Adaptive Refinement and Clustering of $\mathcal{H}$-Matrices

by

L. Grasedyck, W. Hackbusch, S. Le Borne

Preprint no.: 106 2001

Adaptive Refinement and Clustering of $\mathcal{H}$-Matrices

L. Grasedyck, **Kiel**, W. Hackbusch, **Leipzig**, and S. Le Borne, **Cookeville**

Abstract

In [4], a class of (data-sparse) $\mathcal{H}$-matrices is introduced which allows an approximate matrix arithmetic of nearly optimal complexity. In several subsequent papers (e.g., [5], [6]), $\mathcal{H}$-matrices were shown to be applicable in the boundary element as well as finite element context, again yielding nearly optimal complexity estimates for storage and work requirements of the respective stiffness matrices. The analyses were based on the assumption of the underlying cluster trees being balanced. This assumption might be violated in the case of adaptive mesh refinement. The present paper provides an extension of $\mathcal{H}$-matrix techniques to (a sequence of) problems on adaptively refined meshes. A measure to monitor the actual storage and work complexities is introduced and employed to decide whether an adaptively refined (unbalanced) cluster tree is still acceptable or should be reconstructed in a balanced way.

AMS Subject Classification: 65F05, 65F30, 65F50, 65N50
Key words: Hierarchical matrices, data-sparse approximations, formatted matrix operations, fast solvers, adaptive mesh refinement

1 Introduction

In [4], a class of hierarchical matrices ($\mathcal{H}$-matrices) has been introduced that allows a sparse approximation to large, fully populated stiffness matrices arising in boundary element method or finite element method applications. In the FEM case, it is the inverse of the stiffness matrix that is fully populated and can be approximated by an $\mathcal{H}$-matrix.

The construction of an $\mathcal{H}$-matrix is based on a (hierarchical) block partitioning of the product index set $I \times I$ which itself is based on a (hierarchical) partitioning of the index set I. These hierarchical partitionings are organised in so-called (block) $\mathcal{H}$-trees. If the finite index set I and hence its size $n := |I|$ is fixed, then these (block) $\mathcal{H}$-trees can be constructed in a balanced way which will lead to nearly optimal complexities between $\mathcal{O}(n)$ and $\mathcal{O}(n \log_2^2 n)$ for the storage, matrix-vector multiplication and standard matrix operations like (approximate) matrix-matrix multiplication or (approximate) matrix inversion of the stiffness matrix $A \in \mathbb{R}^{n,n}$.

A local refinement of the discretisation leads to an enlarged index set I' of the size $n' := |I'|$ with $n' > n$ and a stiffness matrix A'. The above mentioned (nearly) optimal case could be obtained for A' by reconstructing a balanced $\mathcal{H}$-tree for the index set I'. However, due to the local character of the refinement, the majority of the indices of A and A' coincide. This observation suggests to only update the matrix format locally in the positions corresponding to the refinement. In terms of the $\mathcal{H}$-tree this corresponds to expanding some vertices of the tree, more precisely replacing some leaves by small subtrees.

Such an update of a (block) $\mathcal{H}$-tree might lead to an unbalanced tree and hence to worse storage and work complexities, the worst case being $\mathcal{O}(n^2)$ for storage and $\mathcal{O}(n^3)$ for matrix inversion. Hence one has to carefully weigh the advantages (w.r.t. matrix assembly) of updating the tree over reconstructing it in a balanced way compared to the disadvantages (w.r.t. storage and subsequent work requirements) of dealing with an unbalanced tree.

In this paper we introduce a measure which indicates how much the storage and work complexity of an $\mathcal{H}$-matrix based upon an adaptively refined cluster tree differs from that of a balanced cluster tree. Hence this measure can be used to decide when a reconstruction of the cluster tree becomes necessary in order to maintain optimal complexity estimates. Since an implementation of $\mathcal{H}$-matrices that allow local updates as a consequence of local updates of the cluster tree is not straightforward, we provide one such exemplary implementation.

The rest of the paper is organised as follows: In Section 2, we introduce the class of $\mathcal{H}$-matrices and estimate their storage and work complexities under rather general assumptions. In Section 3, we introduce a measure and explain how it can be employed in an adaptive process in order to keep storage and work requirements as low as possible. Section 4 provides an implementation for the relevant data structures and matrix operations. The theoretical results are confirmed by numerical tests which are presented in Section 5.

2 $\mathcal{H}$-Matrices

Let I be a finite index set, and consider the vector space $\mathbb{K}^I$ over the field $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$. A (fixed) partitioning P of I into disjoint subsets is described by

$$P = \{I_j \mid 1 \leq j \leq |P|\} \qquad \text{with} \qquad I = \cup_{j=1}^{|P|} I_j, \qquad I_j \cap I_i = \emptyset \text{ for } i \neq j.$$

We consider a set of such partitionings, including coarse as well as fine ones, which is hierarchically structured and defined by an $\mathcal{H}$-tree $T = T(I)$ which will be introduced after establishing some notation. Given a tree $T = (V, E)$ with a vertex $t \in V$, we will also write $t \in T$ instead. A vertex $x \in V$ is called a *predecessor* (*successor*, *son*) of $y \in V$ if $(x, y) \in E$ ($(y, x) \in E$). A vertex with no successor is called *leaf*. We use the notations

$$\begin{aligned}
\mathcal{S}(t) &= \{s \in T \mid s \text{ is successor of } t\} \qquad \text{for } t \in T,\\
\mathcal{F}(t) &= v \text{ if } t \in \mathcal{S}(v) \quad \text{for} \quad t \in T \setminus \{\operatorname{root}(T)\},\\
\mathcal{L}(T) &= \{t \in T \mid S(t) = \emptyset\},\\
T^{(0)} &= \{\operatorname{root}(T)\},\\
T^{(i)} &= \{t \in T \mid \mathcal{F}(t) \in T^{(i-1)}\} \qquad \text{for } i \in \mathbb{N},\\
\mathcal{L}(T, i) &= \mathcal{L}(T) \cap T^{(i)} \qquad \text{for } i \in \mathbb{N}_0,\\
p_T &= \max\{i \in \mathbb{N}_0 \mid T^{(i)} \neq \emptyset\}
\end{aligned}$$

for the set of successors of a vertex, the predecessor (father) of a vertex, the set of leaves of a tree, the vertices of level 0, the vertices of level i, the leaves of level i and the depth of the tree, respectively.

Definition 2.1 *($\mathcal{H}$-tree) Let I be an index set, and let $\mathcal{P}(I)$ denote its power set. A tree $T = (V, E)$ with $V \subset \mathcal{P}(I)$ is called an $\mathcal{H}$-tree (based on I), if the following conditions hold:*

(i) $I \in T$.
(ii) *If $t \notin \mathcal{L}(T)$, then $S(t)$ contains disjoint subsets of I and t is the union of its sons, i.e., $t = \cup_{s \in S(t)} s$.*

The name $\mathcal{H}$-tree is due to its hierarchical structure. Since the vertices $t \in T$ are named *clusters*, the $\mathcal{H}$-tree is also called *cluster tree.*

Remark 2.2 *For any $\mathcal{H}$-tree T of I and $i \in \{0, \ldots, p_T\}$ there holds*

$$I = \dot{\bigcup}\{t \mid t \in T^{(i)} \cup (T, j), j < i\}, \qquad \textit{in particular} \qquad I = \dot{\bigcup}\{t \mid t \in (T)\},$$

i.e., the leaves of an $\mathcal{H}$-tree yield a partitioning for the index set I.

In order to obtain a (hierarchical) set of partitionings for the index set $I \times I$ we will construct a so-called *block $\mathcal{H}$-tree*, denoted by $T_2 = T(I \times I)$ in contrast to $T_1 = T(I)$ for the previous cluster tree.

Definition 2.3 ***(block $\mathcal{H}$-tree)*** *Given an index set I and an $\mathcal{H}$-tree $T_1(I) = (V_1, E_1)$, a block $\mathcal{H}$-tree (block cluster tree) $T_2(I \times I) = (V_2, E_2)$ corresponding to T_1 is an $\mathcal{H}$-tree of $I \times I$ with root $I \times I$ such that for all $i \in \{0, \ldots, p_{T_2}\}$ and $t = t_1 \times t_2 \in T_2^{(i)}$ it holds $t_1, t_2 \in T_1^{(i)}$.*

Remark 2.4 *A more general definition of a block $\mathcal{H}$-tree allows $t_1, t_2 \in T_1$ of different depths for blocks $t = t_1 \times t_2 \in T_2$. This would, however, complicate the subsequent theoretical considerations without yielding significantly improved results.*

The vertices of a block $\mathcal{H}$-tree are called *block clusters* or just *blocks*. There exist several constructions to obtain a block $\mathcal{H}$-tree from a cluster tree in a unique way. Two such algorithms that construct the more general block $\mathcal{H}$-trees of Remark 2.4 are described in [6]. A simple algorithm to construct a block $\mathcal{H}$-tree as defined in Definition 2.3 will be given in Construction 2.7 below. There we distinguish between admissible and non-admissible blocks as defined in

Definition 2.5 ***(admissibility)*** *a) An admissibility condition for a block $\mathcal{H}$-tree T_2 is a mapping $Adm : T_2 \to \{TRUE, FALSE\}$. A block $b = t_1 \times t_2 \in T_2$ is called admissible if $Adm(b) = TRUE$.*
b) Given a parameter n_{min} indicating a minimal block size, a block $\mathcal{H}$-tree T_2 is called admissible if all leaves $b = t_1 \times t_2$ are either admissible or $\max\{|t_1|, |t_2|\} \leq n_{min}$. We denote the set of admissible (inadmissible) leaves of T_2 by $\mathcal{L}^+(T_2)$ ($\mathcal{L}^-(T_2)$).

For the remainder of the paper we assume the following condition to hold:

Assumption 2.6 *T_2 is an admissible block $\mathcal{H}$-tree.*

Construction 2.7 *Start with $I \times I = \operatorname{root}(T_2)$ and define sons of $b = t_1 \times t_2 \in T_2$ (where $t_1, t_2 \in T_1$) by*

$$S(b) := \begin{cases} \emptyset & S(t_1) = \emptyset \quad \vee \quad S(t_2) = \emptyset \quad \vee \quad t_1 \times t_2 \textit{ admissible} \\ & \vee \quad (|t_1| \leq n_{min} \wedge |t_2| \leq n_{min}), \\ \{s_1 \times s_2 \mid s_1 \in S(t_1), s_2 \in S(t_2)\} & \textit{otherwise.} \end{cases}$$

Each block $b \in T_2$ corresponds to a location of a matrix block $M^b = (m_{ij})_{(i,j) \in b}$ of a matrix $M = (m_{ij})_{(i,j) \in I \times I} \in \mathbb{K}^{I \times I}$.

Definition 2.8 ***($\mathcal{H}$-matrix)*** *Let $k \in \mathbb{N}_0$. The set of $\mathcal{H}$-matrices induced by a block $\mathcal{H}$-tree T_2 is*

$$\mathcal{M}_{\mathcal{H},k}(T_2) := \{M \in \mathbb{K}^{I \times I} \mid \forall b \in \mathcal{L}^+(T_2) : \operatorname{rank}(M^b) \leq k\}.$$

The purpose of the admissibility condition is to ensure a sufficient approximation of the $\mathcal{H}$-matrix $M_{\mathcal{H}}$ to a given matrix M. The parameter n_{min} has been introduced taking into account practical aspects: Under a certain size, typically 4 to 32 times k, it is more efficient to implement a matrix as a full matrix than as a structured one.

Matrices of rank at most k can be represented in several ways. For the matrix blocks in an $\mathcal{H}$-matrix corresponding to admissible block clusters we choose the representation as an $\mathbf{R}k$-matrix, which is defined in

Definition 2.9 ***(Rk-matrix)*** *We call a matrix $R \in \mathbb{K}^{n,m}$ an **R**k-matrix if R is given as*

$$R = \sum_{i=1}^{k} a_i b_i^T$$

with $a_i \in \mathbb{K}^n$ and $b_i \in \mathbb{K}^m$ for $i = 1, \ldots, k$.

Lemma 2.10 *The costs $N_{\boldsymbol{R}k,St}, N_{F,St}$ for the storage and $N_{\boldsymbol{R}k\cdot V}, N_{F\cdot V}$ for the matrix-vector multiplication of $n \times m$ **R**k- and full matrices are given by*

$$\begin{aligned} N_{\boldsymbol{R}k,St}(n,m) &= k(n+m), \qquad & N_{\boldsymbol{R}k\cdot V}(n,m) &= 2k(n+m) - k - n \leq 2k(n+m), \\ N_{F,St}(n,m) &= nm, & N_{F\cdot V}(n,m) &= 2nm - n \leq 2nm. \end{aligned}$$

Definition 2.11 ***(sparse block $\mathcal{H}$-tree)*** *A block $\mathcal{H}$-tree $T_2(I \times I)$ that results from a cluster tree $T_1(I)$ by Construction 2.7 is called* sparse *if there exists a constant C_{sp} such that for all $t_1 \in T_1$:*

$$\max\left\{ \left| \{t_2 \in T_1 \mid t_1 \times t_2 \in \mathcal{L}(T_2)\} \right|, \left| \{t_2 \in T_1 \mid t_2 \times t_1 \in \mathcal{L}(T_2)\} \right| \right\} \leq C_{sp}. \tag{2.1}$$

The sparsity can be proven for rather arbitrary block $\mathcal{H}$-trees (see, e.g., [2],[3]) with a small constant C_{sp}. For the block $\mathcal{H}$-tree corresponding to the block-partitioning B_2 in [6, Section 2.2.2] (see also Figure 2 (left)), there holds $C_{sp} = 1$. The sparsity enables us to estimate the storage requirements of an $\mathcal{H}$-matrix belonging to $\mathcal{M}_{\mathcal{H},k}(T_2)$.

Lemma 2.12 *Let $L := \{i \in \mathbb{N}_0 \mid \exists\, t_1 \times t_2 \in \mathcal{L}(T_2, i)\}$. The costs for the storage $N_{\mathcal{H},St}(T_2)$ of a matrix $M \in \mathcal{M}_{\mathcal{H},k}(T_2)$ for a sparse block $\mathcal{H}$-tree $T_2(I \times I)$ are bounded by $2|L| C_{sp} \max\{k, \frac{1}{2} n_{min}\}|I|$.*

Proof.

$$\begin{aligned} N_{\mathcal{H},St}(T_2) &= \sum_{t_1 \times t_2 \in \mathcal{L}^+(T_2)} N_{\mathbf{R}k,St}(|t_1|,|t_2|) \quad + \sum_{t_1 \times t_2 \in \mathcal{L}^-(T_2)} N_{F,St}(|t_1|,|t_2|) \\ &\overset{L.2.10}{\leq} \sum_{t_1 \times t_2 \in \mathcal{L}^+(T_2)} k|t_1| \quad + \sum_{t_1 \times t_2 \in \mathcal{L}^+(T_2)} k|t_2| \quad + \sum_{t_1 \times t_2 \in \mathcal{L}^-(T_2)} \frac{1}{2} n_{min}(|t_1| + |t_2|) \\ &\leq \sum_{t_1 \times t_2 \in \mathcal{L}^+(T_2) \cup \mathcal{L}^-(T_2)} \max\{k, \frac{1}{2} n_{min}\}|t_1| \quad + \sum_{t_1 \times t_2 \in \mathcal{L}^+(T_2) \cup \mathcal{L}^-(T_2)} \max\{k, \frac{1}{2} n_{min}\}|t_2| \\ &\overset{(2.1)}{\leq} \sum_{i \in L} \sum_{t_1 \in T_1^{(i)}} C_{sp} \max\{k, \frac{1}{2} n_{min}\}|t_1| \quad + \sum_{i \in L} \sum_{t_2 \in T_1^{(i)}} C_{sp} \max\{k, \frac{1}{2} n_{min}\}|t_2| \\ &\overset{R.2.2}{\leq} \sum_{i \in L} C_{sp} \max\{k, \frac{1}{2} n_{min}\}|I| \quad + \sum_{i \in L} C_{sp} \max\{k, \frac{1}{2} n_{min}\}|I| \\ &\leq 2|L| C_{sp} \max\{k, \frac{1}{2} n_{min}\}|I|. \end{aligned}$$

■

Lemma 2.13 *The costs for the storage $N_{\mathcal{H},St}(T_2)$ and for the matrix-vector multiplication $N_{\mathcal{H}\cdot V}(T_2)$ of a matrix $M \in \mathcal{M}_{\mathcal{H},k}(T_2)$ based on a block $\mathcal{H}$-tree $T_2(I \times I)$ are related by*

$$N_{\mathcal{H},St}(T_2) \leq N_{\mathcal{H}\cdot V}(T_2) \leq 2 N_{\mathcal{H},St}(T_2).$$

Proof.

$$
\begin{aligned}
N_{\mathcal{H},St}(T_2) &= \sum_{t_1\times t_2\in\mathcal{L}^+(T_2)} N_{\mathbf{R}k,St}(|t_1|,|t_2|) \quad + \sum_{t_1\times t_2\in\mathcal{L}^-(T_2)} N_{F,St}(|t_1|,|t_2|) \\
&\overset{L.2.10}{=} \sum_{t_1\times t_2\in\mathcal{L}^+(T_2)} k(|t_1|+|t_2|) \quad + \sum_{t_1\times t_2\in\mathcal{L}^-(T_2)} |t_1|\cdot|t_2| \\
&\leq \sum_{t_1\times t_2\in\mathcal{L}^+(T_2)} 2k(|t_1|+|t_2|)-k-|t_1| \quad + \sum_{t_1\times t_2\in\mathcal{L}^-(T_2)} (2|t_1|\cdot|t_2|-|t_2|) \\
&\overset{L.2.10}{=} \sum_{t_1\times t_2\in\mathcal{L}^+(T_2)} N_{\mathbf{R}k\cdot V}(|t_1|,|t_2|) \quad + \sum_{t_1\times t_2\in\mathcal{L}^-(T_2)} N_{F\cdot V}(|t_1|,|t_2|) \\
&= N_{\mathcal{H}\cdot V}(T_2) \\
&\overset{L.2.10}{\leq} \sum_{t_1\times t_2\in\mathcal{L}^+(T_2)} 2k(|t_1|+|t_2|) \quad + \sum_{t_1\times t_2\in\mathcal{L}^-(T_2)} 2|t_1|\cdot|t_2| \\
&= 2N_{\mathcal{H},St}(T_2).
\end{aligned}
$$

■

The fact that the costs for the matrix-vector multiplication can be bounded by the storage requirements for an $\mathcal{H}$-matrix (and vice versa) will lead to the introduction of a measure for the overall complexity of an $\mathcal{H}$-matrix in Section 3.

3 Adaptive Refinement and Clustering

3.1 Motivation

We begin this section with a simple example. Given the index set $I = \{1,\cdots,n\}$ with $n = 2^p$, $p \in \mathbb{N}$, we construct two different cluster trees $T_1^{balanced}$ and $T_1^{unbalanced}$ which are depicted in Figure 1. We then construct block cluster trees $T_2^{balanced}$ and $T_2^{unbalanced}$ by Construction 2.7 and impose

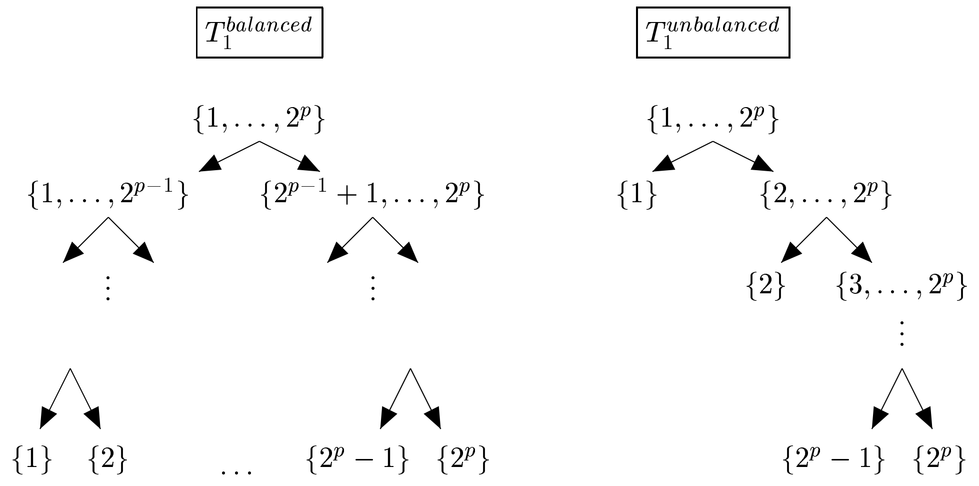


Figure 1: A balanced and an unbalanced cluster tree for the index set I.

the admissibility condition

$$b = t_1 \times t_2 \text{ is admissible} \Longleftrightarrow t_1 \cap t_2 = \emptyset. \tag{3.1}$$

The block structures for the resulting $\mathcal{H}$-matrices $M_{\mathcal{H}}^{balanced}$ and $M_{\mathcal{H}}^{unbalanced}$ for $p = 4$, i.e., $n = 16$, are depicted in Figure 2. One can easily check that the storage requirements for the matrix formats are

$$N_{\mathcal{H},St}(M_{\mathcal{H}}^{balanced}) = O(pkn), \qquad N_{\mathcal{H},St}(M_{\mathcal{H}}^{unbalanced}) = O(n^2),$$

where we assumed rank k for the individual matrix blocks (more precisely, $\text{rank}(b) = \min\{k, r, s\}$ for an admissible $r \times s$ matrix block b). This example demonstrates that a balanced cluster tree leads

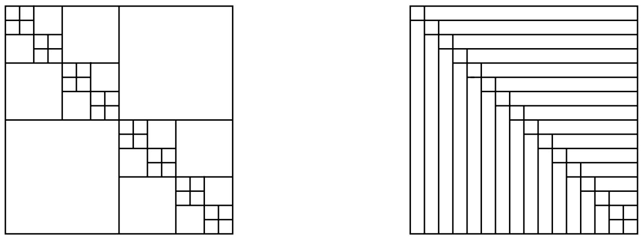


Figure 2: The matrix block structure resulting from a balanced and an unbalanced tree, resp.

to much better storage requirements (and therefore work requirements, see Lemma 2.13) than an unbalanced tree. In practical applications, however, one might encounter somewhat unbalanced trees due to an adaptive refinement process as will be illustrated by the following example: Let $J = [0, 1]$ be an interval that is subdivided into $n = 2^p$ disjoint subintervals J_i, $1 \leq i \leq n$, of size $1/n$. If we identify each subinterval J_i with an index i, we can construct a balanced cluster tree $T_1^{balanced}$ for the index set $I = \{1, \cdots, n\}$ as illustrated in Figure 1(left). Due to the results of some error estimator, we might want to locally refine our partitioning, e.g., further divide only the leftmost interval J_1 into two subintervals, and then again continue to divide only the leftmost interval of these two new intervals, etc. The resulting interval partitioning for $p = 2$ and four further local refinements of the leftmost interval is given in Figure 3. Starting with a balanced cluster tree for the equidistant

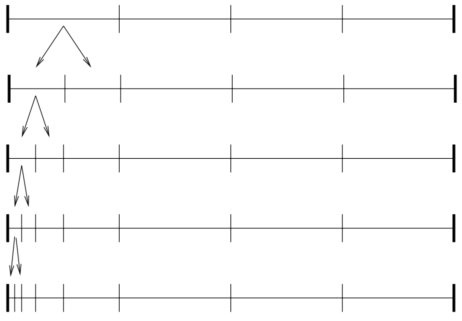


Figure 3: An adaptively refined interval partitioning.

interval partitioning of J into n subintervals, we can iteratively update this tree by simply replacing the cluster corresponding to the refined interval J_1 by a new cluster consisting of the two new subintervals with the corresponding sons. Alternatively, one could rebuild the complete cluster tree in order to obtain a tree as balanced as possible. Whereas in the case of the balanced cluster tree the amount of work to update the tree is larger than compared to the unbalanced case, the subsequent arithmetic operations with the resulting $\mathcal{H}$-matrix will be less expensive. This circumstance raises the following questions:

- How unbalanced does the tree have to be for it to be worth to be rebuilt completely?
- How can this be measured efficiently and reliably?

- And how does one have to implement $\mathcal{H}$-matrices in order to benefit from the locality of the update?

3.2 The Density Measure in an Adaptive Scheme

For a sparse block $\mathcal{H}$-tree T_2 that is almost cardinality balanced, i.e., $p = \log_2(n)$, and $n_{min} \leq 2k$, there holds $N_{\mathcal{H},St}(T_2) = O(kn\log_2(n))$ and $N_{\mathcal{H}\cdot V}(T_2) = O(kn\log_2(n))$ (see Lemmata 2.12,2.13). Therefore the quotient $N_{\mathcal{H},St}(T_2)/(kn\log_2(n))$ should be bounded in the balanced case.

Definition 3.1 ***(density measure)*** *Given a block $\mathcal{H}$-tree T_2, we define the density of $\mathcal{M}_{\mathcal{H},k}(T_2)$ by*

$$D(T_2) := \frac{N_{\mathcal{H},St}(T_2)}{2kn\log_2(n)}.$$

For the block $\mathcal{H}$-tree corresponding to the block-partitioning B_2 in [6, Section 2.2.2], there holds $D(T_2) \approx 1$. This can be regarded as the coarsest reasonable block $\mathcal{H}$-tree and therefore as a lower bound for the density of a general admissible block $\mathcal{H}$-tree.

In order to assess the density of a given $\mathcal{H}$-matrix $H \in \mathcal{M}_{\mathcal{H},k}(T_2)$, one needs a reference value D^* for the same matrix represented in a block structure corresponding to a balanced block $\mathcal{H}$-tree $T_2^{balanced}$. In an adaptive process this value is given by the initial (balanced) block cluster tree. If, after some refinement steps, the density exceeds a certain threshold $(1+\delta)D^*$, then the (block) cluster tree is reconstructed in a balanced way and the reference value D^* can be updated. The optimal value of δ depends on how many times the matrix-vector multiplication has to be performed and how long the reconstruction of the (block) cluster tree (and reassembling of the matrix) takes. These values can also be estimated by the initial or most recent reconstruction of the (block) cluster tree and thus the parameter δ can be determined.

Example 3.2 *Let $H \in \mathcal{M}_{\mathcal{H},k}(T_2)$ be an $\mathcal{H}$-matrix. The reference value $D^* = D(T_2^{balanced})$ is given and we assume that the time for the execution of one matrix-vector multiplication is C_tD^* in the balanced and $C_tD(T_2)$ in the unbalanced case. The time for the complete re-clustering is estimated by $C_{recluster}$. If s matrix-vector multiplications should be performed then the optimal value for δ is*

$$\delta := \frac{C_{recluster}}{sC_tD^*}.$$

A crucial point in the adaptive scheme is to update the matrix with a complexity equal to the number of new matrix entries (usually $O(k\log_2(n))$). If the vectors a_i, b_i corresponding to some $\mathbf{R}k$-block in the matrix are stored as arrays, then the complete arrays have to be reallocated if the lengths of the vectors increase, which leads to a complexity equal to (at least) the size of the largest updated block (usually $O(n)$).

If the vectors a_i, b_i are stored as a doubly linked list and one has to find the j-th entry of the vector a_i, then one has to go through the whole list (in the worst case), which leads again to a complexity equal to (at least) the size of the largest updated block (usually $O(n)$). In the next section we present an implementation that overcomes this bottleneck.

4 Implementation of Adaptively Expandable $\mathcal{H}$-Matrices

$\mathbf{R}k$-matrices, full matrices and vectors can be implemented in several ways, depending on factors such as the class of the machine where the calculation takes place, the problem to be discretised and/or solved, the available software to be used (e.g., LAPACK, MATLAB) and several other circumstances. Here we choose to store each vector a_i, b_i of an $\mathbf{R}k$-matrix and each row of a full matrix block as a doubly linked list. The purpose of the linked lists for the storage of vectors is to allow an inexpensive update if one entry i is replaced by a small set $\{i_1, \ldots, i_s\}$ of entries, as it might occur in an adaptive refinement process.

Implementation 4.1 ***(List based vectors)*** *A list based vector v is implemented as a doubly linked list of entries. Each entry $v.i$ has a pointer $v.i.\mathrm{prev}$ to its predecessor, a pointer $v.i.\mathrm{next}$ to its successor and a value $v.i.\mathrm{value}$. The first entry $v.\mathrm{first}$ of the list has no predecessor ($v.\mathrm{first.prev} = \mathrm{NULL}$) and the last entry $v.\mathrm{last}$ of the list has no successor ($v.\mathrm{last.next} = \mathrm{NULL}$). The vector v itself stores only the pointer to $v.\mathrm{first}$ and $v.\mathrm{first}$ stores a pointer $v.\mathrm{first.v}$ to the vector v.*

Implementation 4.2 ***(Rk-matrices)*** *An **R**k-matrix R stores $2k$ pointers $R.a[i], R.b[i], i = 1, \ldots, k$, to the (list based) vectors a_i, b_i (as they appear in Definition 2.9) in two respective arrays.*

Implementation 4.3 ***(full matrices)*** *Each row of a full matrix F is stored as a (list based) vector. F stores only the pointer to the first row $F.\mathrm{firstrow}$. Each entry i of a row-vector has (in addition to his predecessor $i.\mathrm{prev}$ and successor $i.\mathrm{next}$) a pointer to the lower entry in the same column ($i.\mathrm{lower}$) and the upper entry in the same column ($i.\mathrm{upper}$) (if they exist, otherwise they are NULL).*

Implementation 4.4 ***($\mathcal{H}$-matrices)*** *An $\mathcal{H}$-matrix $H \in \mathcal{M}_{\mathcal{H},k}(T_2)$ is recursively defined over the block $\mathcal{H}$-tree T_2. The structure of H is similar to that of T_2. H corresponds to the root of T_2 and has pointers $H.\mathrm{sub[i,j]}$, $i, j = 1, \ldots, |S(\mathrm{root}(T_1))|$, to the (sub-)matrices corresponding to the sons of $\mathrm{root}(T_2)$. The submatrices are **R**k-, full or again $\mathcal{H}$-matrices. Recursively this defines H. Each $\mathcal{H}$-submatrix H corresponding to $(t_1, t_2) \in T_2$ stores the number of sons $H.r := |S(t_1)|$, $H.s := |S(t_2)|$ of r, s.*

Next we will illustrate how to perform the matrix-vector multiplication $w = H \cdot v$ for the list based vectors $v, w \in \mathbb{R}^I$ and an $\mathcal{H}$-matrix H. Since H is recursively defined it is advantageous to define subroutines that perform $w' := w' + M \cdot v'$ for **R**k-, full or $\mathcal{H}$-submatrices M of H and subvectors w', v' of w, v. Here it is convenient to call the subroutines with the first entries of the (sub-)vectors. The algorithmic details in the context of list based vectors are given in Figures 4, 5.

Implementation 4.5 ***(degrees of freedom)*** *Each degree of freedom i (e.g., basis function, grid-point, triangle or abstract class) stores the pointers $*.\mathrm{entry}$ to its respective list entry of every vector (a_i, b_i of the **R**k-submatrices and v, w from the matrix-vector multiplication) it belongs to. These pointers are organised in a linked list where $i.\mathrm{first}$ is the first entry. Additionally, i stores pointers to each full matrix it belongs to in a doubly linked list, where $i.\mathrm{fullfirst}$ is the first entry.*

On each level of a sparse block $\mathcal{H}$-tree of depth p there are at most $2C_{sp}$ leaves that contain i, and thus there are at most $2(p+1)C_{sp} \max\{k, \frac{1}{2}n_{min}\}$ pointers to vectors of **R**k- or full matrices to be stored. Additionally, pointers to the vectors v, w for the matrix-vector multiplication $w = H \cdot v$ have to be stored. Overall, the complexity for the storage increases by $O(n(p+1)kC_{sp})$, thus the complexity for an $\mathcal{H}$-matrix implemented as above is of the same order but slightly larger than that of a standard (non-adaptable) $\mathcal{H}$-matrix.

An algorithm for the local update of the **R**k-blocks of the matrix and of the vectors (for the matrix-vector-multiplication) is given in Figure 5 (right). We restrict ourselves to the case where one degree of freedom i, specifying a row and column of the matrix, is replaced by exactly two new ones. Here, we are only concerned with providing storage space and accessing the entries, not providing the values for the entries. Furthermore, we assume that all admissible block clusters, where the new degrees of freedom are inserted, remain admissible, i.e., remain **R**k-matrices.

```
Procedure $\mathbf{R}k$-MVM$(R, pv, pw)$

call with pointers pv=v.first,pw=w.first
uses real number $s$, pointers $x, y, x'$;
for $i = 1 \ldots k$ do begin
    $s := 0$;
    $x := pv$; $y := R.b[i]$.first;
    while $y$.next $\neq$ NULL do begin
        $s := s + y$.value $\cdot$ $x$.value;
        $y := y$.next; $x := x$.next;
    end;
    $s := s + y$.value $\cdot$ $x$.value;
    $x' := x$;
    $x := pw$; $y := R.a[i]$.first;
    while $y$.next $\neq$ NULL do begin
        $x$.value := $x$.value + $y$.value $\cdot$ $s$;
        $y := y$.next; $x := x$.next;
    end;
    $x$.value := $x$.value + $y$.value $\cdot$ $s$;
end;
$pv := x', pw := x$;

returns $pv = v$.last$, pw = w$.last
```

```
Procedure F-MVM$(F, pv, pw)$

call with pointers pv=v.first,pw=w.first
uses real number $s$, pointers $r, x, y, c$;
$r := F$.firstrow.first; $y := pw$;
while $r$.lower $\neq$ NULL do begin
    $s := 0$; $x := pv$; $c := r$;
    while $c \neq$ NULL do begin
        $s := s + c$.value $\cdot$ $x$.value;
        $c := c$.next; $x := x$.next;
    end;
    $y$.value := $y$.value + $s$;
    $r := r$.lower; $y := y$.next;
end;
$c := r$; $s := 0$; $x := v$.first;
while $c$.next $\neq$ NULL do begin
    $s := s + c$.value $\cdot$ $x$.value;
    $c := c$.next; $x := x$.next; end;
$s := s + c$.value $\cdot$ $x$.value;
$y$.value := $y$.value + $s$;
$pv := x, pw := y$;

returns $pv = v$.last$, pw = w$.last
```

Figure 4: $\mathbf{R}k$ - and full matrix-vector multiplication $w := w + R \cdot v$, $w := w + F \cdot v$, resp.

```
Procedure $\mathcal{H}$-MVM$(H, pv, pw)$

call with pointers pv=v.first,pw=w.first
uses integers $i, j$, pointers $x, x', y$;
$y := pw$;
for $i = 1, \ldots, H.r$ do begin
    $x := pv$; $y' := y$;
    for $j = 1, \ldots, H.s$ do begin
        if $j \neq H.s$ then $y := y'$;
        if $H$.sub[i,j] is an $\mathcal{H}$-matrix then
            call $\mathcal{H}$-MVM($H$.sub[i,j]$, x, y$);
        if $H$.sub[i,j] is an $\mathbf{R}k$-matrix then
            call $\mathbf{R}k$-MVM($H$.sub[i,j]$, x, y$);
        if $H$.sub[i,j] is a full matrix then
            call F-MVM($H$.sub[i,j]$, x, y$);
        if $i \neq H.r$ then $x := x$.next;
    end;
    if $j \neq H.s$ then $y := y$.next;
end;
pv:=x; pw:=y;

returns $pv = v$.last$, pw = w$.last
```

```
Procedure $\mathcal{H}$-Update$(H, i, i_1, i_2)$

uses pointers $x, x_1, x_2, x', j_1, j_2, j_1', j_2', xe$;
$x := i$.first; $x' :=$ NULL;
while $x \neq$ NULL do begin
    $x_1 :=$ new list entry; $x_2 :=$ new list entry;
    $j_1 :=$ new list entry; $j_2 :=$ new list entry;
    $xe := x$.entry;
    if $x \neq i$.first then begin
        $j_1'$.next $:= j_1$; $j_2'$.next $:= j_2$; end;
    else begin
        $i_1$.first $:= j_1$; $i_2$.first $:= j_2$; end;
    $j_1$.entry $:= x_1$; $j_1' := j_1$; $j_2$.entry $:= x_2$; $j_2' := j_2$;

if $xe$.prev $\neq$ NULL then $xe$.prev.next $:= x_1$;
    if $xe$.next $\neq$ NULL then $xe$.next.prev $:= x_2$;
    $x_1$.next $:= x_2$; $x_2$.prev $:= x_1$;
    if $xe$.prev $\neq$ NULL then $x_1$.prev $:= xe$.prev;
      else begin $x_1$.v $:= xe$.v; $xe$.v.first $:= x_1$; end;
    if $xe$.next $\neq$ NULL then $x_2$.next $:= xe$.next;
    $x' := x$.next; delete $x$.entry; delete $x$; $x := x'$;
end;
```

Figure 5: $\mathcal{H}$-matrix-vector multiplication $w := w + H \cdot v$ and the local $\mathbf{R}k$ -matrix/vector expansion.

A special case that has not been mentioned yet occurs if an (inadmissible) full matrix block has to be expanded. Due to the expansion, the size of the block may exceed n_{min} and thus the corresponding index sets have to be further divided and checked for admissibility. Since the size of the block is limited by n_{min}, the complexity for the update is $O(n_{min})$ if the size of the block does not exceed n_{min} and $O(n_{min}^2)$ otherwise. The implementational techniques are similar to those in Figure 5 (right).

5 Numerical Results

In this section we provide a numerical example that employs the adaptive update procedures of the previous section. We assume that each admissible block of the matrix to be stored is an $\mathbf{R}k$-matrix and that the entries of the vectors a_i, b_i (Definition 2.9) are given, thus omitting the discretisation process as well as the $\mathbf{R}k$-approximation. The blockwise $\mathbf{R}k$-approximation can be performed, e.g., by some explicit separable kernel approximation in the case of the boundary element method ([9], [8]) or some low rank/incomplete cross approximation ([1], [10]).

Each entry $A_{ij}^{(l)}$ of the matrix to be stored and evaluated (in the l-th adaptive step) is related to a pair of basis functions $\phi_i^{(l)}, \phi_j^{(l)} : [0,1]^2 \to \mathbb{R}$.

The one-dimensional example in Section 3, Figure 2, was introduced to demonstrate that the local matrix expansion *might* lead to an expensive arithmetic. For two-dimensional grids, a similar negative effect theoretically occurs when a grid is adaptively refined towards a single point. The situation in practical applications is somewhat different: Here we typically begin with some refinement (and a corresponding balanced cluster tree), and then perform a few adaptive refinement steps around (possibly several) critical points. The number of unknowns created through the adaptive refinement is relatively small compared to the number of unknowns created in the initial refinement. This leads to only a small portion of the adaptively updated cluster tree to be unbalanced. As a consequence, the overall complexity estimates for storage and matrix arithmetic are dominated by the initial balanced part of the cluster tree, i.e. they are still nearly optimal.

If the adaptive refinement concerns not just a single point but a (larger) region, hence creating more than just a few extra unknowns, then the complexities for storage and matrix arithmetic are still nearly optimal even for the adaptive update scheme. This is illustrated by the following example.

The reference grid $\mathcal{G}_0$ that defines the supports $X_i^{(0)}$ of the basis functions is a regular tensor-product grid of $[0,1]^2$ with 32×32 panels ($n_0 = 1024$ degrees of freedom). Subsequent grids $(\mathcal{G}_l)_{l=1}^{l_{max}}$ are obtained by refining the panels adjacent to the y-axis (see Figure 6) and define the index sets $(I_l)_{l=1}^{l_{max}}$ with $n_l := |I_l| = 992 + \sum_{i=0}^{l-1} 2^{i+6} + 2^{l+5}$ elements and the basis functions $(\phi_i^{(l)})_{i=1}^{n_l}$. The supports $(X_i^{(l)})_{i=1}^{n_l}$ of the basis functions $\mathcal{B}_l := \{\phi_i^{(l)} \mid i \in I_l = \{1, \dots, n_l\}\}$ are squares. Here we have assumed $\mathcal{B}_l$ to be the canonical basis of the space of piecewise constant functions on $\mathcal{G}_l$.

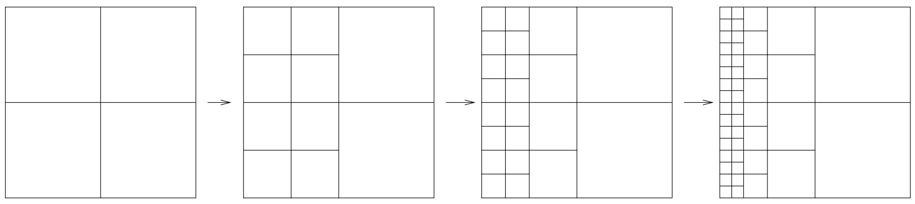


Figure 6: A locally refined reference grid.

If $X_i^{(l)} \subset \tau$, $X_j^{(l)} \subset \sigma$ for two subsets $\tau, \sigma \in \mathbb{R}^2$ and the admissibility condition

$$\min\{\operatorname{diam}(\tau), \operatorname{diam}(\sigma)\} \leq 2\eta \operatorname{dist}(\tau, \sigma) \tag{5.1}$$

holds (the diameter diam and distance dist are based on the Euclidean norm in $\mathbb{R}^2$, $\eta \in [0,1]$), then

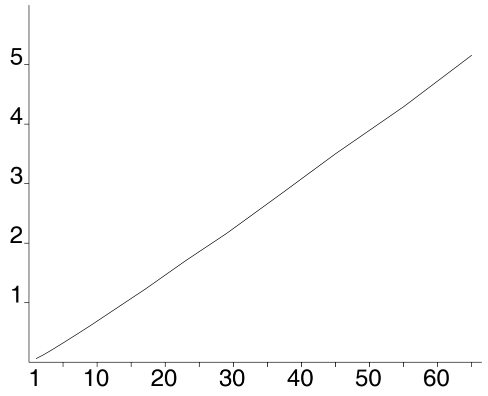
$$A_{ij}^{(l)} = \sum_{\nu=1}^{k} f_\nu^\tau(\phi_i^{(l)}) g_\nu^\sigma(\phi_j^{(l)}),$$

for some $f_\nu^\tau, g_\nu^\sigma : \mathcal{B}_l \to \mathbb{R}$. If the admissibility condition does not hold, then

$$A_{ij}^{(l)} = h^{\tau,\sigma}(\phi_i^{(l)}, \phi_j^{(l)}),$$

for some $h^{\tau,\sigma} : \mathcal{B}_l \times \mathcal{B}_l \to \mathbb{R}$.

The balanced tree $T_1(I_0)$ is constructed by binary space partitioning (sometimes called geometric bisection, see e.g. [2]) and will result in a cardinality-balanced tree. The block $\mathcal{H}$-tree $T_2(I_0 \times I_0)$ is given by Construction 2.7 (we set the parameter $n_{min} := 4$). The $\mathcal{H}$-matrices H_l corresponding to the block $\mathcal{H}$-trees $T_2(I_l \times I_l)$ are obtained by the local matrix expansion (Figure 5 (right)). In Figure 7



l	n_l	$D(T_2(I_l \times I_l))$	$t(H_l \cdot v)$
0	1024	38	0.06
4	2464	35	0.13
5	4000	35	0.27
6	7072	34	0.49
7	13216	34	0.97
8	25504	34	1.85
9	50080	33	4.0

Figure 7: Left: The storage requirements for $n = 1{\cdot}10^3, \ldots, 60{\cdot}10^3$ degrees of freedom in $100 MegaByte$. Right: The level l, the number of basis functions n_l, the density $D(T_2(I_l \times I_l))$, where we replaced $N_{\mathcal{H},St}(T_2)$ by the actual amount of storage required for the matrix (the unit is sizeof(double)), and the time for one matrix-vector multiplication in seconds (SunEnterprise, 250 MHz).

we have depicted the storage requirements for the matrices H_l and the corresponding density-indices $D(T_2(I_l \times I_l))$, that do *not* increase. Here we set the local matrix ranks to $k = 1$, and the parameter η in the admissibility condition (5.1) to $\eta = 0.8$. We point out that the storage requirements and the complexity for the matrix-vector multiplication are roughly twice as large as in the standard (not adaptively expandable) $\mathcal{H}$-matrix case.

If the (block) $\mathcal{H}$-tree $T_2(I_l \times I_l)$ is built by local matrix expansion and the supports of the basis functions are nested (as in the numerical example chosen here), then the tree is identical to the one built for the index set $I_l \times I_l$ by geometrically-balanced binary space partitioning (see [3]). The consequence is that all grids that allow for a geometrically-balanced construction of the $\mathcal{H}$-matrix are immediately suited for the local matrix expansion. If the grid is not suited for the local matrix expansion, then this is indicated by the density measure D and one can try to re-cluster the tree T_2 in a cardinality-balanced way.

References

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

[2] Grasedyck, L.: Theorie und Anwendungen Hierarchischer Matrizen (German). PhD thesis, University Kiel, Germany, 2001.

[3] Grasedyck, L. and Hackbusch, W.: Construction and arithmetic of $\mathcal{H}$-matrices. In preparation.

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

[5] Hackbusch, W. and Khoromskij, B. N.: A sparse $\mathcal{H}$-matrix arithmetic: General complexity estimates. J. Comp. Appl. Math. 125, 479–501 (2000).

[6] Hackbusch, W. and Khoromskij, B. N.: A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems. Computing 64, 21–47 (2000).

[7] Hackbusch, W. and Khoromskij, B. N.: $\mathcal{H}$-matrix approximation on graded meshes. In John R. Whiteman (ed.): *The Mathematics of Finite Elements and Applications X*, pages 307–316. Elsevier, 2000.

[8] Hackbusch, W. and Nowak, Z. P.: On the fast matrix multiplication in the boundary element method by panel clustering. Numer. Math. 54, 463–491 (1989).

[9] Rokhlin, V.: Rapid solution of integral equations of classical potential theory. J. Comput. Phys. 60, 187–207 (1985).

[10] Tyrtyshnikov, E. E.: Incomplete cross approximation in the mosaic-skeleton method. Computing 64, 367–380 (2000).

Sabine Le Borne
Department of Mathematics
Box 5054
Tennessee Technological University
Cookeville, TN 38505
USA

sleborne@tntech.edu

Lars Grasedyck
Mathematisches Seminar II
Universität zu Kiel
Hermann-Rodewald-Str. 3
D-24098 Kiel
Germany

lgr@numerik.uni-kiel.de

Wolfgang Hackbusch
Max-Planck-Institut für
Mathematik in den
Naturwissenschaften
Inselstr. 22-26
D-04103 Leipzig
Germany

wh@mis.mpg.de