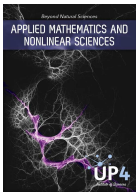
Applied Mathematics and Nonlinear Sciences 2(1) (2017) 201–212



Applied Mathematics and Nonlinear Sciences

http://journals.up4sciences.org

Parallel Solution of Hierarchical Symmetric Positive Definite Linear Systems

José I. Aliaga, Rocío Carratalá-Sáez[†], Enrique S. Quintana-Ortí.

Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I Castellón
Spain

Submission Info

Communicated by Juan L.G. Guirao
Received 6th March 2017
Accepted 22th June 2017
Available online 22th June 2017

Abstract

We present a prototype task-parallel algorithm for the solution of hierarchical symmetric positive definite linear systems via the $\mathscr{H}$-Cholesky factorization that builds upon the parallel programming standards and associated runtimes for OpenMP and OmpSs. In contrast with previous efforts, our proposal decouples the numerical aspects of the linear algebra operation from the complexities associated with high performance computing. Our experiments make an exhaustive analysis of the efficiency attained by different parallelization approaches that exploit either task-parallelism or loop-parallelism via a runtime. Alternatively, we also evaluate a solution that leverages multi-threaded parallelism via the parallel implementation of the Basic Linear Algebra Subroutines (BLAS) in Intel MKL.

Keywords: Hierarchical matrices, hierarchical arithmetic, Cholesky factorization, high performance, multicore processors.
AMS 2010 codes: 65F05, 68W10, 68M20.

1 Introduction

Hierarchical matrices (abbreviated as $\mathscr{H}$-matrices), in combination with $\mathscr{H}$-arithmetic, offer an efficient mathematical abstraction to deal with problems appearing in boundary element methods, elliptic partial differential operators or related integral equations [8]. The motivation is that $\mathscr{H}$-matrices can be leveraged to compress an $n \times n$ dense matrix using only $O(n\,c\log n)$ elements, where the parameter c can be tuned to control the accuracy of the approximation [7]. Furthermore, basic linear algebra operations, such as matrix addition and matrix-matrix multiplication, as well as more complex matrix factorizations, can all be (approximately) computed in $\mathscr{H}$-arithmetic with a cost of $O(n\log^d n)$ floating-point operations (flops), for a small constant d [6]. In addition, for large-scale instances of these problems, the approximation errors introduced by $\mathscr{H}$-arithmetic are often in the order of the discretization error.

[†]Corresponding author.
Email address: rcarrata@uji.es

ISSN 2444-8656 doi:10.21042/AMNS.2017.1.00017

Over the last years, there have been a number of research efforts to develop mature libraries of linear algebra operations for $\mathscr{H}$-matrices. Some of the most relevant cases have resulted in the research/teaching packages Hlib and H2Lib; and the commercial library HLibPro.[a] In principle, the routines in these software packages can (or be easily modified to) run in parallel on multicore architectures by linking in a multi-threaded implementation of the *Basic Linear Algebra Subprograms* (BLAS) [4] such as that in Intel MKL. More sophisticate parallel codes for $\mathscr{H}$-matrices exploit the task-parallelism implicit to linear algebra operations in order to orchestrate a parallel execution on a multicore processor [10]. However, these codes correspond to *ad-hoc* implementations that strongly couple the numerical method to the parallelization of the algorithm.

In this paper we pursue the development of a prototype task-parallel algorithm for the solution of hierarchical symmetric positive definite (SPD) linear systems via the $\mathscr{H}$-Cholesky factorization. Our approach departs from previous efforts in that we rely on the parallelization mechanisms/runtimes underlying two standard programming tools, namely OpenMP [12] and OmpSs [11]. Our proposal thus decouples the numerical aspects of the linear algebra operation, which are left in the hands of the expert mathematician, physicist or computational scientist, from the difficulties associated with high performance computing, which are more naturally addressed by computer scientists and engineers.

In [2] we presented a prototype parallel implementation of the $\mathscr{H}$-LU factorization for dense linear systems using OmpSs. In this paper we continue our effort towards the analysis and development of parallel algorithms for $\mathscr{H}$-linear algebra by targeting the Cholesky factorization. We select this particular decomposition because it corresponds to the most expensive computational stage for the solution of SPD linear systems. Furthermore, this type of decomposition presents a richer set of dependencies between the linear algebra building blocks that allows us to illustrate and discuss the difficulties and benefits of a runtime-assisted parallelizations for multicore architectures when compared with a simpler multi-threaded BLAS-based alternative. In comparison with [2], which was focussed on the implementation details, this paper makes a more exhaustive experimental analysis of the efficiency attained by different parallelization approaches.

The OmpSs programming model has been applied to extract task-parallelism for the LU factorization of dense matrices in [3], for the $\mathscr{H}$-LU factorization of $\mathscr{H}$-matrices in [2], and in iterative methods for sparse linear systems in [1]. In this paper we also adopt the OmpSs solution to exploit the implicit task-parallelism from the $\mathscr{H}$-Cholesky factorization. In addition, we analyze the use of the standard OpenMP in order to exploit either loop-parallelism or task-parallelism, highlighting the programming and performance differences. Moreover, we explore the limits of an alternative that simply extracts parallelism from within the BLAS kernels.

The rest of the paper is structured as follows. In Section 2 we provide a short review on the mathematical foundations of $\mathscr{H}$-matrices and $\mathscr{H}$-arithmetic. In Section 3 we visit the $\mathscr{H}$-Cholesky factorization, paying special attention to the data dependencies, and in Section 4 we describe several approaches to parallelize this decomposition. Finally, in Section 5 we assess the parallel performance of the different parallel implementation; and in Section 6 we summarize our insights.

2 Background on $\mathscr{H}$-Matrices

In this section, we briefly review a few basic concepts related to $\mathscr{H}$-matrices. For more details, see [7, 10]. We commence with the definition of a *cluster tree* for a given index set.

Definition 1. Let I be an index set with cardinality $n = \#I$. The graph $T_I = (V, E)$, with vertices V and edges E, is a *cluster tree* over I, if I is the root of T_I and, for all $v \in V$, either v is a leaf of T_I or $v = \dot{\cup}_{v' \in S(v)} v'$, where $S(v)$ denotes the set of direct descendents of v.

$\mathscr{H}$-matrices represent a hierarchical partitioning of an index set in the form of a *cluster tree*. Moreover, when partitioning the product index set $I \times I$ using the collection of subsets of I defined by T_I, we obtain block cluster trees over cluster trees.

[a] Visit https://github.com/gchavez2/awesome-hierarchical-matrices for a survey of software for $\mathscr{H}$-matrices.

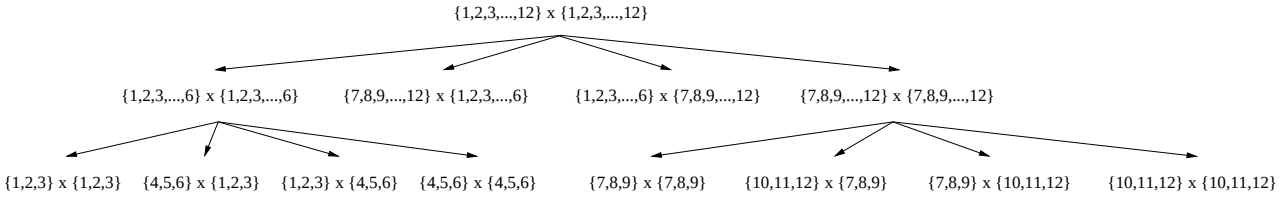


Fig. 1 Block cluster tree obtained from the index set $I = \{1,2,3,\ldots,12\}$.

Definition 2. Let T_I be a cluster tree over I and consider a node $b = p \times q$. The *block cluster tree* $T_{I\times I}$ over T_I can be defined recursively for the node b, starting with the root $I \times I$, as follows:

$$S(b) = \begin{cases} \emptyset & \text{if } b \text{ is admissible or } S(p) = \emptyset \text{ or } S(q) = \emptyset, \\ S' & \text{otherwise,} \end{cases},$$

where $S' := \{\ p' \times q' \ :\ p' \in S(p),\ q' \in S(q)\ \}$.

In order to obtain the hierarchy of blocks that defines the $\mathscr{H}$-matrix structure, an *admissibility* condition must be used, which ensures that a $p \times q$ admissible block D in the block cluster tree can be approximated by a low-rank factorization, up to a certain accuracy, as $D = PQ^T$, where P and Q are respectively $p \times c$ and $q \times c$ matrices, and $c \ll p,q$.

Figure 1 shows an example of a block cluster tree consisting of leaves that conform a partition of $I \times I$, with $I = \{1,2,3,\ldots,12\}$ using weak admissibility criteria; see [9] for details.

The collection of low-rank matrices with maximal rank k is known as the set of $\mathscr{H}$-matrices.

Definition 3. The set of $\mathscr{H}$-matrices for a block cluster tree $T_{I\times I}$ over a cluster tree T_I is defined as:

$$\begin{aligned}\mathscr{H}(T_{I\times I},k) := \{\ M \in \mathbb{R}^{I\times I} \mid \forall p\times q \in \mathscr{L}(T_{I\times I})\ :\\ \operatorname{rank}(M|_{p\times q}) \le k \vee \\ \{p,q\} \cap \mathscr{L}(T_I) \neq \emptyset\ \},\end{aligned}$$

where $\mathscr{L}(T_I)$ is the set of leaves of T_I and $k \in \mathbb{N}$.

Following with the example in Figure 1, the block cluster tree defined in that example yields the partitioning of a matrix of size 12×12 into the $\mathscr{H}$-matrix illustrated in Figure 2.

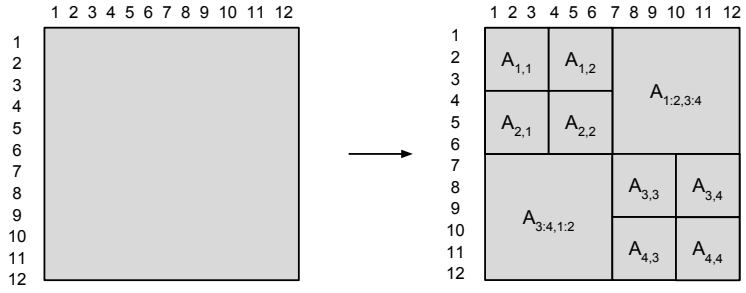


Fig. 2 $\mathscr{H}$-matrix obtained by applying the partitioning defined by the block cluster tree in Figure 1 over an 12×12 matrix.

The abstract concept underlying $\mathscr{H}$-matrices can be understood from the last definition: an $\mathscr{H}$-matrix can be regarded as a data-sparse representation of a dense matrix. Following this approach, the storage costs are reduced by exploiting the low-rank factorized representation of the data in some of the leaf nodes of the block cluster tree. (The rest of the leaf nodes are simply dense matrices of full or almost full rank and, therefore, there is little to be gained by representing them in factorized form.)

In addition, $\mathscr{H}$-matrices employ $\mathscr{H}$-arithmetic to compute matrix addition, inversion, multiplication and the LU factorization with log-linear computation costs; see [6]. For this purpose, the sum of low-rank matrices is truncated to rank c (or, preferably, with respect to a given precision ε) via the singular value decomposition (SVD) [5]. In other words, operations on $\mathscr{H}$-matrices (except the matrix-vector product) are approximative in order to attain a log-linear arithmetic complexity.

3 Cholesky Factorization of $\mathscr{H}$-Matrices

3.1 Cholesky factorization

In this section we initially provide a definition for the Cholesky factorization. Next, we present a blocked algorithm that will be later evolved in order to obtain a procedure for the decomposition of hierarchical matrices.

Definition 4. The Cholesky factorization of an SPD matrix $A \in \mathbb{R}^{n\times n}$ returns a lower triangular factor $L \in \mathbb{R}^{n\times n}$ such that $A = LL^T$. (Alternatively, the Cholesky factorization can deliver an upper triangular factor $U \in \mathbb{R}^{n\times n}$ such that $A = U^T U$, where $L = U^T$.)

There exist three algorithmic variants (procedures) to compute the Cholesky factorization [5]. We next review the blocked right-looking (BRL) procedure, which underlies the scheme to compute the $\mathscr{H}$-Cholesky factorization detailed later in this section.

Let us assume that the $n \times n$ SPD matrix A is partitioned into $n_t \times n_t$ blocks of dimension $t_s \times t_s$ each where, for simplicity, we consider that $n = n_t t_s$. This consideration simplifies the description of the algorithms next. However, our codes operate on blocks on any dimension and this consideration has no effect on the performance results later in the paper. Let us denote the blocks defined by this partitioning as A_{ij}, with $i, j \in \{1, 2, \ldots, n_t\}$. The BRL algorithm for the Cholesky factorization then commences by computing the factorization of the top-left block of matrix A:

$$A_{1,1} = L_{1,1} L_{1,1}^T.$$

Next, it solves $n_t - 1$ lower triangular systems (with t_s right-hand side vectors each) to obtain the first column of blocks of L:

$$L_{i,1} := A_{i,1} L_{1,1}^{-T}, \quad i = 2, 3, \ldots, n_t;$$

and then it performs the update of the trailing sub-matrix:

$$A_{i,j} := A_{i,j} - L_{i,1} \cdot L_{1,j}^T, \quad i = 2, 3, \ldots, n_t;\ j = 2, 3, \ldots, i.$$

Note that, because of the symmetry of the matrix, it is not necessary to compute the first row of blocks of L nor update those blocks in the upper triangular part of A. Therefore, these operations complete the first iteration of the BRL algorithm, and the process is then (recursively) repeated by computing the Cholesky factorization of the trailing sub-matrix $A_{2:n_t,2:n_t}$; see Algorithm 1. A practical implementation of this algorithm overwrites the entries in the lower triangular part of A with those of L.

An inspection of the operations that compose the BRL procedure exposes a collection of data dependencies between the blocks of A. For example, $L_{2,1}$ cannot be computed until the factorization of $A_{1,1}$ is ready. Similarly, $A_{2,2}$ cannot be updated until $L_{2,1}$ has been computed. Figure 3 illustrates the data dependencies for the operations that compose the first iteration of the Cholesky factorization applied to a 4×4 (blocked) matrix. In addition to these dependencies, there exist others connecting the operations of the first iteration of loop k to those appearing in subsequent iterations of the procedure. For example, at the beginning of the second iteration $A_{2,2}$ cannot be factorized until it has been updated with respect to $L_{2,1}$ during the first iteration.

The BRL algorithm for the Cholesky factorization specifies a unique ordering of the operations, different from that dictated by the two other algorithmic variants for this factorization [5]. However, this schedule can be modified, provided the data dependencies are fulfilled, while still computing the same result. For example,

Algorithm 1 BRL algorithm for the Cholesky factorization.

Require: $A \in \mathbb{R}^{n \times n}$
1: **for** $k = 1, 2, \ldots, n_t$ **do**
2: $A_{kk} = L_{kk} L_{kk}^T$
3: **for** $i = k+1, k+2 \ldots, n_t$ **do**
4: $L_{ik} := A_{ik} L_{kk}^{-T}$
5: **end for**
6: **for** $i = k+1, k+2, \ldots, n_t$ **do**
7: **for** $j = k+1, k+2, \ldots, i$ **do**
8: $A_{ij} := A_{ij} - L_{ik} \cdot L_{kj}^T$
9: **end for**
10: **end for**
11: **end for**

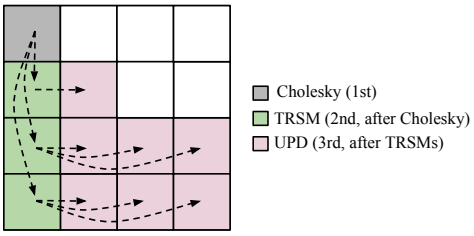


Fig. 3 Operations and data dependencies in the first iteration ($k = 1$) of the BRL algorithm for the Cholesky factorization.

the solution of the $n_t - 1$ lower triangular systems for $L_{i,1}$, $i = 2, 3, \ldots, n_t$, during the first iteration of the BRL algorithm, can be obtained in any order, as these operations are independent from each other.

3.2 $\mathscr{H}$-Cholesky factorization

We next present a procedure for the $\mathscr{H}$-Cholesky factorization that is obtained from a specialization of the BRL algorithm that takes into account the hierarchical structure of A. For this purpose, assume that $A \in \mathbb{R}^{n \times n}$ is a hierarchical SPD matrix, partitioned as in Figure 2, and consider that all blocks in that partitioning are dense. Then, the following sequence of operations O1–O10 computes the $\mathscr{H}$-Cholesky factorization of A:

Sequence of operations for the $\mathscr{H}$-Cholesky factorization of A:	
O1 : $A_{1,1}$	$= L_{1,1} L_{1,1}^T$
O2 : $L_{2,1}$	$:= A_{2,1} L_{1,1}^{-T}$
O3 : $A_{2,2}$	$:= A_{2,2} - L_{2,1} \cdot L_{2,1}^T$
O4 : $A_{2,2}$	$= L_{2,2} L_{2,2}^T$
O5 : $L_{3:4,1:2}$	$:= A_{3:4,1:2} L_{1:2,1:2}^{-T}$
O6 : $A_{3:4,3:4}$	$:= A_{3:4,3:4} - L_{3:4,1:2} \cdot L_{3:4,1:2}^T$
O7 : $A_{3,3}$	$= L_{3,3} L_{3,3}^T$
O8 : $L_{4,3}$	$:= A_{4,3} L_{3,3}^{-T}$
O9 : $A_{4,4}$	$:= A_{4,4} - L_{4,3} \cdot L_{4,3}^T$
O10 : $A_{4,4}$	$= L_{4,4} L_{4,4}^T$

The operations in this sequence correspond to three basic linear algebra building blocks (or computational kernels):

- Cholesky factorization: O1, O4 O7, O10;
- triangular system solve with transposed lower triangular factor: O2, O5, O8; and

- matrix-matrix multiplication, which can be specialized in the form of symmetric rank-t_s updates: O3, O6, O9.

These kernels are also present in the BRL Cholesky factorization procedure introduced in Algorithm 1. The only difference is that some of the kernels in the $\mathscr{H}$-Cholesky factorization involve operands of different sizes.

Figure 4 provides a graphical representation of the $\mathscr{H}$-Cholesky factorization process and the corresponding data dependencies between blocks. The five plots in the figure are to be read from left to right, starting with those in the top row first. They respectively correspond to the groups of operations:

$$\mathsf{O1}-\mathsf{O3} \mid \mathsf{O4} \mid \mathsf{O5}-\mathsf{O6} \mid \mathsf{O7}-\mathsf{O9} \mid \mathsf{O10}.$$

that appeared separated with horizontal lines in the sequence of operations for the $\mathscr{H}$-Cholesky factorization listed above. In the figure, the top-left matrix specifies the sequence of dependencies O1→O2→O3; and the top-right matrix corresponds to the implicit dependency O5→O6. For simplicity, some of the data dependencies are not shown, but they can be easily derived from the context.

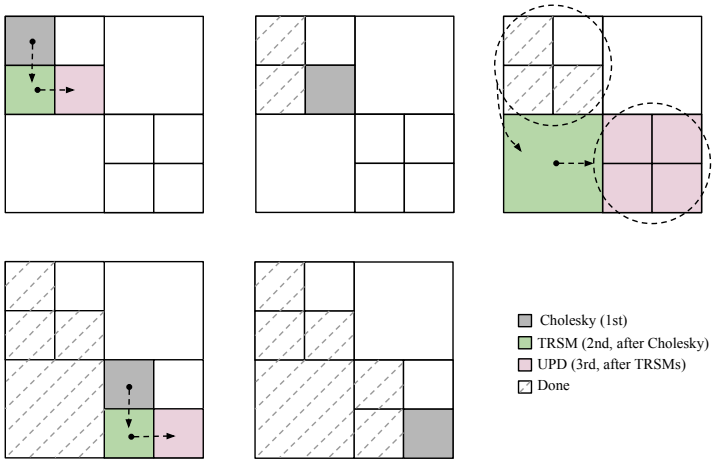


Fig. 4 Operations and data dependencies in the generalization of the BRL algorithm for the $\mathscr{H}$-Cholesky factorization.

We note that if any of the matrix blocks of A is compressed in low-rank factorized form, as is natural in for an $\mathscr{H}$-matrix, the operations involving this block will have to be performed in the appropriate $\mathscr{H}$-arithmetic. In any case, the dependencies remain the same. Conversely, in case a matrix block only contains zeros, some of the operations may become unnecessary. For example, if $A_{2,1}$ is "null" (that is $A_{2,1} \equiv 0$), then $L_{2,1} \equiv 0$, and O2, O3 do not need to be computed. Furthermore, in that case O1 and O4 can be computed in parallel as there exist no dependency connecting them.

4 Parallel $\mathscr{H}$-Cholesky factorization

In this section we describe three approaches for the parallel computation of the $\mathscr{H}$-Cholesky factorization. For this purpose, we will employ the BRL procedure for the Cholesky factorization in Algorithm 1 as a case study. Generalizing the ideas presented next carry over to the hierarchical case in a straight-forward manner.

4.1 Multi-threaded BLAS

The BRL algorithm decomposes the Cholesky factorization into a collection of four types of basic computational kernels operating on the matrix blocks: Cholesky factorization (of a diagonal block A_{kk}), triangular solve (line 4), symmetric rank-t_s update (line 8, when $i = j$), and a general matrix-matrix multiplication (line 8, when

$i \neq j$). Efficient implementations of these four building blocks are provided in vendor as well as open-source libraries for dense linear algebra such as Intel MKL, NVIDIA cuBLAS, IBM ESSL, GotoBLAS, OpenBLAS and BLIS. Most of these implementations are also multi-threaded yielding a direct parallel execution on a multicore architecture by simply linking any of them to the application that invokes these kernels. In particular, this is the case of an implementation of the BRL procedure (or its hierarchical generalization) that relies on a multi-threaded instance of the BLAS to perform these computations.

This first approach, based on a multi-threaded BLAS and transparent to the programmer (as it does not require any change in the original code), will only deliver a substantial fraction of the machine peak performance in case the matrix blocks involved in the operations performed inside the BLAS are sufficiently large with respect to the number of cores of the platform. Unfortunately, for hierarchical matrices, many of the blocks exhibit a reduced dimension, constraining the parallel efficiency of this simple approach.

4.2 Loop-parallelism with independent iterations

The BRL algorithm consists of three nested loops, indexed by variables k, j, i. Among them, it was already argued that the triangular solves inside the loop in lines 3–5 are independent of each other; and the same property applies to the operations comprised by nested loops in lines 6–10. This implies that a direct approach to extract loop-parallelism from this code can employ OpenMP to parallelize these two cases using a *parallel for* directive. When applied to the first case, for example, this means that the iterations of loop j are distributed among the threads in charge of executing the code in parallel. The type of distribution can be adjusted via specific clauses. However, given that all iterations of the loop have the same theoretical cost (and can be expected to contribute a similar practical cost), a simple static schedule will be, in general, appropriate.

The parallelization of the second case, corresponding to the loops in lines 6–10, is slightly more complex because of the nested organization of the loops. Here, these loops cannot be tackled by using a collapse clause that internally transforms them into a single loop, because of the dependency between the counter of the innermost loop on the outermost one (index j iterates from $k+1$ till i.) A solution is to parallelize the outermost loop only, but tune this with an appropriate scheduling clause that enforces a fair distribution of the workload among the threads.

In summary, this second approach provides an easy-to-program strategy to parallelize the BRL procedure (as well as the corresponding generalization to hierarchical matrices). However, the performance will depend on the number of iterations of the loops and may be constrained because, at execution, each loop parallelized in this manner implicitly requires a synchronization point for the threads upon completion.

4.3 Dependency-aware task-parallelism

The third parallelization approach is the most sophisticated strategy but also the option that can be expected to render higher performance. The idea is to rely on the "sequential" BRL procedure in order to identify the real data dependencies between the operations of the algorithm. Each one of these operations becomes then a task/node and the dependencies act as vertices connecting pairs of nodes of a task dependency graph (TDG). This graph defines a partial order that specifies multiple correct orderings for the operations. Concretely, it depends only on the actual dependencies of the mathematical operation but not on the algorithm itself. For example, when applied to all three known variants for the computation of the Cholesky factorization, the result is the same TDG. In other words, the execution of the operations comprised by the Cholesky factorization, executed in any order that fulfills the dependencies registered in the TDG, will produce the same correct solution as a sequential run (except for small variations due to the use of finite-precision arithmetic and rounding error).

This third approach aims to maximize the degree of concurrency exposed to the hardware, which offers multiple (correct) alternatives in the sequencing of the operations involved by the factorization. In principle, exploiting this type of parallelization approach may seem complex, because it requires to encode the algorithm, discover the data dependencies between operations (preferably, at execution time), and keep track of the order in which they are issued for execution. Fortunately, this burden can be greatly alleviated by employing a

n	n_l	Block granularity in each level
5,000	2	5,000, 100
	3	5,000, 500, 100
	4	5,000, 2,500, 1,250, 250
10,000	2	10,000, 500
	3	10,000, 500, 100
	4	10,000, 1,000, 500, 100

Table 1 Configurations for the experimental evaluation of the $\mathscr{H}-Cholesky$ factorization.

proper parallel programming model, assisted by a runtime with support for task-parallelism. This is the case of OmpSs, which was originally conceived to exploit this type of parallelism, and also for releases 3.0 and higher of OpenMP. The mechanism to automatically detect dependencies, the implications from the point of view of decomposing tasks into finer-grain sub-operations, and the consequences of the data storage layout are all discussed in [2], which was focussed on the implementations details.

5 Experimental results

All the experiments in this section were performed using IEEE double precision arithmetic, on a server equipped with two Intel E5-2603v3 sockets, each with a 6-core processor (1.6 GHz), and 32 Gbytes of DDR3 RAM. Our codes were linked with Intel MKL (`composer_xe_2011_sp1`) for the BLAS kernels and the Cholesky factorization, and OmpSs (version 16.06).

As argued at the beginning of this paper and following previous works (see [2]), our goal is to expose the performance benefits of leveraging a task-parallel programming model such as OmpSs and OpenMP for the solution of linear algebra operations on $\mathscr{H}$-matrices. In contrast, we do not aim to develop a mature library that competes with other implementations. For this reason, the experiments in this section are designed to assess the scalability of our codes, in a simplified yet practical scenario.

The $\mathscr{H}$-matrices used for the experimental analysis comprise dense and null blocks, but present no low-rank ones. (In any case, including this last type of blocks has no effect on the task-based parallelization effort but requires special numerical kernels that change the implementation and costs of the tasks operating on them.) Concretely, for the evaluation of the parallel implementations, we use two SPD $\mathscr{H}$-matrices of dimensions $n = 5,000$ and $n = 10,000$ with random entries following a normal distribution in (0,1). To avoid numerical difficulties, the matrices were enforced to be diagonally dominant. Moreover, for each $\mathscr{H}$-matrix we varied the number of levels (n_l) and the granularity of the blocks in each level, as displayed in Table 1. Finally, we performed experiments for four ratios of null blocks (dispersion): 0% (full matrix), 25%, 50% and 75%, which specify the number of blocks that are null compared to the total amount of blocks. Note that a higher ratio of null blocks does not necessarily represent a sparser matrix.

Figures 5 and 6 report the GFLOPS (billions of floating point operations per second) rates attained by the different parallel approaches for each of the matrix configurations on the Intel server using 4, 8 and 12 threads/cores. These performance rates take into account that some of the blocks may be null when calculating the actual flops necessary to factorize each matrix. A higher GFLOPS rate thus indicates higher performance/shorter execution time. The parallel variants evaluated there include:

- MKL Multithread: Parallel algorithm that exploits parallelism inside the BLAS kernels invoked during the execution of the BRL variant of the hierarchical Cholesky factorization; see subsection 4.1.
- OpenMP - Simple: Loop-parallel algorithm that exploits the loop-parallelism present in the BRL variant of the hierarchical Cholesky factorization; see subsection 4.2.

- OmpSs and OpenMP - Tasks: Task-parallel algorithms that exploit the parallelism defined by the TDG associated with the hierarchical Cholesky factorization; see subsection 4.3.

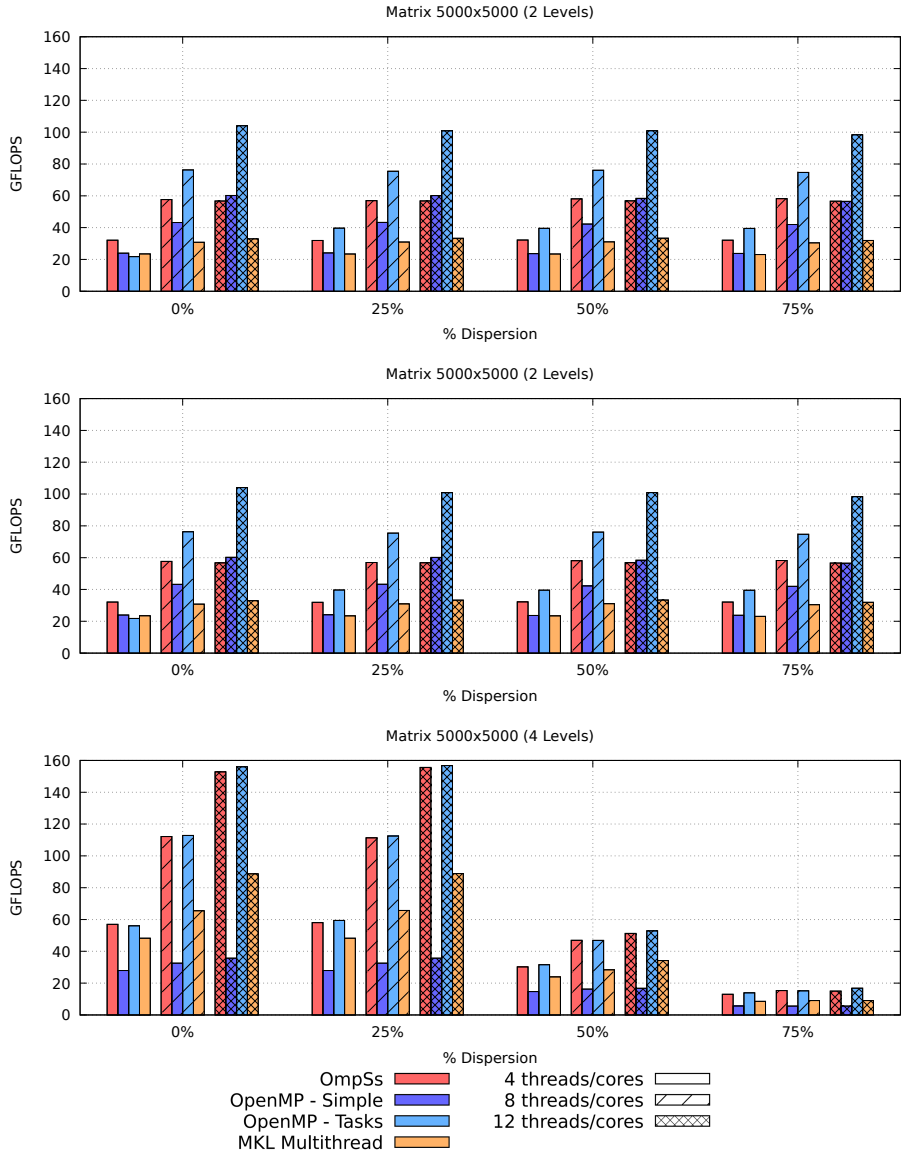


Fig. 5 Performance of the task-parallel $\mathscr{H}$-Cholesky factorization of a matrix of order 5,000 using OpenMP and OmpSs in the Intel E5-2603v3 server using 4, 8 and 12 threads/cores.

The results of this evaluation offer some general conclusions:

- The GFLOPS rates observed with the four parallelization approaches grow with the number of cores for all tested configurations: problem size, number of levels of the hierarchical matrix, and sparsity (dispersion); but there are notable differences depending on the specific approach.

- In general, the task-parallel versions (OmpSs and OpenMP - Tasks) outperform OpenMP - Simple or MKL Multithread variants. Furthermore, when the number of tasks is small, the OpenMP task-parallel implementation offers higher parallel performance than the task-parallel version that relies OmpSs. For larger matrices and, when the number of tasks grows, the differences between the two task-parallel solu-

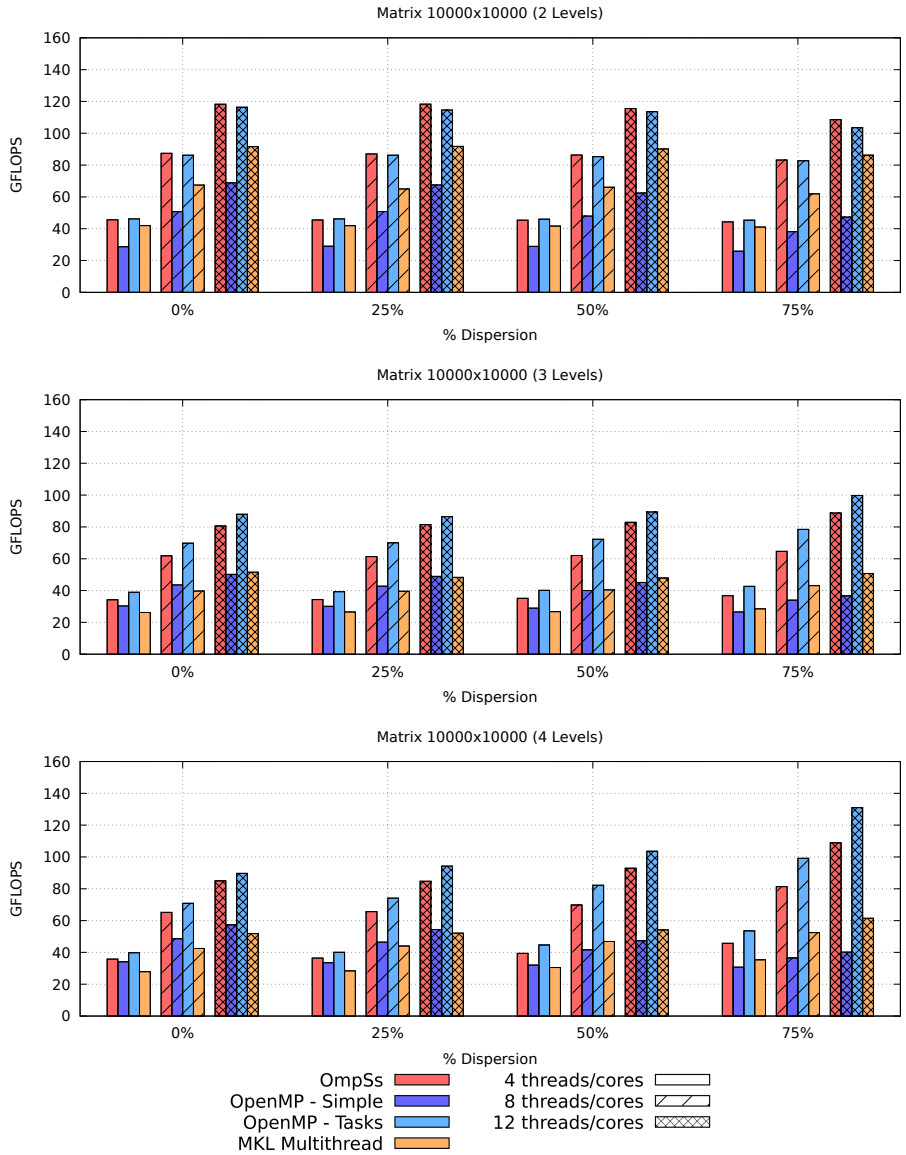


Fig. 6 Performance of the task-parallel $\mathscr{H}$-Cholesky factorization of a matrix of order 10,000 using OpenMP and OmpSs in the Intel E5-2603v3 server using 4, 8 and 12 threads/cores.

tions is reduced.

- In most cases the parallel performance is slightly reduced as the amount of null blocks is increased in comparison to denser configurations.

6 Concluding remarks

We have developed and evaluated several parallel algorithms for the solution of hierarchical SPD linear systems, via the $\mathscr{H}$-Cholesky factorization, on multicore architectures. In particular, our algorithms explore the benefits of extracting concurrency from within a multi-threaded implementation of the BLAS; from the loops present in the BRL variant of the Cholesky factorization; or from the tasks that appear when decomposing this operation via a task-parallel implementation with the support of a parallel programming model as those

behind OmpSs and OpenMP. Our results clearly demonstrate that this third option, when combined with the runtimes underlying OmpSs/OpenMP, clearly outperforms the somehow trivial approaches that target multi-threaded BLAS and loop-parallelism.

As part of future work, we intend to analyze the possibility of extracting concurrency from a combination of two or more of these complementary mechanisms, and to shift our parallelization effort to work on a mature library for $\mathscr{H}$-matrices.

Acknowledgments

This work was supported by project TIN2014-53495-R of MINECO and FEDER. Rocío Carratalá-Sáez is supported by a fellowship grant of the FPU program of MINECO.

References

[1] José I. Aliaga, Rosa M. Badia, María Barreda, Matthias Bollhöfer, and Enrique S. Quintana-Ortí. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned cg method. In *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2014)*, pages 262–269, 2014.

[2] José I. Aliaga, Rocío Carratalá-Sáez, Ronald Kriemann, and E. S. Quintana-Ortí. Task-parallel LU factorization of hierarchical matrices using OmpSs. In *Proceedings of the 19th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2017*, 2017. To appear.

[3] Rosa M. Badia, Jose R. Herrero, Jesus Labarta, Jose M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21:2438–2456, 2009.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. on Mathematical Software*, 16(1):1–17, March 1990.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[6] Lars Grasedyck and Wolfgang Hackbusch. Construction and arithmetics of $\mathscr{H}$-matrices. *Computing*, 70(4):295–334, August 2003.

[7] Wolfgang Hackbusch. A sparse matrix arithmetic based on $\mathscr{H}$-matrices. part i: Introduction to $\mathscr{H}$-matrices. *Computing*, 62(2):89–108, May 1999.

[8] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49 of *Springer Series in Computational Mathematics*. Springer-Verlag Berlin Heidelberg, 2015.

[9] Mohammad Izadi. *Hierarchical matrix techniques on massively parallel computers*. Ph.D. dissertation, Universität Leipzig, 2012.

[10] Ronald Kriemann. $\mathscr{H}$-LU factorization on many-core systems. *Computing and Visualization in Science*, 16(3):105–117, 2013.

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

[12] The OpenMP API specification for parallel programming. http://www.openmp.org/.