A New Generation of Task-Parallel Algorithms for Matrix Inversion in Many-Threaded CPUs

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

[1] Universidad Complutense de Madrid.
[2] Universitat Politècnica de Catalunya.
[3] Universitat Politècnica de València.

February 2021

Abstract

We take advantage of the new tasking features in OpenMP to propose advanced task-parallel algorithms for the inversion of dense matrices via Gauss-Jordan elimination. Our algorithms perform a partitioning of the matrix operand into two levels of tasks: The matrix is first divided vertically, by column blocks (or panels), in order to accommodate the standard partial pivoting scheme that ensures the numerical stability of the method. In addition, depending on the particular kernel to be applied, each panel is partitioned either horizontally by row blocks (tiles) or vertically by μ-panels (of columns), in order to extract sufficient task parallelism to feed a many-threaded general purpose processor (CPU).

The results of the experimental evaluation show the performance benefits of the advanced tasking algorithms on an Intel Xeon Gold processor with 20 cores.

Keywords: Task parallelism, OpenMP, matrix inversion, high performance.

1 Introduction

Task parallelism has been proposed as a response to tackle algorithms with an irregular and runtime dependent execution flow [17]. Furthermore, task parallelism is also appealing as a means to deal with the increasing number of cores of current and future general-purpose processors (hereafter, CPUs) with a large count of cores.

Following the two opening assertions in this section, in recent years a number of efforts have demonstrated the benefits of extracting task parallelism for dense linear algebra operations; see, e.g., PLASMA [4, 11], libFLAME [14, 18], StarPU [15], and OmpSs [2, 10]. In the case of the PLASMA library, its native runtime system (Quark) was recently abandoned in favour of OpenMP 4.5 [6]. In contrast, OmpSs and StarPU are research tools to explore the design space

The final publication is available at ACM via https://doi.org/10.1145/3448290.3448563

for multi-threading parallelism, with the objective of extending OpenMP with new directives to support, among others, asynchronous parallelism and heterogeneity.

In response to the integration of an increasing number of cores in CPUs, in this paper we target the parallelization of matrix inversion, using OpenMP tasking, on many-threaded architectures. While matrix inversion can be viewed as a representative operation for many other dense matrix factorizations [7], there exist a few relevant applications (arising, e.g., in statistics, numerical integration in superconductivity computations, and computation of the stable subspace in control theory) in which a matrix inverse must be explicitly computed [12]. In this case, the Gauss-Jordan elimination (GJE) [9, 8] yields a highly parallel method, with a constant (regular) workload per iteration that is alluring from the parallelization perspective on clusters as well as platforms equipped with hardware accelerators [12, 1, 3].

The integration of partial pivoting to ensure the (practical) numerical stability of matrix inversion via GJE [8] restricts the options to exploit task parallelism for this particular operation, much like it occurs for the LU factorization [4, 14]. The reason is that partial pivoting is realized via a sequence of row permutations, which advocates for a sort of 1D (column-wise) workload distribution among the threads that limits the parallel scalability of the algorithm. The same problem has been previously tackled for the LU factorization in a number of manners: The authors of [13] described a more flexible (incremental) pivoting scheme that was the basis for subsequent work on communication-avoiding algorithms; the work in [5] introduced a very fine-grain, cache-aware (and complex) multi-threaded parallelization of the *panel factorization* that stands on the critical path; and PLASMA leverages OpenMP 4.5 to track dependencies at the granularity of individual columns, via dummy tasks [6].

In this paper we propose advanced task-parallel (TP) algorithms for matrix inversion via GJE that combine simplicity with row-wise and column-wise fine-grain task decompositions of the algorithm's operations in order to accommodate partial pivoting while exposing ample task parallelism. In particular, our paper makes the following contributions:

- We introduce a two-level task partitioning of the workload, analyzing the impact on performance of distinct horizontal/vertical divisions depending on the type of kernel: panel factorization, panel permutation, and panel update.
- We exploit the advanced tasking features in OpenMP to develop simple yet efficient codes for matrix inversion via GJE that can easily accommodate these distinct partitioning schemes.
- We present a complete experimental evaluation on a top-of-the-shelf Intel processor, with 20 cores, that includes performance experiments, analysis of optimal block size, and use of traces to illustrate the parallel behaviour and detect performance bottlenecks.

To close this part, we point out that the ideas discussed in the following for matrix inversion are likely to carry over to other "column-wise"-oriented matrix factorizations: the LU and LDL^T decompositions for the solution of linear systems; the QR factorization for linear least squares problems; and two-sided

orthogonal reduction to condensed forms for the solution of eigenproblems and the singular value decomposition [7].

After a short introduction of notation in the next subsection, the rest of the paper is structured as follows. In Section 2 we briefly review the GJE method for matrix inversion and discuss the conventional parallelization approach advocated by LAPACK, applied to this procedure. Next, in Section 3 we introduce a basic TP algorithm for this matrix operation that preserves partial pivoting by enforcing a simple 1D distribution of the workload by columns. In Section 4 we describe our two-level TP approach to expose additional task parallelism for the operation. Finally, we close the paper with a few concluding remarks and a discussion of future work in Section 5.

1.1 Notation

For the remainder of the paper, we will consider a nonsingular $n \times n$ matrix A where, for simplicity, we assume that n is an integer multiple of the algorithmic block size b. Furthermore, for the presentation of the algorithms, we will partition A into $s = n/b$ column blocks (also referred to as panels), of dimension $n \times b$ each. In our notation, $A(:, c_1 : c_2)$ denotes the submatrix of A that spans panels $c_1, c_1 + 1, \ldots, c_2$ of A, which comprise columns $c_1 \cdot b, c_1 \cdot b + 1, \ldots, c_2 \cdot b + b - 1$ of the matrix. (Note that our indices for vector arrays and matrices start at 0.) In some cases, the panels will be further partitioned into row blocks, or tiles.

2 Matrix inversion via GJE and conventional parallelization

In this section we introduce the algorithm for matrix inversion via GJE and review the conventional multi-threaded parallelization approach for this type of operations adopted in LAPACK.

2.1 Baseline GJE algorithm

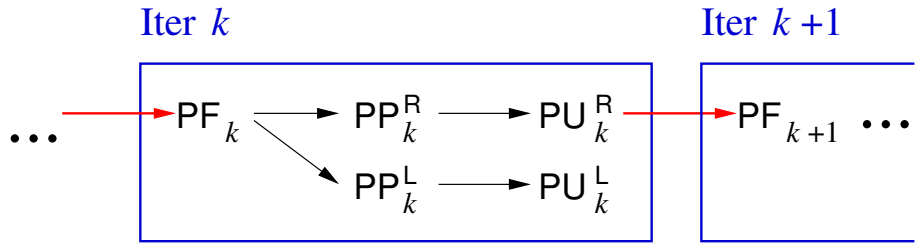
Figure 1a displays the GJE algorithm for matrix inversion [9, 12]. The formulation there corresponds to a baseline blocked (BSB) algorithm, presented with a high level of abstraction that is especially suited for the discussion of the distinct parallelization strategies that are applied to this operation in the paper. The three types of kernels inside the algorithm's loop, indexed by k, perform the following calculations:

1. Compute the "factorization" of panel k of the matrix (consisting of the matrix columns $k \cdot b : (k+1) \cdot b - 1$) via routine PF (for panel factorization). The practical numerical stability of the GJE procedure is ensured via the integration within this kernel of a partial pivoting scheme akin to that used in the LU decomposition [7, 8]. In the formulation of the algorithm, the row permutations due to the application of the pivoting technique are assumed to be stored in the array (vector) p.

2. Permute the submatrix (panels) to the left and right of panel k, respectively comprising the matrix columns $0 : k \cdot b - 1$ and $(k + 1) \cdot b : n - 1$, via routine PP (for panel permutation).

```
1  #define K ((k)*b : ((k)+1)*b-1)
2  #define S ((s)*b : ((s)+1)*b-1)
3
4  void BSB_GJE( matrix A, vector p, int s, int b )
5  {
6    for ( k = 0; k < s; k++ ) {
7      // Factorize panel k
8      PF( A(:,K), p(K) );
9
10     // Permute panels 0:k-1 w.r.t. panel k
11     PP( p(K), A(:,0:K-1) );
12     // Update panels 0:k-1 w.r.t. panel k
13     PU( A(:,K), A(:,0:K-1) );
14
15     // Permute panels k+1:s-1 w.r.t. panel k
16     PP( p(K), A(:,K+1:S-1) );
17     // Update panels k+1:s-1 w.r.t. panel k
18     PU( A(:,K), A(:,K+1:S-1) );
19   }
20 }
```

(a) Routine. 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).



(b) Data dependencies. PF_k, $\mathsf{PP}^{\mathsf{L}}_k/\mathsf{PP}^{\mathsf{R}}_k$, $\mathsf{PU}^{\mathsf{L}}_k/\mathsf{PU}^{\mathsf{R}}_k$, respectively stand for the panel factorization, panel permutation, and panel update performed during iteration k of the BSB algorithm in Figure 1a. In the latter two types of kernels, the superindices indicate whether the operations involves the submatrix to the "L"eft or the "R"ight of panel k.

Figure 1: BSB GJE algorithm for matrix inversion.

3. Update of the submatrix (panels) to the left and right of panel k via routine PU (for panel update).

A couple of observations about this algorithm are relevant for the discussion in the rest of the paper:

- Provided the algorithmic block size b is selected to be moderately large, the BSB algorithm can hide the memory accesses with sufficient floating-point operations, overcoming the memory bandwidth bottleneck.
- As discussed in detail in the next subsection, the use of partial pivoting introduces certain data dependencies that restrict the parallelism among the kernels in the loop body.

2.2 Dependency analysis and conventional parallelization

In this type of implementations, parallelism is extracted at the kernel level, using a *fork-join* approach; that is, sequential implementations are linked with a parallel BLAS library to gain performance and improve core occupation. As illustrated in Figure 1b, due to data dependencies the execution of the PF kernel in iteration k (denoted as PF_k,) cannot be overlapped with any of the permutations and updates in the same iteration (respectively, PP_k and PU_k). Furthermore, for the range of values that b takes in practice, the PF kernel exhibits a limited degree of parallelism due to its reduced number of columns, the row permutations required for partial pivoting, and the complex data dependencies existing within this particular kernel. Therefore, Amdahl's law dictates that a parallelization scheme that only exploits the intrinsic parallelism within each of the kernels in the loop body will face a scalability bottleneck as the number of threads tends to grow.

Unfortunately, that is precisely the approach adopted by the conventional parallelization scheme in LAPACK, which simply extracts all parallelism by invoking a multi-threaded instance of the BLAS. (Internally, the multi-threaded BLAS leverage POSIX threads to extract loop parallelism.)

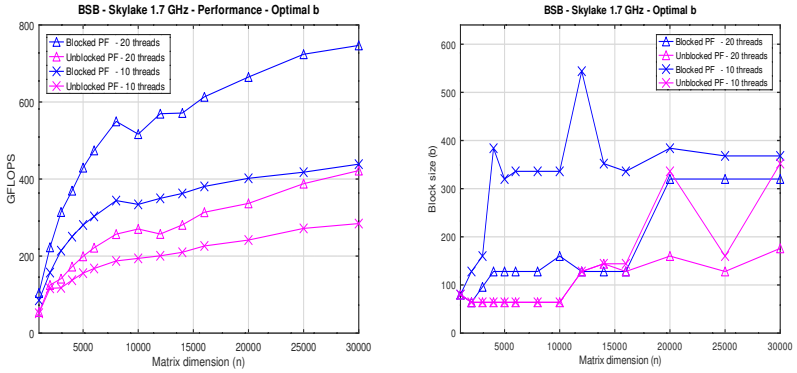
The experiments that close this section show the negative impact of the panel factorization on the global performance of the matrix inversion algorithm via GJE. Prior to the discussion of these results, in the next subsection we introduce the hardware and software setup that will be utilized during all the experimental evaluation conducted in this work.

2.3 Setup

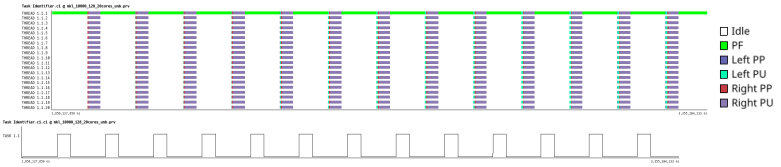
The experiments in this paper were performed using double precision (DP) arithmetic on a server equipped with a 20-core Intel Xeon Gold 6138 processor (Skylake micro-architecture) running at 1.7 GHz. Each core features two AVX-512 FMA (512-bit wide) units, yielding a peak performance of 51.4 DP GFLOPS (billions of floating point operations, or flops, per second) per core and a total of 1,028 DP GFLOPS for the complete socket. 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. On the software side, we employed the Intel compilers for C/Fortran (`icc`/`ifort`) version 18.

In order to avoid the performance distortions caused by the utilization of the power modes (and associated processor core frequencies) in the Intel Skylake architecture, the operating frequency was set to 1.7 GHz for all cores. One single thread was mapped per physical core and thread migration was prevented via the appropriate Linux configuration commands.

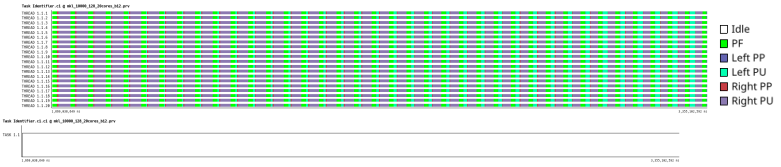
Unless otherwise stated, the algorithmic block size b (in our work, also referred to as "outer" block size or panel width) was selected individually for each algorithm, matrix dimension, and number of cores in order to optimize performance. For the variants that compute the PF kernel using a blocked procedure, the inner block size b_{in} was selected via extensive experimentation with values ranging between 8 and the (outer) block size b.



(a) Performance (left) and optimal block size (right).



(b) Trace of the first 1.5 seconds of execution with $n = 10,000, b = 128$ and 20 cores showing the task type (top) and core occupation (bottom), using an unblocked procedure for PF.



(c) Trace of the first 1.5 seconds of execution with $n = 10,000, b = 128, b_{in} = 12$ and 20 cores showing the task type (top) and core occupation (bottom), using a blocked procedure for PF.

Figure 2: Performance study of the BSB GJE routine linked with Intel MT MKL on Skylake.

2.4 Performance analysis

Figure 2 reports the results for the BSB GJE routine when linked with the multi-threaded (MT) instance of Intel MKL, mimicking the conventional parallelization scheme for LAPACK. The left-hand side plot in the figure reports the GFLOPS rate for matrices of dimension n up to 30,000 with 10 and 20 cores. In addition, we evaluate there two versions of the BSB routine, which respectively compute the PF kernel via either an unblocked algorithm or a blocked version based on the same BSB routine (with a smaller "inner" block size). The optimal values for the block size b determined for each version are reported in the right-hand side plot in the same figure. Furthermore, the trace in the bottom part of the figure shows the tasks that are executed during the initial iterations of the algorithm and the core occupancy, for a particular matrix dimension, number of cores, and block size.

The results in the top-left plot of Figure 2 report a growth of performance

with the matrix dimension, reaching 410+ and 750+ GFLOPS for the version that performs the PF kernel via a blocked algorithm applied to tackle the largest matrix dimension using 10 and 20 cores, respectively. Furthermore, there is a clear performance gap (close to a factor 1.5×) between this version and the counterpart that computes PF using an unblocked procedure, see the corresponding trace. As reported in the top-right plot, and could be expected in practice, in both cases the optimal block size tends to grow with the dimension of the matrix, with smaller values in the unblocked version to reduce the impact of the suboptimal PF factorization.

The traces in the bottom half of the figure expose the synchronous behaviour of the LAPACK-like conventional parallelization scheme, which strictly alternates the execution of PF with that of the matrix permutations/updates. The unblocked procedure is composed of Level-2 BLAS kernels, which are executed sequentially. In contrast, the blocked procedure mainly consists of Level-3 BLAS kernels, but applied to a narrow panel, as in this case, result in low parallel performance. The application of the row permutations is parallelized by adding the appropriate OpenMP parallelization pragma in the outermost loop of the realization of the routine for this purpose in LAPACK (`dlaswap`).

The insight to take away from these traces is that a non-overlapped execution of the PF kernels with other operations will likely result on low performance, as the panel factorization is intrinsically close-to-sequential, yielding a significant waste of resources (threads/cores) that will remain idle or infra-utilized.

3 Basic task-parallel Algorithms for Matrix Inversion

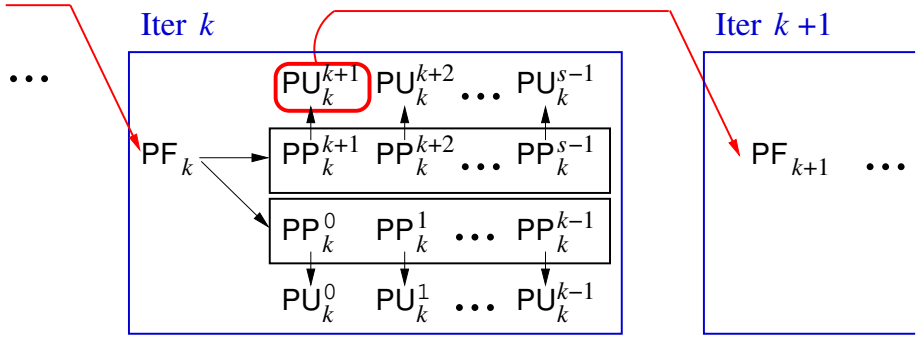
In this section we describe a basic task-parallel (hereafter BTP) alternative for the multi-threaded execution of the GJE algorithm that aims to overcome the performance bottleneck imposed by the PF kernel. One of the main differences between the TP approach and the LAPACK-like parallelization scheme lies in the fact that, in the former the kernels are purely sequential, and parallelism is exposed by the programmer in terms of tasks via OpenMP annotations, which serve as hints to the runtime in order to control (task) data dependencies, as explained next. In contrast, for the LAPACK-like scheme the parallelism is internal to the (multi-threaded) BLAS routines and does not require any action from the programmer.

3.1 GJE algorithm with dynamic look-ahead via a runtime

Let us divide the matrix into a collection of panels, of b columns each, so that at iteration k, there are k panels in the leading submatrix (i.e., to the left of panel k) and $s - k - 1$ panels in the trailing one (i.e, to the right of panel k). The dependencies of this TP algorithm, with the permutations and updates decomposed into a collection of tasks, are illustrated in Figure 3a. (For simplicity, some of the dependencies, which are not relevant for the following discussion, are omitted in the figure.)

The key aspect revealed in Figure 3a is that, at iteration k, the trailing permutation/updates for panels $j = k + 2, k + 3, \ldots, s - 1$ (i.e., all except $j =$

.



(a) Task dependencies. PF_k stands for the panel factorization at iteration k; PP_k^j and PU_k^j respectively denote the panel permutations and updates of panel j performed during the same iteration k of the BTP routine in Figure 3b.

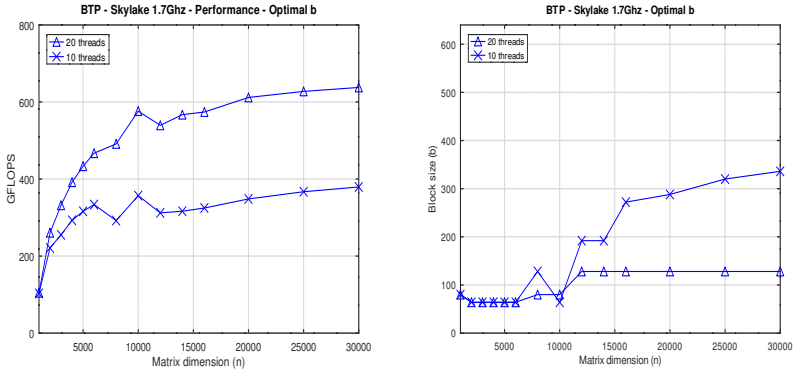
```
1  #define K ((k)*b : ((k)+1)*b-1)
2  #define J ((j)*b : ((j)+1)*b-1)
3
4  void BTP_GJE( matrix A, vector p, int s, int b )
5  {
6    #pragma omp parallel {
7    #pragma omp single {
8    for ( k = 0; k < s; k++ ) {
9      // Factorize panel k
10     #pragma omp task depend( inout: A(:,K), out: p(K) )
11     PF( A(:,K), p(K) );
12
13     // Permute and update panels 0:k-1 w.r.t. panel k
14     for ( j = 0; j < k; j++ ) {
15       #pragma omp task depend( in: p(K), inout: A(:,J) )
16       PP( p(K), A(:,J) );
17       #pragma omp task depend( in: A(:,K), inout: A(:,J) )
18       PU( A(:,K), A(:,J) );
19     }
20
21     // Permute and update panels k+1:s-1 w.r.t. panel k
22     // Omitted for brevity
23   }}}
24 }
```

(b) Routine. For brevity, we omit the `firstprivate` and `private` clauses that are included in the actual code.

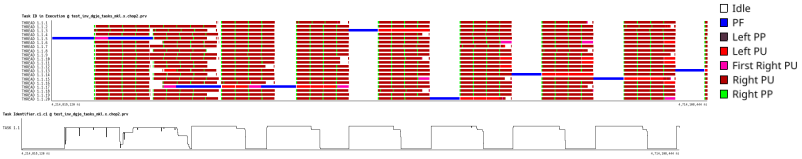
Figure 3: BTP GJE routine for matrix inversion, with task parallelism extracted via OpenMP.

$k+1$), plus all the leading permutation/updates, for panels $j = 0, 1, \ldots, k-1$, can proceed in parallel with the factorization of panel $k+1$. This technique is often referred to as *look-ahead* [16] and allows to overlap the execution of the largely sequential PF with the highly parallel PP and PU.

The re-formulation of the code to accommodate a dynamic variant of the



(a) Performance (left) and optimal block size (right).



(b) Trace of the first 0.5 seconds of the execution with $n = 10,000, b = 128$ and 20 cores showing the task type (top) and core occupation (bottom).

Figure 4: Performance study of the BTP GJE algorithm on SKYLAKE.

look-ahead technique while extracting task parallelism using OpenMP is shown in Figure 3b. Note that the specific task schedule is non-deterministic, as it is decided 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. Thus, the depth or degree of the look-ahead is dynamically controlled by the runtime.

3.2 Performance analysis

The top-left plot in Figure 4 reports the GFLOPS rates for the BTP GJE algorithm, using 10 and 20 cores, applied to matrices of dimension n up to 30,000. As in other experiments in this paper, the inner and outer block sizes are optimized via an extensive experimental evaluation. The bottom trace shows the parallel behaviour of this routine, for a specific case configuration.

In the trace, we can observe how look-ahead is automatically in place via the runtime data-dependency management: for a given iteration k, as soon as the update of the $k + 1$ panel is accomplished (task in magenta and labelled as First Right PU), the factorization of panel $k + 1$ can proceed, effectively overlapping tasks from different iterations. A relevant aspect to note in the trace is that, for a few initial iterations, this BTP realization offers a certain degree of overlapping between the PF kernel and the remaining operations, but not as much as it is possible. The reason for this suboptimal behaviour is that, although we included the appropriate OpenMP `priority` clauses to advance the execution of the PF kernels (and dependent kernels), the scheduler in Intel's OpenMP runtime does not seem to use these hints. (In a separate experiment,

we could confirm that this version of the compiler does not take into account OpenMP task priorities.) As a result, the execution of the BTP version does not offer a significant advantage with respect to a conventional parallelization. In fact, as the large updates performed on the trailing/leading submatrices are divided into multiple panels/tasks, each to be executed by a single thread, the threads compete for shared resources, such as the memory bandwidth. The net outcome is that the performance diminishes with respect to that of the BSB routine, as can be observed from a direct comparison between Figures 2a and 4a.

To conclude this section, we note the critical role of the algorithmic block size b for the TP scheme. Concretely, choosing a large value for b improves the performance of the individual BLAS that are invoked for the panel updates, at the cost of reducing the degree of task parallelism and making more difficult to hide the cost of the PF kernels. Conversely, a small value of b reduces the performance of the BLAS, though it augments the amount of task parallelism and decreases the contribution of the panel factorization to the global cost.

4 A New Generation of task-parallel Algorithms for Matrix Inversion

In this section we propose several advanced task-parallel (ATP) algorithms that extract additional tasks within the PF kernels as well as the panel permutations and updates.

4.1 Horizontal partitioning of the panel update

The introduction of partial pivoting seems to impede a "horizontal" division of the operations on the panels into tasks, constraining the amount of task parallelism that can be exposed and, therefore, exploited during the inversion. However, while this is the case for the panel permutations due to the (row) partial pivoting, it does not hold for the arithmetic operations that are necessary to perform a panel update.

The code in Listing 1 illustrates how to take advantage of this idea in order to obtain a TP scheme with additional task parallelism via a horizontal partitioning of each ("parent") panel update into $r = n/t$ ("child") tasks, each processing a "'tile" of dimension $t \times b$. (For simplicity, we assume that the problem dimension n is an integer multiple of the tile height t.) Note that the code for this ATP algorithm includes an OpenMP `taskwait` pragma to synchronize a group of child tasks with the corresponding parent task. In addition, the tile partitioning is only performed on the panel update kernels, but not on the panel permutations.

An aspect to highlight is that, in order to ensure that the data dependencies are correctly identified, we need to leverage the support for "array sections" in OpenMP, passing the complete specification of the memory regions covered by each panel/tile task. In comparison, for the basic parallelization scheme in Section 3, the panels operated each on a disjoint region of memory. Therefore, in order to detect data dependencies in the BTP routine there, it could have been sufficient to compare the address in memory of the top-left entry of each panel instead of the whole regions.

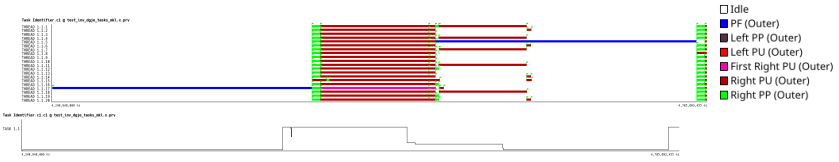
```
1  #define K ((k)*b:((k)+1)*b-1)
2  #define J ((j)*b:((j)+1)*b-1)
3  #define I ((i)*t:((i)+1)*t-1)
4
5  void ATP_GJE( matrix A, vector p, \
6                int s, int r, int t, int b )
7  {
8    #pragma omp parallel {
9    #pragma omp single {
10   for ( k = 0; k < s; k++ ) {
11     // Factorize panel k
12     #pragma omp task depend( inout: A(:,K), out: p(K) )
13     PF( A(:,K), p(K) );
14
15     // Permute and update panels 0:k-1 w.r.t. panel k
16     for ( j = 0; j < k; j++ ) {
17       #pragma omp task depend( in: p(K), inout: A(:,J) )
18         PP( p(K), A(:,J) );
19
20       // Update tiles in panel j w.r.t. those in panel k
21       #pragma omp task depend( in: A(:,K), inout: A(:,J) )
22       {
23         for ( i = 0; i < r; i++ ) {
24           #pragma omp task depend( in: A(I,K), \
25                                    inout: A(I,J) )
26           PU( A(I,K), A(I,J) );
27         }
28         #pragma omp taskwait
29       }
30     }
31
32     // Permute and update panels k+1:s-1 w.r.t. panel k
33     // Omitted for brevity
34   }}}
35 }
```

Listing 1: ATP GJE routine with two-level task parallelism extracted via OpenMP and the panel updates divided horizontally into tiles to expose additional $(t \times b)$ tasks.

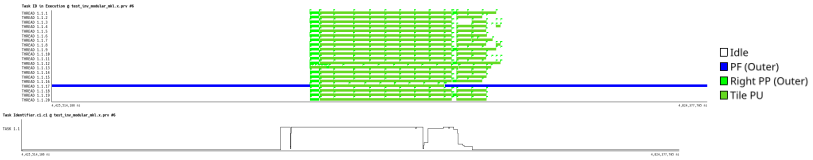
The top two traces in Figure 5 compare the execution of the BTP algorithm and the ATP counterpart that divides the panel updates horizontally into tasks. There is a clear unbalance in the workload distribution for the former, where a significant amount of the cores remain idle during the update part of the computation due to an insufficient number of tasks. This is tackled in the ATP variant via the partitioning of the panels into finer-grain tiles, allowing a better distribution of the work for the panel updates. Unfortunately, for the combination of block size, matrix dimension, and number of cores in this experiment, the panel factorization still imposes a major performance bottleneck for both TP algorithms.

4.2 Partitionings of the panel factorization

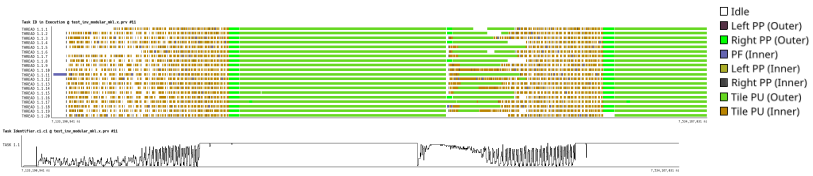
The obvious solution to the high cost of a sequential (single-task) execution of the PF kernel is to divide this operation into multiple tasks as well, exposing



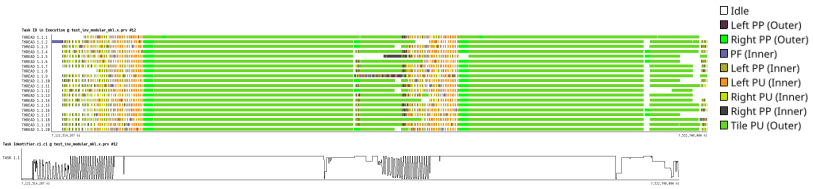
(a) BTP GJE algorithm.



(b) ATP GJE algorithm with horizontal partitioning of PU.



(c) ATP GJE algorithm with horizontal partitioning of both PF and PU.



(d) ATP GJE algorithm with vertical partitioning of PF into μ-panels and horizontal partitioning of PU.

Figure 5: Traces of the first two iterations of the execution of the TP GJE routines, with $n = 10,000, b = 384, t = 1,250$ and 20 cores, on SKYLAKE.

task parallelism which can be leveraged by the system cores to accelerate the otherwise sequential execution of this type of kernel.

At this point, we remind that, for performance reasons, the PF kernel is realized using a blocked procedure based on the BSB routine (see Section 2). Thus, at iteration k of the inversion algorithm, the implementation of this procedure processes panel k of the matrix by dividing its columns into μ-panels of width $b_{in} < b$. Therefore, we can consider two ways of parallelizing the *inner μ-panel updates* within this BSB procedure:

1. divide each individual inner μ-panel update horizontally into multiple "μ-tile tasks" (of dimension $t_{in} \times b_{in}$ each); or

2. consider each inner μ-panel update itself as "μ-panel task".

The algorithms for realizing these two parallelizations of the PF kernel are analogous to those in Figure 3b and Listing 1 (applied to the panel that has to be factorized instead of the full matrix).

The effect of these two partitioning schemes for the PF kernel, by μ-tiles or by μ-panels, combined in both cases with a horizontal division of the (outer)

panel updates, is reported in the bottom two traces in Figure 5. From this experiment, it seems clear the superiority of the second option, which tackles the PF kernel via a BSB procedure that considers the inner μ-panel updates appearing within the execution of this kernel as μ-panel tasks. The reason is that the alternative that divides PF into μ-tiles yields a three-level partitioning of the matrix operand into tasks, exposing an excessive number of tasks of too-fine granularity.

4.3 Vertical partitioning of the panel update

The previous discussions in this section leads to a natural question: Can we combine the version of PF that operates with μ-panels with an analogous vertical division of the (outer) panel updates (that is, into μ-panels as well)? While the answer is yes, we note that, in principle, this is equivalent to reducing the block size for the BTP algorithm.

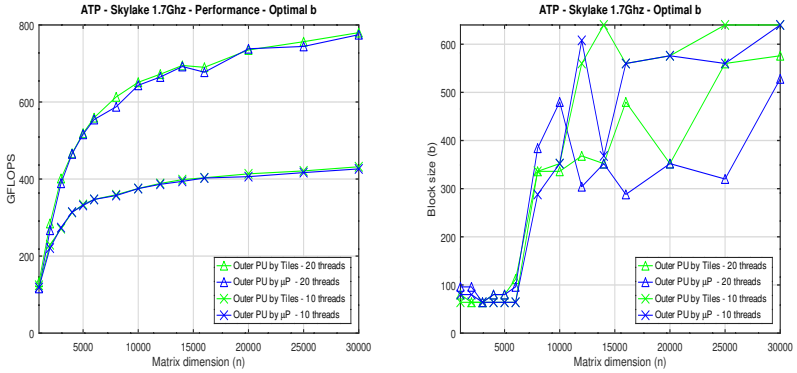
Figure 6 evaluates the performance of two ATP algorithms: In both cases, the inner μ-panel updates appearing in the PF kernel are considered as μ-panel tasks, so that they differ only in that the outer panel updates are partitioned into either μ-tiles or μ-panels. The results from this experiment show clear differences in the optimal block size as well as the execution traces for the first iterations. Nonetheless, there is a remarkable similarity in the performance attained by both schemes for 10 and 20 cores and all matrix dimensions.

4.4 Global comparison

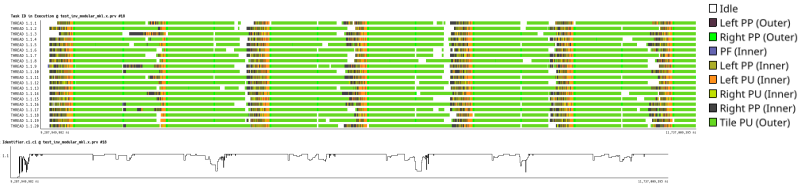
We close the experimental evaluation in this paper with a global comparison of the following four matrix inversion algorithms:

- An standard matrix inversion algorithm based on the LU factorization using the appropriate calls to the LAPACK routines in Intel MT MKL (`dgetrf`+`dgetri`).
- The BSB GJE routine in Section 2, with parallelism extracted only from the MT instance of the BLAS in Intel MKL.
- 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.

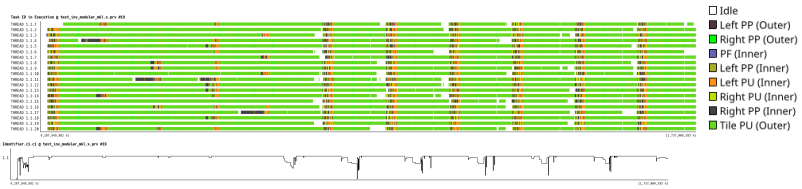
This evaluation involves two different types of algorithms for matrix inversion: one based on the LU factorization and our alternative based on GJE. We note that the two types of algorithms perform exactly the same arithmetic operations, though in a different order and, therefore, have the same computational 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 using 10 and 20 cores of the Skylake platform, 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



(a) Performance (left) and optimal block size (right).



(b) Trace of the first 0.5 seconds of the execution for the TP GJE algorithm with vertical partitioning of PF and horizontal partitioning of PU, with $n = 16,000, b = 480, b_{in} = 12, t = 4,000$ and 20 cores, showing task type (top) and core occupancy (bottom).



(c) Trace of the first 0.5 seconds of the execution for the TP GJE algorithm with vertical partitionings of both PF and PU, with $n = 16,000, b = 480, b_{in} = 12, t = 4,000$ and 20 cores, showing task type (top) and core occupancy (bottom).

Figure 6: Performance study of the ATP GJE algorithm on Skylake.

fixed value for the outer block size (concretely, set to $b = 256$) in order to assess the sensibility of performance to this parameter.

The results in Figure 7 show the benefits of the two-level task partitioning of the workload integrated in our ATP routine, which is only outperformed in one case: by the BSB routine, for the largest matrix dimension (n=30,000) and lowest number of cores (10). In all other 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 also a clear winner, both when selecting the best block size and fixing a default value (again, set to $b = 256$).

We close this section by noting that the use of the OpenMP array sections results in simpler parallel codes for our ATP routines, which do not require

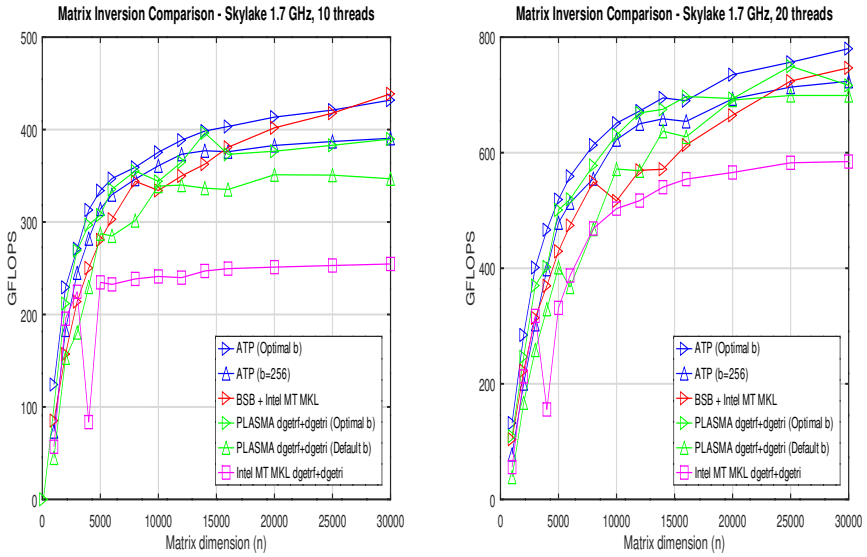


Figure 7: Performance of the algorithms for matrix inversion on SKYLAKE using 10 and 20 cores (left and right, respectively).

the use of dummy tasks, as in PLASMA, to control the execution for the LU factorization that appears during the matrix inversion [6].

5 Concluding Remarks and Open Research Lines

In this paper, we have designed several strategies to map a representative matrix factorization routine (matrix inversion via Gauss-Jordan elimination –GJE–) into a many-core general-purpose architecture. Specifically, 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 GJE algorithm, necessary to address the increasing number of cores in present and future many-core architectures.

In more detail, we have proposed a two-level task generation strategy that accommodates a first partitioning scheme based on a vertical (panel-wise) division of the matrix operand, which is combined with a second horizontal (tile-wise) division of each panel in order to expose an extra amount of task parallelism and hence improve core occupation. This strategy has been proved to be valid both at an outer level as well as (in a complementary way) at an inner level, recursively applying the same strategy for the factorization of the panel that emerges as a critical task in the global inversion procedure.

By means of a thorough evaluation, we have demonstrated that, by selecting proper values for block and tile sizes, the proposed TP scheme yields a high core occupation, and its parallel performance excels that of 1) a blocked implementation of GJE (based on multi-threaded Intel MKL BLAS), 2) state-of-the-art dense linear algebra libraries (such as PLASMA), and 3) the commercial implementation of the LU-based matrix inversion procedure in Intel MKL.

The extraction of additional degrees of task parallelism will actually be more relevant as the number of cores increases in future architectures. Hence, it will be of wide appeal to test our implementations on massively parallel architectures, including the new-generation ARM servers with up to 96 cores per system. Additionally, the efficient exploitation of task- and data-parallelism for each specific architecture is also in our roadmap as a future research line.

The improvements of the latest versions of OpenMP regarding task parallelism open a plethora of new research lines that will be explored not only for the GJE-based matrix inversion routine, but also for other matrix factorization routines with similar features. Among these new functionalities, those improving tasking support are of wide interest to us:

- *Multidependency support using iterators,* that will ease and improve the efficiency of nested task generation and ease the management of complex data dependencies.
- *Task-to-data affinity clauses*, that may yield more efficient cache hierarchy exploitation and data locality on NUMA architectures for runtime-based task-parallel implementations. The evaluation of our implementations armed with task-to-data affinity hints and its comparison with NUMA-oblivious parallel BLAS implementation is kept as a future research line as soon as the future OpenMP runtimes offer mature support for this functionality.
- *Task priority support in OpenMP runtimes*, still immature or non-existent in modern implementations, will critically affect the overall performance of many matrix factorization implementations.
- *CUDA support within OpenMP*, when fully available, will facilitate the integration of GPUs within portable OpenMP codes, improving performance and gaining in portability. In this case, the use of heterogeneous block/tile size will become mandatory and impact a correct performance portability across heterogeneous architectures.
- *Recursive/nested task generation*, with a dynamic recursion depth depending on the instantaneous necessity of task parallellism, is also an interesting research line, both in symmetric or asymmetric multicore processors.

Acknowledgments

This research was sponsored by projects RTI2018-093684-B-I00 and TIN2017-82972-R of *Ministerio de Ciencia, Innovación y Universidades*; project S2018/TCS-4423 of Comunidad de Madrid; and project PR65/19-22445 of Universidad Complutense de Madrid.

References

[1] Anzt, H., Dongarra, J., Flegar, G., and Quintana-Ortí, E. S. Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors. *Parallel Computing* (2018).

[2] Badia, R. M., Herrero, J. R., Labarta, J., Pérez, J. M., Quintana-Ortí, E. S., and Quintana-Ortí, G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Conc. and Comp.: Pract. and Exper. 21* (2009), 2438–2456.

[3] Benner, P., Ezzatti, P., Quintana-Ortí, E. S., and Remón, A. Matrix inversion on CPU–GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience 25*, 8 (2013), 1170–1182.

[4] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35*, 1 (2009), 38 – 53.

[5] Castaldo, A. M., Whaley, R. C., and Samuel, S. Scaling LAPACK panel operations using parallel cache assignment. *ACM Trans. Math. Softw. 39*, 4 (July 2013).

[6] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., Yarkhan, A., Abalenkovs, M., Bagherpour, N., Hammarling, S., Šístek, J., Stevens, D., Zounon, M., and Relton, S. D. PLASMA: Parallel linear algebra software for multicore using OpenMP. *ACM Trans. Math. Softw. 45*, 2 (May 2019).

[7] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, 1996.

[8] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[9] Householder, A. S. *The Theory of Matrices in Numerical Analysis*. Dover, New York, 1964.

[10] OmpSs project home page. `http://pm.bsc.es/ompss`.

[11] PLASMA project home page. `http://icl.cs.utk.edu/plasma`.

[12] Quintana, E. S., Quintana, G., Sun, X., and van de Geijn, R. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22*, 5 (2001), 1762–1771.

[13] Quintana-Ortí, E. S., and van de Geijn, R. A. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software 35*, 2 (July 2008), 11:1–11:16.

[14] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Van Zee, F. G., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw. 36*, 3 (2009), 14:1–14:26.

[15] StarPU project. `http://runtime.bordeaux.inria.fr/StarPU/`.

[16] Strazdins, P. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[17] van der Pas, R., Stotzer, E., and Terboven, C. *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press, 2017.

[18] Van Zee, F. G. `libflame`*: The Complete Reference*. `www.lulu.com`, 2012.