Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig

$\mathcal{H}$-LU Factorization on Many-Core Systems

(revised version: November 2014)

by

Ronald Kriemann

Preprint no.: 5 2014

$\mathcal{H}$-LU Factorization on Many-Core Systems

R. Kriemann*

November 28, 2014

A version of the $\mathcal{H}$-LU factorization is introduced, based on the individual computational tasks occurring during the block-wise $\mathcal{H}$-LU factorization. The dependencies between these tasks form a directed acylic graph, which is used for efficient scheduling on parallel systems. The algorithm is especially suited for many-core processors and shows a much improved parallel scaling behavior compared to previous $\mathcal{H}$-LU factorization algorithms.

AMS Subject Classification: 65F05, 65Y05, 65Y20, 68W10, 68W40
Keywords: hierarchical matrices, parallel algorithms, many-core processors, DAG-based

1 Introduction

Hierarchical matrices ($\mathcal{H}$-matrices), introduced in [11], are a powerful tool to represent dense matrices coming from integral equations or partial differential equations in a hierarchical, block-oriented, data-sparse way with log-linear memory costs. Furthermore, a matrix arithmetic, e.g., matrix addition, multiplication, inversion and LU factorization, is possible with log-linear computation costs (see [7]).

In [15] this $\mathcal{H}$-arithmetic was modified to use parallel, shared memory machines. The basic building block of this parallel $\mathcal{H}$-arithmetic was the $\mathcal{H}$-matrix multiplication, which relied on the fact that all per-sub-block computations could be collected and executed in parallel for all matrix blocks.

$\mathcal{H}$-inversion and $\mathcal{H}$-LU factorization are formulated as recursive block algorithms, similar to standard blocked algorithms for dense matrices. Beside the actual recursion, both algorithms are based on $\mathcal{H}$-matrix multiplication. Furthermore, an intrinsic sequential order, e.g., for diagonal elements, is present in these algorithms. Therefore, only the matrix multiplications can be used for parallelization.

For $\mathcal{H}$-inversion, this form of parallelization was usable for machines up to 16 computing cores (see [15]), e.g., *multi-core* CPUs. On the other hand, $\mathcal{H}$-LU factorization only showed a disappointing performance, since in addition to the diagonal, recursion also took place in the off-diagonal part due to matrix solves with upper or lower block tridiagonal matrices.

Especially on today's *many-core* systems with many more than 16 processing cores, this algorithm was not applicable. The only exception are $\mathcal{H}$-matrices from domain-decomposition problems, which consist of several, decoupled systems and therefore provide more room for concurrent computations.

*Max-Planck-Institute for Mathematics in the Sciences, Inselstr. 22–26, 04103 Leipzig, Germany, rok@mis.mpg.de

For dense matrices sequential linear algebra implementations are widely available by the BLAS and LAPACK software libraries ([3]). The algorithms in these software packages are optimized for optimal usage of memory hierarchies in classical (and modern) CPUs. As an example, the LU factorization splits the matrix in blocks of columns, so called *panels*, which are handled at once (see Figure 1). Within a panel, each column is handled one after the other, e.g., finding pivots, swapping rows and updating the remaining elements in the column. Afterwards, the block structure of the panel allows to use matrix multiplication to update the trailing submatrix, thereby optimizing CPU cache usage.

On parallel machines such algorithms are often implemented by using a parallel version of the BLAS library, i.e., the actual algorithm is unmodified and only parallel matrix-vector or parallel matrix-matrix operations are used. However, when using multi- or many-core CPUs, this LU factorization algorithm possesses a critical bottleneck, since all updates need to wait for the current panel to have been computed. Furthermore, parallelization within a panel computation is limited, finally leading to a limited performance on such computer systems (see [5]).

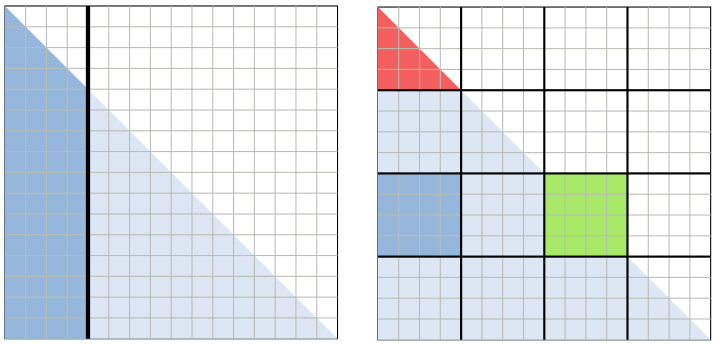

Figure 1: Panel (left) and tile (right) based LU factorization

To fix this problem, the panel was further split into square submatrices or *tiles* (see Figure 1). As soon as a tile in a panel was finished, the update operations for blocks in the trailing submatrix can be started, thereby eliminating the synchronization problem.

In such a dense LU factorization, each operation, e.g., LU factorization of a diagonal tile or the update of a block via matrix multiplication, forms a single *task*, i.e., atomic unit of computation with data dependencies. The tasks and their dependencies hereby form a *directed acyclic graph* (DAG), which can be used to efficiently schedule the tasks onto the different CPU cores. For a detailed introduction into the specific tasks used in a tile-based LU factorization we refer the reader to [4, 5, 17, 18, 21].

Due to the high amount of work per task, these algorithms are also especially suited for graphics processing units (GPUs), for which programming interfaces like CUDA or OpenCL (see [14, 20]) are available. For GPUs or hybrid CPU+GPU systems, the PLASMA software package ([1]) implements dense linear algebra based on tiles with the aim to provide the same function set as LAPACK. The MAGMA software ([2]) extends this to multiple GPUs in a single host system.

An analog approach can also be applied to sparse matrix factorization. In particular the *multi-frontal* approach introduced in [6] is suited for task-based DAG computations due to its *elimination* or *assembly tree* which is used to organize the work. Nodes in an assembly tree correspond to one or more columns of the matrix. While a node can only be processed after all its child nodes, independent subtrees may be handled in parallel. Furthermore, for large nodes, e.g., representing several columns, the work may be split into subtasks. In [12] and [19] the reader will find more details about the design of such DAG-based sparse solvers.

However, although highly efficient in terms of raw CPU power, due to the quadratic/cubic computational complexity, dense linear algebra is no replacement for algorithms with (log-) linear complexity like $\mathcal{H}$-matrices, assuming the latter is applicable to a specific problem. A similar property holds for sparse solvers as the complexity is dependent on the fill-in during LU factorization, which may destroy near-linear complexity (see [8]).

Therefore, in this work the concept of task-based DAG computations is used to split the $\mathcal{H}$-LU factorization into single tasks and to define corresponding dependencies to form a DAG. This task/DAG-based algorithm is able to utilize parallel CPUs much more efficiently compared to the classical recursive algorithm and in particular demonstrates an optimal parallel scaling behavior on many-core systems.

In Section 2 basic terms for $\mathcal{H}$-matrices are defined together with the standard $\mathcal{H}$-LU factorization algorithm. The new task-based algorithm is described in Section 3. Numerical experiments are presented in Section 4, followed by a conclusion in Section 5.

2 Definitions

2.1 $\mathcal{H}$-Matrices

Let I be an index set and $n = \#I$ its cardinality. The basic building block of an $\mathcal{H}$-matrix is the *hierarchical* partitioning of I in the form of a *cluster tree* (or $\mathcal{H}$-tree):

Definition 2.1 (Cluster Tree) *The tree $T_I = (V, E)$ is called a* cluster tree *over I if*

1. *I is the root of T_I and*
2. *for all vertices $v \in V$ either $v = \dot\cup_{v' \in \mathcal{S}(v)} v'$ or v is a leaf of T_I.*

Here, $\mathcal{S}(v)$ is the set of sons of v. The set of leaves of T_I is denoted by $\mathcal{L}(T_I)$. A node in T_I is also called a cluster *and we write $t \in T_I$ if $t \in V$. Nodes in the cluster tree with the same distance from the root are said to be on the same* level *of the tree. Finally, let $T_I^\ell := \{t \in T_I \ : \ \text{level}(t) = \ell\}$.*

Remark 2.2 *The leaves of the cluster tree T_I have an upper limit $n_{\min} > 0$ for their size, e.g., for all $t \in \mathcal{L}(T_I) : \#t \leq n_{\min}$. In practical applications, this is also used to stop further partitioning of clusters, i.e., for $t \in T_I$ if $\#t \leq n_{\min}$, t will be a leaf in T_I. In this sense, $n_{\min}$ is the analog to a minimal block size when working with dense matrices, e.g., to optimize cache performance.*

The cluster tree T_I defines the set of subsets of I which may be used to partition the product index set $I \times I$ and hence, the matrix index set. For an $\mathcal{H}$-matrix to be efficient, the partitioning has to be restricted to a minimal set of *block clusters* $t \times s$ with $t, s \in T_I$, such that $t \times s$ is either small, i.e., $t, s \in \mathcal{L}(T_I)$, or *admissible*. The admissibility of a node is in general defined by the corresponding application. For typical problems involving partial differential equations or integral equations the classical admissibility condition is given by:

$$\min(\text{diam}(t), \text{diam}(s)) \leq \eta \, \text{dist}(t, s) \tag{1}$$

Here, $\text{diam}(\cdot)$ and $\text{dist}(\cdot, \cdot)$ denote the diameter of and distance between geometrical entities associated with the clusters, e.g., the support of basis functions.

Using the admissibility condition, the actual partitioning of $I \times I$ can be constructed. For this, a cluster tree over $I \times I$ is built, with its leaves forming the partition.

Definition 2.3 (Block Cluster Tree) *Let T_I be a cluster tree over I. The* block cluster tree *$T_{I\times I}$ over T_I is defined recursively for a node $b = t \times s$, starting with the root $I \times I$:*

$$\mathcal{S}(b) = \begin{cases} \emptyset & \text{if } b \text{ is admissible or } \mathcal{S}(t) = \emptyset \text{ or } \mathcal{S}(s) = \emptyset, \\ \mathcal{S}' & \text{otherwise} \end{cases}$$

with $\mathcal{S}' := \{t' \times s' : t' \in \mathcal{S}(t), s' \in \mathcal{S}(s)\}$ and repeated for all nodes in $\mathcal{S}(b)$.

The admissibility condition ensures that admissible blocks in $T_{I\times I}$ can be approximated up to a predefined accuracy $\varepsilon \geq 0$ by low-rank matrices with maximal rank k. The set of all such matrices forms the set of $\mathcal{H}$-matrices:

Definition 2.4 ($\mathcal{H}$-Matrix) *For a block cluster tree $T_{I\times I}$ over a cluster tree T_I and $k \in \mathbb{N}$, the set of $\mathcal{H}$-matrices $\mathcal{H}(T_{I\times I}, k)$ is defined as*

$$\mathcal{H}(T_{I\times I}, k) := \Big\{ M \in \mathbb{R}^{I\times I} \;\Big|\; \forall t \times s \in \mathcal{L}(T_{I\times I}) : \operatorname{rank}(M|_{t\times s}) \leq k \vee \{t, s\} \cap \mathcal{L}(T_I) \neq \emptyset \Big\}$$

Low-rank submatrices $M|_{t\times s}$ in $\mathcal{H}$-matrices are represented in the factorized form $M = AB^T$ with $A \in \mathbb{R}^{t\times k}$ and $B \in \mathbb{R}^{s\times k}$. With this, the storage complexity of $\mathcal{H}$-matrices is in $\mathcal{O}(kn \log n)$ (see [7]).

$\mathcal{H}$-matrices also allow matrix arithmetic, e.g., matrix-vector multiplication, matrix addition, multiplication, inversion and LU factorization. With the exception of matrix-vector multiplication, these operations are approximative to ensure the log-linear storage complexity. For this, the sum of low-rank matrices is *truncated* to rank k using a low-rank singular value decomposition (see [7]). This results in a complexity of $\mathcal{O}(k^2 n \log n)$ for matrix addition and $\mathcal{O}\left(k^2 n \log^2 n\right)$ for matrix multiplication, inversion and LU factorization.

Instead of a fixed rank k, construction and arithmetics of $\mathcal{H}$-matrices may also be performed with respect to the precision ε. For this, during the truncation of low-rank matrices, only singular values $s_\ell, 0 \leq \ell < k$, (and corresponding vectors) with $s_\ell \geq \varepsilon \cdot s_0$ are used for the truncation result, while $s_0 \geq s_1 \geq s_2 \ldots$ is assumed. This leads to a variable rank in the subblocks of the $\mathcal{H}$-matrix. In practice, such a fixed-accuracy arithmetics is often more efficient than a fixed-rank arithmetics, which is why it is used for all subsequent examples.

2.2 Sequential $\mathcal{H}$-LU Factorization

For an $\mathcal{H}$-matrix A the factorization $A = LU$ with a unit diagonal lower triangular $\mathcal{H}$-matrix L and an upper triangular $\mathcal{H}$-matrix U is to be computed.

Let $t \times t \in T_{I\times I}$ with $\mathcal{S}(t) = \{t_0, \ldots, t_{\ell-1}\}$. Then, due to the block structure of the matrices $A|_{t\times t}, L|_{t\times t}$ and $U|_{t\times t}$, i.e.,

$$A|_{t\times t} = \begin{pmatrix} A|_{t_0\times t_0} & \cdots & A|_{t_0\times t_{\ell-1}} \\ \vdots & \ddots & \vdots \\ A|_{t_{\ell-1}\times t_0} & \cdots & A|_{t_{\ell-1}\times t_{\ell-1}} \end{pmatrix} =$$

$$\begin{pmatrix} L|_{t_0\times t_0} & & \\ \vdots & \ddots & \\ L|_{t_{\ell-1}\times t_0} & \cdots & L|_{t_{\ell-1}\times t_{\ell-1}} \end{pmatrix} \begin{pmatrix} U|_{t_0\times t_0} & \cdots & U|_{t_0\times t_{\ell-1}} \\ & \ddots & \vdots \\ & & U|_{t_{\ell-1}\times t_{\ell-1}} \end{pmatrix} =$$

$$L|_{t\times t} U|_{t\times t},$$

the $\mathcal{H}$-LU factorization can be formulated recursively as in Algorithm 1.

Algorithm 1 Recursive $\mathcal{H}$-LU factorization.

```
procedure LU( in: A|_{t×t}, out: L|_{t×t}, U|_{t×t} )
    for i := 0, ..., ℓ − 1 do
        LU( A|_{t_i×t_i} );
        for j := i + 1, ..., ℓ − 1 do
            SOLVE_UPPER( A|_{t_j×t_i}, L|_{t_j×t_i}, U|_{t_i×t_i} );
            SOLVE_LOWER( A|_{t_i×t_j}, L|_{t_i×t_i}, U|_{t_i×t_j} );
        for j, k := i + 1, ..., ℓ − 1 do
            A|_{t_j×t_k} := A|_{t_j×t_k} − L_{t_j×t_i} U_{t_i×t_k};
```

In a similar way, a recursive algorithm for solving $B|_{s\times t} = L|_{t\times t}C|_{s\times t}$, $s \times t \in T_{I\times I}, \mathcal{S}(s) = \{s_0, \ldots, s_{\ell-1}\}$, with a given lower triangular $\mathcal{H}$-matrix L, general $\mathcal{H}$-matrix B and unknown $\mathcal{H}$-matrix C is obtained and shown in Algorithm 2.

Algorithm 2 Solve with lower triangular $\mathcal{H}$-matrix.

```
procedure SOLVE_LOWER( in: B|_{s×t}, L|_{t×t}, out: C|_{s×t} )
    for i := 0, ..., ℓ − 1 do
        for j := 0, ..., ℓ − 1 do
            SOLVE_LOWER( B|_{t_i×s_j}, L|_{t_i×t_i}, C|_{t_i×s_j} );
        for k := i + 1, ..., ℓ − 1 do
            for j := 0, ..., ℓ − 1 do
                B|_{t_k×t_j} := B|_{t_k×t_j} − L_{t_k×t_i} C_{t_i×t_j};
```

A similar recursive algorithm can be formulated for solving with an upper triangular $\mathcal{H}$-matrix.

Remark 2.5 *In Algorithm 1 and Algorithm 2 only the recursive part of the actual algorithms is presented, which is used as long as all arguments of the routines are block matrices. If one of the input matrices is a dense or low-rank matrix, special routines are used. This algorithm structure appears in many other algorithms of $\mathcal{H}$-algebra as well.*

Unfortunately, these algorithms only provide very limited opportunities for parallelization. In the case of binary cluster trees, the standard case in practical applications, there are just two matrix solves to be performed in parallel in Algorithm 1. Similarly, the two matrices in a block row may be handled in parallel in Algorithm 2. Only the matrix updates themselves can be computed with optimal parallel scaling. Especially the strict, local execution order together with the recursive nature of these algorithms inhibits parallel execution. Therefore, the achievable parallel speedup is small (cf. [15]).

Remark 2.6 *In [13] an alternative $\mathcal{H}$-LU factorization was introduced, based on an* index-wise *data partitioning where also individual operations per block may be distributed onto several processors. In theory, this yields a much better parallel scaling behavior.*

3 Task-based $\mathcal{H}$-LU Factorization

Before the task-based $\mathcal{H}$-LU factorization is defined, the corresponding formulation for dense matrices is described. The concepts can then be generalized to block structures in $\mathcal{H}$-matrices.

3.1 LU Factorization of Dense Matrices

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix, i.e., represented by storing all matrix coefficients explicitly, and the dense LU factorization $A = LU$ is sought. In particular, the *blocked* (or *tiled*) LU factorization is of interest, i.e., for a given block size $N > 0$ with $n = n_b \cdot N$ for some $n_b \in \mathbb{N}$, A decomposes into blocks $A_{ij} \in \mathbb{R}^{N \times N}, 0 \leq i, j < n_b$. After factorizing the diagonal block $A_{ii} = L_{ii}U_{ii}$, all blocks L_{ji} and $U_{ij}, i < j < n_b$ in the current block row and column have to be computed via matrix solves $L_{ii}U_{ij} = A_{ij}$ and $L_{ji}U_{ii} = A_{ji}$. Afterwards all blocks $A_{k\ell}, i < k, \ell < n_b$ of A in the trailing submatrix are updated via $A_{k\ell} := A_{k\ell} - L_{ki}U_{i\ell}$.

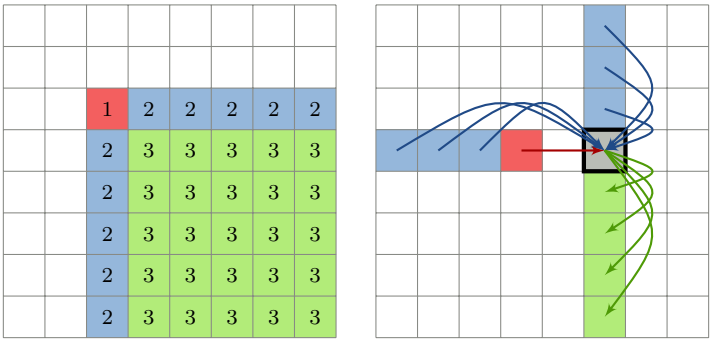

Figure 2: Dense LU factorization: separate algorithm stages (left), incoming and outgoing dependencies (right).

The parallel dense LU factorization algorithm can be described in terms of distinct stages (see Figure 2 (left)), e.g., first factorize the diagonal block, then solve in the current row/column and finally update the trailing matrix:

for $i := 0, \ldots, n_b - 1$ **do**
  FACTORIZE(i);
  parallel for $j := i + 1, \ldots, n_b - 1$ **do**
    SOLVE_UPPER(j, i);
    SOLVE_LOWER(i, j);
  parallel for $k, \ell := i + 1, \ldots, n_b - 1$ **do**
    UPDATE(k, i, ℓ);

where the following functions are used:

- FACTORIZE(i) : compute $A_{ii} = L_{ii}U_{ii}$,
- SOLVE_UPPER(j, i) : solve $L_{ji}U_{ii} = A_{ji}$,
- SOLVE_LOWER(i, j) : solve $L_{ii}U_{ij} = A_{ij}$ and
- UPDATE(k, i, ℓ) : compute $A_{k\ell} := A_{k\ell} - L_{ki}U_{i\ell}$.

Remark 3.1 *Except for the recursion in the case of $\mathcal{H}$-matrices, the block-wise dense LU factorization and the $\mathcal{H}$-LU factorization in Algorithm 1 are identical if the block index is replaced by the corresponding index set. The main difference between dense and $\mathcal{H}$-arithmetic is due to the special routines for handling various combinations of dense, low-rank and block matrices in the $\mathcal{H}$-matrix case (see Remark 2.5).*

Algorithm 3 Constructs a DAG for the dense LU factorization.

```
for i := 0, ..., n_b − 1 do
    task( FACTORIZE( i ) );
    parallel for j := i + 1, ..., n_b − 1 do
        task( SOLVE_UPPER( j, i ) );  task( FACTORIZE( i ) ) → task( SOLVE_UPPER( j, i ) );
        task( SOLVE_LOWER( i, j ) );  task( FACTORIZE( i ) ) → task( SOLVE_LOWER( i, j ) );
    parallel for k, ℓ := i + 1, ..., n_b − 1 do
        task( UPDATE( k, i, ℓ ) );
        task( SOLVE_UPPER( k, i ) ) → task( UPDATE( k, i, ℓ ) );
        task( SOLVE_LOWER( i, ℓ ) ) → task( UPDATE( k, i, ℓ ) );
        if      k = ℓ then task( UPDATE( k, i, ℓ ) ) → task( FACTORIZE( k ) );
        else if k > ℓ then task( UPDATE( k, i, ℓ ) ) → task( SOLVE_UPPER( k, ℓ ) );
        else               task( UPDATE( k, i, ℓ ) ) → task( SOLVE_LOWER( k, ℓ ) );
```

Unfortunately, after the diagonal factorization and the matrix solves, synchronization points exists, where CPU cores have to wait for all others to finish. Especially on many-core CPUs, this is a serious drawback, significantly limiting the parallel speedup of such algorithms.

However, each of these computations, being diagonal factorizations (FACTORIZE(i)), off-diagonal matrix solves (SOLVE_UPPER(j, i), SOLVE_LOWER(i, j)) or matrix updates (UPDATE(k, i, ℓ)), forms a separate computational task with several dependencies between them. Wrapping each function into a **task**($\cdot$) construct yields an algorithm to define the set of all tasks for the dense LU factorization.

Still missing are the dependencies between these tasks, in the following denoted by "$\rightarrow$", i.e., $\mathbf{task}_1 \rightarrow \mathbf{task}_2$ implies $\mathbf{task}_2$ may only start after $\mathbf{task}_1$ has finished.

For $0 \leq i < n_b$ the first part of the dependencies can be directly taken from the above algorithm:

1. (*Diagonal Factorization*) $\forall i < j < n_b$:

$$\begin{aligned}\mathbf{task}(\text{FACTORIZE}(i)) &\rightarrow \mathbf{task}(\text{SOLVE\_UPPER}(j, i)),\\ \mathbf{task}(\text{FACTORIZE}(i)) &\rightarrow \mathbf{task}(\text{SOLVE\_LOWER}(i, j)),\end{aligned}$$

2. (*Update Prerequisites*) $\forall i < k, \ell < n_b$:

$$\begin{aligned}\mathbf{task}(\text{SOLVE\_UPPER}(k, i)) &\rightarrow \mathbf{task}(\text{UPDATE}(k, i, \ell)),\\ \mathbf{task}(\text{SOLVE\_LOWER}(i, \ell)) &\rightarrow \mathbf{task}(\text{UPDATE}(k, i, \ell)).\end{aligned}$$

Furthermore, each factorization or matrix solve also requires all corresponding updates to be applied, which completes the set of dependencies:

3. (*Contributing updates*) $\forall \ell < i < j$:

$$\begin{aligned}\mathbf{task}(\text{UPDATE}(i, \ell, i)) &\rightarrow \mathbf{task}(\text{FACTORIZE}(i)),\\ \mathbf{task}(\text{UPDATE}(j, \ell, i)) &\rightarrow \mathbf{task}(\text{SOLVE\_UPPER}(j, i)),\\ \mathbf{task}(\text{UPDATE}(i, \ell, j)) &\rightarrow \mathbf{task}(\text{SOLVE\_LOWER}(i, j)).\end{aligned}$$

The final algorithm for the dense LU factorization DAG is shown in Algorithm 3, with the matrix block A_{00} being the start node and the corresponding task for matrix block A_{n_b-1,n_b-1}

the end node of the DAG. Incoming edges of a node correspond to data needed before executing the task, e.g., due to an update to the local matrix block, whereas outgoing edges correspond to locally computed data needed by other tasks (see Figure 2 (right)).

Algorithm execution using a DAG has the advantage that any task may be scheduled for execution as soon as all its dependencies are met, thereby overlapping diagonal factorization tasks with off-diagonal solve or update tasks. By this the above limitation of stage-based algorithms with their synchronization points and idle CPU cores can be overcome.

In practice, an optimal scheduling of the tasks depends on various characteristics of the computer system, e.g., processor layout and memory configuration, which usually involves a careful fine tuning of corresponding scheduling algorithms. However, the definition of the DAG remains independent of the computer system as it only reflects the mathematical data dependency. This significantly simplifies algorithm development.

3.2 $\mathcal{H}$-LU Factorization

When extending the task-based algorithm to $\mathcal{H}$-matrices the hierarchy of the matrix blocks in the $\mathcal{H}$-matrix has to be taken into account. As an example, the factorization of diagonal blocks in Figure 3 (left) induce only the matrix solve of blocks on the *same* level (marked blue). Furthermore, matrix blocks may form prerequisites for other blocks on different levels (Figure 3 (right)).

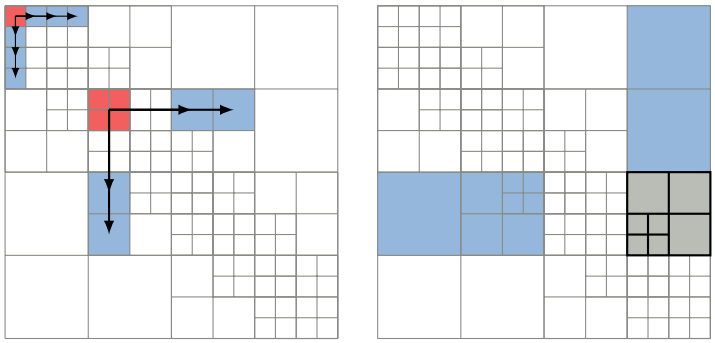

Figure 3: Data dependency between tasks during $\mathcal{H}$-LU factorization.

Similar to the dense case, an algorithmic formulation of the LU factorization helps in the definition of the tasks and their dependencies. However, the algorithms for the $\mathcal{H}$-LU factorization in Section 2.2 have several disadvantages as they only affect sub blocks in the currently handled matrix block. Furthermore, the recursion is split into different algorithms, e.g., the factorization algorithm, the triangular matrix solves and matrix multiplication. All this results in hiding relations between matrix blocks (and tasks), even if they are on the same level of the block cluster tree but in different parts of the $\mathcal{H}$-matrix. Furthermore, additional unnecessary dependencies are introduced due to the structure of these algorithms.

Therefore, a reformulation is needed, where *all* blocks on the same level of the block cluster tree are considered. With this, after the factorization of a diagonal block, all off-diagonal matrix blocks are known, which are ready to be solved. This corresponds to the *global scope* of the LU factorization algorithm in the dense matrix case.

However, one important aspect of the algorithms in Section 2.2 shall *not* be changed: as long as all arguments of a routine are block matrices, recursion is applied (see Remark 2.5). For parallel computations this is furthermore extended as functions involving low-rank or dense matrices are

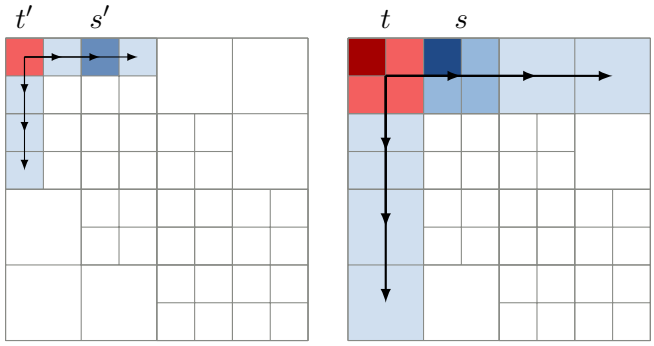

Figure 4: Example of multiple solves per matrix block due to double recursion.

considered *atomic*, i.e., should be handled by a single CPU core. In practice, such operations can be split into subtasks, e.g., when solving a low-rank matrix all vectors may be handled in parallel, but to simplify the presentation this is not considered in this work.

In the following, we need to generalize the block dependencies for the dense LU factorization for the case of $\mathcal{H}$-matrices. For this, the block index is replaced by (sub) index sets.

Definition 3.2 *Let I be an index set. For $t, s \subseteq I$ the relation $<_I$ is defined as: $t <_I s$ iff $\forall i \in t, i' \in s : i < i'$. Furthermore, let $t \leq_I s$ iff $t <_I s \vee t = s$.*

In a first step to a global reformulation, the loop over sons of the current index set for matrix solves and updates in Algorithm 1 is replaced by loops over all index sets on the same level of the block cluster tree. For a diagonal block $A|_{t\times t}, t \times t \in T_{I\times I}$, this implies that all matrix blocks $A|_{s\times t}$ with $s \in T_I^{\text{level}(t)}, s >_I t, s \times t \in T_{I\times I}$, i.e., the block row to the right of the diagonal block, can be solved (Figure 3, left). The analog holds for the block column below and the trailing sub matrix. Let again $\mathcal{S}(t) = \{t_0, \ldots, t_{\ell-1}\}$ be the set of sons of the cluster t.

procedure LU(**in**: $A|_{t\times t}$, **out**: $L|_{t\times t}, U|_{t\times t}$)
  for $i := 0, \ldots, \ell - 1$ **do**
    LU($A|_{t_i\times t_i}$);
    for $s \in T_I^{\text{level}(t_i)}, s >_I t_i$ **do**
      SOLVE_UPPER($A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}$);
      SOLVE_LOWER($A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}$);
    for $s, r \in T_I^{\text{level}(t_i)}, s, r >_I t_i, s \times r \in T_{I\times I}$ **do**
      $A|_{r\times s} := A|_{r\times s} - L_{r\times t_i} U_{t_i\times s}$;

Remark 3.3 *In the above and all subsequent algorithms it is assumed that for a matrix block $A|_{t'\times t''}$ (same for L and U) $t' \times t'' \in T_{I\times I}$ holds. Tests for this important but natural condition for $\mathcal{H}$-arithmetic was left out of the algorithms to improve their readability.*

Unfortunately, this change alone is not sufficient as it introduces potential multiple computations of matrix solves (or updates) for a particular matrix block. For $t, t' \in T_I$ with $t' \subset t$, the factorization of $A|_{t'\times t'}$ will induce the matrix solve of $A|_{s'\times t'}$ with $s' \in T_I^{\text{level}(t')}, s' >_I t'$. However, the same holds for $A|_{t\times t}$ and $A|_{s\times t}$ with $s \in T_I^{\text{level}(t)}, s >_I t$ and $s' \subset s$ (see Figure 4).

The reason for this behavior is, that recursion is still allowed in all algorithms of the $\mathcal{H}$-LU factorization, i.e., also in matrix solves and updates. However, due to the global scope of the

factorization algorithm itself, such a recursion is no longer needed in SOLVE_UPPER/LOWER and matrix updates. All matrix blocks corresponding to leaves are already in the scope of the function LU.

Therefore, to prevent recursion in all functions except LU, at least one operand to matrix solve and matrix update functions has to be associated with a leaf in $T_{I\times I}$. Adding corresponding tests yields the global $\mathcal{H}$-LU Algorithm 4.

Algorithm 4 $\mathcal{H}$-LU factorization with global scope.

```
procedure LU( in: A|_{t×t}, out: L|_{t×t}, U|_{t×t} )
    for i := 0, ..., ℓ − 1 do
        LU( A|_{t_i×t_i} );
        for s ∈ T_I^{level(t_i)}, s >_I t_i do
            if {t_i × t_i, s × t_i} ∩ L(T_{I×I}) ≠ ∅ then
                SOLVE_UPPER( A|_{s×t_i}, L|_{s×t_i}, U|_{t_i×t_i} );
            if {t_i × t_i, t_i × s} ∩ L(T_{I×I}) ≠ ∅ then
                SOLVE_LOWER( A|_{t_i×s}, L|_{t_i×t_i}, U|_{t_i×s} );
        for s, r ∈ T_I^{level(t_i)}, s, r >_I t_i do
            if {r × s, r × t_i, t_i × s} ∩ L(T_{I×I}) ≠ ∅ then
                A|_{r×s} := A|_{r×s} − L_{r×t_i} U_{t_i×s};
```

Remark 3.4 *Algorithm 4 already provides a much improved degree of parallelism compared to Algorithm 1, e.g., all matrix blocks in the current (global) block row/column can be solved in parallel. However, it still contains synchronization points between all CPUs and hence, will not show optimal parallel scaling.*

Having a working algorithm for the global $\mathcal{H}$-LU factorization allows to define an algorithm to construct a corresponding DAG. We start with the definition of all involved tasks. Again, this is done by wrapping all function calls in task constructs. Before this, we simplify the function calls by using the following abbreviations:

- FACTORIZE(t) for LU$(A|_{t\times t})$,
- SOLVE_UPPER(s,t) for SOLVE_UPPER$(A|_{s\times t}, L|_{s\times t}, U|_{t\times t})$,
- SOLVE_LOWER(t,s) for SOLVE_LOWER$(A|_{t\times s}, L|_{t\times t}, U|_{t\times s})$ and
- UPDATE(r,t,s) for $A|_{r\times s} := A|_{r\times s} - L_{r\times t}U_{t\times s}$.

Using these functions and wrapping all function calls for matrix solves and matrix updates in Algorithm 4 in tasks one obtains Algorithm 5 for the task set.

In addition to matrix solves/updates in Algorithm 5 also tasks for diagonal factorizations are used. However, factorization tasks only perform computations if $t_i \times t_i \in \mathcal{L}(T_{I\times I})$. In particular, no recursive LU factorization takes place for non-leaf block clusters (see also Remark 3.5).

Furthermore, the (block) index sets for individual matrix solve and update tasks are collected in the sets $\mathcal{T}_{\text{solve}}$ and $\mathcal{T}_{\text{update}}(\cdot)$ as they will be needed in the following for setting up the dependencies between tasks.

For the task dependencies we start with the relations between diagonal factorization and off-diagonal matrix solves as defined by Algorithm 5. For this, let $t \in T_I$:

Algorithm 5 Defines task set of $\mathcal{H}$-LU factorization.

```
procedure LU_DAG( in: A|_{t×t}, out: L|_{t×t}, U|_{t×t} )
  for i := 0, ..., ℓ − 1 do
    LU_DAG( A|_{t_i×t_i} );
    task(FACTORIZE(t_i));
    for s ∈ T_I^{level(t_i)}, s >_I t_i do
      if {t_i × t_i, s × t_i} ∩ L(T_{I×I}) ≠ ∅ then
        task(SOLVE_UPPER( s, t_i ) );
        T_solve := T_solve ∪ {s × t_i};
      if {t_i × t_i, t_i × s} ∩ L(T_{I×I}) ≠ ∅ then
        task(SOLVE_LOWER( t_i, s ) );
        T_solve := T_solve ∪ {t_i × s};
    for s, r ∈ T_I^{level(t_i)}, s, r >_I t_i do
      if {r × s, r × t_i, t_i × s} ∩ L(T_{I×I}) ≠ ∅ then
        task(UPDATE(r, t_i, s));
        T_update(r × s) := T_update(r × s) ∪ {t_i};
```

1. (*Diagonal Factorization*) $\forall s \in T_I^{\text{level}(t)}, s >_I t$:

$$\{t \times t, s \times t\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset \quad \Longrightarrow \quad \textbf{task}(\text{FACTORIZE}(t)) \rightarrow \textbf{task}(\text{SOLVE\_UPPER}(s,t)),$$
$$\{t \times t, t \times s\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset \quad \Longrightarrow \quad \textbf{task}(\text{FACTORIZE}(t)) \rightarrow \textbf{task}(\text{SOLVE\_LOWER}(t,s)),$$

Now let $s, r \in T_I^{\text{level}(t)}$ with $s, r >_I t$. Unfortunately, for the update task for $A|_{r\times s}$, the dependencies cannot be expressed from Algorithm 4 as directly as before, since no tasks for solving $L_{r\times t}$ or $U_{t\times s}$ may exist, e.g., if the diagonal and off-diagonal blocks are both non-leaf matrix blocks. Instead, all tasks corresponding to subblocks of $L_{r\times t}$ and $U_{t\times s}$ are needed to ensure that updates take place only after all computations for these subblocks have been finished.

For this, the set $\mathcal{T}_{\text{solve}}$ can be used as it contains all block index sets for which matrix solve tasks exist due to Algorithm 5. One notes that by construction, $t \times s \in \mathcal{T}_{\text{solve}} \implies \forall\, t' \times s' \in T_{I\times I} \setminus \{t \times s\} : t \times s \cap t' \times s' = \emptyset$, i.e., a matrix block is only solved once.

Let again $t \in T_I$. Then the dependencies between solve and update tasks can be formulated as:

2. (*Update Prerequisites*) $\forall r, s \in T_I^{\text{level}(t)}, r, s >_I t$:

$$\forall r' \times t' \in \mathcal{T}_{\text{solve}} \text{ with } r' \times t' \subseteq r \times t : \textbf{task}(\text{SOLVE\_UPPER}(r', t')) \rightarrow \textbf{task}(\text{UPDATE}(r,t,s)),$$
$$\forall t' \times s' \in \mathcal{T}_{\text{solve}} \text{ with } t' \times s' \subseteq t \times s : \textbf{task}(\text{SOLVE\_LOWER}(t', s')) \rightarrow \textbf{task}(\text{UPDATE}(r,t,s)).$$

Finally, update tasks for a matrix block $A|_{r\times s}, r \times s \in T_{I\times I}$, induce dependencies to all matrix solve or factorization tasks acting on $r' \times s' \in T_{I\times I}$ with $r' \times s' \cap r \times s \neq \emptyset$. In contrast to the previous dependency set, here $r' \times s'$ can be on an arbitrary level, i.e., below, equal to or above $r \times s$.

To set up all corresponding dependencies regarding $r \times s$, the set $\mathcal{T}_{\text{update}}(r \times s)$ from Algorithm 5 is used:

3. (*Contributing updates*) :

$$\forall r' \times s' \in \mathcal{T}_{\text{solve}} \cup \{r'' \times r'' \in T_{I\times I}\}, r' \times s' \cap r \times s \neq \emptyset\text{:}$$

$$\forall t \in \mathcal{T}_{\text{update}}(r \times s):$$

$$\begin{cases} r' >_I s' & : \textbf{task}(\text{UPDATE}(r,t,s)) \rightarrow \textbf{task}(\text{SOLVE\_UPPER}(r',s')) \\ r' <_I s' & : \textbf{task}(\text{UPDATE}(r,t,s)) \rightarrow \textbf{task}(\text{SOLVE\_LOWER}(r',s')) \\ r' = s' & : \textbf{task}(\text{UPDATE}(r,t,s)) \rightarrow \textbf{task}(\text{FACTORIZE}(r')) \end{cases}$$

Here, the set $\{r'' \times r'' \in T_{I\times I}\}$ of diagonal block clusters is included for all factorization tasks.

In Figure 5 the corresponding dependencies are shown for the marked matrix block. Here, the pairs of blue matrix blocks form contributing updates, whereas the marked matrix block itself forms an update prerequisite for all green matrix blocks. Finally, the diagonal factorization dependency is marked red.

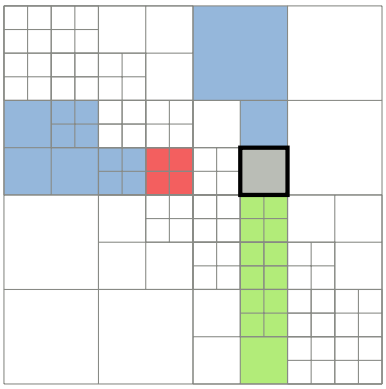

Figure 5: Data dependency between tasks during $\mathcal{H}$-LU factorization.

Dependencies due to diagonal factorization and update prerequisites can be integrated into Algorithm 5. However, setting up dependencies due to contributing updates during the computation of the task set is more involved, since first all corresponding tasks from all levels in $T_{I\times I}$ have to be known. Therefore, this is performed in a post-processing step after computing the whole task set by a simple tree traversal from the root to the leaves and shifting down dependencies from $\mathcal{T}_{\text{update}}(\cdot)$ until a block cluster in $\mathcal{T}_{\text{solve}}$ is reached (see Algorithm 6).

Algorithm 6 Assigning dependencies due to contributing updates.

```
procedure ASSIGN_DEPEND( in: t × s )
  if t × s ∈ T_solve then
    for t' × s' ∈ T_{I×I} with t' × s' ⊆ t × s do
      for r ∈ T_update(t' × s') do
        if t <_I s then
          task(UPDATE(t', r, s'))→task(SOLVE_LOWER(t, s));
        else
          task(UPDATE(t', r, s'))→task(SOLVE_UPPER(t, s));
  else
    for i, j := 0, ..., ℓ − 1 do
      T_update(t_i × s_j) := T_update(t_i × s_j) ∪ T_update(t × s);
      ASSIGN_DEPEND(t_i × s_j);
```

The remaining procedure for computing the DAG for an $\mathcal{H}$-LU factorization is shown in Algorithm 7. Here, the factorization of the upper left (leaf) matrix block represents the start node of the DAG and the factorization of the lower right (leaf) matrix block its end node.

Algorithm 7 Computes DAG for $\mathcal{H}$-LU factorization (except contributing updates).

procedure LU_DAG(**in**: $A|_{t\times t}$, **out**: $L|_{t\times t}, U|_{t\times t}$)
  for $i := 0, \ldots, \ell - 1$ **do**
    LU_DAG($A|_{t_i \times t_i}$);
    task(FACTORIZE(t_i));
    for $s \in T_I^{\text{level}(t_i)}, s >_I t_i$ **do**
      if $\{t_i \times t_i, s \times t_i\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
        task(SOLVE_UPPER(s, t_i));
        $\mathcal{T}_{\text{solve}} := \mathcal{T}_{\text{solve}} \cup \{s \times t_i\}$;
        task(FACTORIZE(t_i)) $\rightarrow$ **task**(SOLVE_UPPER(s, t_i));
      if $\{t_i \times t_i, t_i \times s\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
        task(SOLVE_LOWER(t_i, s));
        $\mathcal{T}_{\text{solve}} := \mathcal{T}_{\text{solve}} \cup \{t_i \times s\}$;
        task(FACTORIZE(t_i)) $\rightarrow$ **task**(SOLVE_LOWER(t_i, s));
    for $r, s \in T_I^{\text{level}(t_i)}, r, s >_I t_i$ **do**
      if $\{r \times s, r \times t_i, t_i \times s\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
        task(UPDATE(r, t_i, s));
        $\mathcal{T}_{\text{update}}(r \times s) := \mathcal{T}_{\text{update}}(r \times s) \cup \{t_i\}$;
        for all $r' \times t' \in \mathcal{T}_{\text{solve}}, r' \times t' \subseteq r \times t_i$ **do**
          task(SOLVE_UPPER(r', t')) $\rightarrow$ **task**(UPDATE(r, t_i, s));
        for all $t' \times s' \in \mathcal{T}_{\text{solve}}, t' \times s' \subseteq t_i \times s$ **do**
          task(SOLVE_LOWER(t', s')) $\rightarrow$ **task**(UPDATE(r, t_i, s));

An example of the resulting DAG is presented in Figure 6. To reduce the number of edges, only one half of the graph is shown, i.e., no dependencies with respect to solving with a lower triangular matrix. Furthermore, the DAG is split into subgraphs for each level in the $\mathcal{H}$-matrix to emphasize the dependencies between different levels.

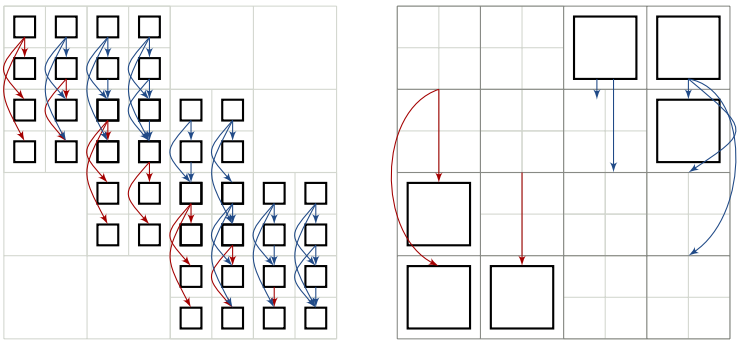

Figure 6: Example for a DAG for $\mathcal{H}$-LU.

Remark 3.5 *The above defined DAG for the $\mathcal{H}$-LU factorization only represents one possible solution to the problem. Alternative DAGs may introduce more nodes to collect dependencies,*

e.g., for matrix solve tasks for non-leaf off-diagonal blocks to agglomerate all dependencies due to subblocks, thereby reducing the number of dependencies to update tasks. Furthermore, different LU strategies, e.g., variations of left-looking or right-looking strategies, may also lead to different tasks and therefore to different DAGs. Also possible subtasks of special routines for handling dense and low-rank matrices can be integrated into the $\mathcal{H}$-LU-DAG (see Remark 2.5).

As in the dense case, tasks in the DAG may be executed as soon as all their dependencies are met, which again allows to overlap diagonal factorization with solve or update operations. In contrast to this, the classical recursive $\mathcal{H}$-LU factorization is restricted by the recursive block hierarchy, which enforces synchronization of all CPU cores at the diagonal blocks. This becomes obvious when the individual stages of both factorization algorithms, i.e., groups of blocks which may by computed simultaneously, are explicitly shown as in Figure 7. There, for the task-based algorithm the execution may follow the anti-diagonal of the matrix (Figure 7, right), while the recursive algorithm has to finish diagonal matrix blocks before computing off-diagonal matrix blocks (Figure 7, left).

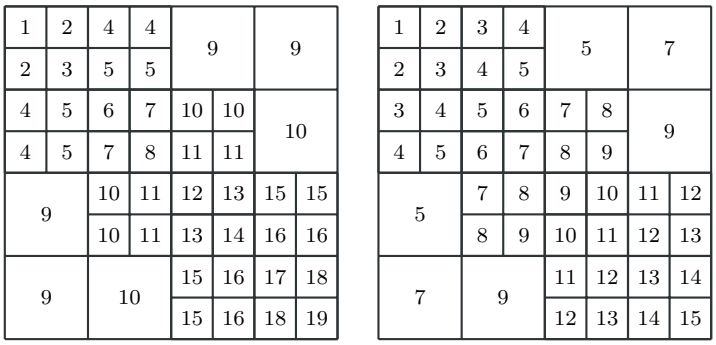

Figure 7: Stages of recursive (left) and task-based (right) $\mathcal{H}$-LU factorization.

Remark 3.6 *In Figure 7, we assumed* linear *computation costs per block to simplify the presentation.*

3.3 $\mathcal{H}$-LU Factorization with Domain Decomposition

$\mathcal{H}$-matrices for the domain decomposition case possess a special structure with large zero blocks, which will remain zero during $\mathcal{H}$-LU factorization (see Figure 8), thereby yielding a more efficient algorithm. This is achieved by decoupling subdomains by an *inner boundary* (or *interface*), and ordering the indices accordingly (refer to [9, 10]). Due to this decoupling, matrix blocks with index sets associated with different domains only have zero coefficients. Matrix blocks with interface indices are non-zero and are handled as standard $\mathcal{H}$-matrices, i.e., based on standard geometrical or algebraic clustering algorithms.

Diagonal matrix blocks corresponding to different subdomains may be handled in parallel, followed by the factorization of the matrix block with indices of the inner boundary. Therefore the parallel scaling behavior of the domain decomposition approach strongly depends on the size of this interface block. For $\mathcal{H}$-matrices geometrical and black-box algorithms for computing a suitable interface have been developed in [9] and [10].

In case of *nested dissection*, the domain decomposition approach is recursively applied to the subdomains.

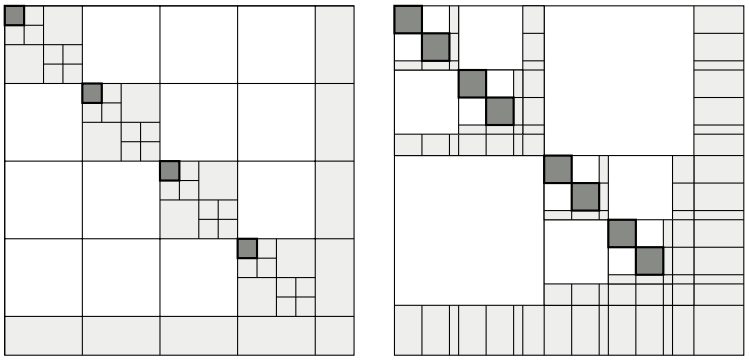

Figure 8: Matrix structure for the domain decomposition case (left) and nested dissection case (right).

Due to this structure, the DAG of the $\mathcal{H}$-LU factorization will not have a single but n_d start nodes in the case of domain decomposition with n_d domains and 2^ℓ start nodes for the nested dissection case, where ℓ is the recursion depth (marked blocks in Figure 8). Hence, the inherent parallelism of the domain decomposition approach is automatically utilized with the task-based approach.

4 Numerical Results

All examples are computed on an Intel XeonPhi 5110P accelerator card with 60 CPU cores and 8 GB main memory.

Due to the high number of CPU cores, this computing platform serves as an example of a many-core system. In contrast to other accelerator cards derived from graphic cards, the XeonPhi card can be programmed very similarly to classical CPUs, e.g., without special programming interfaces like CUDA or OpenCL. In particular, the whole program may run *natively* on the accelerator card.

This native execution mode was also used in the experiments below as it avoids any additional communication between the accelerator card and the host system due to *offloading* certain computations as in the OpenCL case.

However, the native execution mode is limited by the relatively small amount of main memory available for the program, e.g., compared to typical memory sizes of standard compute servers. This limitation will be overcome by the next generation of these processors.

Remark 4.1 *The Intel XeonPhi 5110P also supports the simultaneous execution of up to four threads per core if different threads can use different (sub) units of a single core, referred to as* hyper-threading*. This feature was not used during the benchmarks since the algorithms did benefit only little by it, presumably since the individual tasks are still too large for hyper-threading.*

All computations are performed using the software library HLIBpro ([16]). Low-level linear algebra functions are provided by the *Intel Math Kernel Library* v11.0. For task handling and task scheduling the *Threading Building Blocks* v4.2 are used.

The parallel performance is measured by the parallel speedup $S(p) = t_{\text{seq}}/t(p)$. Here, t_{seq} is the run-time of the best sequential implementation, whereas $t(p)$ is the run-time of the parallel algorithm for p processors or CPU cores. Due to overhead, the task-based algorithm shows a slightly larger run-time compared to the recursive algorithm if only a single CPU core is used,

i.e., in the sequential case. Therefore for $p = 1$ the recursive $\mathcal{H}$-LU factorization was used for all experiments below.

4.1 Integral Equations

We consider the integral equation

$$\int_\Gamma \kappa(x, y)u(x)dy = f(x), \quad x \in \Gamma$$

with $\Gamma = \partial\Omega, \Omega \subset \mathbb{R}^3$, and kernel function κ. Galerkin's method with piecewise constant trial functions is applied for discretisation on a triangulation of Ω.

Two different domains Ω are used for the numerical tests: a spherical grid with 47720 triangles and a cylindrical grid with 41728 triangles (see Figure 9). The cylindrical grid has characteristics of a one-dimensional problem and should give some practical lower bounds for the achievable speedup.

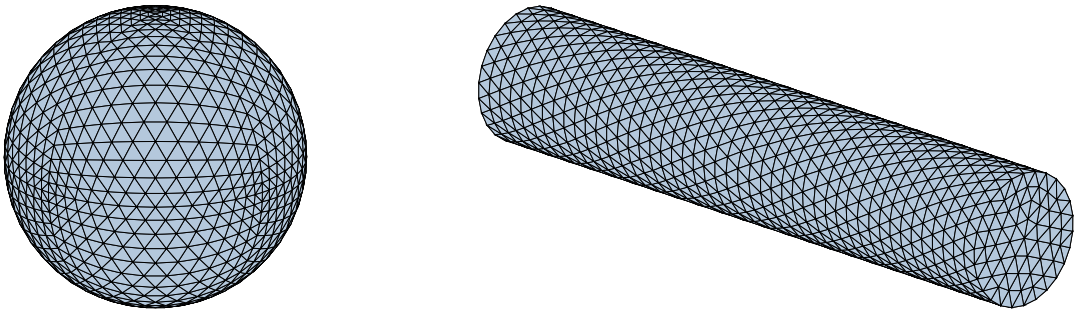

Figure 9: Grids used for $\mathcal{H}$-LU factorization.

Using the standard admissibility (1) with $\eta = 2$ and $n_{\min} = 20$, the resulting block partitions are shown in Figure 10. The similarity of the cylinder problem with the one-dimensional problem is obvious with large off-diagonal blocks.

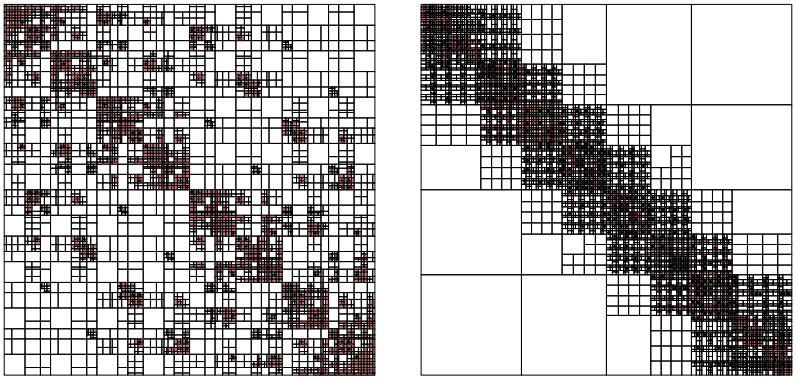

Figure 10: Block partitioning for the sphere (left) and cylinder (right) grid.

Furthermore, different kernel functions are used for the experiments:

- for the Laplace single layer potential $k_1(x, y) = \frac{1}{|x-y|}$ and

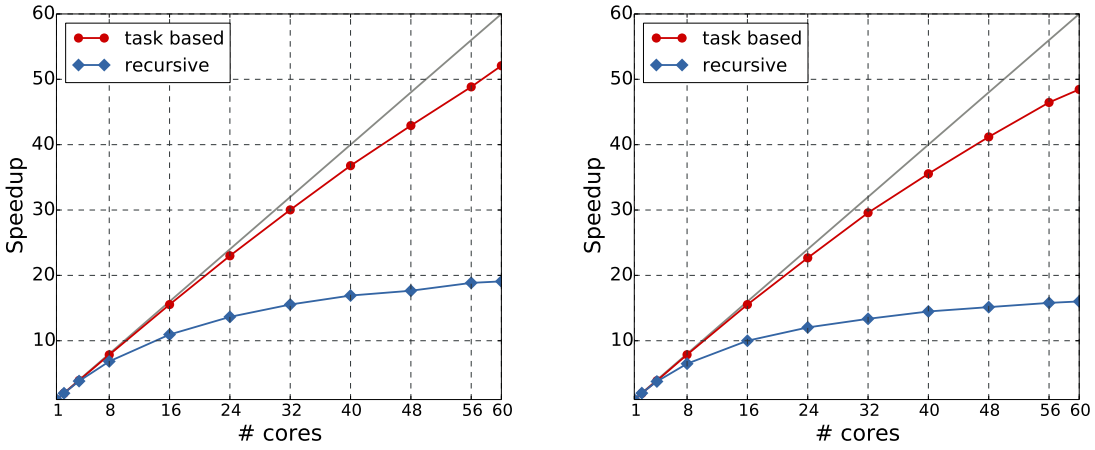

Figure 11: Speedup of task-based and recursive $\mathcal{H}$-LU factorization for Laplace kernel on a sphere (left) and cylinder (right) grid.

- for the Helmholtz single layer potential $k_2(x,y) = \frac{e^{i\kappa|x-y|}}{|x-y|}$

Here, $\kappa = 2 + i$ denotes the wave number (low-frequency case).

For both grids and kernel functions the $\mathcal{H}$-LU factorization was computed for a fixed block-wise accuracy of $\varepsilon = 10^{-4}$ (relative with respect to the spectral norm). The measured time for the task-based $\mathcal{H}$-LU factorization includes the time for constructing the DAG.

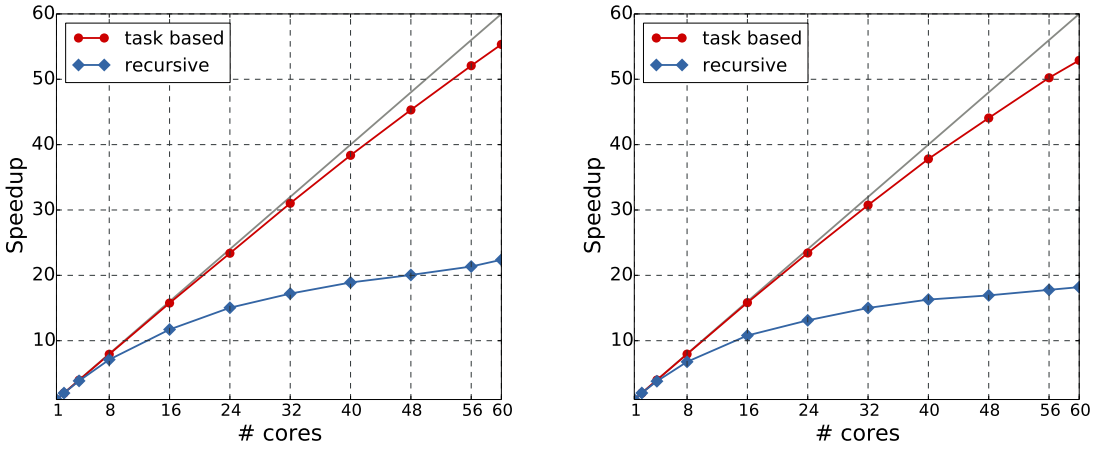

Figure 12: Speedup of task-based and recursive $\mathcal{H}$-LU factorization for Helmholtz kernel on a sphere (left) and cylinder (right) grid.

In Figure 11 the parallel speedup for the task-based and the classical recursive $\mathcal{H}$-LU factorization for the Laplace kernel function is shown. The advantage of the task-based approach is obvious with an almost optimal speedup.

Computations on the cylindrical grid result in a slightly reduced parallel speedup since the number of tasks is smaller and fewer very costly tasks are present compared to the spherical grid. However, the difference of the maximal speedup for both grids is small.

When computing the $\mathcal{H}$-LU factorization for the Helmholtz kernel function, as shown in

Figure 12, the parallel speedup of both algorithms is even higher due to the more costly complex arithmetic and a higher rank of low-rank blocks. This results in a better ratio of computation to overhead compared to the Laplace case.

However, the general picture is identical with the recursive algorithm only benefiting little from the increased computational work per block and still showing a very limited speedup.

Finally, in Figure 13 the parallel speedup of the $\mathcal{H}$-LU factorization in dependence on the problem size is shown for a fixed (maximal) number of CPU cores. Even for very small problem sizes, the task-based algorithm is capable of reaching a substantial portion of the maximal achievable speedup.

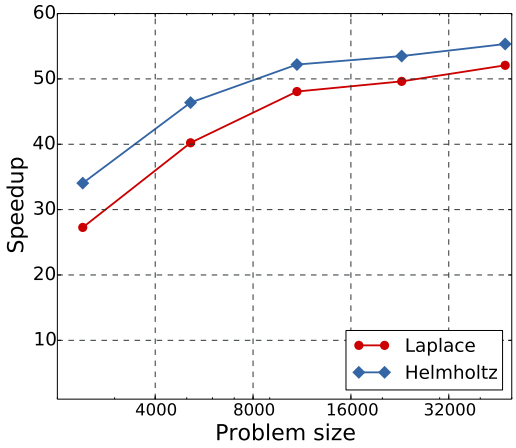

Figure 13: Scaling of $\mathcal{H}$-LU factorization speedup for 60 cores and an increasing problem size for Laplace and Helmholtz kernel.

4.2 Domain Decomposition

As an example for domain decomposition, we consider the *convection-diffusion* equation

$$-\kappa\Delta u + b \cdot \nabla u = f \quad \text{in} \quad \Omega =]0,1[^d, \quad d \in \{2,3\}.$$

with a circular convection direction b:

$$b(x) := \begin{pmatrix} 0.5 - x_2 \\ x_1 - 0.5 \end{pmatrix}.$$

In $\mathbb{R}^3$, the third component of b is zero. The value of κ is set to 10^{-3} resulting in a dominant convection.

The problem is discretised using a finite element method with piecewise linear ansatz functions on a uniform triangulation of Ω with 1046529 unknowns in $\mathbb{R}^2$ and 531441 unknowns in $\mathbb{R}^3$ respectively. The computation of the inner boundary was done using the black-box algorithm described in [10].

In contrast to the integral equation example, the computation of the DAG takes a significant portion of the whole execution time. The main reason for this is that the number of unknowns is much larger and therefore the matrix has more blocks, but on the other hand the costs per unknown are smaller. In the current implementation, the DAG is still computed sequentially, thereby limiting the parallel speedup.

In Figure 14 the speedup of the $\mathcal{H}$-LU factorization for the 2D and the 3D case are shown. In the 2D case, both algorithms show an almost identical parallel speedup if the computation of the DAG is taken into account. Without this, the speedup is much better, demonstrating the currently large portion of the DAG computation on the $\mathcal{H}$-LU-factorization.

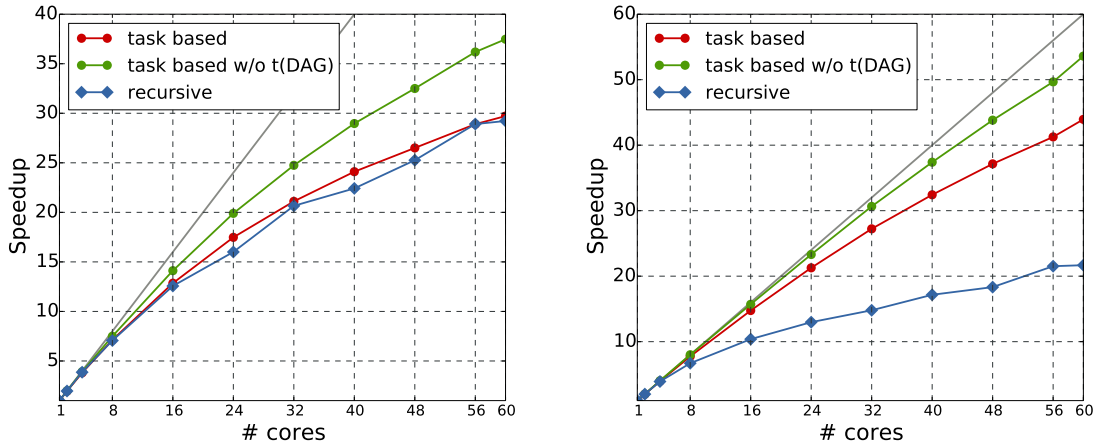

Figure 14: Speedup of $\mathcal{H}$-LU factorization for convection diffusion equation and nested dissection in $\mathbb{R}^2$ (left) and $\mathbb{R}^3$ (right).

Especially for the classical recursive algorithm, the parallel speedup is mainly due to the inherent parallel structure of the factorization because of the block decoupling (see [10]).

However, even without the computation of the DAG, the parallel speedup is less than in the case of dense matrices, e.g., for an integral equation. The reason for this is management overhead for handling the DAG during execution while at the same time the computation costs per block are small.

In the 3D case, the situation changes, since the classical LU factorization is limited by the larger interfaces in $\mathbb{R}^3$. On the other hand, the task-based algorithm can handle tasks associated with the interface blocks as soon as possible, thereby overcoming this limitation.

As can be seen in Figure 15 the parallel speedup is close to the maximal achievable speedup already for small problem sizes. Nevertheless, again the large negative influence of the (still sequential) DAG computation is also visible.

5 Conclusion

On many-core systems a formulation of the $\mathcal{H}$-LU factorization based on individual tasks and the dependencies between them was introduced, which lead to an almost optimal parallel scaling behavior for different problem classes. Furthermore, the principal technique of task-based parallelism is applicable to various other $\mathcal{H}$-matrix algorithms and by this forms a foundation for $\mathcal{H}$-matrix implementations on such architectures for future processors.

The modified execution order of the $\mathcal{H}$-LU factorization with global scope, e.g., the computation of blocks in a global block row instead of just local subblocks, is also applicable on distributed memory machines. Here, the $\mathcal{H}$-matrix is distributed among several computing nodes in some suitable way and node-local computations can be performed as soon as the data is available. First numerical experiments indicate, that again, the modified factorization algorithm yields a higher parallel speedup compared to previous implementations.

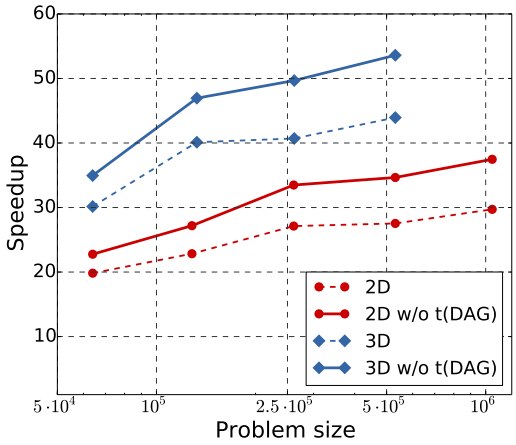

Figure 15: Scaling of $\mathcal{H}$-LU factorization speedup for 60 cores and an increasing problem size for nested dissection in $\mathbb{R}^2$ and $\mathbb{R}^3$.

The existing limitation of the implementation due to the sequential computation of the DAG can be overcome by omitting the DAG and using a dynamic task creation system, e.g., spawn matrix solve tasks after the factorization of a diagonal block. Albeit, such an approach would make the handling of all dependencies for a single task more involved.

References

[1] E. Agullo et al. *PLASMA Users' Guide*. University of Tennessee: Electrical Engineering and Computer Science Department, 1997.

[2] E. Agullo et al. "*Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012037.

[3] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[4] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "*Parallel tiled QR factorization for multicore architectures*". In: *Concurrency and Computation: Practice and Experience* 20.13 (2008), pp. 1573–1590.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "*A class of parallel tiled linear algebra algorithms for multicore architectures*". In: *Parallel Computing* 35.1 (2009), pp. 38–53.

[6] I. S. Duff and J. K. Reid. "*The Multifrontal Solution of Indefinite Sparse Symmetric Linear*". In: *ACM Trans. Math. Softw.* 9.3 (Sept. 1983), pp. 302–325. ISSN: 0098-3500. DOI: 10.1145/356044.356047. URL: http://doi.acm.org/10.1145/356044.356047.

[7] L. Grasedyck and W. Hackbusch. "*Construction and arithmetics of $\mathcal{H}$-matrices*". In: *Computing* 70 (2003), pp. 295–334.

[8] L. Grasedyck, W. Hackbusch, and R. Kriemann. "*Performance of $\mathcal{H}$-LU Preconditioning for Sparse Matrices*". In: *Computational Methods in Applied Mathematics* 8.4 (2008), pp. 336–349.

[9] L. Grasedyck, R. Kriemann, and S. Le Borne. "*Domain-decomposition based $\mathcal{H}$-matrix preconditioners*". In: *Proceedings of DD16*. Vol. 55. LNSCE. Springer-Verlag Berlin, 2006, pp. 661–668.

[10] Lars Grasedyck, Ronald Kriemann, and Sabine LeBorne. "*Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems*". In: *Computing and visualization in science* 11.4-6 (2008), pp. 273–291. ISSN: 1432-9360. DOI: 10.1007/s00791-008-0098-9.

[11] W. Hackbusch. "*A sparse matrix arithmetic based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices*". In: *Computing* 62 (1999), pp. 89–108.

[12] J. Hogg, J. Reid, and J. Scott. "*Design of a Multicore Sparse Cholesky Factorization Using DAGs*". In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3627–3649.

[13] M. Izadi. "*Hierarchical Matrix Techniques on Massively Parallel Computers*". PhD thesis. University of Leipzig, 2012.

[14] Khronos OpenCL Working Group. *The OpenCL Specification*. URL: http://www.khronos.org/registry/cl/.

[15] R. Kriemann. "*Parallel $\mathcal{H}$-matrix arithmetics on shared memory systems*". In: *Computing* 74 (2005), pp. 273–297.

[16] Ronald Kriemann. *HLIBpro*. URL: http://www.hlibpro.com/.

[17] J. Kurzak, A. Buttari, and J. Dongarra. "*Solving systems of linear equations on the CELL processor using Cholesky factorization*". In: *Parallel and Distributed Systems, IEEE Transactions on* 19.9 (2008), pp. 1175–1186.

[18] J. Kurzak and J. Dongarra. "*QR factorization for the Cell Broadband Engine*". In: *Scientific Programming* 17.1-2 (2009), pp. 31–42.

[19] Xavier Lacoste et al. *Sparse direct solvers with accelerators over DAG runtimes*. Anglais. Rapport de recherche RR-7972. INRIA, 2012, p. 11. URL: http://hal.inria.fr/hal-00700066.

[20] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "*Scalable Parallel Programming with CUDA*". In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730.

[21] E. S. Quintana-Ortí and R. A. Van De Geijn. "*Updating an LU factorization with pivoting*". In: *ACM Transactions on Mathematical Software (TOMS)* 35.2 (2008), p. 11.