Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig

Parallel H-Matrix Arithmetic on Distributed-Memory Systems

by

Mohammad Izadi Khaleghabadi

Preprint no.: 73 2012

Parallel $\mathscr{H}$-Matrix Arithmetic on Distributed-Memory Systems

Mohammad Izadi*
Max Planck Institute for Mathematics in the Sciences,
Inselstr. 22, D-04103

Leipzig, Germany
December 24, 2012

Abstract

In the last decade, the hierarchical matrix technique was introduced to deal with dense matrices in an efficient way. It provides a data-sparse format and allows an approximate matrix algebra of nearly optimal complexity. This paper is concerned with utilizing multiple processors to gain further speedup for the $\mathscr{H}$-matrix algebra, namely matrix truncation, matrix-vector multiplication, matrix-matrix multiplication, and inversion.

One of the most cost-effective solution for large-scale computation is distributed computing. Distribute-memory architectures provide an inexpensive way for an organization to obtain parallel capabilities as they are increasingly popular. In this paper, we introduce a new distribution scheme for $\mathscr{H}$-matrices based on the corresponding index set. Numerical experiments applied to a BEM model will complement our complexity analysis.

keywords: Hierarchical matrices, parallel algorithm, distributed-memory systems.

Mathematics Subject Classification (2000) 65F05, 65Y05, 65Y20, 68W10, 68W40.

1 Introduction

Solving integral equations by numerical methods based on boundary element methods, or approximating the inverse of elliptic partial differential operators discretized by finite element methods, lead to a linear system of equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $n \times n$ fully populated matrix. The required storage and execution time for the corresponding standard matrix operations ($\boldsymbol{Ax}, \boldsymbol{A}+\boldsymbol{B}, \boldsymbol{A}\cdot\boldsymbol{B}, \boldsymbol{A}^{-1}$, LU decomposition) quickly become impractical as larger problems are considered.

The technique of hierarchical matrices [6] ($\mathscr{H}$-matrices) proposes a strategy to reduce these requirements significantly up to $\mathcal{O}(n\log^c n)$ with moderate constant c. Therefore, this method with logarithmic-linear complexity can even compete with the standard iterative methods in case of solving large sparse linear systems.

*Email:izadi@mis.mpg.de

Unfortunately, due to the large hidden constant involved in the computational complexity estimates, $\mathscr{H}$-matrices are limited to compete with the state-of-the-art iterative solvers which also can be found in highly efficient parallel versions. Therefore, to be still one of the fastest techniques, one has to accelerate their performances. One common way to do that is to exploit the parallelism features of these methods in an efficient manner.

Parallel algorithms in the context of $\mathscr{H}$-matrices were already implemented on shared-memory systems in [9]. These algorithms are shown to behave nearly optimal w.r.t. speedup and efficiency on such systems while utilizing a block-wise $\mathscr{H}$-matrix distribution. Furthermore, implementing $\mathscr{H}$-matrix construction as well as matrix-vector multiplication based on the block-wise distribution on distributed-memory systems has proved to have a similar behavior as on shared-memory [1]. However applying the block-wise $\mathscr{H}$-distribution does not scale well for the more complicated $\mathscr{H}$-algebra, namely matrix-matrix multiplication and inversion due to a significant inter-processor communication overhead on distributed-memory architectures. As an alternative, we investigate the index-wise $\mathscr{H}$-distribution with a higher applicability on distributed machines.

For large-scale computations, the platform of choice will be large-scale distributed-memory systems. Distributed-memory machines allow a cost-effective way to achieve scalability as the problem size and number of processors grow. However, on these computers, communication is typically much slower than computation, so to achieve peak performance on distributed-memory we need to keep the amount of communication low when designing parallel algorithms. In the context of $\mathscr{H}$-matrices, this aim can be obtained by means of an index-wise $\mathscr{H}$-distribution.

The rest of this paper is structured as follows. In the next section, the basic concepts of $\mathscr{H}$-matrices are defined together with a model problem from BEM that describes the applicability of $\mathscr{H}$-matrices and which also is used for computational experiments. Section 3 is devoted to how to partition an $\mathscr{H}$-matrix among processors such that in related parallel algorithms we have a low communication cost. Finally in Section 4, parallel algorithms for the $\mathscr{H}$-matrix truncation, matrix-vector multiplication, matrix multiplication, and inversion are proposed. Numerical test for each $\mathscr{H}$-matrix operation, applied to our BEM model will confirm the corresponding theoretical complexity.

This paper is a summary of the original work [7], where we address the algorithms and complexity estimates of parallel $\mathscr{H}$-matrix algebra in detail.

2 Hierarchical Matrices

The technique of $\mathscr{H}$-matrices uses a tree like data-sparse structure to store a dense matrix such that the leaves of the tree are dense or *low-rank* matrices. The $\mathscr{H}$-matrix format relies on a hierarchical tree structure called *block cluster* tree, which is obtained by a hierarchical partitioning of the index set into subblocks. During the construction of the block cluster tree at each level of partitioning, these subblocks have to be tested using a so-called *admissibility condition*, that determines whether they are leaf or should be further partitioned. We briefly introduce these key concepts of $\mathscr{H}$-matrices and for details refer the reader to [2] and [5].

2.1 Definitions and Notations

Definition 2.1 (Cluster tree) *Let I be a finite index set. By $T(I)=(V,E)$ we denote a tree with vertices V and edges E ($\subseteq V\times V$). For a vertex $v\in V$ we define the set of all its sons by $S(v):=\{w\in V\mid (v,w)\in E\}$. The tree $T(I)$ is called a cluster tree over I, if the following conditions are fulfilled:*

- *$I\in V$ is the root of $T(I)$ and is denoted by $root(T)$ and $\forall v\in V, v\neq\emptyset\Rightarrow v\subseteq I$.*
- *If $v\in V$ is not a leaf, i.e. $S(v)\neq\emptyset$ then it is equal to the disjoint union of its sons, that is $v=\dot{\cup}_{w\in S(v)}w$.*

A node $v\in V$ is called cluster.

Let $T:=T(I)$ be a cluster tree. The set of all leaves of the tree T is defined by $\mathscr{L}(T):=\{v\in V\mid S(v)=\emptyset\}$. The set of leaves leads to a partition of the index set I, or $I=\dot{\bigcup}\{v\mid v\in\mathscr{L}(T)\}$. The *level* of a tree T is defined recursively by

$$T^{(0)}:=\{root(T)\},\quad T^{(l)}:=\{v\in V\mid\exists w\in T^{(l-1)}:(w,v)\in E\},$$

for $l\in\mathbb{N}$ and we write level$(v)=l$ if $v\in T^{(l)}$. The *depth* of a tree T is defined as $d(T):=\max\{l\in\mathbb{N}_0\mid T^{(l)}\neq\emptyset\}$. The leaves on level $l=0,\ldots,d(T)$ are denoted by $\mathscr{L}(T,l):=\mathscr{L}(T)\cap T^{(l)}$.

Given two possibly different index sets I and J, the concept of a block cluster tree comes as follows:

Definition 2.2 (Block cluster tree) *Let $T(I)$ and $T(J)$ be two cluster trees over the index sets I and J. A cluster tree $T(I\times J):=T(T(I)\times T(J))=(V,E)$ is a block cluster tree over the product index set $I\times J$ if for all $v\in V$ the following conditions hold:*

- *$T^{(0)}(I\times J)=\{I\times J\}$.*
- *If $v\in T^{(l)}(I\times J)$ then there exist $\tau\in T^{(l)}(I)$ and $\sigma\in T^{(l)}(J)$ such that $v=\tau\times\sigma$.*
- *For sons of $v=\tau\times\sigma$, $\tau\in T(I)$ and $\sigma\in T(J)$ we have*

$$S(v):=\begin{cases}\emptyset & \text{if } S(\tau)=\emptyset \text{ or } S(\sigma)=\emptyset,\\ \{\tau'\times\sigma':\tau'\in S(\tau)\text{ and }\sigma'\in S(\sigma)\} & \text{otherwise.}\end{cases}$$

From a practical point of view and in this paper the clusters $T(I)$ and $T(J)$ are *binary trees,* that is the number of sons for each inner $v\in V$ is exactly two. Consequently, the resulting block cluster tree $T(I\times J)$ is a quad-tree. The leaves of the block cluster tree $T:=T(I\times J)$ provide a block partition of the product index set $I\times J$, namely $I\times J=\dot{\bigcup}_{0\leq l\leq d(T)}\mathscr{L}(T,l)$.

A block cluster tree $T(I\times J)$ contains a hierarchy of partitions over $I\times J$ which terminates at blocks $\tau\times\sigma$, for $\tau\in T(I),\sigma\in T(J)$ provided either the minimum cluster size $n_{min}\geq 1$ is reached or they can be approximated by low-rank matrices in the following data representation:

Definition 2.3 ($\mathscr{R}(k)$-matrix representation) *Let $k \in \mathbb{N}_0$. Any block $\boldsymbol{R} \in \mathbb{R}^b$, $b = \tau \times \sigma$ is called to be stored in an $\mathscr{R}(k)$-matrix representation, if the following factorized form holds*

$$\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T, \quad \boldsymbol{A} \in \mathbb{R}^{\tau \times k}, \boldsymbol{B} \in \mathbb{R}^{\sigma \times k},$$

with $\boldsymbol{A}$ and $\boldsymbol{B}$ stored in full matrix representation. We call $\boldsymbol{R}$ a low-rank or $\mathscr{R}(k)$-matrix.

The rank k is assumed to be the same for all $\mathscr{R}(k)$-blocks and sufficiently small, compared to the size of clusters τ and σ. Since the storage requirement for an $\mathscr{R}(k)$-matrix is $k(|\tau| + |\sigma|)$ instead of $|\tau| \cdot |\sigma|$ for standard full matrices, we obtain considerable savings in the storage and reduce the complexity of the corresponding $\mathscr{H}$-matrix operations (see [5]).

During the construction of the block cluster tree $T(I \times J)$ which can be done recursively, we need some auxiliary condition that check whether a block $b = \tau \times \sigma \in T(I \times J)$ is of appropriate size or if it can be approximated by an $\mathscr{R}(k)$-matrix. This identification is done by an admissibility condition Adm:$T(I \times J) \mapsto \{\text{true, false}\}$. By this, a block cluster b is admissible if Adm(b) =true, otherwise, it is inadmissible.

Definition 2.4 ($\mathscr{H}$-matrix) *Let $k, n_{min} \in \mathbb{N}_0$. The set of $\mathscr{H}$-matrices based on the block cluster tree $T := T(I \times J)$ is defined as*

$$\mathscr{H}(T,k) = \{\boldsymbol{M} \in \mathbb{R}^{I \times J} \mid \forall \tau \times \sigma \in \mathscr{L}(T) : rank(\boldsymbol{M}|_{\tau \times \sigma}) \leq k \quad or \quad \min(|\tau|, |\sigma|) \leq n_{min}\}.$$

The matrix $\boldsymbol{M} \in \mathscr{H}(T,k)$ is said to be stored in $\mathscr{H}$-matrix representation if all admissible blocks are stored in $\mathscr{R}(k)$-matrix representation and all inadmissible blocks with $\min(|\tau|, |\sigma|) \leq n_{min}$ are stored as full matrices. Predefining n_{min} in the range 30-60 has proved efficient in most practical computations.

2.2 Model Problem

As an application of $\mathscr{H}$-matrices we consider the one-dimensional Fredholm integral equation of the first kind. Let the function $\mathscr{F} : [0,1] \mapsto \mathbb{R}$ be given. We are looking for a function $u : [0,1] \mapsto \mathbb{R}$ such that it satisfies the following integral equation

$$\int_0^1 \log|x-y| u(y) dy = \mathscr{F}(x), \quad x \in [0,1]. \tag{2.1}$$

Note that the kernel function $\log|x-y|$ has a singularity along $x = y$. In the following we refer to this example as BEM-example.

Applying the standard *Galerkin* method with piecewise constant ansatz functions $\{\varphi_i\}_{i \in I}$, where $I = \{0, \ldots, n-1\}$ is the corresponding index set, the equation (2.1) will be projected onto the space $\mathscr{V}_h = \text{span}\{\varphi_0, \ldots, \varphi_{n-1}\}$ of finite dimension. This leads to looking for an approximate solution $u_h = \sum_{i \in I} u_i \varphi_i \in \mathscr{V}_h$ with u_j being the solution of a linear system with the coefficient matrix $\boldsymbol{G} := (\boldsymbol{G}_{ij})_{i,j \in I}$,

$$\boldsymbol{G}_{ij} := \int_0^1 \int_0^1 \varphi_i(x) \log|x-y| \varphi_j(y) dy\, dx.$$

The success of low-rank matrix approximations of certain blocks of a block cluster tree $T(I \times J)$ depends on the smoothness properties of the given kernel function. In the context of BEM, the following admissibility condition is frequently used for determining admissible blocks:

Definition 2.5 *Let $\eta > 0$ be a fixed parameter. A block $b = \tau \times \sigma$ is said to satisfy the standard admissibility condition (or η-admissible) if*

$$Adm_\eta(b) = true :\Longleftrightarrow \min(diam(\Omega_\tau), diam(\Omega_\sigma)) \leq \eta \; dist(\Omega_\tau, \Omega_\sigma),$$

where Ω_τ and Ω_σ are the union of the supports of the respective basis functions, i.e. $\Omega_\tau := \bigcup_{i\in\tau} supp(\varphi_i)$, $\Omega_\sigma := \bigcup_{i\in\sigma} supp(\varphi_i)$.

Thanks to the smoothness of the kernel function $\log|x-y|$ far away from the singularity, the above admissibility condition (see [2]) is satisfied and therefore each matrix block $\boldsymbol{R} \in \mathbb{R}^{\tau\times\sigma}$ can be approximated by an $\mathscr{R}(k)$-matrix. Otherwise near the diagonal, each matrix block is inadmissible and then can be represented by a dense full rank matrix. An example of an approximated $\mathscr{H}$-matrix $\tilde{\boldsymbol{G}}$ of $\boldsymbol{G}$ is shown in Fig. 1.

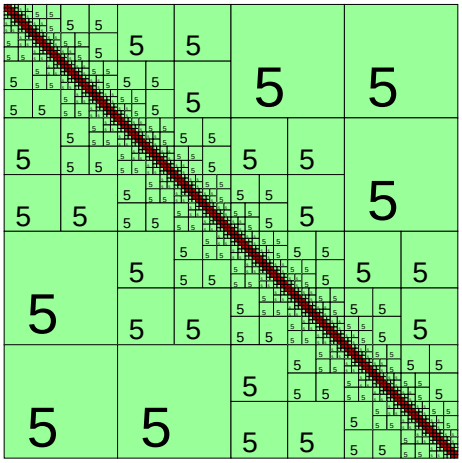

Figure 1: An $\mathscr{H}$-matrix for $n = 4096$, $n_{min} = 32$, and fixed rank $k = 5$.

3 Index-Wise $\mathscr{H}$-Distribution

Partitioning an $\mathscr{H}$-matrix across processors is the primary step towards designing any parallel algorithms on distributed-memory machines. Our approach towards an $\mathscr{H}$-distribution relies on the splitting of the index set $I = \{0,\ldots,n-1\}$ by specifying a block size n_b as the number of contiguous indices from I to be assigned to a single processor. Let $n = 2^d, d \in \mathbb{N}$ and $T(I)$ be a cluster tree over I. Let $P = \{0,\ldots,p-1\}$ be the set of all processors, with $p = 2^{d'} (d' \in \mathbb{N})$ as the number of processors.

In the following we will consider a data distribution among processors based on the cluster tree. A similar distribution map was first discussed in [8] for the special case of the block size $n_b = L := \lceil n/p \rceil$ and only utilized for the $\mathscr{H}$-matrix-vector multiplication algorithm.

Definition 3.1 (Distribution Map) *Assume that I is an index set and P the set of processors. Let*

$$\pi_I : I \longrightarrow P,$$

be a mapping from index set to processor set. Then by the function

$$\begin{aligned} \pi : T(I) &\longrightarrow \mathbb{P}(P) \\ v &\longmapsto \{\pi_I(i) \mid i \in v\}, \end{aligned} \tag{3.1}$$

we will denote the mapping of clusters to associated processor sets, where $\mathbb{P}$ is the power set of P. Additionally, by $I(q)$, we will denote the set of indices local to processor $q \in P$, namely

$$I(q) := \{i \in I \mid \pi(i) = q\}.$$

In many applications, the definition of π_I follows from the definition of $\pi := \pi(I)$. As an example, in the case $n_b = L$ the definition of $\pi = \pi^L$ can be done recursively, starting with the root of the cluster tree by $\pi^L(T^{(0)}) = P$. For $v \in T \setminus \mathscr{L}(T)$ with sons v_0, v_1, and $\pi^L(v) = P' = \{p_0, \ldots, p_1\}$ such that $|P'| > 1$, let $\pi^L(v_0) = P'_0$ and $\pi^L(v_1) = P'_1$ with

$$P'_0 := \{p_0, \ldots, (p_0 + p_1)/2 - 1\} \quad \text{and} \quad P'_1 := \{(p_0 + p_1)/2, \ldots, p_1\},$$

e.g., the processor set P' is halved. If P' contains only one processor, let $\pi^L(v_0) = \pi^L(v_1) = P'$. In fact, π^L is the *block* distribution of the index set I and on each processor $q \in P$ resides the following portion of I

$$I(q) = \{qL, \ldots, (q+1)L - 1\}.$$

It should be noted that, if the underlying cluster tree has an unbalanced structure or an adaptive rank is used for constructing the low-rank blocks in an $\mathscr{H}$-matrix, the definition of π^L leads to a load imbalance situation. Thus we make the following assumption:

Assumption 3.2 *In this work, the distribution mapping π is based on the facts that we have equal costs per index set $i \in I$ and the specially structured block cluster trees from BEM-example are utilized.*

As soon as all clusters are mapped by π to their associated processor sets, the set of all processor groups starting from the root to the leaves constitutes a tree structure similar to $T(I)$. We call this a *processor tree* induced by the mapping $\pi(I)$ and denote it by $P(I)$. An example of $P(I)$ for $p = 4$ processors is shown in Fig. 2.

Generally, we consider a *block-cyclic* distribution of the index set induced by the mapping π that depends on the block size parameter n_b. The values for the size of n_b that we employ in this work are

$$n_b = 2^l n_{min}, \quad l = 0, \ldots, l' := \log(n/(p \cdot n_{min})).$$

Obviously, the largest l corresponds to the block distribution with the block size $n_b = L$. Local indices $I(q)$ consists of multiple groups of successive entries of I of size $2^l n_{min} > 1$ that are alloted to the processor q cyclically which can be expressed as

$$I(q) = \{i \in I \mid q = \lfloor i/(2^l n_{min}) \rfloor \bmod p\}.$$

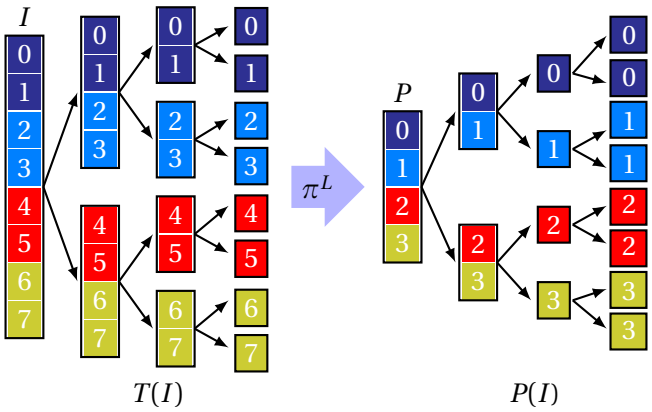

Figure 2: Processor tree $P(I)$ induced by mapping π^L with $L=2$ and $I=\{0,\cdots,7\}$.

This type of distribution means that as soon as the size of a cluster v is greater than $p \cdot n_b$, all processors of its father are completely assigned to that node again, otherwise a smaller number of processors is allocated to the sons of v in a hierarchical manner as the block distribution. The mapping π for $0 \leq l < l'$ can be represented as

$$\pi^l(v) := \begin{cases} P & \text{if } |v| > p \cdot n_b, \\ \pi^L(v) & \text{otherwise,} \end{cases}$$

where π^L is the block distribution with block size $L = \lceil |v|/(2^l n_{min}) \rceil$.

Equipped with a processor mapping for clusters, the data distribution of the $\mathscr{H}$-matrix can be defined. Let us suppose that $T(I \times J)$ is a block cluster tree over the cluster trees $T(I)$ and $T(J)$, where we assume that for the index set J, the mapping $\pi(J)$ and its corresponding processor tree $P(J)$ are defined similarly. Once the mappings $\pi(I)$ and $\pi(J)$ are applied to distribute the index sets I and J across P respectively, the Cartesian mapping $\pi(I \times J) := \pi(I) \times \pi(J)$ with

$$\pi(I \times J) : T(I \times J) \longmapsto P(I) \times P(J),$$

will distribute an $\mathscr{H}$-matrix over all processors.

To be more precise, for all low-rank blocks $\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T \in \mathbb{R}^{\tau \times \sigma}$ with $\tau \in T(I)$ and $\sigma \in T(J)$ each processor q stores only that part of the matrix $\boldsymbol{A}$ corresponding to local indices, e.g., for the set $I(q) \cap \tau$. In the same way, only the local part of $\boldsymbol{B}$ defined by $J(q) \cap \sigma$ is stored (see Fig. 3). Analogously, a similar approach can be applied for each dense matrix $\boldsymbol{D} \in \mathbb{R}^{\tau \times \sigma}$.

4 Parallel Algebra

4.1 Parallel Performance Model

To assess the performance of an algorithm, we require to obtain its communication and arithmetic costs. Due to inter-processor communication for exchanging messages on distributed-memory systems, a cost model is required. Indeed, in each communication step, there are

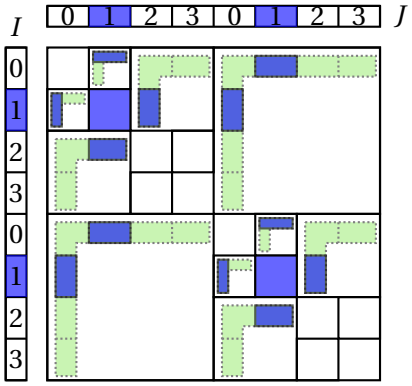

Figure 3: Local data of processor $q = 1$ for an $\mathscr{H}$-matrix for 4 processors.

two sources of overhead: latency and bandwidth. The communication cost is evaluated using the formula [4]

$$t_{comm} = \alpha + \beta \cdot m,$$

for sending a message of m words for a single transfer operations, where α is the latency, and β is the inverse of the network bandwidth. Therefore the total running time of an algorithm while ignoring overlap of communication and computation can be measured as follows:

$$T = \alpha \cdot (\#\text{messages}) + \beta \cdot (\#\text{words sent}) + (\#\text{flops}). \tag{4.1}$$

Based on this model we obtain a parallel complexity for the corresponding $\mathscr{H}$-matrix algebra and justify these complexities by numerical experiments. All algorithms run on a computer cluster, which nodes are equipped with an AMD Opteron 254 processor, with 2.8 GHz frequency, and interconnected via a high speed Infiniband network.

4.2 Low-Rank Truncation

Devising an $\mathscr{H}$-matrix truncation algorithm depends only on the underlying $\mathscr{R}(k)$-block truncations. The truncation algorithm is particularly utilized in the matrix multiplication procedure, since the set $\mathscr{H}(T,k)$ is not closed under addition.

A truncation of an $\mathscr{R}(k)$-matrix $\boldsymbol{R}$ to rank $k' \leq k$ is defined as the best approximation in the set of $\mathscr{R}(k')$-matrices. The truncated singular value decomposition (SVD) gives the optimal rank-k' approximation of a matrix $\boldsymbol{R}$ w.r.t Frobenius and spectral norm. To get the truncated SVD we omit all singular values, which are smaller than some ϵ or we choose a fixed number of singular values.

An efficient sequential algorithm for the truncation of a low-rank matrix $\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T$ ($\boldsymbol{A} \in \mathbb{R}^{\tau \times k}, \boldsymbol{B} \in \mathbb{R}^{\sigma \times k}$) to matrix $\tilde{\boldsymbol{R}}$ with rank $k' \leq k$ can be computed in $\mathscr{O}(k^2(|\tau| + |\sigma|) + k^3)$ complexity ([5]). Thus, we first compute a QR factorization of $\boldsymbol{A} = \boldsymbol{Q}_A \boldsymbol{R}_A$ and $\boldsymbol{B} = \boldsymbol{Q}_B \boldsymbol{R}_B$. Then, apply an SVD for the product of the two upper triangular factors:

$$\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T = \boldsymbol{Q}_A(\boldsymbol{R}_A \boldsymbol{R}_B^T)\boldsymbol{Q}_B^T = \boldsymbol{Q}_A(\boldsymbol{U}\Sigma\boldsymbol{V}^T)\boldsymbol{Q}_B^T \Longrightarrow \tilde{\boldsymbol{R}} := \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}^T = (\boldsymbol{Q}_A\tilde{\boldsymbol{U}})(\boldsymbol{Q}_B\tilde{\boldsymbol{V}}\tilde{\Sigma})^T, \tag{4.2}$$

where $\tilde{\boldsymbol{U}}, \tilde{\boldsymbol{V}}$ are the first k' columns of the unitary matrices $\boldsymbol{U}, \boldsymbol{V}$ and the diagonal matrix $\tilde{\boldsymbol{\Sigma}} = \mathbf{diag}(s_0, s_1, \cdots, s_{k'-1}, 0, \cdots, 0)$ is obtained by retaining the first k' diagonal elements of $\boldsymbol{\Sigma}$ with $s_0 \geq s_1 \geq \cdots \geq s_{k-1} \geq 0$ as singular values.

In the sequential truncation, the basic tools are truncated QR and SVD. Since the rank k is usually small compared to p, applying a parallel SVD is not efficient. Apart from the SVD, the remaining part of the truncation procedure (4.2), i.e. computing a QR factorization and performing the matrix multiplications can be parallelized.

To have numerical stability in the truncation process, we have used three parallel QR schemes: Givens, Householder, and tall and skinny QR (TSQR, see [3]). Although the two classical QR factorizations are scalable for large matrix sizes, the new TSQR has proved to be efficient also for small matrices as they appear in an $\mathscr{H}$-matrix structure. Indeed, the TSQR algorithm is superior in both theory and practice compared to existing competitive QR algorithms on distributed-memory systems:

Lemma 4.1 *Computing a parallel TSQR factorization for a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ with p processors has the complexity*

$$\mathscr{W}_{TSQR}(n,m,p) := \mathscr{O}(\frac{nm^2}{p} + m^3 \log p + \alpha \cdot \log p + \beta \cdot m^2 \log p), \quad (m \leq n/p).$$

Now, for truncating an $\mathscr{R}(k)$-matrix $\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T$ in parallel suppose that matrices $\boldsymbol{A} \in \mathbb{R}^{\tau\times k}$ and $\boldsymbol{B} \in \mathbb{R}^{\sigma\times k}$ are distributed across two disjoint processor sets, each consists of half of the p processors; $P_\tau := \{p_{\tau_0}, \ldots, p_{\tau_{p'-1}}\}$ and $P_\sigma := \{p_{\sigma_0}, \ldots, p_{\sigma_{p'-1}}\}$ ($p' := p/2$). Thus the parallel TSQR factorizations of $\boldsymbol{A}$ and $\boldsymbol{B}$ can be done simultaneously. After TSQR, all processors in P_τ will store $\boldsymbol{R}_A$ and $\boldsymbol{R}_B$ is stored by all processors in P_σ. Then if we exchange $\boldsymbol{R}_A$ and $\boldsymbol{R}_B$ between two processor sets, all processors in $P_\tau \cup P_\sigma$ will continue the truncation process without any further communications by doing just a serial SVD. Finally updating $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ will be done by both processor sets independently (see Algorithm 4.1).

procedure Rk_truncate($\boldsymbol{R}, k, k', P_\tau, P_\sigma$)
{Compute parallel TSQR of $\boldsymbol{A}$ and $\boldsymbol{B}$}
if $q \in P_\tau$ **then**
  $\boldsymbol{A} = \boldsymbol{Q}_A \boldsymbol{R}_A$; `Send` ($p_{\sigma_j}$, $\boldsymbol{R}_A$); `Recv` (p_{σ_j}, $\boldsymbol{R}_B$); $(0 \leq j < p')$
else if $q \in P_\sigma$ **then**
  $\boldsymbol{B} = \boldsymbol{Q}_B \boldsymbol{R}_B$; `Recv` ($\boldsymbol{R}_A$, p_{τ_j}); `Send` ($\boldsymbol{R}_B$, p_{τ_j}); $(0 \leq j < p')$
$\hat{\boldsymbol{R}} := \boldsymbol{R}_A \boldsymbol{R}_B^T$;
{Compute an SVD of $\hat{\boldsymbol{R}}$ on all processors}
$\hat{\boldsymbol{R}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$;
if $q \in P_\tau$ **then**
  $\tilde{\boldsymbol{A}} = \boldsymbol{Q}_A \tilde{\boldsymbol{U}}$; $(\tilde{\boldsymbol{U}} := [\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{k'-1}])$
else if $q \in P_\sigma$ **then**
  $\tilde{\boldsymbol{B}} = \boldsymbol{Q}_B \tilde{\boldsymbol{V}}\tilde{\boldsymbol{\Sigma}}$; $(\tilde{\boldsymbol{\Sigma}} := \mathbf{diag}(s_0, \ldots, s_{k'-1})$ and $\tilde{\boldsymbol{V}} := [\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{k'-1}])$
end;

Algorithm 4.1: Parallel $\mathscr{R}(k)$-matrix truncation with fixed rank on processor q.

The complexity of Algorithm 4.1 for truncating an $\mathscr{R}(k)$-matrix $\boldsymbol{R} \in \mathbb{R}^{n\times n}$ is basically bounded by the cost of the corresponding QR factorization and SVD:

$$\mathscr{W}_{Rk}(n,k,p) := \mathscr{O}(\frac{nk^2}{p} + k^3 + (\alpha + \beta \cdot k^2)\log p).$$

Note that, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are distributed among all p processors, we need to apply TSQR and update them in order, which does not modify the above overall complexity. Experimental results for a low-rank truncation for a different number of n and from rank $k = 20$ to rank $k' = 10$ are shown in Fig 4.

n	$p=1$ t[s]	$p=2$ t[s]	$p=4$ t[s]	$p=8$ t[s]	$p=16$ t[s]	$p=32$ t[s]
2^{12}	2.15_{-2}	1.25_{-2}	6.80_{-3}	5.52_{-3}	4.71_{-3}	4.58_{-3}
2^{13}	6.13_{-2}	3.41_{-2}	1.32_{-2}	9.08_{-3}	6.81_{-3}	5.62_{-3}
2^{14}	1.44_{-1}	7.72_{-2}	3.46_{-2}	1.73_{-2}	1.09_{-2}	7.93_{-3}
2^{15}	3.01_{-1}	1.61_{-1}	8.10_{-2}	4.30_{-2}	2.09_{-2}	1.26_{-2}
2^{16}	7.13_{-1}	3.77_{-1}	1.66_{-1}	9.70_{-2}	5.23_{-2}	2.43_{-2}
2^{17}	1.75_{+0}	9.17_{-1}	3.95_{-1}	2.02_{-1}	1.16_{-1}	6.01_{-2}
2^{18}	3.55_{+0}	1.84_{+0}	9.54_{-1}	4.87_{-1}	2.44_{-1}	1.34_{-1}
2^{19}	7.14_{+0}	3.69_{+0}	1.92_{+0}	1.30_{+0}	5.71_{-1}	2.65_{-1}
2^{20}	1.44_{+1}	7.23_{+0}	3.81_{+0}	2.38_{+0}	1.54_{+0}	6.23_{-1}

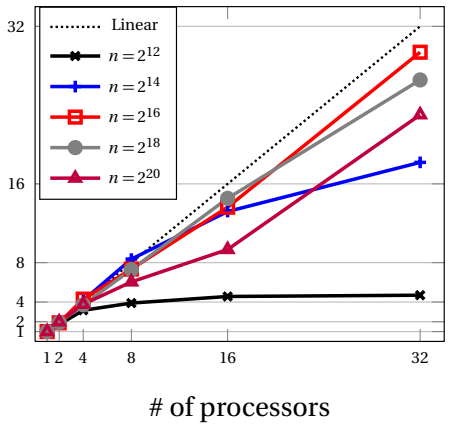

Figure 4: Parallel timing for different p and n (left) and the corresponding speedup (right) in $\mathscr{R}(k)$-matrix truncation from rank $20 \to 10$.

4.3 Matrix-Vector Multiplication

In contrast to $\mathscr{H}$-matrix truncation which is an approximative algorithm, $\mathscr{H}$-matrix-vector multiplication is an exact operation. We are aiming at the more general case

$$y := \alpha' \boldsymbol{M}\boldsymbol{x} + \beta' \boldsymbol{y}, \tag{4.3}$$

with an $\mathscr{H}$-matrix $\boldsymbol{M} \in \mathscr{H}(T,k)$, vectors $\boldsymbol{x},\boldsymbol{y} \in \mathbb{R}^n$, and scalars α', β'. Performing (4.3) is restricted to the set of leaves of T and hence, uses only dense and $\mathscr{R}(k)$-matrix-vector multiplications.

To multiply an $\mathscr{R}(k)$-matrix $\boldsymbol{R} = \boldsymbol{A}\boldsymbol{B}^T \in \mathbb{R}^{\tau\times\sigma}$ with a vector $\boldsymbol{x} \in \mathbb{R}^\tau$, let P_τ, P_σ be the processor sets to which $\boldsymbol{A}$ and $\boldsymbol{B}$ are assigned to; i.e. $P_\tau := \{p_{\tau_0}, \ldots, p_{\tau_{p'-1}}\}$ and $P_\sigma := \{p_{\sigma_0}, \ldots, p_{\sigma_{p'-1}}\}$ ($p' := p/2$). Then we have the following decompositions

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 & \cdots & \boldsymbol{A}_{p'-2} & \boldsymbol{A}_{p'-1} \end{pmatrix}^T, \quad \boldsymbol{B} = \begin{pmatrix} \boldsymbol{B}_0 & \boldsymbol{B}_1 & \cdots & \boldsymbol{B}_{p'-2} & \boldsymbol{B}_{p'-1} \end{pmatrix}^T,$$
$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_{p'-2} & \boldsymbol{x}_{p'-1} \end{pmatrix}^T, \quad \boldsymbol{y} = \begin{pmatrix} \boldsymbol{y}_0 & \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_{p'-2} & \boldsymbol{y}_{p'-1} \end{pmatrix}^T,$$

where $\boldsymbol{A}_i, \boldsymbol{y}_i \in p_{\tau_i}$ and $\boldsymbol{B}_j, \boldsymbol{x}_j \in p_{\sigma_j}$ for $i,j = 0, \cdots p'-1$. The product $\boldsymbol{R}x = y$ is computed as follows:

1: $\boldsymbol{t}_q := \boldsymbol{B}_q^T \boldsymbol{x}_q, \forall q \in P_\sigma$;

2: Reduce $\boldsymbol{t}_q$ at P_σ; Broadcast $\boldsymbol{t}(:=\sum_{q\in P_\sigma} \boldsymbol{t}_q)$ to P_τ;

3: $\boldsymbol{y}_q := \boldsymbol{A}_q \boldsymbol{t}, \forall q \in P_\tau$;

The products in step 1 can be computed in parallel on processors in P_σ as well as multiplications in step 3 on processors in P_τ. Assuming a minimum spanning tree in communication step 2, i.e. reduction of $\boldsymbol{t}_q$ in P_σ and then broadcast on P_τ, the costs for multiplications and communications is therefore bounded by

$$\mathcal{O}((|\sigma|+|\tau|)k/p' + (\alpha+\beta\cdot k)\log p').$$

The drawback of the above procedure is that we need to perform steps 1 and 3 in order, which leaves half of the processors idle. This implies that the maximum achievable speedup is bounded by $p/2$. Therefore, in order to perform $\mathcal{R}(k)$-matrix-vector multiplication more efficiently, we use the fact that computing $\boldsymbol{B}^T\boldsymbol{x}$ and $\boldsymbol{A}\boldsymbol{t}$ in steps 1 and 3 can be done independently by all processors in P_τ and P_σ. The idea is to decouple all low-rank block multiplications such that we first perform step 1 by all column processors, then communicate only one time followed by doing step 3 by all row processors. This can be seen in Algorithm 4.2.

procedure mv_mult($\alpha', \boldsymbol{A}, \boldsymbol{x}, \beta', \boldsymbol{y}, q$)
$\boldsymbol{y} := \beta'\boldsymbol{y}$;
{Step 1}
for all low-rank blocks $b = \tau \times \sigma$ **do**
  $\boldsymbol{t}_q(b) = \boldsymbol{B}^T\boldsymbol{x}|_\sigma$;
{Step 2: Communication}
Reduce $\boldsymbol{t}_q$ at all column processors;
Broadcast $\boldsymbol{t}$ to all row processors;
{Step 3}
for all low-rank blocks $b = \tau \times \sigma$ **do**
  $\boldsymbol{y}|_\tau := \boldsymbol{y}|_\tau + \alpha'\boldsymbol{A}\boldsymbol{t}(b)$;
end;

Algorithm 4.2: Parallel $\mathcal{H}$-matrix-vector multiplication.

The same procedure is applied for dense matrices, which need n_{min} unit of storage for each local $\boldsymbol{t}$. Assuming $\tilde{k} = \min\{k, n_{min}\}$ we obtain the following cost for the $\mathcal{H}$-matrix-vector multiplication on p processors:

$$\mathcal{W}_{\mathcal{H},MV}(n,k,p) = \frac{\mathcal{W}_{\mathcal{H},MV}(n,k,1)}{p} + \mathcal{O}\left(\frac{n}{p\cdot n_b}(\log p)(\log n)(\alpha+\beta\cdot\tilde{k})\right).$$

Numerical results for the $\mathcal{H}$-matrix-vector multiplication applied to the BEM-example with $n_{min} = 64$, $k = 10$, and the largest block size $n_b = L$ as the optimum block size (minimal communication) are reported in Fig. 5.

As the results indicate, reaching a weak scalability is possible for the $\mathcal{H}$-matrix-vector multiplication by increasing n and p simultaneously.

n	$p=1$ t[s]	$p=2$ t[s]	$p=4$ t[s]	$p=8$ t[s]	$p=16$ t[s]	$p=32$ t[s]
2^{15}	0.071	0.036	0.019	0.010	0.006	0.007
2^{16}	0.157	0.079	0.042	0.029	0.011	0.009
2^{17}	0.342	0.172	0.090	0.053	0.023	0.016
2^{18}	0.750	0.376	0.194	0.104	0.053	0.031
2^{19}	1.657	0.836	0.424	0.220	0.121	0.059
2^{20}	3.673	1.843	0.947	0.479	0.249	0.128

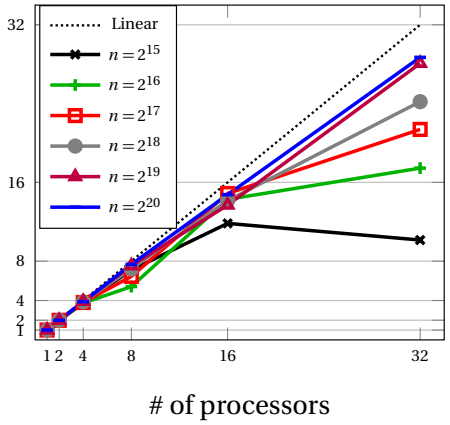

Figure 5: Parallel timing for different p and n (left) and the corresponding speedup (right) in $\mathscr{H}$-matrix-vector multiplication with rank $k=10$ ($n_b=L$).

4.4 Matrix-Matrix Multiplication

So far, the basic operations used in the $\mathscr{H}$-matrix-matrix multiplication are discussed. In this section we consider the following general multiplication form

$$\boldsymbol{C} := \beta' \boldsymbol{C} + \alpha' \boldsymbol{A} \cdot \boldsymbol{B}, \tag{4.4}$$

with $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathscr{H}(T,k)$ and $\alpha', \beta' \in \mathbb{R}$. A sequential algorithm for matrix multiplication can be implemented recursively. Since $\mathscr{H}$-matrices are (recursive) block matrices, performing (4.4) is done by multiplying the corresponding (sub)blocks of $\boldsymbol{A}$ and $\boldsymbol{B}$ followed by adding the intermediate results to the (sub)blocks of $\boldsymbol{C}$. After each operation, a truncation is needed to bring back the result of the multiplication to the initial rank k.

For parallel multiplication, we use the fact that many block multiplications $\boldsymbol{C}_{\tau\sigma} := \boldsymbol{C}_{\tau\sigma} + \boldsymbol{A}_{\tau\gamma} \cdot \boldsymbol{B}_{\gamma\sigma}$ ($\tau, \gamma, \sigma \subset I$) as suboperations in (4.4) are independent and therefore one can process them in parallel. In addition, these independent block multiplications are the ones whose execution order can be changed without modifying the final result in (4.4). However, there are still some dependencies between these block multiplications, e.g. they may have the same destination block, or they share the same processors leading to idleness of processors. Thus we require some synchronization to enforce the simultaneous execution of such blocks on all participating processors. Therefore our goal is to arrange the list of block multiplications in such a way that the idling time of processors is minimized.

To proceed, we denote the set of block multiplications by $\mathscr{M} := \{(\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i)\}_{i=0}^{m-1}$ with m being the number of all block multiplications. For a single matrix block $\boldsymbol{A}'$ we denote the associated processor set by $p(\boldsymbol{A}')$ and for each element $M = (\boldsymbol{A}_j, \boldsymbol{B}_j, \boldsymbol{C}_j) \in \mathscr{M}$ by $p(M)$, i.e. $p(M) := p(\boldsymbol{A}_j) \cup p(\boldsymbol{B}_j) \cup p(\boldsymbol{C}_j)$. The set of all processor sets involved in processing $\mathscr{M}$ will be denoted by $P_{\mathscr{M}} := \{p(M) \mid M \in \mathscr{M}\}$. The set $\mathscr{M}$ and the corresponding processor sets $P_{\mathscr{M}}$ can be obtained by simulating the $\mathscr{H}$-matrix-matrix multiplication sequentially as is shown in Algorithm 4.3.

procedure sim($\boldsymbol{A},\boldsymbol{B},\boldsymbol{C},\mathscr{M},P_{\mathscr{M}}$)
if ($\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are block matrices) **then**
  for all $i,j,l\in\{0,1\}$ **do**
    sim($\boldsymbol{A}_{il},\boldsymbol{B}_{lj},\boldsymbol{C}_{ij},\mathscr{M},P_{\mathscr{M}}$);
else
  $\mathscr{M}:=\mathscr{M}\cup\{(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C})\}$;
  $P_{\mathscr{M}}:=P_{\mathscr{M}}\cup\{p(\boldsymbol{A})\cup p(\boldsymbol{B})\cup p(\boldsymbol{C})\}$;
end;

Algorithm 4.3: Simulating the list of all block multiplication $\mathscr{M}$

Now we are aiming at scheduling the set $\mathscr{M}$ onto the set of all involved processors P such that a maximal parallel efficiency is obtained. This can be done either by a direct scheduling of $\mathscr{M}$ or by factorizing $\mathscr{M}$ into some smaller subsets.

Each block multiplication strictly depends on the set of processors which are assigned to it and this is the minimum information one can use to schedule $\mathscr{M}$. An efficient scheduling algorithm based on splitting of $\mathscr{M}$ into smaller sets uses the cardinality of the elements of $P_{\mathscr{M}}$. Instead of directly scheduling $\mathscr{M}$ onto P, subsets $\mathscr{M}|_{\tilde{p}\in P_{\mathscr{M}}}=\{M\in\mathscr{M}:\ p(M)=\tilde{p}\}$ are considered. Now, for the moment, we assume that the computational cost for each set $\mathscr{M}|_{\tilde{p}\in P_{\mathscr{M}}}$ is equal for all $\tilde{p}\in P_{\mathscr{M}}$. Thus we schedule $P_{\mathscr{M}}$ only according to an increasing size of $\tilde{p}\in P_{\mathscr{M}}$, for example starting from sets with only one processor, then two and so on. The cost of this algorithm is $\mathcal{O}(p\cdot|P_{\mathscr{M}}|\log|P_{\mathscr{M}}|)$.

Example 4.2 *Let assume that we have $m=6$ block multiplications*

$$\mathscr{M}=\{M_0,M_1,M_2,M_3,M_4,M_5\},\ P_{\mathscr{M}}=\{\{0,1,2\},\{0,1,2,3\},\{3\},\{0,1\},\{2,3\},\{0,2\}\},$$

where their corresponding processor sets are in $P_{\mathscr{M}}$. Then we get a schedule with four parallel steps: ❶*:* $\{0,1,2\},\{3\}$ ❷*:* $\{0,1\},\{2,3\}$ ❸*:* $\{0,1,2,3\}$ ❹*:* $\{0,2\}$.

Once the scheduling is accomplished, one can immediately reorder the initial list of block multiplications $\mathscr{M}$ w.r.t the list of processor sets derived by the scheduling algorithm. We refer to this list as $\mathscr{M}_{opt}$ which has an (almost) optimal arrangement of block multiplications ready for parallel execution. Surprisingly, utilizing the above scheduling give us the best $\mathscr{M}_{opt}$ in practice, rather than any scheduling algorithm uses the real cost of block multiplications (see Remark 4.3).

Remark 4.3 *In the above scheduling for two processors sets $\tilde{p}\neq\tilde{q}\in P_{\mathscr{M}}$ we have assumed that the costs of $\mathscr{M}|_{\tilde{p}}$ and $\mathscr{M}|_{\tilde{q}}$ are equal, which does not hold in practice. Beside $p(M_i)$, one may also utilize the actual cost of an individual block multiplication as defined by a cost function $c:\mathscr{M}\mapsto\mathbb{R}_{\geq 0}$. However, we emphasize that the cost function needed for scheduling is based on coarse approximation and is very complex, which depends on many parameters like the underlying hardware, etc. On the other hand, obtaining a scheduling in this way is more costly and most importantly the time of $\mathscr{H}$-matrix multiplication (i.e. execution of $\mathscr{M}_{opt}$) induced by this scheduling shows no significant improvement over the scheduling without costs.*

The next lemma will estimate the cost of the $\mathscr{H}$-matrix matrix multiplication:

Lemma 4.4 *The parallel computation of the product* (4.4) *on p processors has the following complexity for a block size* $n_{min} \leq n_b = 2^l n_{min} \leq L$

$$\mathscr{W}_{\mathscr{H},MM}(n,k,p) = \frac{\mathscr{W}_{\mathscr{H},MM}(n,k,1)}{p} + \mathscr{O}\left(\frac{n}{p\cdot n_b}\left(\log^5 p + \log p \log^2 n_b\right)(\alpha+\beta\cdot k^2)\right)$$
$$+ \mathscr{O}\left(\frac{n}{p\cdot n_b}\log\frac{n}{p\cdot n_b}\left(\log^3 p + \log p \log n\right)(\alpha+\beta\cdot k^2)\right).$$

The next table will report the results of matrix multiplication for different values of n, rank $k = 10$, and $n_{min} = 64$. The timings are only for the actual matrix multiplication $\mathscr{M}_{opt}$ without scheduling timings which are small and can be neglected. The block size used for the experiments is $n_b = L$, since due to Lemma 4.4, it is the optimal block size, which reflects a lower communication cost. Obviously, the efficiency of the $\mathscr{H}$-matrix multiplication will

n	$p=1$ t[s]	$p=2$ t[s]	$p=4$ t[s]	$p=8$ t[s]	$p=16$ t[s]	$p=32$ t[s]
2^{15}	40.16	23.61	14.38	8.52	5.93	4.37
2^{16}	97.31	55.49	33.60	19.84	13.24	9.58
2^{17}	235.08	132.64	79.19	46.04	30.79	21.44
2^{18}	544.72	305.94	182.69	102.77	69.91	48.47
2^{19}	1337.40	716.22	419.29	238.89	158.79	106.83
2^{20}	3666.10	1881.70	1017.70	537.65	376.64	282.94

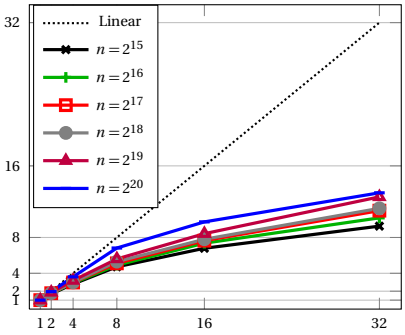

Figure 6: Parallel timing for different p and n (left) and the corresponding speedup (right) in $\mathscr{H}$-matrix-matrix multiplication with rank $k = 10$ ($n_b = L$).

increase when a larger matrix size is used. However, the growth is less compared to $\mathscr{H}$-matrix vector multiplication.

4.5 Matrix Inversion

An efficient way for inverting a block matrix is by block Gaussian elimination. The inverse of a block matrix $\boldsymbol{A}$ is given by

$$\boldsymbol{A} = \left(\begin{array}{c|c} \boldsymbol{A}_{00} & \boldsymbol{A}_{01} \\ \hline \boldsymbol{A}_{10} & \boldsymbol{A}_{11} \end{array}\right), \quad \boldsymbol{C} := \boldsymbol{A}^{-1} = \left(\begin{array}{c|c} \boldsymbol{A}_{00}^{-1} + \boldsymbol{A}_{00}^{-1}\boldsymbol{A}_{01}\boldsymbol{S}^{-1}\boldsymbol{A}_{10}\boldsymbol{A}_{00}^{-1} & -\boldsymbol{A}_{00}^{-1}\boldsymbol{A}_{01}\boldsymbol{S}^{-1} \\ \hline -\boldsymbol{S}^{-1}\boldsymbol{A}_{10}\boldsymbol{A}_{00}^{-1} & \boldsymbol{S}^{-1} \end{array}\right),$$

where $\boldsymbol{S} := \boldsymbol{A}_{11} - \boldsymbol{A}_{10}\boldsymbol{A}_{00}^{-1}\boldsymbol{A}_{01}$ is the Schur complement. The existence of $\boldsymbol{C}$ depends only on the existence of $\boldsymbol{A}_{00}^{-1}$ and $\boldsymbol{S}^{-1}$ and computing $\boldsymbol{C}$ is done recursively by inverting $\boldsymbol{A}_{00}$ and $\boldsymbol{S}$ along the diagonal as is shown in Algorithm 4.4.

The parallel version of the algorithm is basically identical to the sequential one. As one observes, it only contains matrix multiplication operations in the form of (4.4). Although the

procedure invert(A, C)
if A is 2×2 block matrix **then**
  `invert`(A_{00}, C_{00});
  $T_{01} := C_{00}A_{01}$; $T_{10} := A_{10}C_{00}$;
  $A_{11} := A_{11} - A_{10}T_{01}$; `invert`($A_{11}, C_{11}$);
  $C_{01} := -T_{01}C_{11}$; $C_{10} := -C_{11}T_{10}$;
  $C_{00} := C_{00} - T_{01}C_{10}$;
else
  $C := A^{-1}$;
end;

Algorithm 4.4: $\mathcal{H}$-matrix inversion by Gaussian elimination

performance of the inversion algorithm depends on the performance of the invoked matrix multiplications, its inherent part (due to diagonal inversion) has also an substantial influence on the idling time of processors.

In other words, choosing the block size n_b has a direct and visible impact on the idleness of processors. Unlike the other $\mathcal{H}$-matrix operations which have the best performance corresponding to the largest block size $n_b = L$, for this block size, inverting an $\mathcal{H}$-matrix has the poorest performance. The reason is that for $n_b = L$, each specific processor q needs to invert a submatrix of size $L \times L$ sequentially while other processors have nothing to do and just wait for processor q to finish and hereafter the processor q will be idle while the trailing submatrix is computed. However, our expectation is to find some l^* for which we have the best possible efficiency. In fact, selecting $n_b = 2^{l^*} n_{min}$ is a trade-off between idle time and communication costs (see Lemma 4.4).

The complexity of the $\mathcal{H}$-matrix inversion, which consists of two terms, one for the parallel matrix multiplication, one for the serial diagonal inversion is expressed as follows:

Lemma 4.5 *The parallel $\mathcal{H}$-matrix inversion of matrix $\mathbf{A} \in \mathcal{H}(T,k)$ on p processors has the following complexity for block size $n_{min} \leq n_b = 2^l n_{min} < L$:*

$$\begin{aligned}\mathcal{W}_{\mathcal{H},Inv}(n,k,p) = \mathcal{O}\left(\frac{n k^2 \log s \log^2 n}{p}\right) + \mathcal{O}(n k^2 \log^2 n_b) \\ + \mathcal{O}\left(s(p+\log s)\left(\log^5 p + \log p \log^2 n_b\right)(\alpha + \beta \cdot k^2)\right) \\ + \mathcal{O}\left(s \log^2 s \left(\log^3 p + \log p \log n\right)(\alpha + \beta \cdot k^2)\right),\end{aligned}$$

with $s := n/(p \cdot n_b)$.

To see the behavior of different block sizes n_b, we consider the inversion of the BEM-example with $n = 524288$, $k = 10$, and $n_{min} = 64$ as presented in Fig. 7. As the results show, the parallel efficiency depends on p as well as n_b. The best performance is attained for $n_b = 256, 512$ while the worst performance is corresponding to $n_b = L$. The numerical results for different values of n corresponding to an optimum block size $n_b = 256$ are reported in Fig. 8.

As one can see, the drop of efficiency is clear for a large number of processors, which is due to the high communication cost in the parallel matrix multiplications for a small block

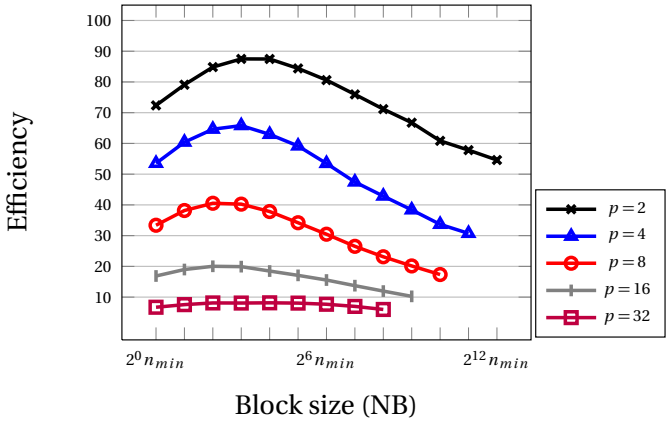

Figure 7: Parallel efficiency for $\mathscr{H}$-matrix inversion in terms of varying n_b ($k=10$).

n	$p=1$ t[s]	$p=2$ t[s]	$p=4$ t[s]	$p=8$ t[s]	$p=16$ t[s]	$p=32$ t[s]
2^{15}	57.52	39.09	30.18	26.58	26.09	26.78
2^{16}	142.92	93.97	70.36	60.50	60.46	64.37
2^{17}	356.20	227.41	159.85	135.77	137.06	154.86
2^{18}	857.29	541.46	373.89	306.24	310.83	396.50
2^{19}	2252.10	1334.90	881.15	698.50	698.50	873.13

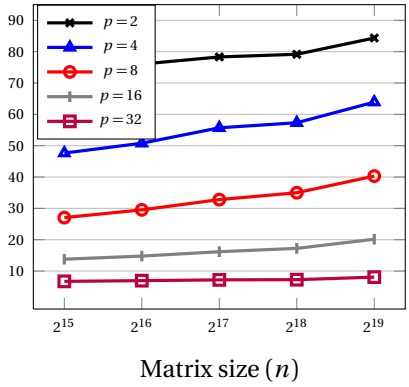

Figure 8: Parallel timing for different p and n (left) and the corresponding efficiency (right) in $\mathscr{H}$-matrix inversion with rank $k=10$ ($n_b=256$).

size. This means that the usability of the $\mathscr{H}$-matrix inversion for more than $p=8$ is severely limited.

5 Conclusions

The parallel algorithms for the arithmetic of standard $\mathscr{H}$-matrices are developed and the corresponding complexity estimates are obtained. The proposed algorithms exhibit high scalability and nearly optimal speedup for a sufficiently large system size when implemented on a distributed-memory systems.

One of the prominent factor that dramatically improves the performance of algorithms on distributed systems is the choice of a good data distribution strategy. Applying the index-wise $\mathscr{H}$-matrix distribution enables us to achieve the desired values of parallel efficiency on these machines when the number of processors and problem size increase especially for the largest block size $n_b = L$

Unfortunately, the $\mathscr{H}$-matrix inversion algorithm is not weakly scalable when a larger number of processors is used. In fact, for a large n_b, its inefficiency is partly due to sequential inversion of diagonal blocks while for a small n_b it stems from the higher communication costs.

In this work we have not considered a general $\mathscr{H}$-matrix or when a variable rank is used for the underlying low-rank matrices. Assuming an equal cost per index for distributing a general $\mathscr{H}$-matrix or an $\mathscr{H}$-matrix with adaptive ranks, does not yield an optimal load balancing. These need further investigations and is the subject of ongoing research.

Acknowledgments

The author would like to thank Dr. Ronald Kriemann for several fruitful discussions on this paper.

References

[1] M. Bebendorf, R. Kriemann: Fast Parallel Solution of Boundary Integral Equations and Related Problems, Comput. Vis. Sci, 8(3-4), 121-135 (2005).

[2] S. Börm, L. Grasedyck, W. Hackbusch: Hierarchical matrices. Technical report. Lecture Note 21, MPI Leipzig (2003).

[3] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou: Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput., 34(1), 206-239 (2011).

[4] A. Grama, A. Gupta, G. Karypis, and V. Kumar: Introduction to Parallel Computing (2nd Edition) Addison-Wesley, (2003).

[5] L. Grasedyck, W. Hackbusch: Construction and arithmetics of $\mathscr{H}$-matrices. Computing 70(4), 295-334 (2003).

[6] W. Hackbusch: A sparse matrix arithmetic based on $\mathscr{H}$-matrices I: Introduction to $\mathscr{H}$-matrices, Computing 62(2), 89-108 (1999).

[7] M. Izadi: Hierarchical Matrix Techniques on Massively Parallel Computers, PhD thesis, Universität Leipzig, (2012).

[8] R. Kriemann: Parallele Algorithmen für $\mathscr{H}$-Matrizen, PhD thesis, Universität Kiel, (2004).

[9] R. Kriemann: Parallel $\mathscr{H}$-Matrix Arithmetics on Shared Memory Systems. computing 74(3), 273-297 (2005).