Tiled Algorithms for Efficient Task-Parallel H-Matrix Solvers

Rocío Carratalá-Sáez, Mathieu Faverge, Grégoire Pichon, Guillaume Sylvand, Enrique Quintana-Ortí

Tiled Algorithms for Efficient Task-Parallel $\mathcal{H}$-Matrix Solvers

Rocío Carratalá-Sáez*, Mathieu Faverge†, Grégoire Pichon‡, Guillaume Sylvand§, Enrique S. Quintana-Ortí¶
*Universitat Jaume I, Castelló de la Plana, Spain, rcarrata@uji.es
†Bordeaux INP, CNRS,Inria, Univ. Bordeaux, Talence, France, mathieu.faverge@inria.fr
‡Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, Lyon, France, gregoire.pichon@univ-lyon1.fr
§Airbus Central R&T / Inria, Bordeaux, France, guillaume.sylvand@airbus.com
¶Universitat Politècnica de València, València, Spain, quintana@disca.upv.es

***Abstract*—In this paper, we describe and evaluate an extension of the CHAMELEON library to operate with hierarchical matrices ($\mathcal{H}$-Matrices) and hierarchical arithmetic ($\mathcal{H}$-Arithmetic), producing efficient solvers for linear systems arising in Boundary Element Methods (BEM). Our approach builds upon an open-source $\mathcal{H}$-Matrices library from Airbus, named HMAT-OSS, that collects sequential numerical kernels for both hierarchical and low-rank structures; the tiled algorithms and task-parallel decompositions available in CHAMELEON for the solution of linear systems; and the STARPU runtime system to orchestrate an efficient task-parallel (multi-threaded) execution on a multicore architecture.**

Using an application producing matrices with features close to real industrial applications, we present shared-memory results that demonstrate a fair level of performance, close to (and sometimes better than) the one offered by a pure $\mathcal{H}$-Matrix approach, as proposed by Airbus HMAT proprietary (and non open-source) library. Hence, this combination CHAMELEON + HMAT-OSS proposes the most efficient fully open-source software stack to solve dense compressible linear systems on shared memory architectures (distributed memory is under development).

***Index Terms*—Hierarchical matrices, LU factorization, linear systems, task-parallelism, multicore processors**

I. INTRODUCTION

Many real-life simulations require handling large collections of data together with a considerable number of computations. Solving linear systems of equations is often a key kernel in these applications, and usually represents one of the most expensive operations, both in terms of memory usage and computational cost. The matrices that realize these systems present considerable structural variations, covering a wide spectrum of different configurations, from dense to sparse, depending on the underlying simulation they represent. Thus, reducing the cost of this operation is interesting in order to 1) solve larger problems, and 2) reduce the time-to-solution as well as other associated costs (e.g., memory or energy).

Since their introduction [1], [2], hierarchical Matrices ($\mathcal{H}$-Matrices) have gained substantial momentum in Boundary Element Methods (BEM) and elliptic partial differential operators [3]. In particular, thanks to the combination of dense and low-rank blocks in a nested structure, $\mathcal{H}$-Matrices nowadays provide a powerful numerical tool to reduce storage costs and execution time significantly [2]. For example, the LU Factorization of an $n \times n$ $\mathcal{H}$-Matrix ($\mathcal{H}$-LU) requires $\Theta(n\ k^2\ log^2\ n)$ floating-point operations (flops) in $\mathcal{H}$-Arithmetic, where the parameter k can be tuned to control the accuracy of the approximation (that is, it establishes the local rank for the $\mathcal{H}$-Matrix subblocks). In contrast, the same factorization costs $\Theta(\frac{2}{3}n^3)$ flops in the dense case.

Improving the performance of $\mathcal{H}$-Arithmetic operations is an active area of research that has recently produced a fair number of libraries as well as many interesting algorithmic developments (see Section III). These research efforts are surely motivated by the relevance of the applications that can be efficiently tackled with $\mathcal{H}$-Matrices, but also because of the complexity and benefits of $\mathcal{H}$-Arithmetic. On the one hand, $\mathcal{H}$-operations require dealing with both low-rank and dense blocks – which often implies re-compressing some of the intermediate results – while following a nested structure, and usually a recursive algorithm. On the other hand, in general, the definition and storage of $\mathcal{H}$-Matrices implies complex data accesses. This fact promoted the appearance of alternative structures, such as Block Low-Rank (BLR) [4]–[6] and lattice $\mathcal{H}$-Matrices [7], [8], that trade off slightly higher time and memory costs in exchange for superior simplicity. One asset of these approaches it that it is easier to exploit parallelism, as they present more regular structures.

In this paper we combine existing efficient numerical kernels for hierarchical, low-rank and full-rank matrices, together with an efficient task-based implementation designed to solve dense linear systems. As we will show, this allows us to leverage modern programming models and runtime systems yielding to the single open-source solution that achieves fair parallel performance, while avoiding re-implementing pure $\mathcal{H}$-Arithmetic. Concretely, our work integrates the following three components:

- For $\mathcal{H}$-Arithmetic, we choose the HMAT-OSS [9] kernels to operate with low-rank blocks, as they have been proved to offer fair efficiency in industrial applications [10]. This is a library for $\mathcal{H}$-Matrices maintained by Airbus, whose public version is sequential.
- In addition, for the task-based implementation of matrix operations, we leverage the CHAMELEON [11]–[13] library, a dense linear algebra package that relies on the sequential task flow programming model to schedule tiled algorithms on top of multiple runtime systems such as

OpenMP [14], PaRSEC [15], StarPU [16], [17], or Quark [18].
- Finally, we focus on the specific StarPU runtime system [17] support of the Chameleon library to exploit the task-parallelism in our solution. StarPU is in charge of issuing the tasks to the systems cores while fulfilling inter-task dependencies.

Our strategy to re-utilitze structures that are similar to $\mathcal{H}$-Matrices while reducing their complexity is close to what is referred to as "lattice $\mathcal{H}$-Matrices" in [7]. As opposed to that work, in our case, we need to split the initial matrix into regular tiles (i.e., tiles of the same size) to meet the requirements of Chameleon algorithms requirements, and each of these tiles are individually turned into $\mathcal{H}$-Matrices. We will refer to our structure as "Tile $\mathcal{H}$-Matrix".

The rest of the paper is structured as follows: in Section II we present in brief details about $\mathcal{H}$-Matrices, the LU factorization, the $\mathcal{H}$-LU algorithm, and its challenges; in Section III we present some related work, such as existing libraries, different compressed structures approaches, and runtime strategies to tackle $\mathcal{H}$-Arithmetics. In Section IV we describe $\mathcal{H}$-Chameleon, giving details about the libraries that conform the basis of this work, our approach to represent the data, the original kernels from Chameleon and Hmat-oss we leverage, and how we construct the matrices. In Section V we describe the test case employed in the subsequently presented experiments, as well as the target platform in which we perform the executions. Finally, in Section VI we expose the main remarks and conclusions we extract from this work.

II. Solution of $\mathcal{H}$-Matrix Linear Systems

We open this section with a concise review of $\mathcal{H}$-Matrices and $\mathcal{H}$-Arithmetic. More details can be found in [1]–[3]. Then, we detail the principle of the LU factorization for $\mathcal{H}$-Matrices and how it can be parallelized with modern task based runtime systems.

A. A brief introduction to $\mathcal{H}$-Matrices

From a theoretical perspective, $\mathcal{H}$-Matrices are built upon block cluster trees that represent a hierarchical partitioning of the column and row index sets of the original matrix. Mathematically, this is summarized in Definitions 1 and 2.

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 sons of v. Then, considering the node $b = p \times q$ of the cluster tree T_I, 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:

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

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

The admissibility condition applied to decide whether to re-partition a block ensures that an admissible block (that is, one that is not re-partitioned) can be approximated by a low-rank block. Mathematically, $\mathcal{H}$-Matrices are equivalent to a collection of low-rank blocks with a maximal rank k, as exposed in Definition 2.

Definition 2: The set of $\mathcal{H}$-Matrices for a block cluster tree $T_{I\times I}$ over a cluster tree T_I is defined as:

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

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

In practice, most of the operations involving $\mathcal{H}$-Matrices need to truncate the rank of low-rank blocks via the Singular Value Decomposition (SVD) [19] (cf Section II-B), in order to maintain the log-linear arithmetic complexity. Alternative solutions exist to reduce the cost of this operation by computing approximated low-rank compression such as Adaptive Cross Approximation (ACA) [20] or randomized techniques [21].

B. LU factorization of general and $\mathcal{H}$-Matrices

The LU factorization is an extremely useful decomposition for the solution of linear systems that, given a non singular matrix $A \in \mathbb{R}^{n\times n}$ (for the coefficients of the linear system), returns a unit lower triangular factor $L \in \mathbb{R}^{n\times n}$, and an upper triangular factor $U \in \mathbb{R}^{n\times n}$, such that $A = LU$.

There exists different variants of the algorithm to compute the LU factorization of a full-rank matrix. Over the years, the factorizations have evolved from blocked algorithms working by panels (LAPACK [22]), to the modern and now common tiled-algorithms, used in many libraries exploiting runtime systems to schedule the computation on the target architecture.

Considering a matrix A of $n_t \times n_t$ tiles/blocks with each block $A_{ij} \in \mathbb{R}^{n_b\times n_b}$, the tiled LU factorization algorithm can be performed following the blocked Right-Looking (RL) variant [19], which is exposed in Algorithm 1. It consists of an outer loop that computes, at each iteration k, the LU factorization of the corresponding diagonal block A_{kk} (`xGETRF` LAPACK operation); next it computes the k-th block column panel (from L) and the k-th block row panel (from U) by solving several triangular systems (`TRSMs`); and then it updates (via the matrix-matrix product or `GEMM`) the remaining blocks of A with respect to the block panels of L and U calculated in this iteration.

The task-based version of Algorithm 1 identifies each `GETRF`, `TRSM` and `GEMM` kernel as a task, which can be enriched by keywords to specify the inputs and outputs of each kernel. With this additional information and the sequential execution of this algorithm, runtime systems are able to automatically infer data dependencies between the tasks and generate a directed acyclic graph (DAG) of the tasks in the algorithm. Each node refers to a task, and each edge to a data dependency. The DAG of a full-rank LU factorization for a tiled matrix of 3×3 is illustrated in Figure 1.

When moving to $\mathcal{H}$-Matrices, the algorithm for the $\mathcal{H}$-LU factorization can be directly constructed from Algorithm 1 by adding recursion into the kernels. The algorithm is applied

Algorithm 1 Tiled GETRF algorithm

Require: $A \in \mathbb{R}^{n \times n}$ with $n = n_t * n_b$
1: **for** $k = 1, 2, \ldots, n_t$ **do**
2: $(L_{kk}, U_{kk}) \leftarrow$ GETRF(A_{kk})
3: **for** $j = k+1, k+2, \ldots, n_t$ **do**
4: $U_{kj} \leftarrow$ TRSM(Left, Lower, Unit, L_{kk}, A_{kj})
5: **end for**
6: **for** $i = k+1, k+2, \ldots, n_t$ **do**
7: $L_{ik} \leftarrow$ TRSM(Right, Upper, NonUnit, U_{kk}, A_{ik})
8: **end for**
9: **for** $i = k+1, k+2, \ldots, n_t$ **do**
10: **for** $j = k+1, k+2, \ldots, n_t$ **do**
11: $A_{ij} \leftarrow$ GEMM(L_{ik}, U_{kj}, A_{ij})
12: **end for**
13: **end for**
14: **end for**

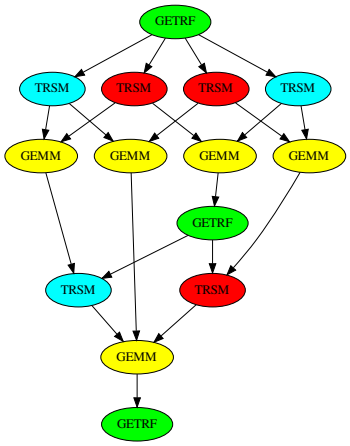

Fig. 1: DAG of the tasks that conform a full-rank LU factorization (following Algorithm 1) for a tiled matrix of 3×3 tiles.

on the first level of the hierarchy of the $\mathcal{H}$-Matrix where the partitioning discussed previously defines the number of tiles. Then, the three kernels: `GETRF`, `TRSM` and `GEMM` are replaced by their $\mathcal{H}$-Arithmetic versions, which recursively apply the tile algorithm for the partition of the lower level.

In the case of `H-GETRF`, this consists simply in recursively calling the Algorithm 1 in line 2, as long as the considered block is an $\mathcal{H}$-Matrix. When reaching a leaf, the traditional LAPACK kernel is called.

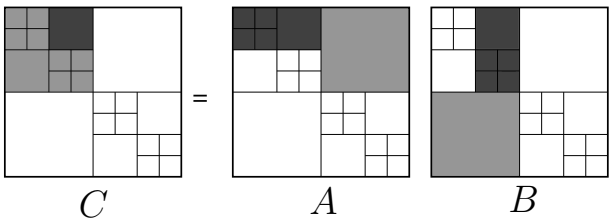

Fig. 2: Example of `H-GEMM` configuration. The dark gray part of C is updated by contributions coming from both light gray and dark grays parts of B and C, with some of these blocks being full rank and others being low rank.

The `H-TRSM` and `H-GEMM` are a little more complex, as they may involve matrices with different $\mathcal{H}$-Structure. In the case of `H-GEMM`, with 3 matrices involved and 3 possible formats for each (low rank, full rank or subdivided), 27 different configurations exist. The simplest case occurs when all three matrices are subdivided: we then progress in the $\mathcal{H}$-Matrix structure to the lower level. All the remaining cases have to be handled individually. Figure 2 illustrates the type of complex configuration that must be dealt with.

In this work, as detailed in section IV-A, we chose to rely on an existing implementation of the $\mathcal{H}$-Matrix algorithm in order to skip this hassle: concretely, we picked the HMAT-OSS [9] library, an open source sequential package for $\mathcal{H}$-Arithmetic. Hence, in Algorithm 1, from CHAMELEON's point of view, the tiles manipulated become $\mathcal{H}$-Matrices created by HMAT-OSS, and the calls to LAPACK `GEMM`, `TRSM` and `GETRF` are replaced by calls to the HMAT-OSS implementations of `H-GEMM`, `H-TRSM` and `H-GETRF`.

III. Related Work

Over the past few decades, several libraries have been made available for $\mathcal{H}$-Matrices and $\mathcal{H}$-Arithmetic. Starting from sequential approaches, shared memory or distributed memory versions have been proposed, as well as some solutions to exploit GPUs. Those works have demonstrated that implementing a fast and efficient library for general $\mathcal{H}$-Matrix format is not an easy task, and as a consequence they often proposed solutions to simplify the problem.

We focus here on the libraries implemented with the task-based paradigm model on top of runtime supports. The key feature of the $\mathcal{H}$-Matrices resides in the recursive structure of the matrix that both saves flops and memory. However, none of the modern runtime systems for task-based algorithms propose simple and easy to use features to describe recursive algorithms. Many works have thus explored different techniques to bear with this issue, starting by exploiting OpenMP task to increase the performance for multicore architectures [23]–[25]. These solutions realized a *bulk-synchronous* parallelism that was limited by synchronizations at each level of the $\mathcal{H}$-Structure. This was eventually extended to an MPI+OpenMP version in [26].

To alleviate the limitations of bulk-synchronous parallelism, [10] proposed an implementation of the HMAT library on top of the STARPU runtime system by enumerating all the required dependencies for each submitted task. One drawback of this solution, is that it may end up with a very large number of dependencies that the runtime system must deal with. Furthermore, in a distributed memory environment, it becomes even more difficult to track all dependencies and the runtime system struggles to deliver scaling performances. In H2Lib [27], HLibPro [28] and libHLR [29], the authors have been working on a semi-automatic DAG generation to generate this set of dependencies while being able to trim the DAG from unnecessary edges [30]. In [31], the authors introduce fake dependencies while submitting the DAG similar to *anchors* they attach to only if needed. This helps them to recursively

introduce the missing dependencies, while not overloading the runtime systems with too many of them. Similarly to this work, the authors proposed in [32] an implementation of H2lib on top of OmpSs, where a fair parallel performance is achieved thanks to novel features included in OmpSs-2 programming model, such as weak dependencies and early release. These features have been made available through the Nanos++ [33] runtime, which extends OpenMP.

Other approaches have looked into different compression formats that could better fit the task-based programming model by compromising lower memory and/or flops benefits in exchange for simpler programming and better efficiency. The most common one is the Block Low-Rank [4] (BLR) format. This partitions the matrix into blocks which can be either in a low-rank representation or full-rank. This provides a regular tiling of the matrix easy to parallelize following all the previous works made in dense linear algebra. The Block Separable (BS) [34]–[36] format is very close to BLR but, as opposed to it, it uses a weak admissibility condition such that all off-diagonal blocks are low-rank. The main advantage of those last two formats is that they remove the complex hierarchical block clustering of the $\mathcal{H}$-Matrices. In [37], a new intermediate format called multilevel BLR, was introduced to bridge the gap between $\mathcal{H}$-Matrices and BLR, by reaching similar asymptotic costs while providing a regular structure easier to maintain and parallelize. However, there is still no efficient implementation of this format.

Finally, the Lattice $\mathcal{H}$-Matrices, or Tile-$\mathcal{H}$ format, proposed in [7], [8], presents another alternative format to attain better asymptotic costs with respect to BLR, while presenting a similar regular structure. The idea relies in flattening the first levels of the clustering tree to obtain an $n_t \times n_t$ tile layout where each tile is itself an $\mathcal{H}$-Matrix. In [8], they proposed an OpenMP + MPI implementation of the LU factorization with this format, developed in the HACApK library [38]. The parallelization model follows the one developed in the dense linear algebra library SLATE: OpenMP tasks working on each tile are submitted to perform both the computation and the MPI communications, and computations within each of the tiles are done with the sequential version of HACApK. This format has also been tested in `libHLR` [29] and in [31].

The main differences between our approach and the existing ones are that 1) we focus on the tiling idea to be able to expose a higher concurrency degree as BLR/BS do, but without losing efficiency, because each tile contains an $\mathcal{H}$-Matrix; 2) we leverage existing libraries (CHAMELEON and HMAT-OSS) instead of reimplementing the whole $\mathcal{H}$-Arithmetic, contrarily to what is done in the referenced Lattice $\mathcal{H}$-Matrices approach; 3) as a result from using StarPU combined with CHAMELEON, we already have tradidional task based parallelism arising from blocks/tiles-based structures, so we are able to attain a fair parallel performance in this context with an already existing and efficient runtime.

IV. $\mathcal{H}$-CHAMELEON

The main goal of this work is to validat whether it is possible to ontain an efficient open-source $\mathcal{H}$-Matrices library at a small developer cost. For that purpose, we leverage existing libraries, software, programming models and runtime systems to efficiently scale $\mathcal{H}$-Matrices solvers, via the classic tile-based approach commonly used in dense linear algebra, in order to deliver good parallel performance. In this section, we expose the main details of the proposed solution.

A. The building blocks

The work of this paper relies on three building blokcs or components to provide this new $\mathcal{H}$-Matrix library: the dense linear algebra library CHAMELEON, the runtime system STARPU, and the $\mathcal{H}$-Matrices library HMAT-OSS.

CHAMELEON [11], [12] is a dense linear algebra open source software written in C. It relies on the sequential task flow programming model supported by runtime systems such as OpenMP tasks, StarPU, PaRSEC-DTD, or Quark. All dense linear algebra algorithms are expressed as tile algorithms with sequential tasks that are submitted to the underlying runtime system. The runtime automatically infers the data dependencies in these algorithms thanks to keywords that specify the data accesses. CHAMELEON covers all BLAS subroutines with these algorithms, as well as one-sided factorization (Cholesky, LU, QR), and supports multiple runtime systems to schedule the tasks. STARPU is one of them, and the most integrated one into CHAMELEON, besides being the second building brick of our proposal.

STARPU provides tools to describe the pieces of data of the user, such that the data transfers from device to device or node to node, are transparent to the algorithm developer. This is a key feature when developing $\mathcal{H}$-Matrices algorithms, as it will suffice to provide the runtime with pack/unpack functions in order to transfer data (if needed). The fact that STARPU through this feature provides support for heterogeneous distributed architectures oriented our choice to use it for future work.

HMAT-OSS is the open source version of the HMAT library developed by Airbus. It is a sequential library written in C++. This library provides up-to-date implementations of $\mathcal{H}$-Matrix operations and techniques, clustering algorithms and orderings, and real world applications samples. The open source HMAT library, HMAT-OSS, will be employed as a performance reference or baseline in the experiments in Section V. Note that any other $\mathcal{H}$-Matrix library such as H2Lib or libHLR could have been used, and HMAT-OSS was used for its open-source license and by convenience.

B. Tile-$\mathcal{H}$ matrices

The baseline realization of CHAMELEON only supports dense matrices, which are stored employing a descriptor that specifies, among other information, the matrix dimensions, number and size of the tiles which define the matrix partitioning, pointers to data addresses in memory, and some control

parameters to test certain features (e.g., whether there is an overall data pointer).

In order to accommodate Tile-$\mathcal{H}$ Matrices in our hierarchical extension of CHAMELEON, we have expanded the reference descriptor structure to become a collection of tiles. Each of these tiles is potentially an $\mathcal{H}$-Matrix, a low-rank block, or a full-rank matrix. To do this change, we enriched the CHAMELEON matrix descriptor (Structure 1) with an array for the new tile structures, and a helper function, `get_blktile`, to extract the correct tile pointer from the tile indices in the matrix.

```
typedef struct chameleon_desc_s {
    ...
    blktile_fct_t  get_blktile;
    CHAM_tile_t   *tiles;
    ...
} CHAM_desc_t;
```

Structure 1: `CHAM_desc_t` datatype modifications to handle more generic tile formats.

Additionally to the main data structure, the tile description has been modified from the simple data pointer used in runtime systems, such as OpenMP or Quark, to handle the dependencies toward a more complex structure able to handle different data formats. The new datatype, `CHAM_tile_t`, described in Structure 2, allows to simply store different matrix formats, defined by the `format` field, and a pointer to the matrix, `mat`, which can be either a full-rank matrix, or a more complex datatype, such as an $\mathcal{H}$-Matrix coming from an external library. The data dependencies that were initially tracked down using the pointer to the data in the full-rank matrix are thus now followed by the pointer to this tile descriptor. This means that all the algorithms from the CHAMELEON library could work out of the box with an $\mathcal{H}$-Matrix format, considering that there exist kernels to handle this type of matrices.

```
typedef struct chameleon_tile_s {
    int8_t format;
    int    m, n, ld;
    void  *mat;
} CHAM_tile_t;
```

Structure 2: `CHAM_tile_t` data structure to accommodate any format of tiles in the CHAMELEON library.

Finally, the global representation of the matrix that links a CHAMELEON descriptor (`CHAM_desc_t`) with an HMAT-OSS descriptor (`hmat_matrix_t`) in a unique element is done via an additional new data structure (Structure 3). In this new datatype, `super` represents the CHAMELEON library tile descriptor; `clusters` stores an array of the cluster trees created to partition the original data; `admissibilityCondition` contains this parameter value, necessary to determine whether a certain block is already admissible (and consequently converted to a low-rank block) or needs to be re-partitioned; `hi` is the interface defined in the HMAT-OSS library to deal with $\mathcal{H}$-Matrices, which contains general information (for example, the clustering algorithm needed to construct the $\mathcal{H}$-Matrix; and data precision format); `hmatrix` contains the $\mathcal{H}$-Matrix built employing HMAT-OSS construction kernels (which is the descriptor of the library whose content will be employed in the HMAT-OSS kernels operations); and `perm` stores the permutation array.

```
struct HCHAM_desc_s {
    CHAM_desc_t             *super;
    hmat_cluster_tree_t    **clusters;
    hmat_admissibility_t    *admissibilityCondition;
    hmat_interface_t        *hi;
    hmat_matrix_t           *hmatrix;
    int                     *perm;
};
```

Structure 3: `HCHAM_desc_s` structure created to handle the complexity of the Tile-$\mathcal{H}$ structure.

C. Clustering algorithm

In order to use Tile-$\mathcal{H}$ matrices in the CHAMELEON library, we need an adapted clustering tree. Indeed, the CHAMELEON library works exclusively on regular tile sizes, with the exception of the padding row and column. Thus, it is not sufficient to flatten the first levels of the clustering tree as it is done in [8]. We extended the HMAT-OSS library with a recursive tile clustering algorithm, named "*NTilesRecursive*", that recursively divides a given cluster tree (CT) into clusters that respect the regular partitioning into tiles of size NB.

This process is illustrated in Algorithm 2, and the parameters and functions employed are described next:

- *CT* is the cluster tree to partition;
- *axis* is the main axis of the current slice, which is exploited by the geometric clustering techniques to split along the largest dimension;
- *NB* is the desired tile size;
- *offset* represents the coordinates (values) of the first value in the current cluster tree;
- *size* is the size of the current cluster tree;
- the function *slice* returns a portion of the current cluster according to the given offset and size;
- the function *largestDimension* returns the largest dimension in the current cluster;
- and the function *sortByDimension* orders the current cluster of unknowns accordingly to the given dimension;

At each level, this function performs a pseudo-bisection aligned with the tile size along the largest dimension and returns the concatenation of the recursive call to each subset of unknowns. This provides a regular clustering of the unknowns that matches both the constant size requirement of the CHAMELEON library and the Tile-$\mathcal{H}$ format. A median bisection algorithm is then called within each cluster to refine the clustering of each tile.

D. HMAT-OSS *kernels*

In order to implement a hierarchical LU factorization, we have leveraged a number of sequential numerical kernels from the CHAMELEON and HMAT-OSS libraries, which provide

Algorithm 2 NTilesRecursive(CT, n_b , offset, size, axis)

$n_t = \lceil \frac{\text{size}}{NB} \rceil$
if $n_t == 1$ **then**
  return CT
end if
dim =getLargestDimension(CT, axis)
sortByDimension(CT, dim)
$\text{offset}_L = \text{offset}$
$\text{size}_L = NB * \lceil \frac{n_t}{2} \rceil$
$\text{offset}_R = \text{offset} + \text{size}_L$
$\text{size}_R = \text{size} - \text{size}_L$
$\text{CT}_L = \text{slice(}\ \text{CT}, \text{offset}, \text{offset} + \text{size}_L\ \text{)}$
$L = \text{NTilesRecursive(}\ \text{CT}_L, NB, \text{offset}_L, \text{size}_L, \text{dim)}$
$\text{CT}_R = \text{slice(}\ \text{CT}, \text{offset} + \text{size}_L, \text{size}_R\ \text{)}$
$R = \text{NTilesRecursive(}\ \text{CT}_R, NB, \text{offset}_R, \text{size}_R, \text{dim)}$
return (L, R)

the necessary operations to factorize our Tile-$\mathcal{H}$ matrix. In order to do these changes, we modified the main kernels of the CHAMELEON libraries. These kernels, as the data dependencies tracking system, integrated pointers to full-rank matrices. To limit the impact on the library, we just introduced an intermediate layer to enable the switch between full-rank and $\mathcal{H}$ kernels. Thus, the task insertion functions are not modified by this change, and the `CHAM_tile_t` datatype helps us to switch from one kernel type to another, thanks to the `format` field.

On the HMAT-OSS library side, we provide a similar interface to BLAS for `GETRF`, `TRSM`, and `GEMM` operations, and an intermediate internal layer which allows us to switch from this $\mathcal{H}$-Matrix interface to the more classical BLAS interface.

V. NUMERICAL EXPERIMENTS

In this section, we first present the environment from which the data comes from, and a description of the PlaFRIM platform used for the experiments. Afterwards, we analyze the parallel performance of $\mathcal{H}$-CHAMELEON in a multicore system.

A. Experimental context

The test case used is the application TEST_FEMBEM [39] that generates a real or complex matrix, which has features similar to real industrial applications in aeronautics. For any number of unknowns n, we create a cloud of points $(x_i)_{1 \leq i \leq n}$ located on the surface of a cylinder of chosen height and width, as illustrated on Figure 3. These points are equally spaced in both directions on the cylinder surface. Then, we define the interaction kernel between two points x_i and x_j separated by a distance $d = |x_i - x_j|$ as $K(d) = \exp(ikd)/d$ in the complex case, and $K(d) = 1/d$ in the real case. In the complex case, k plays the role of a wave number, it is chosen with the "rule of thumb" of having 10 points per wavelength, which is a rule commonly used in the wave propagation community. The singularity at $d = 0$ is simply removed by setting d equal to half the mesh step in that case. Element (i, j) in the matrix is $a_{ij} = K(|x_i - x_j|)$.

In the real case, the rank of the $\mathcal{H}$-Matrix blocks will be mostly independent of their sizes, as can be seen on Figure 3. In the middle matrix, the ranks of all blocks oscillates around an average value of 9, no matter how big or small those blocks are. Therefore, most of the data (in terms of storage) will be located near the diagonal of the matrix. On the other hand, in the complex case, the rank will grow with the size of the blocks (due to the oscillatory nature of the kernel), and the data will be much more evenly distributed in the matrix. Hence, the amount of storage and work is a lot more important in the complex case, and the work distribution is much more difficult too.

B. Experiments platform

All our experiments have been performed on the PlaFRIM [40] test bench, and more specifically on the `bora` cluster. Each node is equipped with two INTEL `Xeon Skylake Gold 6240` of 18-cores, running at 2.60 GHz, and equipped with 192 GB of memory. The application is compiled with GCC 9.2.0, and INTEL MKL `2019` is used for the BLAS and LAPACK kernels. The 1.3.0 version of STARPU is used, and our proposal is built on revisions 33AA719 of HMAT-OSS [9] and 08CF0CD1 of CHAMELEON [13].

C. Experiments

We next present the experiments performed to evaluate $\mathcal{H}$-CHAMELEON. Our first experiment pursues to highlight that our implementation, though simpler than constructing a classical $\mathcal{H}$-Matrix, still enables a good compression ratio. To this end, Figure 4 shows a comparison between HMAT-OSS (dashed lines) and $\mathcal{H}$-CHAMELEON (full lines) compression ratios for double (left plot) and complex (right plot) precisions, employing different matrix dimensions, from $10K$ to $200K$, and various tile sizes, from 500 to $10K$. The results show that the difference is negligible in all cases, so that we can affirm that the clustering with fixed tile sizes does not impact the compression ratio on the studied test case, and can even provide better results than the classical median bisection used in HMAT (`-oss`). Note that the HMAT-OSS compression rate is stable as it is not influenced by the tile size.

Second, as precision is the *bargaining chip* in $\mathcal{H}$-Scenarios to permit time and memory savings, it is also necessary to control that the proposed clustering does not impact the numerical accuracy of the $\mathcal{H}$-CHAMELEON $\mathcal{H}$-LU factorization operation. With the purpose of evidencing that precision differences are negligible, Figure 5 presents forward error measurements: $||x - x_0||_f / ||x||_f$, for different $\mathcal{H}$-LU executions with the same matrix configurations employed in the previous experiment on the compression ratio. Note that the accuracy parameter is set at 10^{-4}, both in HMAT and $\mathcal{H}$-CHAMELEON. The largest observable differences are around $1.5e-4$, which means we stay in the same magnitude order.

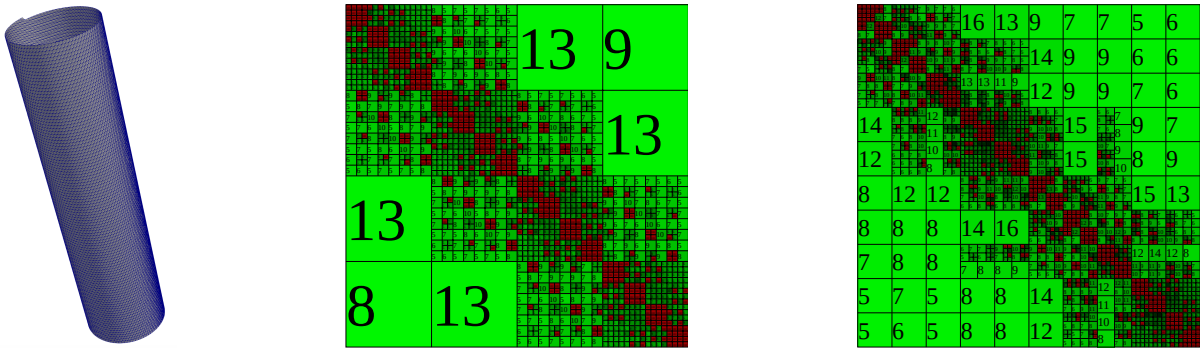

Fig. 3: Illustration of the test case used for the experiments. On the left, the mesh of the cylinder with the distribution of the unknowns on the surface for 10k points. On the middle, the associated compressed real matrix in the HMAT format (classical $\mathcal{H}$-Matrix). On the right, the associated compressed real matrix in the proposed fixed-sized Lattice or Tile-$\mathcal{H}$ based matrix format. In the matrices, low-rank blocks are represented in green (with a number specifying the rank), while dense blocks are coloured in red.

Fig. 4: Comparison of the compression ratio between HMAT-OSS original clustering algorithm (dashed line) and $\mathcal{H}$-CHAMELEON (full lines), in function of the tile size, for double precision (left) and complex precision (right).

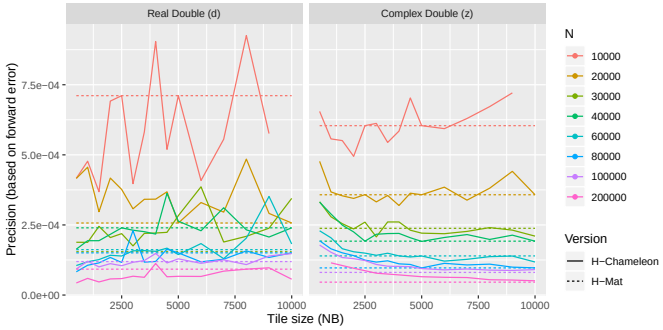

Fig. 5: Comparison of the precision of the solver (based on forward error) between HMAT-OSS original clustering algorithm (dashed line) and $\mathcal{H}$-CHAMELEON ntiles clustering algorithm (full lines) in function of the tile size, for double precision (left) and complex precision (right).

Note that in both Figure 4 and 5, the HMAT-OSS results are constant as the $\mathcal{H}$-Matrix structure used by HMAT-OSS is fixed and independent from the tile size as opposed to the structure generated with the tile-$\mathcal{H}$ format.

Figures 6 and 7 offer multicore parallel performance comparisons employing up to 36 threads (35 in the case of $\mathcal{H}$-CHAMELEON), with various matrix dimensions from $10K$ to $200K$, both in real and complex double precision scenarios. These figures study different scheduling strategies proposed by the STARPU runtime system, while comparing their performance to that provided by the STARPU based implementation of the HMAT library that deals with all the fine grain dependencies.

Three scheduling strategies are studied:

- The work stealing strategy (`ws`) uses a queue per worker and schedules the tasks on the worker which released them by default. Whenever a worker becomes idle, it steals a task from the most loaded worker.
- The locality work stealing (`lws`) strategy similarly uses a queue per worker that is now sorted by the priorities assigned to the tasks. New ready tasks are scheduled on the worker which released them by default. Whenever a worker becomes idle, it steals a task from neighbor workers respecting the priority order.
- The priorities based (`prio`) approach uses a central task queue in which ready tasks are sorted by decreasing priority. All threads try to pull work out of this central queue.

Note that for our proposed implementation we never go over 35 worker threads to keep a dedicated core to the task submission. Experiments have shown that this was more efficient than oversubscribing with 36 worker threads.

In the case of the $\mathcal{H}$-CHAMELEON implementation, all tile sizes presented in compression and accuracy curves have been tested and, based on performance results, we chose the best one for each dimension and precision.

In most of the test cases, we observe that $\mathcal{H}$-CHAMELEON presents slower execution times when using 1, 2, or 3 threads. The tile size being optimized for the 35 threads case induces an overhead of memory and required flops, which impacts the

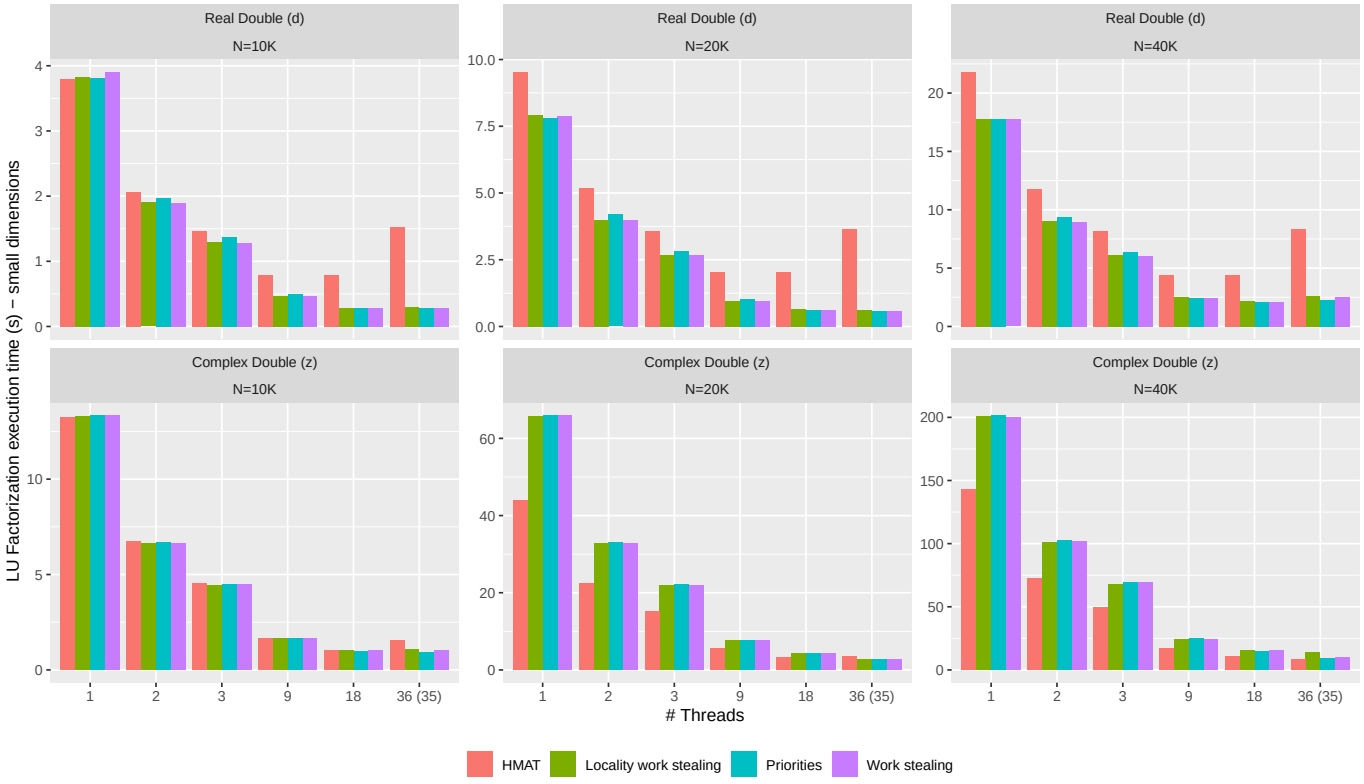

Fig. 6: Comparison of the multicore parallel executions between $\mathcal{H}$-CHAMELEON and HMAT-OSS LU factorization, employing up to 36 threads (35 threads in the case of $\mathcal{H}$-CHAMELEON). Results are shown for real double (top) and complex double (bottom) precisions, and for small matrix dimensions: $10K$ (with $NB = 250$ for d, $NB = 500$ for z), $20K$ (with $NB = 500$), $40K$ (with $NB = 1000$).

executions with a low number of threads. However, when a larger number of threads is used, this is compensated by the parallelism it provides, and so it enables a good scalability of the library. HMAT is not impacted by the tile size and manages to get better performance for the execution on the small numbers of threads.

The comparison of the real and complex double precision results shows that the HMAT performs better on the complex cases, while $\mathcal{H}$-CHAMELEON has a better scaling on the real ones. This can be explained by the difference in the number of operations of the two test cases, due to the arithmetic and to their configuration, as previously explained in Section V-A. In the complex-arithmetic scenario, the cost of the kernels is high enough to cover the overhead of handling the large number of dependencies, which benefits HMAT. However, in the real case, the cost of handling all fine grain dependencies becomes too important with respect to the computational tasks, and therefore $\mathcal{H}$-CHAMELEON outperforms HMAT.

The comparative study of the STARPU scheduling strategies in these figures reflects that, in general, the three variants deliver similar execution times. However, the strategies based on priorities provide higher performance, and the simple priority strategy turns to be the best in most of the cases, except the smaller dimensions. In the real double precision cases with $N = 10K$ and $N = 20K$, the priority scheduler does not provide the fastest solution, as the computational tasks are too small and the idle threads create a contention in the single global task queue of this scheduler.

VI. CONCLUDING REMARKS

In this paper, we have proposed an extension of the CHAMELEON library that takes advantage of $\mathcal{H}$-Matrices and $\mathcal{H}$-Arithmetic to accelerate the execution time and reduce the memory footprint of the LU factorization. More precisely, our approach takes advantage of the sequential kernels of HMAT-OSS to perform $\mathcal{H}$-Arithmetic, and the task-based approach of CHAMELEON to manage parallelism. The original large matrix is split into a set of tiles, and each tile can be represented either as a dense or an $\mathcal{H}$-Matrix. Then, a runtime system, as STARPU does in our experiments, schedules the tasks on the system, and manages the parallelism, following the approaches developed in CHAMELEON for the dense case.

We have conducted experiments on a shared memory machine for large real-life cases, and our results have demonstrated that this approach is competitive with the proprietary HMAT library. Thus, it provides one of the first open-source libraries that is able to reach a good level of performance using $\mathcal{H}$-Matrices. A main asset of the approach is that it will

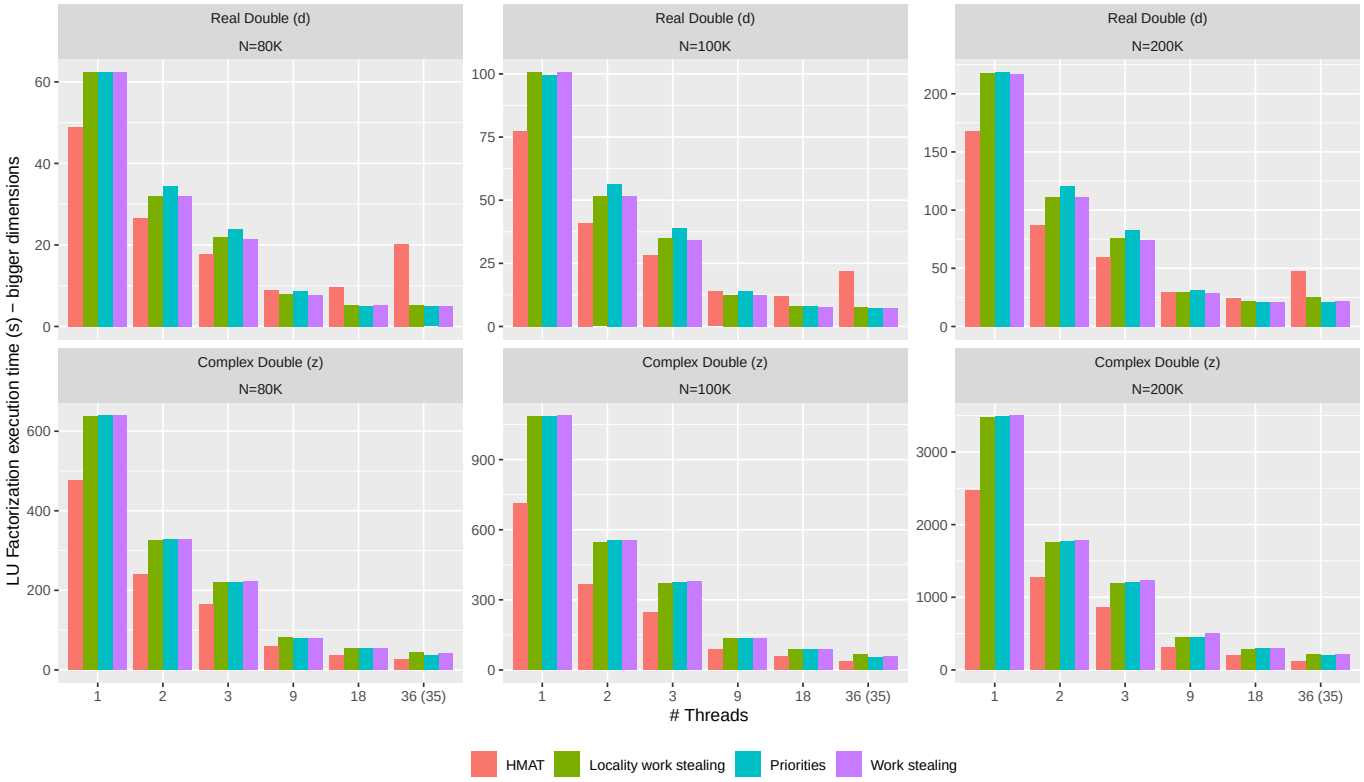

Fig. 7: Comparison of the multicore parallel executions between $\mathcal{H}$-CHAMELEON and HMAT-OSS LU factorization, employing up to 36 threads (35 threads in the case of $\mathcal{H}$-CHAMELEON). Results are shown for real double (top) and complex double (bottom) precisions, and for larger matrix dimensions: $80K$ (with $NB = 1000$ for d, $NB = 2000$ for z), $100K$ (with $NB = 1000$ for d, $NB = 2000$ for z), $200K$ (with $NB = 2000$ for d, $NB = 4000$ for z).

directly benefit from all improvements on the runtime side, as it is now integrated in the CHAMELEON library.

For future work, we plan to study the behavior of this approach for the distributed case, where the main challenge is to correctly handle communications, when the size of the structures, depending on the ranks of matrices, cannot be known statically. It is particularly challenging to study such an approach with the use of dynamic runtime systems. The distributed $\mathcal{H}$-Matrices implementations are also known to be largely unbalanced, and this work will provide a large test suite to work on data distribution and distributed load-balancing algorithms. Moreover, defining a way to discover the best tile size for a given matrix size and number of threads without having the necessity of testing several combinations is also an interesting open research area that remains as an active field for full-rank computations. Solutions based on compression estimations could be studied to give hints to the user based on the matrix structure.

ACKNOWLEDGEMENTS

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

R. Carratalá-Sáez and E. S. Quintana-Ortí were supported by project TIN2017-82972-R from the *Ministerio de Ciencia, Innovación y Universidades*, Spain. R. Carratalá-Sáez was supported by the FPU program of the *Ministerio de Educación, Cultura y Deporte*, and a *HiPEAC collaboration Grant (2019)*.

This work is also supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009.

REFERENCES

[1] W. Hackbusch, "A sparse matrix arithmetic based on hmatrices. part i: Introduction to h-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[2] L. Grasedyck and W. Hackbusch, "Construction and arithmetics of h-matrices," *Computing*, vol. 70, no. 4, pp. 295–334, 2003.

[3] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*. Springer-Verlag Berlin Heidelberg: Springer Series in Computational Mathematics, 2015.

[4] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J. Y. L'Excellent, and C. Weisbecker, "Improving multifrontal methods by means of block low-rank representations," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1451–A1474, 2015.

[5] T. Mary, *Block Low-Rank multifrontal solvers: complexity, performance and scalability*. Université Toulouse 3 Paul Sabatier: Ph.D. Dissertation, 2017.

[6] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman, "Sparse supernodal solver using block low-rank compression: Design, performance and analysis," *Journal of computational science*, vol. 27, pp. 255–270, 2018.

[7] A. Ida, "Lattice h-matrices on distributed-memory systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brasil, May 2018, pp. 389–398.

[8] I. Yamazaki, A. Ida, R. Yokota, and J. Dongarra, "Distributed-memory lattice h-matrix factorization," *The International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 1046–1063, 2019.

[9] Hmat-oss library github repository. [Online]. Available: https://github.com/jeromerobert/hmat-oss

[10] B. Lizé, *Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique: H-Matrices. Parallélisme et applications industrielles.* École doctorale Galilée, Université Sorbonne Paris Nord and Airbus: Ph.D. Dissertation, 2014.

[11] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, cheaper, better a hybridization methodology to develop linear algebra software for gpus," CEA/DAM ; Total E&P ; Université de Bordeaux, Research Report, 2010. [Online]. Available: https://hal.inria.fr/inria-00547847

[12] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," CEA/DAM ; Total E&P ; Université de Bordeaux, Research Report, October 2017. [Online]. Available: https://hal.inria.fr/hal-01618526/file/tpds14.pdf

[13] Chameleon software home page. [Online]. Available: https://solverstack.gitlabpages.inria.fr/chameleon/

[14] The openmp api specification for parallel programming home page. [Online]. Available: http://www.openmp.org/

[15] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2017, pp. 1–8.

[16] Starpu runtime system home page. [Online]. Available: http://starpu.gforge.inria.fr/

[17] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[18] A. Yarkhan, J. Kurzak, and J. Dongarra, "Quark users guide," *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee*, vol. 268, 2011.

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

[20] S. Rjasanow, "Adaptive cross approximation of dense matrices," in *Int. Association Boundary Element Methods Conf., IABEM*, 2002, pp. 28–30.

[21] P.-G. Martinsson and J. Tropp, "Randomized numerical linear algebra: Foundations & algorithms," *arXiv preprint arXiv:2002.01387*, 2020.

[22] E. A. et al., *LAPACK Users guide*. SIAM, 1999.

[23] M. Izadi, *Hierarchical matrix techniques on massively parallel computers*. Universitat Leipzig: Ph.D. Dissertation, 2012.

[24] R. Kriemann, "H-lu factorization on many-core systems," *Computing and Vis. in Science*, vol. 16, no. 3, pp. 105–117, 2013.

[25] J. I. Aliaga, R. Carratalá-Sáez, R. Kriemann, and E. S. Quintana-Ortí, "Task-parallel lu factorization of hierarchical matrices using ompss," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Orlando, United States of America, May 2017, pp. 290–294.

[26] A. Ida, T. Iwashita, T. Mifune, and Y. Takahashi, "Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters," *Journal of Information Processing*, vol. 22, no. 4, pp. 642–650, 2014.

[27] H2lib library home page. [Online]. Available: http://www.h2lib.org/

[28] Hlibpro library home page. [Online]. Available: https://www.hlibpro.com/

[29] libhlr library home page. [Online]. Available: https://www.hlibpro.com/hlr/

[30] S. Boerm, S. Christophersen, and R. Kriemann, "Semi-automatic task graph construction for h-matrix arithmetic," *https://arxiv.org/abs/1911.07531*, 2019.

[31] C. Augonnet, D. Goudin, M. Kuhn, X. Lacoste, R. Namyst, and P. Ramet, "A hierarchical fast direct solver for distributed memory machines with manycore nodes," CEA/DAM ; Total E&P ; Université de Bordeaux, Research Report, Oct. 2019. [Online]. Available: https://hal-cea.archives-ouvertes.fr/cea-02304706

[32] R. Carratalá-Sáez, S. Christophersen, J. I. Aliaga, V. Beltran, S. Boerm, and E. S. Quintana-Ortí, "Exploiting nested task-parallelism in the h-lu factorization," *Journal of Computational Science*, vol. 33, pp. 20–33, 2019.

[33] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the integration of task nesting and dependencies in openmp," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, United States of America, May 2017, pp. 809–818.

[34] H. Cheng, Z. Gimbutas, P. G. Martinsson, and V. Rokhlin, "On the compression of low rank matrices," *SIAM Journal on Scientific Computing*, vol. 26, no. 4, pp. 1389–1404, 2005.

[35] A. Gillman, *Fast direct solvers for elliptic partial differential equations.* University of Colorado: Ph.D. Dissertation, 2011.

[36] A. Gillman, P. Young, and P. G. Martinsson, "A direct solver with o(n) complexity for integral equations on one-dimensional domains," *Frontiers of Mathematics in China*, vol. 7, no. 2, pp. 217–247, 2012.

[37] P. Amestoy, A. Buttari, J.-Y. l'Excellent, and T. Mary, "Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel blr format," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, 2018.

[38] Hacapk library github repository. [Online]. Available: https://github.com/hoshino-UTokyo/hacapk-gpu

[39] TEST_FEMBEM library github repository. [Online]. Available: https://gitlab.inria.fr/hiepacs/papers/hmat/hmat-comparison/

[40] Plafrim home page. [Online]. Available: https://www.plafrim.fr/en/home/