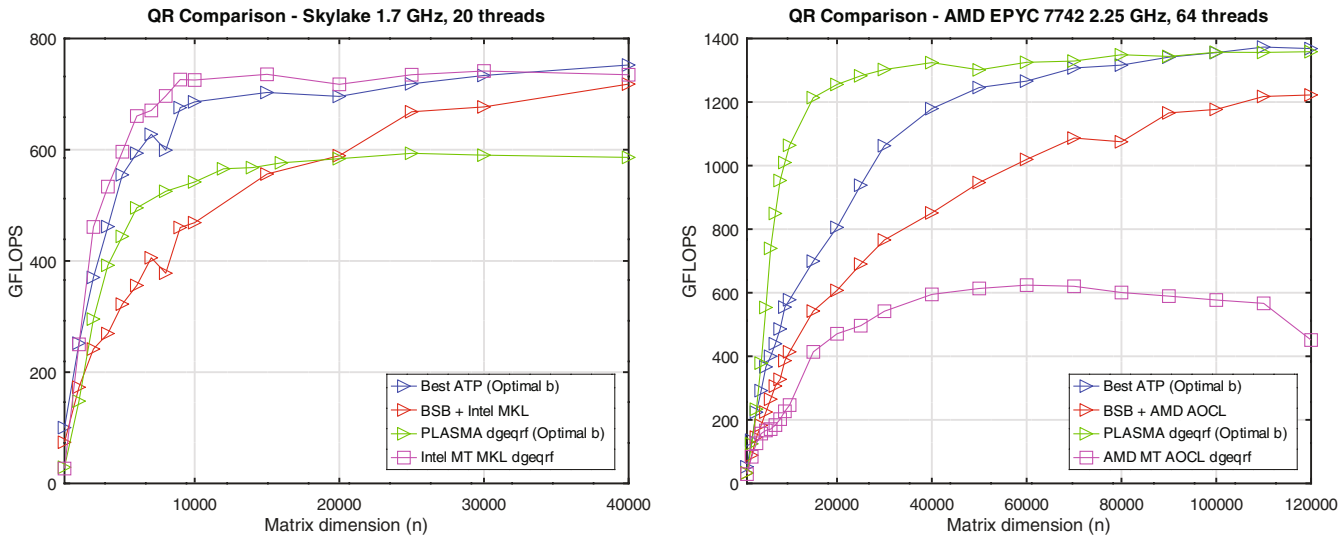


FIGURE 13 Comparative performance of different implementations for the QR factorization on SKYLAKE and ROME using 20 and 64 cores (left and right, respectively)

The comparison with proprietary libraries depends on the specific target machine and proprietary library. On SKYLAKE, our codes are highly competitive with Intel MKL and mimic its behavior for all matrix sizes; also, the comparison with PLASMA yields higher performance for our realization. On ROME, however, the parallel implementation of the QR factorization in AOCL yields lower performance numbers than ours, but the PLASMA implementation outperforms the best ATP implementation; our analysis in this case indicates that PF still poses a critical bottleneck in our realizations, and additional optimizations are mandatory in this stage when combined with AOCL in order to remove this performance bottleneck.

As an additional explanation of the different behavior of these algorithms, depending on the target platform, we note that PLASMA employs a specific tile-based solution for the QR factorization that exposes additional task-parallelism at earlier stages of the algorithm. This approach requires customized linear algebra kernels in order to perform the internal operations with the tiles, corresponding to the calculation of tiny QR factorizations and application of orthogonal transforms. In contrast, our two-level task-parallel algorithms only employ standard LAPACK routines for these two kernels. The results point in the direction that the use of these customized kernels on the Intel server has a significant (negative) impact on performance while this is not the case on the AMD platform.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

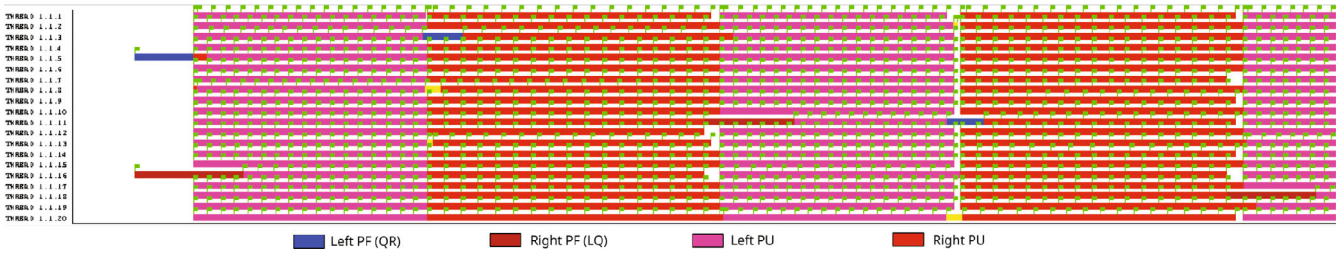


FIGURE 14 Task analysis of the band reduction routine linked with Intel MKL on SKYLAKE. The trace corresponds to the execution of the first two iterations with $n = 30{,}000$, $w = 256$, $b_p = 96$, and 20 cores

4.4 | Orthogonal reduction to band form

As argued in Section 3.3, the algorithm for the band reduction differs from the QR factorization (as well as matrix inversion via GJE) in that it performs two PFs per iteration: a QR factorization that updates the trailing matrix from the left and an LQ factorization that updates the trailing matrix from the right. In principle, this could seem to impose an additional performance challenge as, for the two other operations, we reiterate that a large number of the optimizations were aimed at mitigating the performance bottleneck that the panel factorization represents. However, as depicted in the execution trace of the BTP band reduction algorithm in Figure 14, this is actually not the case for this operation.

In the trace we observe that the left PF (in blue) and the right PF (in dark red) are perfectly overlapped with the execution of the right and left PUs (in red and pink, respectively). Concretely, in the figure we can observe that the left PF corresponding to the second iteration runs in parallel with the execution of the right PU of the first iteration, and the right PF corresponding to the second iteration occurs simultaneously with the execution of the left PU of the second iteration. This is possible due to the dynamic look-ahead exploited by the OpenMP runtime. Overall, for the reference band reduction algorithm, there are only small idle periods and, therefore, little opportunities to improve performance.

The performance results for the band reduction algorithm are reported in Figure 15. First, the plots reveal that, in both servers and for all problem dimensions, the task-based implementations greatly outperform the results of the LAPACK-like BSB implementation. Second, the division of the PU kernels into μ-panels yields minor performance benefits. In relation with this, an additional aspect that deserves some discussion is the constraint on block sizes utilized during the band reduction algorithm. Focusing on the block size for PF, we encounter an upper bound on its size that depends on the target bandwidth. Specifically, this block cannot exceed half the bandwidth, as otherwise the OpenMP runtime will not be able to exploit the dynamic look-ahead and, as a consequence, it will be impossible to overlap PF and PU, which introduces significant idle periods during the execution of the algorithm. In addition, focusing on the optimal block size for PU, our experiments revealed that, for mid lo large-size problems, this tends to equal the block size for PF, while for small- to midsize problems, the former block size tends to be slightly smaller. It is at these points where the division of PU into μ-panels produces visible performance improvements.

4.5 | Discussion

In order to close this section, and given the mixed character of the results, we recap the main highlights, insights and caveats identified during the evaluation, which can be organized into the following three main groups:

- *Comparison with proprietary libraries (Intel MKL/AMD AOCL)*: Our solutions outperform Intel's MKL counterpart for matrix inversion and the AMD AOCL routine for the QR factorization. In addition, it is highly competitive in the rest of the architecture/library combinations. (Recall that the third operation, orthogonal reduction to band form, is not supported by proprietary libraries.)
- *Comparison with a state-of-the-art runtime-based infrastructure (PLASMA)*: our solution is highly competitive with PLASMA for all operations, with clear benefits in the case of the QR factorization on the Intel-based server, and only inferior performance for the QR factorization on the AMD-based platform for small and midrange matrices. (Note that the orthogonal reduction to band form is neither supported by PLASMA.)
- *Comparison with basic block-based codes (built on top of high-performance parallel libraries)*: Our solution outperforms its counterparts in all cases and architectures.

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

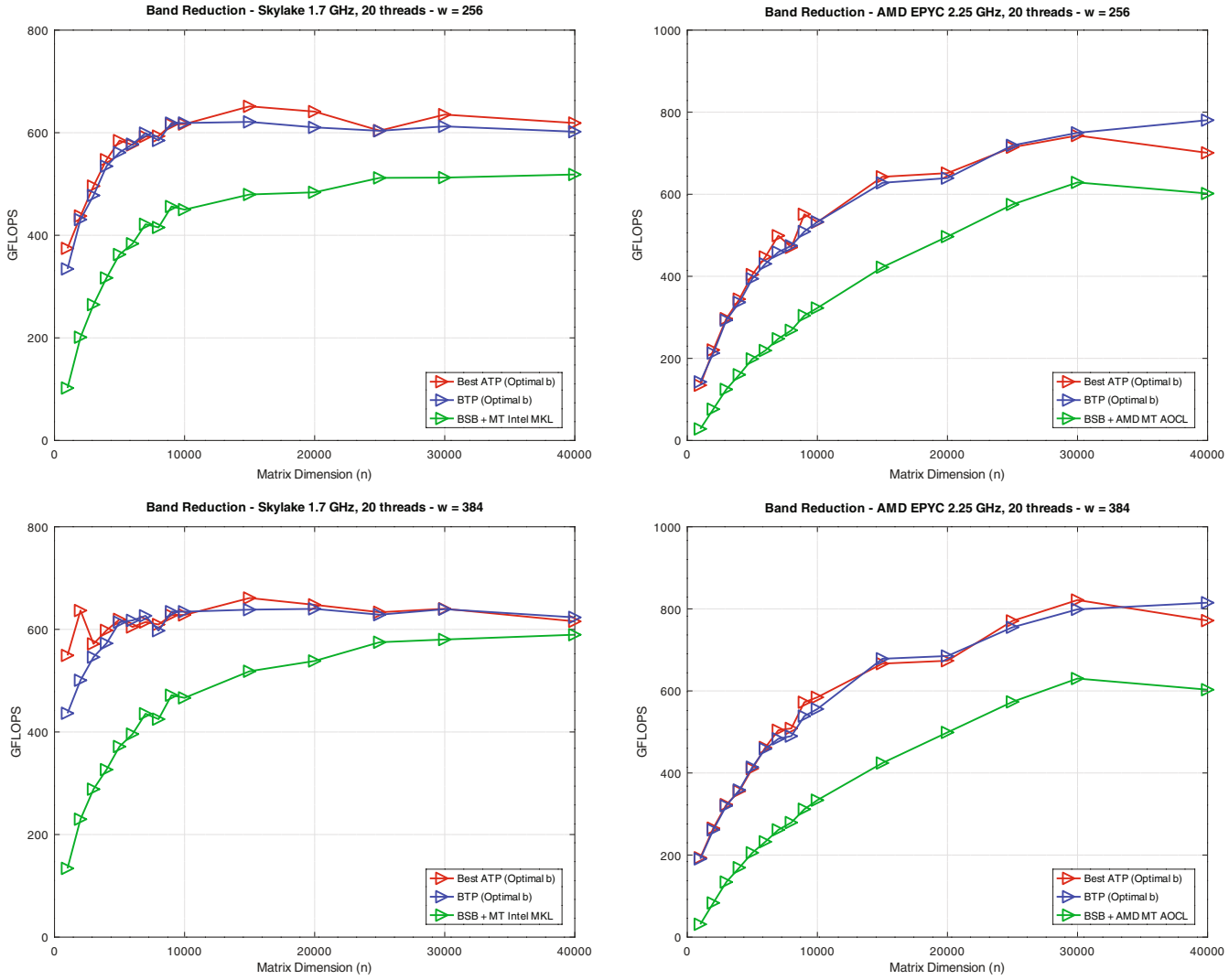


FIGURE 15 Performance of the algorithms for the Band Reduction setting w to 256 (top) and to 384 (bottom) on SKYLAKE and ROME using 20 and 64 cores (left and right, respectively)

5 | CONCLUSIONS

In this article, we have proposed several two-level task-parallel schemes to accelerate the execution of representative matrix factorization routines, such as the matrix inversion via Gauss–Jordan elimination (GJE), the QR factorization and the orthogonal reduction to band form. In more detail, we have leveraged the improved functionality in the newest versions of the OpenMP standard in terms of tasking, together with the maturity of their implementation in modern compilers and runtimes, in order to extract additional levels of task-level parallelism in critical parts of the algorithms, necessary to address the increasing number of cores in present and future many-core architectures.

The two-level task-parallel algorithms rely on a two-level task partitioning strategy. At the outer level, the algorithms follow the classical blocked procedure to decompose the operation by columns, with the purpose of alleviating the performance bottleneck that the costly PFs introduce (i.e., increasing the ratio of high-performance BLAS-3 routines while decreasing the ratio of memory-bound BLAS-1 and BLAS-2 routines). At the inner level, an extra partitioning into finer-grain tasks, either by tiles or by μ-panels, is enforced in order to expose an extra amount of task parallelism and hence improve core occupation.

Our experiments on two recent many-core general-purpose processors from Intel and AMD, with 20 and 64 cores, show the performance benefits of the two-level task partitioning schemes against the classical blocked algorithms as well as, in most cases, state-of-the-art libraries such as PLASMA, Intel MKL and AMD ACML.

Note that the main contribution of this work is two-fold: First, we provide a wider variety of tools and implementations to generic runtimes that exploit multigrained task parallelism so that a *common* family of solutions can be leveraged in order to obtain competitive performance on a wide

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

range of architecture/operation combinations. Second, we carry out a complete performance comparison between task-based and library-based implementations for different operations and architectures. From this point of view, we note that manufacturers often invest ample efforts in performance tuning of linear algebra libraries yet the experience gained with this work is rarely shared with the scientific community.

As part of many other ideas for future work, we would like to explore the effect of enforcing task priorities into the runtime scheduler as current realizations of OpenMP offer limited support for this. In addition, we would also like to investigate the exploitation of task parallelism in combination with loop-level parallelism for finer grain tasks that stand on the critical path.

ACKNOWLEDGMENTS

This research was sponsored by projects RTI2018-093684-B-I00, PID2019-107255GB and TIN2017-82972-R of Ministerio de Ciencia, Innovación y Universidades; project S2018/TCS-4423 of Comunidad de Madrid; project 2017-SGR-1414 of the Generalitat de Catalunya and the Madrid Government under the Multiannual Agreement with UCM in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT, project PR65/19-22445.

DATA AVAILABILITY STATEMENT

Author elects to not share data.

ENDNOTE

*In many of the traces shown in this work, including this one, the block size is set to a suboptimal value, in order to illustrate more clearly a particular problem; for example, this case shows that the amount of task parallelism is scarce.

ORCID

Francisco D. Igual https://orcid.org/0000-0003-4480-9517
Enrique S. Quintana-Ortí https://orcid.org/0000-0002-5454-165X

REFERENCES

1. Buttari A, Langou J, Kurzak J, Dongarra J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput*. 2009;35(1):38-53.
2. PLASMA project home page. http://icl.cs.utk.edu/plasma
3. Quintana-Ortí G, Quintana-Ortí ES, Geijn RA, Zee FG, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans Math Softw*. 2009;36(3):14:1-14:26.
4. Van Zee FG. libflame: the complete reference; 2012. www.lulu.com
5. Agullo E, Beaumont O, Eyraud-Dubois L, Kumar S. Are static schedules so bad? A case study on Cholesky factorization. Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2016:1021-1030; IEEE, 10.1109/IPDPS.2016.90
6. StarPU project. http://runtime.bordeaux.inria.fr/StarPU/
7. Badia RM, Herrero JR, Labarta J, Pérez JM, Quintana-Ortí ES, Quintana-Ortí G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurr Comput Pract Exp*. 2009;21:2438-2456.
8. OmpSs project home page. http://pm.bsc.es/ompss
9. Catalán S, Igual FD, Rodríguez R, Herrero JR, Quintanía-Ort ES. A new generation of task-parallel algorithms for matrix inversion in many-threaded CPUs. Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores–PMAM 2021; 2021:1-10, 10.1145/3448290.3448563
10. Householder AS. *The Theory of Matrices in Numerical Analysis*. Dover; 1964.
11. Quintana ES, Quintana G, Sun X, van de Geijn R. A note on parallel matrix inversion. *SIAM J Sci Comput*. 2001;22(5):1762-1771.
12. Golub GH, Loan CFV. *Matrix Computations*. 3rd ed. Johns Hopkins University Press; 1996.
13. Strazdins P. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical report TR-CS-98-07, Department of Computer Science, Australian National University, Canberra 0200 ACT, Australia 1998.
14. Gunter BC, van de Geijn RA. Parallel out-of-core computation and updating the QR factorization. *ACM Trans Math Soft*. 2005;31(1):60-78. doi:10.1145/1055531.1055534
15. Quintana-Ortí ES, van de Geijn RA. Updating an LU factorization with pivoting. *ACM Trans Math Softw*. 2008;35(2):11:1-11:16. doi:10.1145/1377612.1377615
16. Rodríguez-Sánchez R, Catalán S, Herrero JR, Quintana-Ortí ES, Tomás AE. Look-ahead in the two-sided reduction to compact band forms for symmetric eigenvalue problems and the SVD. *Numer Algorithms*. 2019;80:635-660.

How to cite this article: Catalán S, Herrero JR, Igual FD, Quintana-Ortí ES, Rodríguez-Sánchez R. Fine-grain task-parallel algorithms for matrix factorizations and inversion on many-threaded CPUs. *Concurrency Computat Pract Exper*. 2023;35(27):e6999. doi: 10.1002/cpe.6999

15320634, 2023, 27, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.6999 by Universitaet Politecnica De, Wiley Online Library on [30/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License