A hierarchical fast direct solver for distributed memory machines with manycore nodes

Cédric Augonnet, David Goudin, Matthieu Kuhn, Xavier Lacoste, Raymond Namyst, Pierre Ramet

▶ To cite this version:

Cédric Augonnet, David Goudin, Matthieu Kuhn, Xavier Lacoste, Raymond Namyst, et al.. A hierarchical fast direct solver for distributed memory machines with manycore nodes. [Research Report] CEA/DAM; Total E&P; Université de Bordeaux. 2019. cea-02304706

HAL Id: cea-02304706
https://cea.hal.science/cea-02304706v1

Submitted on 3 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A hierarchical fast direct solver for distributed memory machines with manycore nodes

Augonnet, Cédric[1], Goudin, David[1], Kuhn, Matthieu[1], Lacoste, Xavier[2], Namyst, Raymond[3], and Ramet, Pierre[3]

[1]CEA/DAM, France
[2]Total E&P, France
[3]University of Bordeaux, France

October 3, 2019

Abstract

Compression techniques have revolutionized the Boundary Element Method used to solve the Maxwell equations in frequency domain. In spite of the several orders of magnitude gained in terms of computational cost, and resource consumption, their implementation in a direct solver remains challenging, especially on distributed memory machines. We present the design of an efficient and scalable hierarchical fast direct solver capable of factorizing $\mathcal{H}$-matrices on large scale machines with manycore nodes.

This task-based solver relies on a flexible execution model which features an extension of the sequential task flow (STF) paradigm, enabling seamless expression of complex dependencies between hierarchical data over distributed memory machines. We demonstrate its efficiency and its scalability by solving large scale problems over hundred of manycore nodes, and for example factorize a $\mathcal{H}$-matrix with 4.4 million unknowns compressed at 99% in less than 40 minutes with about 70% of parallel efficiency over 24,320 cores.

1 Introduction

To meet the ever-growing demand for improved accuracy and larger problem sizes in simulations of physical phenomena, numerous research efforts have been devoted to upgrade numerical schemes, algorithms and parallel implementations over the past decades. Although the evolution of modern supercomputers has constantly contributed to address larger problems and heavier computing intensity, the advent of compression techniques allowed to dramatically push the boundaries in terms of reachable problems in several simulation domains.

In this paper, we focus on a scientific application developed at CEA/DAM which simulates the scattering of incident radar waves by an object to predict

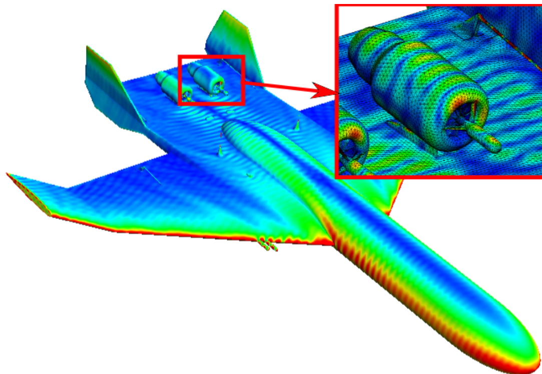


Figure 1: Electric currents at the surface of an UAV at 2.5 GHz

its Radar Cross Section (RCS), which is the ratio between incident energy, and the energy reflected in a specific direction. As illustrated in Figure 1, which depicts a testcase from the ISAE'16 Workshop, the electromagnetic currents are first computed at the surface of the object. The RCS of the object can then be deduced in each direction and polarization.

The Boundary Element Method (BEM) is used to solve the Maxwell equations. For a fixed frequency, this leads to solve a dense complex symmetric (non hermitian) linear system. One right-hand side (RHS) is computed per observation angle and per polarization. During production runs, typical system sizes start from 10^4 and can feature more than 10^7 unknowns, possibly with several thousands of RHS per frequency. Due to the large number of RHS, to the accuracy requirements and to the poor matrix conditioning, a direct method is used to solve the linear problem (Cholesky decomposition). Since this factorization has a $\mathcal{O}(n^3)$ complexity for n unknowns and requires $\mathcal{O}(n^2)$ storage, the code was originally designed (in the early 90s) to run on large clusters of multiprocessor nodes.

However, the accuracy requirements of nowadays production runs implied a heavy redesign of the code. In a previous work, the offloading of computations over accelerators using a tiled algorithm has been explored [9]. This work, which combines improvements regarding both numerical solving techniques and parallel execution models, investigates the use of compression techniques to drastically reduce the memory footprint as well as the number of operations in our Maxwell equations BEM solver.

Single-level compression methods usually involve Block Low Rank approximations. To achieve a higher compression ratio, hierarchical methods such as $\mathcal{H}$-matrices [22], $\mathcal{H}^2$-matrices [23], $\mathcal{H}$SS [42, 41] or $\mathcal{H}$ODLR [7] can be used, at the expense of a significant complexification of the implementation [30, 35]. The challenge is thus to enable the efficient implementation of hierarchical compression techniques over current and future supercomputers.

To fully exploit the computational power of modern clusters in the presence

of many dependencies between computations, the task paradigm has proved to be very effective in many simulation domains. Most interestingly, the majority of existing dense linear algebra solvers already rely on task-based runtime systems as a way to submit dependency-oriented graphs of numerical kernels over shared memory architectures.

In this paper, we explore the design of a task-based $\mathcal{H}$-matrix direct solver over clusters of manycore nodes. To extract maximum parallelism out of the factorization algorithm, we introduce a recursive task decomposition based on a hierarchical read/write dependency mechanism which enables a natural implementation of the algorithm while drastically reducing the number of required synchronizations compared to the traditional fork-join approach. Moreover, our task execution model allows for a smooth integration of asynchronous MPI operations, leading to a high communication overlap ratio.

The main contributions of this work are as follows.

- We propose a new runtime mechanism to resolve implicit task dependencies in applications working on hierarchically decomposed data;
- We show how to seamlessly integrate asynchronous MPI operations using a concept of interruptible tasks;
- We have developed a complete, scalable $\mathcal{H}$-matrix solver on top of these mechanisms;
- We show that our approach achieves high performance over a cluster of KNL processors (70% of parallel efficiency on 24,320 cores).

The remainder of the paper is organized as follows. Section 2 gives a background on task parallelism, solvers, and compression techniques. In Section 3, we describe the execution model we use to implement a direct $\mathcal{H}$-matrix solver in Section 4. We evaluate the efficiency of our solver in Section 5.

2 Background

This section first describes how tasks are used to implement parallel solvers on both shared and distributed memory architectures, showing that mixing tasks and MPI is still a complex problem. Then, it introduces compression techniques in the scope of electromagnetism simulations.

2.1 Task based solvers over distributed memory architectures

Cholesky and LU decompositions are often a central methodology in numerical simulations. These solvers are nowadays commonly implemented with tiled algorithms over tasks runtime systems [18, 40, 20, 38, 2].

A task-based algorithm performing a Cholesky decomposition is illustrated in Figure 2. The left part of the figure details the tiled sequential algorithm.

Each step k of the factorization relies on 4 kernels. First, the POTRF kernel factorizes the diagonal block (line 2). The other blocks on column k (called the k-th *panel*) are updated with a solve operation, namely TRSM (line 4). Remaining blocks are updated through SYRK (line 5) operations for diagonal blocks, and GEMM (line 7) operations for non diagonal blocks. All these operations are available in classical LAPACK implementations.

```
1  for (k=0;k<NB;k++){
2    POTRF (A_kk^RW)
3    for (j=k+1;j<NB;j++){
4      TRSM (A_kk^R, A_jk^RW)
5      SYRK (A_jk^R, A_jj^RW)
6      for (i=k+1;i<j;i++)
7        GEMM (A_jk^R, A_ik^R, A_ij^RW)
8    }
9  }
```

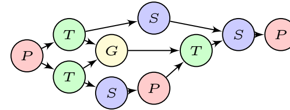


Figure 2: Cholesky factorization using the STF model

Still on the left part of Figure 2, note that the type of access mode is specified for each piece of data. The `R` and `RW` exponents respectively indicate that the data is read or modified by the kernel. The principal advantage of the Sequential Task Flow (STF) paradigm is to leverage these annotations to automatically derive a parallel graph out of such an apparently sequential piece of code. The STF model extracts parallelism with the following coherency model: modifications cannot occur until the end of all pending reads (Write After Read, or WAR), concurrent reads are possible (Read After Read, or RAR), but readers may not access data while they are modified (Read After Write, or RAW), and different modifications cannot occur simultaneously (Write After Write, or WAW). Other advanced types of accesses such as reductions or commutative accesses are also sometimes possible. The right part of Figure 2 shows the Direct Acyclic Graph (DAG) obtained with a matrix with 3×3 sub-blocks. Each vertex corresponds to a task (labeled after the first letter of the corresponding kernel), and the edges between the vertices mark the automatically inferred dependencies. It is therefore natural to parallelize linear solvers thanks to the numerous runtime systems implementing this STF programming model.

Implementing direct dense linear solvers on both machines with shared and distributed memory is a well-studied problem [12]. But the simplicity enabled when using tasks on shared-memory systems is balanced by the difficulty of extending these solvers to machines with distributed memory. Task-based distributed dense solvers have been developed [34, 14, 3], but their implementation is usually a challenge and sometimes rely on domain specific programming environments [20, 15, 8]. Introducing compression techniques into distributed solvers is therefore a delicate problem.

More generally, using tasks in a distributed environment (typically with

MPI) is known to be significantly more complicated [36].

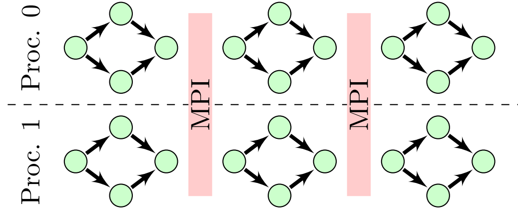


Figure 3: Separated MPI and tasks phases

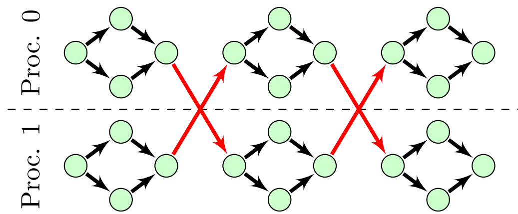


Figure 4: STF and distributed memory

The simplest approach is to design the code as an MPI application, and to take advantage of intra-node parallelism with tasks (*e.g.* using OpenMP). This methodology is natural when adapting a Full-MPI application to exploit intra-node parallelism and reduce the overall number of MPI processes to the benefits of using more threads. Figure 3 for example illustrates such a strategy where we distinguish intra-node task-based phases and synchronous phases with MPI operations such as broadcasts, for instance.

However, the easiness of creating a vast amount of concurrency within a process through tasks is contrasted with the challenge of designing a distributed application with asynchronous communications that overlaps computation and communication. Runtime systems [8, 15, 3] and programming environments [36] can help by automating data transfers in an efficient manner between task-based applications. On Figure 4, this results in an application that is fully based on tassks, and which features much more parallelism and less synchronizations points than on its synchronous counterpart on Figure 3 where communications are not handled as tasks.

2.2 Compression techniques for electromagnetism

Compression techniques are well known to be effective into the frame of BEM for Maxwell equations [11]. Considering a $m \times n$ sub-block B of the matrix obtained with BEM, these techniques are based on the observation that the number of singular values greater than a given threshold ϵ (also called numerical

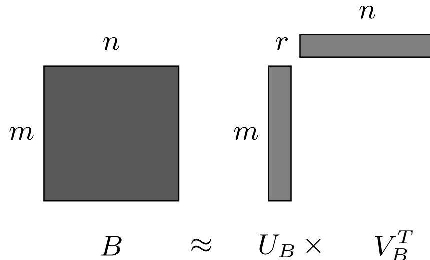


Figure 5: Low-rank approximation of the B block of size $(m \times n)$ as the product of two matrices U_B and V_B of size $(m \times r)$ and $(n \times r)$ where r is the numerical rank of B.

rank at precision ϵ) is generally much smaller than $min(m,n)$. As illustrated on Figure 5, such *low-rank* blocks (also called Rk-matrices) can be approximated under the outer-product form $B \approx UV^t$, where U (resp. V) is a $m \times r$ (resp. $n \times r$), with r the numerical rank of B.

2.2.1 Block Low Rank algorithms

A first strategy to introduce compression in solvers is the Block Low Rank (BLR) approach that consists in potentially compressing blocks of fixed size. Each sub-block of the matrix corresponds to the interaction between two parts of the simulated object. A compressibility criterion, also called admissibility condition, determines whether a block is compressible or not depending on the distance between the two parts. Non-admissible blocks (*e.g.* diagonal blocks) are kept in their dense representation, which we denote as *full-rank* blocks. Figure 6 gives an example of such a matrix stored in the Block Low-Rank format. Besides reducing the overall memory required to numerically solve the equations, compression techniques also lower its computational cost.

The approximation of a compressible block can be computed for instance through Singular Value Decomposition (SVD). A more interesting strategy is to assemble compressible blocks directly in their compressed form through a specialized algorithm such as Adaptive Cross Approximation (ACA) [10] or one of its variants.

Compute kernels must also be adapted to support data in a compressed format. On line 4 of Figure 2, assuming A_{kk} is full-rank and $A_{jk} \approx U_{jk}V_{jk}^T$, the TRSM kernel has to perform $V_{jk} = V_{jk}L_{kk}^{-T}$ instead of $A_{jk} = A_{jk}L_{kk}^{-T}$. It is worth noting that if A_{jk} has a rank r, this kernel has a complexity of $\mathcal{O}\left(rn^2\right)$ instead of $\mathcal{O}\left(n^3\right)$, which leads to dramatic performance improvements.

Introducing compression into the GEMM kernel (line 7) is slightly more complex. For example, with 3 compressible blocks, denoting $(A|B)$ the concatenation of blocks A and B, we have:

$$U_{ij}V_{ij}^T - U_{jk}V_{jk}^T\left(U_{ik}V_{ik}^T\right)^T = \left(U_{ij}|U_{jk}\right)\left(V_{ij}|U_{ik}V_{ik}^TV_{jk}\right)^T.$$

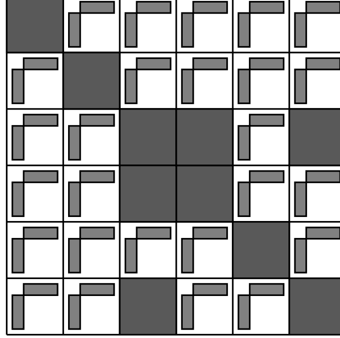


Figure 6: Example of a matrix stored in the Block Low-Rank (BLR) format, where each block can either be full-rank, or compressed as a low-rank block of the form UV^T. Diagonal blocks are always full-rank, but some extra-diagonal blocks can also be full rank.

This corresponds to a compressed block with a rank of $r_{ij} + r_{jk}$, that can be recompressed using a QR-SVD or a Rank Revealing QR (RRQR) kernel to obtain a block with a rank usually close to r_{ij} [11]. The QR-SVD algorithm which we actually implement is detailed in Figure 7.

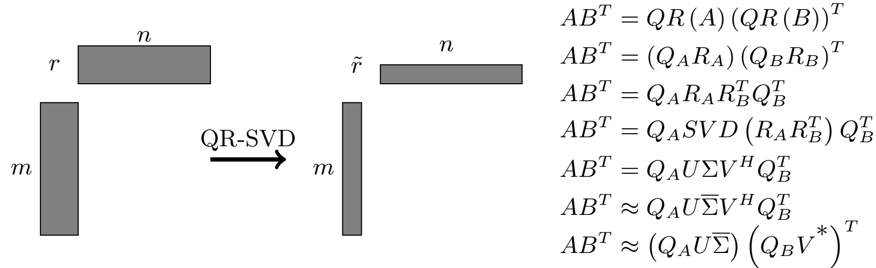


Figure 7: The QR-SVD algorithm recompresses a block written under the form AB^T with a rank r into a block with an optimal numerical rank $\tilde{r}$ (for a given threshold ϵ). It is implemented using two QR decompositions (GEQRF), one SVD kernel (GESVD), and two multiplications with the orthogonal matrices obtained in the QR decompositions (UNMQR). The Σ diagonal matrix is truncated accordingly to ϵ to keep the $\tilde{r}$ largest eigenvalues in $\overline{\Sigma}$ and replace smaller ones with null values.

One notable difficulty is the number of combinations that must be handled when designing these compression-enabled GEMM kernels. Besides the previous example where all blocks are compressible (i.e. approximated under the form UV^T), all configurations require to design totally different algorithms :

- If all blocks are non-compressible, we simply call the GEMM kernel from the BLAS library.

- If we have A_{ij} non-compressible, and that both A_{jk} and A_{ik} compressible, we compute a temporary compressed block D such as $D = A_{jk}A_{ik}^T$, so that $D = U_{jk}V_{jk}^T \left(U_{ik}V_{ik}^T\right)^T$. We can here identify $U_d = A_{jk}$ and $V_d = U_{ik}V_{ik}^T V_{jk}$. Computing V_d only requires calling 2 GEMM kernels, and does not imply recompression operations. We then compute $A_{ij} - U_d V_d^T$ by using another GEMM kernels.

- When A_{ij} is compressible, and both A_{jk} and A_{ik} are non-compressible, the least expensive approach is to explicitely by multiplying $D_1 = A_{jk}A_{ik}^T$ using a GEMM kernel, computing $D_2 = U_{ij}V_{ij}^T - D_1$ with another GEMM kernel, and to apply a costly truncated-SVD operation on D_2 which gives the updated value of A_{ij}.

While the GEMM kernel is dominating dense solvers, recompression kernels take most of the time into BLR solvers. A few large SVD kernels are also very time consuming, but only occur rarely so that they only account for a small portion of the total computation time. These rare but large kernels can however have a significant impact on load balancing.

In addition to dense linear algebra, BLR algorithms are also used to reduce memory footprints and/or computational costs of sparse direct solvers [6, 33].

2.2.2 Hierarchical algorithms

The consideration of $\mathcal{H}$-matrices [22] in place of BLR allows reaching higher compression ratios. Instead of partitioning the overall matrix with blocks with a fixed size, a hierarchical partitioning of the unknowns is performed, leading to a cluster-tree $\mathcal{T}$. The compressibility criterion is then applied on each node of $\mathcal{T}$ to determine if the corresponding block is compressible or not. If the block is not compressible, the criterion is applied again on the node's children.

Similarly to the BLR solver which required to adapt kernels to operate on potentially compressed blocks, an $\mathcal{H}$-matrix solver requires to implement kernels that operate on blocks which are $\mathcal{H}$-matrices themselves. The general approach is to design hierarchical kernels as a combination of kernels operating on either full-rank or low-rank blocks. For example, if $\mathcal{H}$ is a triangular hierarchical block, its Cholesky decomposition can be written recursively:

$$\mathcal{H}\text{-POTRF}\left(\begin{array}{cc} \mathcal{H}_{11} & \\ \mathcal{H}_{21} & \mathcal{H}_{22} \end{array}\right) = \begin{cases} \mathcal{H}\text{-POTRF}(\mathcal{H}_{11}) \\ \mathcal{H}\text{-TRSM}(\mathcal{H}_{11}, \mathcal{H}_{21}) \\ \mathcal{H}\text{-SYRK}(\mathcal{H}_{21}, \mathcal{H}_{22}) \\ \mathcal{H}\text{-POTRF}(\mathcal{H}_{22}) \end{cases}$$

In this case $\mathcal{H}$-POTRF($\mathcal{H}_{11}$) can either operate on a hierarchical block or on a full-rank block, but there are for example $27 = 3^3$ combinations of $\mathcal{H}$-GEMM kernels since each of the 3 blocks can be any either full-rank, low-rank or a hierarchical block. In addition to the recompression kernels required in the BLR solvers, some kernels of the hierarchical solvers are also implemented by the means of format conversion algorithms, such as $\mathcal{H}$-MERGE which converts 4

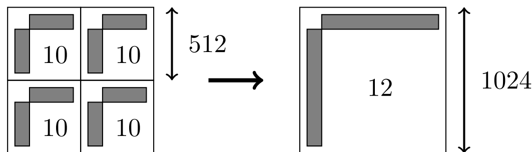


Figure 8: The H-MERGE kernel merges 4 Rk-matrices into a single Rk-matrix. In this example, the initial storage requirement of the $\mathcal{H}$-matrix is $4 \times 2 \times 512 \times 10 = 40960$ entries, compared to $2 \times 1024 \times 12 = 24576$ entries after the merge operation.

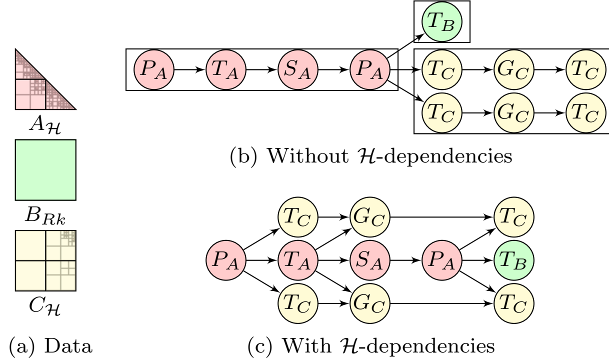
low-rank sub-blocks into a single low-rank block [11], as illustrated on Figure 8. This is for example useful to implement the $\mathcal{H}$-GEMM kernel $\mathcal{H}_C = \mathcal{H}_C - \mathcal{H}_A \mathcal{H}_B^T$ with $\mathcal{H}_A$ and $\mathcal{H}_B$ hierarchical blocks, and $\mathcal{H}_C$ a low-rank block, since we can create $\mathcal{H}_D = \mathcal{H}_A * \mathcal{H}_B$, convert $\mathcal{H}_D$ from a hierarchical into a low-rank block which can be added to $\mathcal{H}_C$.

To illustrate that each configuration can be a problem of its own, let us for instance assume that A is a $\mathcal{H}$-matrix, and that both B and C are Rk-matrices. The $\mathcal{H}$-GEMM which computes $U_C V_C^T = U_C V_C^T - \mathcal{H}_A \left(U_B V_B^T\right)^T$ is here very different from the previous example with three $\mathcal{H}$-matrices. We can indeed implement this kernel by computing $U_D = \mathcal{H}_A V_B$ which is a dense block, and then consider $U_D U_B^T$ as a Rk-matrix with the same rank as B. The new value of $U_C V_C^T$ is then obtained by performing a QR-SVD recompression kernel on $\left(U_C V_C^T + U_D U_B^T\right)$.

Similarly, we meet such a combinatorial challenge when implementing the $\mathcal{H}$-SYRK, $\mathcal{H}$-TRSM and $\mathcal{H}$-POTRF hierarchical kernels over matrices that can either be full-rank, Rk-matrices or $\mathcal{H}$-matrices.

3 Proposed execution model

As highlighted in previous Section, we must address two significant issues in order to design a highly efficient task-based implementation of our $\mathcal{H}$-matrix solver. The first issue is to avoid over-synchronization when recursively generating tasks working on hierarchical data. To this end, we propose a new mechanism to automatically derive task dependencies from sequences of memory accesses to any level of a hierachical data structure. This techniques enables our code to manipulate $\mathcal{H}$-matrices in a practical and efficient way. We then address the problem of "taskifying" MPI communication schemes so as to maximize asynchronous communication progress while resolving task dependencies as soon as possible, by leveraging interruptible tasks to describe complex, multiphase MPI communication schemes. Finally, we show how we implemented this execution model.



(a) Data

(b) Without $\mathcal{H}$-dependencies

(c) With $\mathcal{H}$-dependencies

Figure 9: Panel update where $A_{\mathcal{H}}$ and $C_{\mathcal{H}}$ are $\mathcal{H}$-matrices and B_{Rk} is a Rk matrix ($\mathcal{H}$-POTRF$(A_{\mathcal{H}})$; $\mathcal{H}$-TRSM$(A_{\mathcal{H}}, B_{Rk})$; $\mathcal{H}$-TRSM$(A_{\mathcal{H}}, C_{\mathcal{H}})$;)

3.1 Hierarchical dependencies

In Section 2.1, we have illustrated that the STF model and its implicit data dependencies is a very convenient paradigm to implement scalable solvers. Due to the recursive nature of the algorithms shown in Section 2.2, we now describe how we extended the coherency model used in the STF paradigm to support hierarchical data such as $\mathcal{H}$-matrices by the means of implicit **hierarchical dependencies ($\mathcal{H}$-dependencies)**.

Figure 9 shows the tasks associated when performing three hierarchical operations of a panel update, namely $\mathcal{H}$-POTRF$(A_{\mathcal{H}})$, $\mathcal{H}$-TRSM$(A_{\mathcal{H}}, B_{Rk})$ and $\mathcal{H}$-TRSM$(A_{\mathcal{H}}, C_{\mathcal{H}})$, where $A_{\mathcal{H}}$ and $C_{\mathcal{H}}$ are $\mathcal{H}$-matrices, and B_{Rk} is a low-rank matrix. As depicted by Figure 9a, $A_{\mathcal{H}}$ (resp. $C_{\mathcal{H}}$) is divided into 3 (resp. 4) sub-blocks. In Figure 9b, a traditional approach generating tasks in a nested way would actually lead to wait until all $A_{\mathcal{H}}$ has been updated before providing it as an input of the H-TRSM kernels. In contrast, Figure 9c shows that it is possible to unlock the tasks to update $C_{\mathcal{H}}$ much sooner, as soon as the first sub-block of $A_{\mathcal{H}}$ has been updated. Such a fine-grain synchronization methodology is thus required to obtain enough parallelism on hierarchical workloads.

Different strategies are used to implement those hierarchical workloads in the existing implementations of direct $\mathcal{H}$-matrix solvers. Kriemann *et al.* [28] fully reformulate the $\mathcal{H}$-LU algorithm to extract dependencies between Intel Threading Building Blocks (TBB) [32] tasks in an explicit manner. While extremely efficient, this requires a full redesign of existing algorithms, and makes any algorithmic change a significant burden. It is also unclear how practical this approach would be on a distributed memory machine. Existing direct hierarchical solvers based on the STF model [30, 19] are much easier to develop and to maintain. They actually avoid dealing with hierarchical pieces by either writing kernels that only manipulate $\mathcal{H}$-matrix leaves [19], or by selecting a hierarchical block depth under which all kernels are written using sequential tasks [30].

These *static* approaches either suffer from a massive synchronization overhead, or miss numerous parallelization opportunities within recursive workloads.

A powerful approach to combine the easiness of STF and the efficiency of explicit $\mathcal{H}$-dependencies would be to rely on the STF model directly on hierarchical data sets.

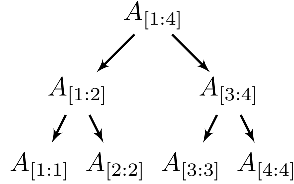


Figure 10: Example of a hierarchical piece of data

Illustrating our approach on actual $\mathcal{H}$-matrix examples would be challenging, so we consider here a simpler but equivalent example on Figure 10 that depicts an array of 4 elements $A_{[1:4]}$, which can be seen as a hierarchical piece of data as well. Each piece of data is assigned a *parent*, which can contain multiple children. For example, $A_{[1:4]}$ is the parent of $A_{[1:2]}$ and $A_{[3:4]}$. Accesses to $A_{[1:2]}$ and $A_{[3:4]}$ are independent, but it is not possible to read $A_{[1:2]}$ while modifying $A_{[1:4]}$. Appropriate dependencies must indeed be introduced to fulfill the coherency model.

Let us assume in Figure 11 that B is a subset of A. If we have a sequence of 4 tasks that respectively modify A (T_1), read B (T_2), read A (T_3), and modify A (T_4), we must ensure that T_2 does not start before the end of task T_1, and we cannot modify A in T_4 until B as been read in T_2. This is achieved by introducing appropriate data dependencies, here denoted as E_1 and E_2 to specify that the execution order of tasks accessing B have to synchronize with tasks accessing A and vice versa. Note that the dependencies between T_1 and T_3, and between T_3 and T_4 are already fulfilled by the STF memory coherency model.

Interestingly, the semantic of those extra data-dependencies is equivalent to RAW, WAR or WAW constraints. A consistent execution between tasks T_1 and T_2 can for instance be obtained by actually *submitting* an empty task, namely $E_1(A^R, B^W)$ in the Figure, accessing A in read mode and B in write mode.

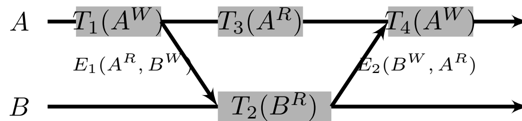
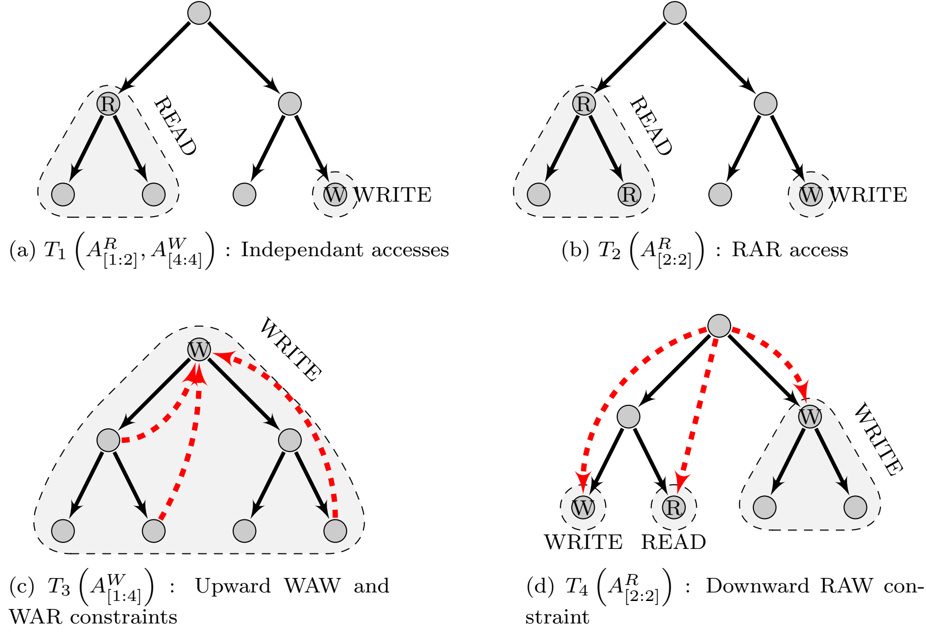


Figure 11: Inserting empty tasks E_1 and E_2 to enforce dependencies for task sequence $T_1; T_2; T_3; T_4$ where $B \subset A$



(a) $T_1\left(A^R_{[1:2]}, A^W_{[4:4]}\right)$: Independant accesses

(b) $T_2\left(A^R_{[2:2]}\right)$: RAR access

(c) $T_3\left(A^W_{[1:4]}\right)$: Upward WAW and WAR constraints

(d) $T_4\left(A^R_{[2:2]}\right)$: Downward RAW constraint

Figure 12: Automatic insertion of dependencies for task sequence $T_1\left(A^R_{[1:2]}, A^W_{[4:4]}\right); T_2\left(A^R_{[2:2]}\right); T_3\left(A^W_{[1:4]}\right); T_4\left(A^R_{[2:2]}\right)$

Indeed, attempting to access A in read mode into E_1 synchronizes with the write access on A performed by T_1. Jointly, reading B in T_2 can only occur after the write action on B is performed into E_1. Hence, we have T_2 executing after E_1, and E_1 executing after T_1, leading to T_2 executing after T_1. Similarly, the execution order between tasks T_2 and T_4 can be ensured by the insertion of the empty task $E_2(B^W, A^R)$ with the appropriate access modes on data A and B.

More elaborated implementations are possible, but this simple *data dependency* mechanism makes it possible to enforce consistent hierarchical data accesses, on top of an existing coherency model solely designed for independant data. Inserting a dependency between A and B in OpenMP for example only requires to submit an empty task with the `depend(inout:A,B)` clause.

Our approach consists in determining automatically, given a hierarchy of data and a set of tasks, which dependencies must be inserted in the task graph to keep data sequentially consistent. To this end, our implementation maintains a state for each piece of data that is updated at **submission time**. We first save the last access *mode* associated with a piece of data: a write access mode for example indicates that the node belongs to a subtree where there is a pending write access. We also have two flags indicating whether there has been a read (resp. write) access on the node to enforce future RAW and WAW (resp. RAR,

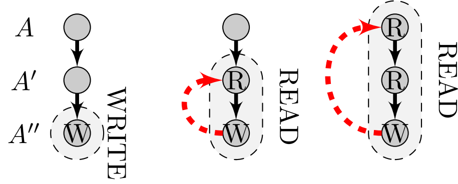


Figure 13: RAW dependencies, $A'' \subset A' \subset A$

WAR) constraints.

Figure 12 illustrates the protocol which allows consistent accesses to the hierarchical data shown in Figure 10. The workload depicted in this example consists of a sequence of four tasks :

$$T_1\left(A^R_{[1:2]}, A^W_{[4:4]}\right); T_2\left(A^R_{[2:2]}\right); T_3\left(A^W_{[1:4]}\right); T_4\left(A^R_{[2:2]}\right).$$

Each sub-figure therefore corresponds to the data status after submitting respectively T_1, T_2, T_3 and T_4. Figure 12b gives the states after submitting T_1, which independently modifies $A_{[4:4]}$ and reads $A_{[1:2]}$. $A_{[4:4]}$ is marked in write mode, and its write access flag is enabled. $A_{[1:2]}$ and the corresponding sub-tree is marked in read mode, and $A_{[1:2]}$ has a read access flag. Note that $A_{[1:1]}$ and $A_{[2:2]}$ have a read mode, but no read flag since there was no task accessing them.

No extra dependencies must be inserted when submitting T_2 which only creates a RAR dependency. Still, a read access flag is set on $A_{[2:2]}$ to protect it against from write accesses.

All accesses to subsets of $A_{[1:4]}$ must be finished before T_3 modifies it. In Figure 12c, we thus introduce dependencies with previous nodes with a flag indicating there was a prior access. Since future accesses on these inner nodes will have to be synchronized with $A_{[1:4]}$, we can unset existing access read and write flags. The entire tree is however denoted as in write mode.

The state obtained after submitting T_4 shown on Figure 12d illustrates how we handle a read access on a data subset which belongs to a tree that has a write mode. To make sure T_3 is over when we access again subsets of $A_{[1:4]}$ (*i.e.* $A_{[1:1]}$,$A_{[2:2]}$ and $A_{[3:4]}$), dependencies are inserted between the root of the subtree and all of these nodes. Note that we could have marked protected $A_{[3:3]}$ and $A_{[4:4]}$ instead of $A_{[3:4]}$, but this would introduce more synchronization points. Our algorithm therefore tries to introduce as few dependencies as possible to maintain data coherency with a minimal synchronization overhead.

Figure 13 illustrates that there are corner cases which must be carefully addressed in this algorithm. In this example, where $A'' \subset A' \subset A$, if a read access on A' (middle) follows a write access on A'' (left), we actually keep the write flag on A'' to denote that future accesses on a sub-tree containing A'' will have to synchronize on it as well. For example, on the right side of Figure 13, a dependency must be inserted between A and A'', even though A' and A'' were in a sub-tree that is marked in read mode. This allows to implement the RAW

constraint efficiently without introducing a superfluous dependency between A' and A, for example.

Our protocol does not access the entire data tree for every data access, and we implemented it with a logarithmic complexity, with respect to the number of nodes in the tree. As a result, the overhead of this protocol is negligible in our execution traces.

3.2 Extending our model for distributed memory with preemptive tasks

Provided we have implicit data-driven dependencies, it is natural to consider our distributed application as a large task graph. When extending STF to a distributed-memory machine, each piece of data is assigned to a process, denoted as the data *owner*. Assuming A and B have two distinct owners in Figure 14, the implicit dependency between T_1 and T_2 would translate into an MPI communication to send A to the owner of B. From the perspective of the task-based application, this is achieved by submitting pair of `send` and `recv` tasks which would be interleaved with the rest of the application.

Implementing this transfer by the means of tasks performing blocking MPI calls (e.g. `MPI_Send`) would result in wasting computing resources while waiting for communications. Besides, execution order might affect program correctness : a process could for example send A_1 and A_2, while another would receive A_2 and A_1, resulting in a deadlock if these are blocking MPI calls.

An intuitive solution is to issue non-blocking MPI operations, and to check *periodically* whether they have completed or not, without blocking the processor in between.

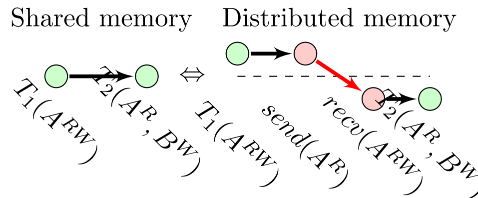


Figure 14: Translating dependencies into communications

Given a *pure* STF approach, we could try to implement the receive operation in Figure 14 with a simple `MPI_Irecv` task which would submit a callback task with an `MPI_Test`, that would recursively submit another callback until the test passes. This approach unfortunately fails because the callback tasks would likely be issued after T_2 has been submitted, so that the implicit dependencies based on task ordering would fail to unlock T_2 only once A has been received.

Instead of submitting multiple tasks, we can consider a single **interruptible** task which would explicitly relinquish the processor until the operation has completed, similarly to the `taskyield` OpenMP directive.

```
#pragma omp task lastprivate(size, buffer)
{
    MPI_Request req;
    MPI_Irecv(&size, sizeof(size), src, tag, &req);
    while (!MPI_Test(&req)) {
        #pragma omp taskyield
    }
    buffer = malloc(size);
    MPI_Irecv(buffer, size, src, tag, &req);
    while (!MPI_Test(&req)) {
        #pragma omp taskyield
    }
}
```

Figure 15: Receiving data of unknown size with an interruptible task split in multiple steps using OpenMP

Such interruptible tasks are especially interesting when exchanging compressed matrices, because their size (and sometimes their structure) may be unknown on the receive side. Interruptions make it possible to **split tasks into multiple steps** to carry out complex data transfers. Figure 15 illustrates how to asynchronously receive a buffer whose size is unknown on the receive side using OpenMP.

A practical issue is that this approach assumes a thread-safe MPI implementation, which is unfortunately often not available on production machines. We thus need to add a notion of *affinity* to ensure that this task will only be executed by the master thread, and that it will be bound to it. It can also be difficult to ensure that this `MPI_Test` is issued often enough, especially if there is a severe workload and/or many simultaneous MPI transfers. In the next section, we will show that a **task pre-emption** mechanism can easily be integrated into a task-based application, and that it is useful to implement our distributed direct $\mathcal{H}$-matrix solver.

3.3 Implementation of the execution model

Due to the lack of a thread-safe MPI implementation on our target platform, and for the sake of flexibility, we have developed a lightweight runtime system which implements the aforementioned execution model. It uses OpenMP internally as a portable threading layer, even though we could rely on other thread libraries such as PTHREADS or ARGOBOTS [37]. Since there is currently no efficient way to pause/restart OpenMP tasks when waiting on an MPI operation, or to restrict MPI-related tasks to a specific processor affinity, we use a minimalistic task implementation which enforces the STF programming model. Provided a thread-multiple MPI, and a few additions to the OpenMP task model, such as those already proposed by OmpSs [36], we could have used OpenMP tasks as well.

Figure 16 illustrates our programming API, which is based on the task API

used in Quark [4]. The `irecv_func` function shows the actual task implementation which will be detailed later on, while `irecv_async` reveals the task submission. In this example which asynchronously receives a piece of data, a `rwlock_t` structure is used to specify data access and its corresponding mode on line 42. Task arguments are stacked on lines 39-41 and retrieved on lines 10. Interestingly, such code could be generated automatically from a code with a semantic close to OpenMP tasks.

Specifying data accesses at task submission makes it possible to infer dependencies. Each piece of data is thus associated to a `rwlock_t` structure which lists all tasks trying to access this piece of data. When a task is submitted, it is appended to the different locks, and a per-task reference count indicates the number of pending data dependencies. Accessed data are unlocked when a task ends. Multiple readers can be unlocked at the same time, but a single writer is unlocked. Tasks with no dependency left (i.e. with a null reference count) are put in the list(s) of ready tasks, from which threads continuously pick up work.

As described in Section 3.1, we extend this STF paradigm for hierarchical data sets. We first let the user describe data hierarchy, by specifying which `rwlock_t` structure is the *father* of another `rwlock_t` structure (or by defining which are the children of a `rwlock_t` structure). We then enforce the algorithms depicted in Figure 12 to automatically determine which data dependencies must be inserted automatically. These dependencies are implemented by the means of empty tasks.

The Nanos runtime system used in OmpSs introduces a pause/restore mechanism which allows it to explicitly preempt and resume the execution of a task, and to let a service thread carry out those MPI transfers efficiently [36]. This is unfortunately not part of the OpenMP standard, so we cannot rely on it directly. In our application, we rely on simple language constructs to implement preemptive tasks.

This is illustrated in Figures 16 which shows a task-based implementation of the `MPI_Recv` primitive. This implementation allows to receive data of unknown size.

Function `irecv_async` submits a task which executes function `irecv_func` when `lock` is accessible in read-write mode. This last function allows to receive a message of unknown size in two communication steps: a first one for the size of the buffer, and a second one for the buffer. The `step` field (line 6) is automatically initialized to 0. The first phase of `irecv_func` on lines 7-14 consists in reading arguments that were packed at submission time into a locally allocated structure accessible through the `priv` field of the task, and submitting a first actual MPI transfer using `MPI_Irecv` (line 12) to receive the size of the buffer. Note that there is no `break` statement between lines 14 and 15, so that we enter in the second phase (lines 15-23) immediately. This second phase consists in testing whether the MPI transfer has finished or not, or to explicitly put the task aside otherwise (lines 18-19). Since the value of the `step` field (line 6) is kept, when a thread executes the task again, it directly jumps to line 15 to perform this test again with a limited overhead. When the test finally succeeds, the memory of the buffer to be received is allocated (line 21).

```
1  void irecv_func(task_t *t)
2  {
3    struct stored_args {size_t *size; void **data;
4                        int src; int tag;
5                        MPI_Request req;} *a;
6    switch (t->step) {
7      case 0:
8        a = malloc(sizeof(struct stored_args));
9        t->priv = a;
10       unpack_args(t,&a->src,&a->tag,
11                   &a->data,&a->size,NULL);
12       MPI_Irecv(a->size,sizeof(size_t),
13                 a->src,a->tag,&a->req);
14       t->step++;
15     case 1:
16       a = t->priv;
17       if (!MPI_Test(&a->req)){
18          t->keep_in_queue = 1;
19          return;
20       }
21       *a->data = malloc(*a->size);
22       MPI_Irecv(*a->data,*a->size,a->src,a->tag,&a->req);
23       t->step++;
24     case 2:
25       a = t->priv;
26       if (!MPI_Test(&a->req)){
27          t->keep_in_queue = 1;
28          return;
29       }
30       t->keep_in_queue = 0;
31       free(a);
32   }
33 }
34
35 void irecv_async(int src, int tag, void **data,
36                  size_t *size, rwlock_t *lock)
37 {
38     add_task(irecv_func,
39         ARG, &src, sizeof(int), ARG, &tag, sizeof(int),
40         ARG, &data, sizeof(void **),
41         ARG, &size, sizeof(size_t *),
42         RWLOCK, &lock, RW, NULL);
43 }
```

Figure 16: Example of a non-blocking MPI transfer implemented using a preemptive task

Then, the MPI transfer to receive the buffer is posted (line 22), and the task goes to the third phase. Similarly to the beginning of phase 2, this last phase first checks if the previous `MPI_Irecv` has finished (line 26), putting the task aside otherwise (lines 27-28). When the test succeeds, resources are liberated and the task finishes its execution normally so the lock accessed in read-write mode (line 42) is released. The corresponding `isend_func` can be implemented in a similar way.

Note that the use of an interruptible task here allows to write this code in the same way we would have written it based on blocking MPI calls, but in a fully asynchronous manner. Both steps could be merged into a single transfer, but the receiving process would have to handle large messages whose sizes are not known in advance: this would result in relying on *unexpected* messages which are challenging for MPI implementations.

Such pieces of code could be compiler-generated, but this illustrates how it is possible to implement task pre-emption using simple language constructs by introducing a mere `step` field and `keep_in_queue` flag into the task, and allowing the application to explicitly put a task back into the list(s) of ready tasks. If `keep_in_queue` flag is not set, a handler is automatically called to release task resources and unlock RW-dependencies.

To reduce the amount of ready tasks with an MPI operation to complete, we also added a *request server* : when a task creates an MPI request, a reference to the request and to the corresponding task is saved in the server, and the task is removed from the ready list(s). The server, which is implemented by the means of a preemptive task as well, periodically tests the completion of all MPI requests at the same time using `MPI_Testany`. Tasks whose request has been completed are put back into the ready list(s). This mechanism significantly reduces the number of calls to the MPI stack, and limits the number of ready tasks waiting for the completion of an MPI operation.

4 Implementation of an $\mathcal{H}$-matrix direct solver

This section describes the implementation of the $\mathcal{H}$-matrix direct solver in use in our application. The first part describes the shared memory implementation which adopts the STF paradigm extended with a coherency model for hierarchical data as described in Section 3.1. The second part shows how the distributed memory version of this solver was written using a 2D block-cyclic distribution scheme and by relying on the interruptible tasks we presented in Section 3.2 to implement the transfer of $\mathcal{H}$-matrix blocks over MPI.

4.1 Design in shared-memory

Similarly to other task-based dense solvers, we have adopted a tiled algorithm which is well known for exhibiting a lot of parallelism. The matrix is partitioned into blocks of fixed size. Each block is associated with a `rwlock_t` structure, as

defined in Section 3.3. The task-based tiled Cholesky factorization implementation is given by Figure 2.

Likewise, the implementation of a BLR solver is achieved using the algorithms previously described in Section 2.2 to take into account the possibility that some blocks are compressed. Interestingly, the design of these kernels and the shape of the blocks are independent from the synchronization mechanisms already used to implement the Tiled algorithm.

Introducing hierarchically compressed blocks puts more challenges because there are many different ways to carry out synchronization between kernels. Similarly to the BLR solver, one possible approach would be to implement the numerous hierarchical kernels sequentially. Obtaining a sufficient amount of parallelism on manycore architectures would require the use of small block sizes, which may significantly degrade compression rates.

Due to the hierarchical nature of $\mathcal{H}$-matrices, it is natural to design those kernels in a recursive fashion. In practice, our task engine enables recursive tasks by allowing task submission directly from another task using a `rwlock_t` structure (**fork**), and to actively wait the completion of these children tasks using an explicit synchronization function (**join**).

While this simple task recursion implementation provides us with a significant amount of parallelism, we have however shown on Figure 9 that such a fork-join synchronization paradigm does not allow to efficiently pipeline multiple kernels that access the same pieces of data. We therefore leveraged the hierarchical data coherency model introduced in Section 3.1 to describe each $\mathcal{H}$-matrix block as a quad-tree of `rwlock_t` where each sub-$\mathcal{H}$-matrix is a node of the tree.

By assigning tasks with an appropriate priority to favor the critical path, we automatically obtain a state-of-the-art task ordering denoted as *dynamic look-ahead* by Kurzak and Dongarra [29], regardless of the complexity introduced by compression techniques.

4.2 Extension to distributed memory

Our strategy to extend our direct hierarchical solver over MPI was to rely on the techniques classically used for task-based dense linear solvers, and to allow manipulating compressed blocks.

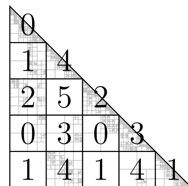


Figure 17: 2D block-cyclic distribution (3x2 processes)

Figure 17 therefore illustrates the 2D block-cyclic distribution used in our

code to distribute $\mathcal{H}$-matrices across the machine. This distribution is not only known to distribute the load evenly on dense problems [12], but also to limit the number of nodes involved in each communication. For example, after the first block is computed on node 0, it must only be sent to nodes 1 and 2 which own the other blocks in the first column. More generally, each communication phase only involves $\mathcal{O}(\sqrt{p})$ among p processes. These traditional results on dense linear algebra are also observed on $\mathcal{H}$-matrices as shown in Section 5.

Once data have been distributed this way, each hierarchical block is associated to a process which *owns* it. All block updates are carried out by the respective *owner* : implementing a distributed version of our solver therefore consists in using the hierarchical kernels already implemented on shared memory, and to exchange hierarchically compressed blocks over MPI when a kernel needs to access a block that was modified remotely.

Even if the structure of a $\mathcal{H}$-matrix can be determined as soon as the mesh is partitioned, the numerical rank of each compressible block is unknown until the matrix has been assembled, and keeps evolving throughout the factorization because the numerical rank of the blocks may evolve when updating them. Compared to existing distributed memory task-based dense solvers, one difficulty when exchanging such $\mathcal{H}$-matrix is therefore that their size (and potentially their structure) are generally not known on the receive side. However, the interruptible tasks we have introduced in Section 3.2 make it straightforward to exchange $\mathcal{H}$-matrix blocks. Sending and receiving a $\mathcal{H}$-matrix block is indeed decomposed into a task with multiple steps. A first step consists in exchanging the structure of the block, along with the numerical ranks of each leaf in the block. On the receive side, the space required to store the block is allocated accordingly. The second send consists in exchanging the actual data content of the $\mathcal{H}$-matrix blocks (*i.e.* U and V).

Given such point-to-point $\mathcal{H}$-matrix transfers, it becomes natural to implement advanced communications schemes, such as asynchronous broadcasts (*e.g.* `MPI_Ibcast`) of $\mathcal{H}$-matrix blocks. Broadcasting a piece of data over a set of processes is indeed achieved by building a spanning tree of the set, with the root process as the root of the tree, and to transmit data throughout this spanning tree. In practice, each process of the tree but its root must receive data from another process by posting a `hmat_irecv` task, and transmit it using `hmat_isend` tasks to at most two processes if we have a binary tree. Note that this approach is simple enough to allow for optimizations such as building the spanning tree with respect to the machine topology to avoid transfers back and forth large scale machines while broadcasting such complex pieces of data.

A significant advantage of our approach is that we can leverage $\mathcal{H}$-dependencies to broadcast hierarchical blocks. Similarly to fine-grain task dependencies which provided us with more parallelism than a fork-join approach in Figure 9, we can efficiently interleave the execution of interdependent remote hierarchical kernels thanks to **hierarchical broadcasts ($\mathcal{H}$-broadcasts)**. For example in Figure 18, we have a node that computes the Cholesky decomposition $\mathcal{H}$-POTRF of a hierarchical block A, and a node that computes BA^{-1} where A is the output of $\mathcal{H}$-POTRF(A). A possible implementation would be to submit a task

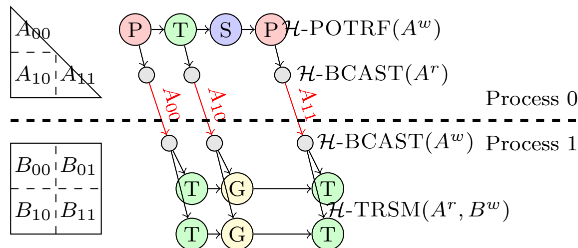


Figure 18: Hierarchical kernels interleaved using hierarchical broadcasts. A is sent piece-wise, which allows to start $\mathcal{H}$-TRSM before the end of kernel $\mathcal{H}$-POTRF.

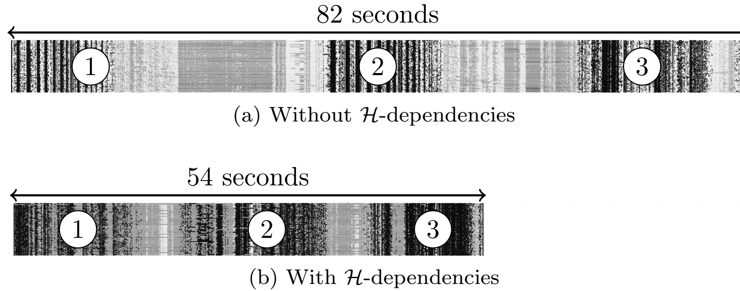
BCAST(A) that broadcasts A after the completion of $\mathcal{H}$-POTRF(A), but it is interesting to note that only A_{00} is required to start updating B_{00} and B_{10}. By transparently decomposing BCAST(A) into multiple independent tasks to broadcast A_{00}, A_{10} and A_{11} as soon as they are ready, we thus obtain a significant performance boost.

5 Evaluation

We now evaluate the performance of our direct $\mathcal{H}$-matrix solver. The experiments were carried on the Tera1000 supercomputer which composed of two separate machines. 1/ Tera1000-1 that is based on 2,125 nodes interconnected with an Infiniband FDR network. Each node has 32 Intel Haswell cores. We use Intel 17.0.4.196 compilers, and OpenMPI 1.8.8. 2/ Tera1000-2 that is based on 8,000 Intel Knights Landing (KNL) boards configured in cache mode and quadrant clustering. It relies on a Bull InterConnect (BXI) network. We use Intel 18.0.3.222 compilers, and OpenMPI 2.0.4.

To maximize the benefits of our shared-memory implementation, we use a single MPI process per node, and we spawn one thread for each physical core (except in Figure 20). All threads were statically bound using the HWLOC library [17], which is paramount on KNL. Our code was linked against Intel MKL. The execution traces were recorded using SCORE-P, and visualized using VAMPIR. Black parts of the traces correspond to $\mathcal{H}$-POTRF kernels, grey parts to all other kernels (*e.g.*, $\mathcal{H}$-TRSM, $\mathcal{H}$-SYRK, $\mathcal{H}$-GEMM and others), and idle time is in white. We first show the benefits of $\mathcal{H}$-dependencies on shared memory architectures, and then study how both $\mathcal{H}$-dependencies and $\mathcal{H}$-broadcasts enhance scalability on distributed memory. We eventually consider the performance obtained on actual industrial testcases. Note that the results obtained using compression techniques were assessed by comparing them against the output of the solver without compression whenever possible. Besides, all implemented optimizations do not degrade accuracy.

On the shared memory side, Figure 19 shows two execution traces that highlight the impact of the $\mathcal{H}$-dependencies on a sphere testcase with 103,000



(a) Without $\mathcal{H}$-dependencies

(b) With $\mathcal{H}$-dependencies

Figure 19: Impact of $\mathcal{H}$-dependencies on 1 KNL (TERA1000-2) on a problem with 103,000 unknowns

unknowns within a single KNL node. While idle time results from inefficient synchronizations on Figure 19a, aggressively unlocking dependencies allows to interleave interdependant kernels in Figure 19b. The second labeled $\mathcal{H}$-POTRF operation therefore occurs sooner, after 19 seconds instead of 35 seconds. Likewise, the overall factorization time is reduced from 82 seconds to 54 seconds. Since both implementations however use the same kernels with different synchronization methodology, we naturally obtain the same performance on a single core (2,620 seconds). We can thus compare the speedup obtained within a KNL in Figure 20. We obtain a parallel efficiency of 78.2% instead of 49.9% on 64 cores. It is also worth noting that this is very challenging testcase because there is only 56MB of data per core in average when using 64 threads.

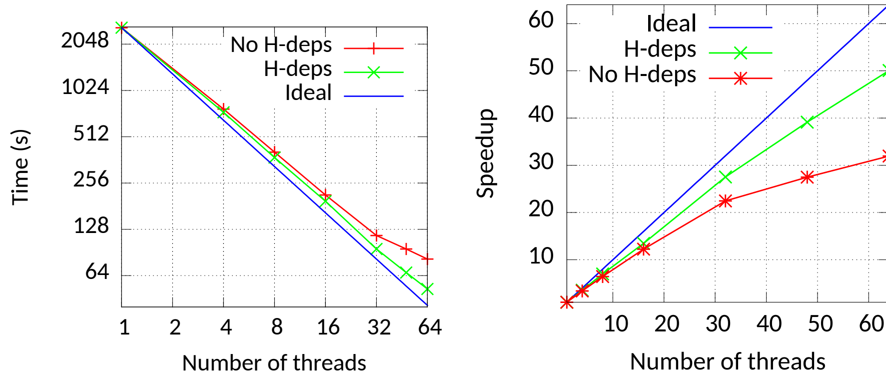
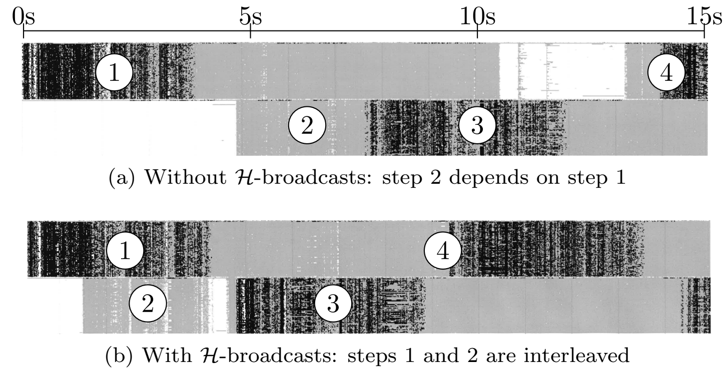


Figure 20: Strong scalability within shared memory on 1 KNL node (TERA1000-2)

On the distributed memory side, Figure 22 shows the strong scalability of a sphere testcase with 867000 unknowns on several nodes of TERA1000-1. In addition to the ideal scaling line, three other lines depict the time performances of the solver using 64 cores (2 nodes) to 3520 cores (110 nodes). Each of these



(a) Without $\mathcal{H}$-broadcasts: step 2 depends on step 1

(b) With $\mathcal{H}$-broadcasts: steps 1 and 2 are interleaved

Figure 21: Impact of $\mathcal{H}$-broadcasts on 2 KNLs (TERA1000-2)(zoom on first the 15 seconds)

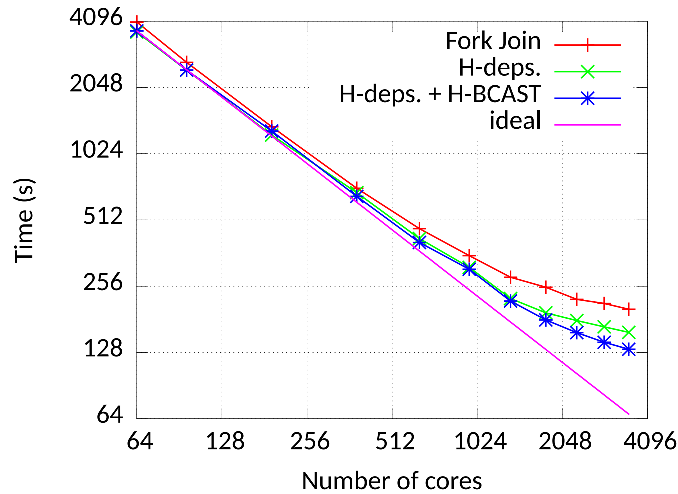


Figure 22: Strong scalability for a testcase with 867,000 unknowns on the Tera1000-1 machine

lines correspond to a different configuration of the solver. As can be seen, using hierarchical dependencies allows to improve both the time performances and the scalability of our solver over the initial fork-join version. Indeed, on 42 nodes, the time to factorize the matrix drops from 281 seconds to 225 seconds, leading to a relative efficiencies of 69% for the fork-join version and 77% for the hierarchical dependencies version. Moreover, adding the hierarchical broadcasts presented into Section 4.2 to the hierarchical dependencies further improves times and scalability: on 72 nodes, the factorization time is improved from 179 to 157 seconds, leading to a relative efficiency of 65% instead of 57%. On 110 nodes, the efficiencies of the three versions degrade to 37%, respectively 42%

and 51% for the fork-join, respectively hierachical dependencies and hierarchical broadcasts, but the volume of computation on this amount of nodes is very low (at most of 0.8 GB of data per node).

The traces in Figure 21 show the impact of $\mathcal{H}$-broadcasts during the first 15 seconds of the factorization step on 2 KNLs for a sphere geometry with 588,000 unknowns. In Figure 21a, the first $\mathcal{H}$-POTRF (labeled as 1) is as efficient as the step 1 of Figure 19b when taking advantage of $\mathcal{H}$-dependencies on shared memory within a single KNL. However, the kernels on the second process labeled with number 2 have to wait for the completion of the entire $\mathcal{H}$-POTRF kernel labeled 1 on the first process before its output is broadcasted. In Figure 21b, we can see that, thanks to the $\mathcal{H}$-broadcasts, the broadcasts of input data for the $\mathcal{H}$-TRSM operations start before the end of the $\mathcal{H}$-POTRF labeled 1 on the first process, allowing the operations to begin earlier. As a result, the execution of the second $\mathcal{H}$-POTRF labeled as 3 on the second process occurs approximatively at 5 seconds of execution time, while it starts at around 7.5 seconds without $\mathcal{H}$-broadcasts.

Table 1: Comparision of the memory footprints with 1D and 2D block-cyclic data distributions on 20 nodes of TERA1000-1 (sphere with 837,000 unknowns testcase)

Distribution	Per-process data footprints	
	Min.	Max.
1×20	3,176 MB	4,526 MB
20×1	3,225 MB	4,488 MB
4×5	3,572 MB	4,054 MB

Table 2: Comparision of amount of data transfers and performance with 1D and 2D block-cyclic data distributions on 20 nodes of TERA1000-1 (sphere with 837,000 unknowns testcase)

Distribution	Communication volumes		Time
	Total (Count)	Per-process Max. (Count)	
1×20	1,788 GB (199,614)	52.9 GB (6,046)	559.2 s
20×1	1,937 GB (203,528)	58.3 GB (6,152)	576.7 s
4×5	688 GB (74,366)	14.7 GB (1,654)	439.8 s

Tables 1 and 2 respectively show the advantages of a 2D block-cyclic distribution over 1D distributions in terms of memory footprints and regarding the amount of data exchanges. As stated in Section 4.2, the amount of data exchanges is reduced by a factor of 4, which allows for a greater scalability. Moreover, there is less than 15% of imbalance (with respect to data footprints) when using a 4×5 distribution, compared to 43% with a 1×20 distribution. Due to a better load balancing, and to a reduction of network solicitation, we thus obtain an improvement of 25% of execution times compared to 1D distributions.

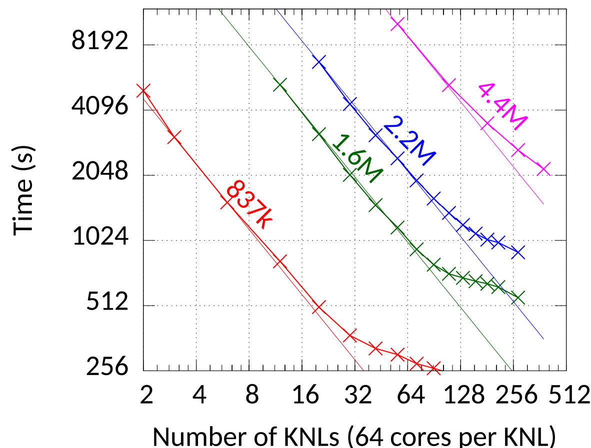


Figure 23: Strong scalability for sphere geometries up to 4.4 million unknowns over KNLs (TERA1000-2)

Figure 23 demonstrates the strong scalability capabilities of our solver over our KNL-based machine. Each curve gives the time required to factorize the matrix obtained with spheres of different sizes, from 837 thousands to 4.4 millions unknowns. In spite of the trivial geometry, the use of a fixed discretization of 10 points per wave-length makes this benchmark ensures that we consider realistic workloads with plausible compression rates around 99%.

For all problems, we observe that the cumulated elapsed CPU time is almost constant when varying the amount of processing ressources. For instance, the resolution of the testcase with 1.6 million unknowns either takes 5,360s on 12 nodes (768 cores), or 1,174s on 56 nodes (3,584 cores). This respectively corresponds to a cumulated CPU time of $768 \times 5,360\text{s} = 1{,}143$ hours and $3,584 \times 1,174\text{s} = 1{,}168$ hours. This corresponds to 97.8% of parallel efficiency even if there only remains 3 GB of data per KNL (out of 192 GB available), or 54MB per core when solving this problem over 56 nodes. The lines attached to the curves on Figure 23 indicate what would be the duration of the problem with a constant cumulated CPU time (e.g. with a perfect scaling), compared to the time required on the smallest number of nodes on which the problem fits. Table 3 shows we obtain similar results on the problem with 4.4 million unknowns, and highlights the gains resulting from our optimizations.

Figure 24 depicts the strong scalability obtained on a machine based on Intel Haswell processors, instead of Intel KNL boards. We observe a behaviour that is very similar to the scalability in Figure 23. Since the TERA1000-1 machine is significantly smaller than TERA1000-2, we thus limited our experiments to the smallest testcases, up to 6,720 cores. This illustrates the suitability of our approach on classical CPU architectures.

Our distributed direct $\mathcal{H}$-matrix solver is also effective on challenging indus-

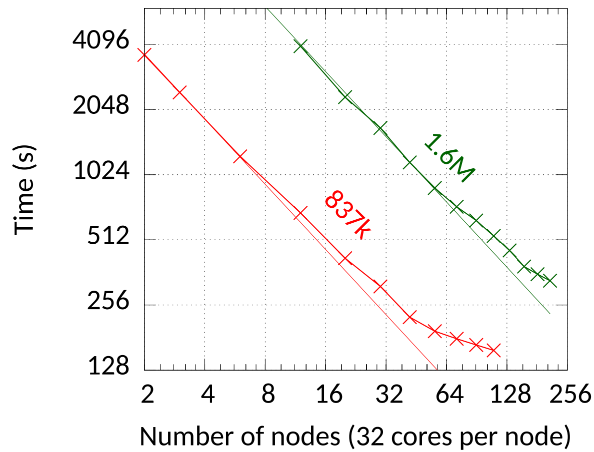
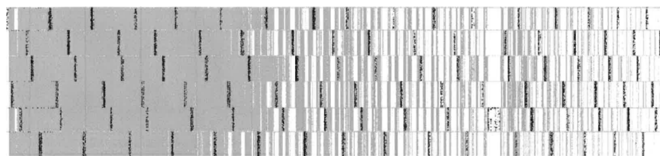
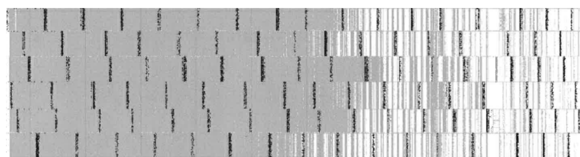


Figure 24: Strong scalability for sphere geometries up to 1.6 million unknowns over HASWELL processors (TERA1000-1)

(a) Without $\mathcal{H}$-dependencies: 325 seconds



(b) With $\mathcal{H}$-dependencies: 235 seconds



(c) With $\mathcal{H}$-dependencies and $\mathcal{H}$-broadcast: 200 seconds

Figure 25: Execution traces of the UAV testcase depending on the synchronization methodology on 6 KNLs (TERA1000-2)

Table 3: Parallel efficiency with 4.4 million unknowns on TERA1000-2 (reference time measured on 56 KNLs)

	#KNLs	56	110	182	272	380
No H-deps	time	11,989 s	6,338 s	4,250 s	3,255 s	2,785 s
	efficiency	100%	96%	87%	76%	63%
H-deps	time	10,233 s	5,338 s	3,562 s	2,676 s	2,188 s
	efficiency	100%	97.5%	88.4%	78.7%	68.9%
memory per node		26 GB	13 GB	7.9 GB	5.3 GB	3.8 GB

trial problems. Figure 25 shows the traces obtained when executing the UAV testcase with 823K unknowns (which output was presented on Figure 1) on $6 \times 64 = 384$ cores. Introducing $\mathcal{H}$-dependencies reduces the execution time from 325s to 235s, as illustrated by the visible diminution of the amount of idle time lost in superfluous or inefficient synchronizations in Figure 25b. Coupling $\mathcal{H}$-dependencies with $\mathcal{H}$-broadcasts reduces this value to 200s. We thus observe a 1.6× improvement of the makespan of the factorization step of this testcase, without changing the amount of computation, but only optimizing its parallelization methodology.

As another example, simulating a 1.7 GHz antenna on a launcher of the CNES (testcase from the ISAE Workshop'14) is done in only 25 minutes using 30 KNLs to solve a problem with 1.65 million unknowns compressed at 98.5%. Likewise, we can compute the RCS of the UAV shown in Figure 1 at 10 GHz in 53 minutes over 72 KNLs for a problem with 4.84 millions unknowns. We here obtain a factorized system with a compression rate of 98.7%, or 1.2 TB of memory. In comparision, this would take more than 2.5 days to solve the same problem on a single KNL node, without a distributed memory $\mathcal{H}$-matrix solver, which is by far less convenient to solve our industrial problems. Likewise, solving the largest problem in Figure 23 would likely require about a week of computation on a single node, assuming we could fit the overall 2 TB of memory required to solve the problem, for example using out-of-core techniques.

6 Related work

Task parallelism have become increasingly popular in the last decade, and many task-based runtime systems have been brought to the community [26]. The Sequential Task Flow programming model, which automatically infers dependencies according to data accesses, has been implemented in runtime systems such as Quark, StarPU, StarSs, and in standardized environments such as OpenMP. Both the OmpSs and StarPU runtime systems also serve as a target to implement $\mathcal{H}$-matrix solvers using the STF model. StarPU makes it possible to manipulate hierarchical data by applying *filters*, which can be invoked asynchronously [39]. In their implementation of a $\mathcal{H}$-matrix solver on top of StarPU, however, Lizé *et al.* [30] still select a depth in the hierarchy under which data are

accessed sequentially to avoid manipulating hierarchical data. OmpSs provides *weak dependencies* to let users lazily lock *all* children of a hierarchical piece of data. In [19], Carratalá-Sáez *et al.* explicitly state that their implementation of $\mathcal{H}$-matrix cannot cope with the whole data hierarchy, and therefore only accessed the $\mathcal{H}$-matrix leaves.

Contrary to the aforementioned solvers, our $\mathcal{H}$-matrix solver fully takes advantage of implicit dependencies between hierarchical pieces of data. Kriemann *et al.* [28] reach almost perfect scalability on shared memory, but manually unfold the DAG of Intel TBB tasks [32], which limits the extensibility and maintainability of their approach compared to $\mathcal{H}$-dependencies.

The interruptible task mechanism we have used is similar to the pausing API provided by Nanos in OmpSs [36], but our approach neither requires a specific non-standard OpenMP implementation nor a thread-multiple MPI implementation. Instead, existing systems can leverage our coherency protocol to easily support hierarchical data on top of the STF model. To the best of our knowledge, there has been no other demonstration that combining interruptible tasks with hierarchical data dependency management significantly helps to scale on distributed memory systems.

Designing a fast scalable parallel solver based on compression techniques is a current hot topic into the field of numerical simulations. In particular, recent studies addressed computational kernels and both shared and distributed memory parallelism. Compression algorithms are often key to get better performances. Alternatives to QR-SVD algorithms, *e.g.* Rank Revealing QR [33] or randomized sampling [24], are investigated. Concurrently, other studies have shown that batched algorithms improve performances over manycore architectures for small matrices computations [31, 21, 16, 1]. Being focused on parallelism, our work is orthogonal to these studies, but the flexibility of our proposition makes it possible to easily integrate such advanced kernels.

Regarding distributed memory, iterative solvers embedding compression techniques are commonly found in the literature (*e.g.* $\mathcal{H}$ACApK [27] or hmglib [25]). Their parallel performances mainly rely on the parallelization of the $\mathcal{H}$-matrix vector product. Building a distributed direct hierarchical solver is more difficult due to the variety of required kernels and the hierarchical nature of the data on which they operate. The closest dense direct hierarchical solver in terms of distributed memory parallelism is the compression component of STRUMPACK [35]. The efficiency of their approach has been proved on large problems, running on up to 8,000 cores. However, it must be noted that their compression technique is based on an $\mathcal{H}$SS representation, which differs from $\mathcal{H}$-matrix. Moreover, on the parallel aspects, this solver is (for now) not multithreaded and the communication schemes are synchronous. Their work could directly take benefits of both $\mathcal{H}$-dependencies and taskified MPI asynchronous operations to get efficient shared memory parallelism and to overlap communications and computations.

7 Conclusion and Future Work

Designing a scalable, efficient $\mathcal{H}$-matrix solver requires to cope with complex, hierarchical dependencies between tasks in order to unleash the maximum degree of parallelism. Expressing such dependencies in a practical way, compatible with the use of MPI communications, is a key to enable applications to be extended with various numerical optimizations.

In this paper, we propose to leverage two runtime system extensions, namely automatic inference of hierarchical data dependencies and interruptible tasks, to achieve these goals. We have developed a complete $\mathcal{H}$-matrix direct solver using MPI and our task-based runtime system. Our experiments show that our approach is up to 1.6 times faster than the traditional fork-join method, and achieves 70% of parallel efficiency on a cluster of 24K cores.

This work enables several promising research directions. In particular, we intend to implement multiple kernel variants to enable dynamic kernel selection. Data locality could be further improved by introducing per-core task queues [13]. Having efficient synchronization techniques will help in offloading kernels for $\mathcal{H}$-matrices over accelerators such as GPUs: advanced runtime system mechanisms will be required to leverage the batching techniques employed by Akbudak *et al.* [5] beyond a single level of compression. Our approach also provides a nice framework to implement dynamic load balancing of tasks across multiple nodes. Last but not least, our approach could apply to other applications, such as AMR-based solvers.

References

[1] Abdelfattah, A., Haidar, A., Tomov, S., and Dongarra, J. Novel hpc techniques to batch execution of many variable size blas computations on gpus. In *Proceedings of the International Conference on Supercomputing* (2017), ACM, p. 5.

[2] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. A hybridization methodology for high-performance linear algebra software for gpus. In *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 473–484.

[3] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. P. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[4] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series* (2009), vol. 180, IOP Publishing, p. 012037.

[5] Akbudak, K., Ltaief, H., Mikhalev, A., and Keyes, D. Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures. In *High Performance Computing* (Cham, 2017), J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds., Springer International Publishing, pp. 22–40.

[6] Amestoy, P. R., Buttari, A., L'Excellent, J.-Y., and Mary, T. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw. 45*, 1 (Feb. 2019), 2:1–2:26.

[7] Aminfar, A., Ambikasaran, S., and Darve, E. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics 304* (2016), 170 – 188.

[8] Augonnet, C., Aumage, O., Furmento, N., Namyst, R., and Thibault, S. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting* (2012), Springer, pp. 298–299.

[9] Augonnet, C., Goudin, D., Pujols, A., and Sesques, M. Accelerating a massively parallel numerical simulation in electromagnetism using a cluster of gpus. In *PPAM* (2013).

[10] Bebendorf, M. Approximation of boundary element matrices. *Numerische Mathematik 86*, 4 (2000), 565–589.

[11] Bebendorf, M. *Hierarchical Matrices - A Means to Efficiently Solve Elliptic Boundary Value Problems*, vol. 63 of *Lecture Notes in Computational Science and Engineering.* Springer, 2008.

[12] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[13] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing 37*, 1 (1996), 55–69.

[14] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011), IEEE, pp. 1432–1441.

[15] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., and Dongarra, J. J. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering 15*, 6 (2013), 36–45.

[16] Boukaram, W. H., Turkiyyah, G., Ltaief, H., and Keyes, D. E. Batched QR and SVD algorithms on gpus with applications in hierarchical matrix compression. *Parallel Computing 74* (2018), 19–33.

[17] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing* (2010), IEEE, pp. 180–186.

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

[19] Carratalá-Sáez, R., Christophersen, S., Aliaga, J. I., Beltran, V., Börm, S., and Quintana-Ortí, E. S. Exploiting nested task-parallelism in the h-lu factorization. *Journal of Computational Science* (2019).

[20] Chan, E., Van Zee, F. G., Bientinesi, P., Quintana-Orti, E. S., Quintana-Orti, G., and Van de Geijn, R. Supermatrix: a multi-threaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (2008), ACM, pp. 123–132.

[21] Charara, A., Keyes, D. E., and Ltaief, H. Tile low-rank GEMM using batched operations on gpus. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings* (2018), M. Aldinucci, L. Padovani, and M. Torquati, Eds., vol. 11014 of *Lecture Notes in Computer Science*, Springer, pp. 811–825.

[22] Hackbusch, W. A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices. *Computing 62*, 2 (May 1999), 89–108.

[23] Hackbusch, W., Khoromskij, B., and A. Sauter, S. *On $\mathcal{H}^2$-Matrices.* 08 2000, pp. 9–29.

[24] Halko, N., Martinsson, P.-G., and Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review 53*, 2 (2011), 217–288.

[25] Harbrecht, H., and Zaspel, P. A scalable h-matrix approach for the solution of boundary integral equations on multi-gpu clusters. *CoRR abs/1806.11558* (2018).

[26] Hoque, R., and Shamis, P. Distributed task-based runtime systems-current state and micro-benchmark performance. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2018), IEEE, pp. 934–941.

[27] Ida, A., Iwashita, T., Mifune, T., and Takahashi, Y. Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of information processing 22*, 4 (2014), 642–650.

[28] Kriemann, R. H-lu factorization on many-core systems. *Comput. Vis. Sci. 16*, 3 (June 2013), 105–117.

[29] Kurzak, J., and Dongarra, J. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *International Workshop on Applied Parallel Computing* (2006), Springer, pp. 147–156.

[30] Lizé, B. *Fast direct solver for the boundary element method in electromagnetism and acoustics : H-Matrices. Parallelism and industrial applications.* Theses, Université Paris-Nord - Paris XIII, June 2014.

[31] Masliah, I., Abdelfattah, A., Haidar, A., Tomov, S., Baboulin, M., Falcou, J., and Dongarra, J. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing* (2016), Springer, pp. 659–671.

[32] Pheatt, C. Intel® threading building blocks. *J. Comput. Sci. Coll. 23*, 4 (Apr. 2008), 298–298.

[33] Pichon, G., Darve, E., Faverge, M., Ramet, P., and Roman, J. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Journal of Computational Science 27* (2018), 255 – 270.

[34] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and Van de Geijn, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 121–130.

[35] Rouet, F.-H., Li, X. S., Ghysels, P., and Napov, A. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw. 42*, 4 (June 2016), 27:1–27:35.

[36] Sala, K., Teruel, X., Perez, J. M., Peña, A. J., Beltran, V., and Labarta, J. Integrating blocking and non-blocking mpi primitives with task-based programming models. *Parallel Computing* (2018).

[37] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kalé, L. V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., and Beckman, P. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems 29*, 3 (March 2018), 512–526.

[38] Song, F., YarKhan, A., and Dongarra, J. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 19:1–19:11.

[39] Thibault, S. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Habilitation à diriger des recherches, Université de Bordeaux, Dec. 2018.

[40] Tomov, S., Dongarra, J., and Baboulin, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing 36*, 5-6 (June 2010), 232–240.

[41] Xia, J., Chandrasekaran, S., Gu, M., and Li, X. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications 31*, 3 (2010), 1382–1411.

[42] Xia, J., Chandrasekaran, S., Gu, M., and Li, X. S. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications 17*, 6 (2010), 953–976.