A scalable $\mathcal{H}$-matrix approach for the solution of boundary integral equations on multi-GPU clusters

H. Harbrecht and P. Zaspel

Departement Mathematik und Informatik
Fachbereich Mathematik
Universität Basel
CH-4051 Basel

Preprint No. 2018-11
June 2018

www.math.unibas.ch

A scalable $\mathcal{H}$-matrix approach for the solution of boundary integral equations on multi-GPU clusters

H. Harbrecht* P. Zaspel*

June 19, 2018

Abstract

In this work, we consider the solution of boundary integral equations by means of a scalable hierarchical matrix approach on clusters equipped with graphics hardware, i.e. graphics processing units (GPUs). To this end, we extend our existing single-GPU hierarchical matrix library `hmglib` such that it is able to scale on many GPUs and such that it can be coupled to arbitrary application codes. Using a model GPU implementation of a boundary element method (BEM) solver, we are able to achieve more than 67 percent relative parallel speed-up going from 128 to 1024 GPUs for a model geometry test case with 1.5 million unknowns and a real-world geometry test case with almost 1.2 million unknowns. On 1024 GPUs of the cluster *Titan*, it takes less than 6 minutes to solve the 1.5 million unknowns problem, with 5.7 minutes for the setup phase and 20 seconds for the iterative solver. To the best of the authors' knowledge, we here discuss the first fully GPU-based distributed-memory parallel hierarchical matrix Open Source library using the traditional $\mathcal{H}$-matrix format and adaptive cross approximation with an application to BEM problems.

1 Introduction

The numerical solution of boundary integral equations is an important task in applications from science and engineering such as electric field computations, electromagnetism, acoustic scattering, or fluid mechanics [29]. Boundary integral equations arise typically from the reformulation of *boundary value problems* with constant coefficients. In many cases, such a reformulation is advantageous, since a discretization of a boundary integral equation requires the introduction of degrees of freedom just *on the boundary*, while the original partial differential equation needs to be discretized *in the full domain*, which might be unbounded in case of exterior boundary value problems.

*Department for Mathematics and Computer Science, Unversity of Basel, Switzerland

Besides the collocation method, the standard approach for discretizing and solving boundary integral equations is based on a Galerkin discretization which amounts to the *boundary element method (BEM)*, see [16, 43, 46] for example. BEM applies standard techniques known from the finite element method (FEM) to the boundary integral equation case. Since the kernel of a boundary integral operator is usually not compactly supported and singular at the diagonal, the stiffness matrix of a BEM discretization is densely populated and computationally expensive to compute. To overcome the cubic complexity of direct factorization approaches, iterative solvers with fast approximate matrix-vector products are used to solve the system of linear equations. Candidates for matrix approximations are the panel clustering [24], the fast multipole method [19], hierarchical ($\mathcal{H}$) matrices [3, 9, 20, 21] or $\mathcal{H}^2$ matrices [6, 22, 23]. We here focus on $\mathcal{H}$-matrices, since these are a widely use approach and, together with adaptive cross approximation (ACA) [5], allow for a purely algebraic construction of the matrix approximation, facilitating its use in real-world applications. For a given fixed accuracy, it is possible to show that the approximate matrix-vector product of $\mathcal{H}$-matrices can be done in $O(N \log N)$ operations.

The objective of this work is to solve large-scale BEM problems by the hierarchical matrix approach. In fact, we aim for solving systems of linear equations from BEM discretizations with hundreds of thousands or millions of unknowns. Such problem sizes arise if the underlying geometry is either very complex or if a solution with a small numerical error is required.

We observe two difficulties when it comes to the solution of large-scale BEM problems. First, the computational runtime becomes excessively large. Second, the required memory is considerable. While the first problem can be addressed by a parallelization of a hierarchical matrix library on a single compute node with high amounts of memory, the second problem can only be fixed by a *distributed-memory* parallelization of the hierarchical matrix approach. Our conclusion is to work on a distributed-memory parallel implementation for $\mathcal{H}$-matrices. In particular, we apply and extend the many-core parallel Open Source library `hmglib` [48, 49] in order to treat BEM-type problems in a distributed-memory parallel way.

`hmglib` is a many-core parallel library allowing to set up and apply $\mathcal{H}$-matrices on a single *graphics processing unit* (GPU). It has been originally developed in the context of the approximation of system matrices from kernel collocation or kernel ridge regression, where it showed decent performance improvements over a parallel $\mathcal{H}$-matrix implementation on standard processors (CPUs). As part of the present work, `hmglib` has been extended such that it can be applied to arbitrary application codes with dense system matrices, as long as the codes provide a means to evaluate matrix entries and the geometric location of the involved degrees of freedom. That is, `hmglib` now provides a general interface e.g. for BEM codes. Moreover, and much more important, the library has been extended such that it is now able to

run on a distributed-memory parallel cluster of GPUs. This allows to scale the solution of BEM problems on up to millions of unknowns. While the original implementation of `hmglib` in [48, 49] could only pre-compute low-rank blocks, the new implementation is further able to pre-compute and store the (non-admissible) dense matrix blocks in (GPU) memory. This is crucial for BEM applications.

Note that there is a series of related CPU libraries for parallel hierarchical matrices. $\mathcal{H}$-Lib^{pro} [9, 18, 32, 34] is a commercial shared-memory parallel library with limited distributed-memory support. `AHMED` (Another software library on hierarchical matrices for elliptic differential equations) [2] and `DMHM` (Distributed-Memory Hierarchical Matrices) [40] provide distributed-memory support on CPUs. `H2Lib` [7] is shared-memory parallel. In the related field of Hierarchically Semi-Separable (HSS) matrices [44], the software `STRUMPACK` [17, 41] is shared- and distributed-memory parallel. In context of many-core processors, i.e. GPUs and e.g. Intel Xeon Phi, there is some recent work. An extension to the `H2Lib` [8] allows to accelerate the quadrature in a $\mathcal{H}^2$ matrix method for BEM by GPUs. A similar approach is used in the BEM library `Bempp` [45, 47]. Furthermore, [33] discusses a many-core parallel LU-factorization for $\mathcal{H}$-matrices on a Xeon Phi device. `BEM4I` [31, 37] provides a BEM library with ACA running on clusters of multi-/many-core hardware by Intel based on an MPI, OpenMP and vectorization parallelization. In [39], the $\mathcal{H}$-matrix vector product (without setup) has been parallelized on a single GPU and on Intel processors. The new *tile low rank* (TLR) format is used in `HiCMA` a library for low-rank Cholesky factorizations on clusters of multi-core and many-core hardware running on Intel hardware [1] and with the main application of Matérn-type covarice matrices. By some of the authors of [1], further work has been carried out, which focused on batched dense linear algebra kernels [11] on a single GPU, batched QR and SVD algorithms [10] on GPUs and a batched TLR GEMM operation on a single GPU [12]. However, to the best of the authors' knowledge, we here discuss the first fully GPU-based distributed-memory parallel hierarchical matrix Open Source library using the traditional $\mathcal{H}$-matrix format and adaptive cross approximation being applied to BEM problems.

To be able to apply our parallel library to a (large-scale) BEM model problem, we further parallelized an existing sequential CPU code for the solution of elliptic problems by the single-layer potential ansatz with piecewise constant basis functions on GPU and coupled that code to `hmglib`. Thereby, we will be able to show that we can solve large-scale BEM problems in the range of millions of unknowns with a descent strong scaling beyond 68 percent strong scaling efficiency on 1024 GPUs of *Titan* at Oak Ridge National Lab (starting from 128 GPUs due to memory limitations).

The remainder of this work is structured as follows. In Section 2, we introduce the mathematical background of boundary integral equations, the boundary element method and the hierarchical matrix approach. Section 3

briefly reviews the computational details of `hmglib` and introduces the new general application interface, the dense block storage and multi-GPU parallelization. Numerical results and parallel scalability studies are discussed for a model application and a model code in Section 4. We finish by conclusions in Section 5.

2 Mathematical background

2.1 Boundary integral equations

Shall $\Omega \subset \mathbb{R}^d$ with $d = 3$ be a Lipschitz domain and $\Gamma := \partial\Omega$ its surface. We aim at solving boundary integral equations of type

$$(\mathcal{A}u)(\boldsymbol{x}) := \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}')u(\boldsymbol{x}')\mathrm{d}\sigma_{\boldsymbol{x}'} = f(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Gamma\,, \tag{1}$$

where $\mathcal{A}$ is supposed to be an invertible boundary integral operator. We assume that $\mathcal{A}$ is a continuous and elliptic operator of order $2q$, which means that it maps from $H^q(\Gamma)$ to $H^{-q}(\Gamma)$. We require the integral kernel $G : \Omega \times \Omega \to \mathbb{R}$ to be *asymptotically smooth*, that is, we require to have constants $C_{as1}, C_{as2} \in \mathbb{R}^{>0}$ such that

$$|\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} \partial_{\boldsymbol{x}'}^{\boldsymbol{\beta}} G(\boldsymbol{x}, \boldsymbol{x}')| \leq C_{as1} \frac{(|\boldsymbol{\alpha}| + |\boldsymbol{\beta}|)!}{(C_{as2}\|\boldsymbol{x} - \boldsymbol{x}'\|)^{|\boldsymbol{\alpha}|+|\boldsymbol{\beta}|}} |G(\boldsymbol{x}, \boldsymbol{x}')|$$

for arbitrary $\boldsymbol{x}, \boldsymbol{x}' \in \Omega$ with $\boldsymbol{x} \neq \boldsymbol{x}'$ and all multi-indices $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{N}_0^d$. This choice allows for kernel functions with singularities at the *diagonal* $\boldsymbol{x} = \boldsymbol{x}'$, while being smooth away from the diagonal.

Example 2.1. Boundary integral equations of the above type arise in context of the solution of the Laplace equation

$$\Delta U = 0 \text{ in } \Omega, \quad U = f \text{ on } \Gamma,$$

where $U \in H^1(\Omega)$ is the solution for given Dirichlet data $f \in H^{1/2}(\Gamma)$. Since we know the fundamental solution of the Laplace operator, we can make the the single-layer potential ansatz

$$U(\boldsymbol{x}) = \int_\Gamma \frac{u(\boldsymbol{x}')}{4\pi\|\boldsymbol{x} - \boldsymbol{x}'\|_2} \mathrm{d}\sigma_{\boldsymbol{x}'} = \tilde{\mathcal{S}}u(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,. \tag{2}$$

That is, we describe the solution of the Laplace equation by means of the unknown density $u \in H^{-1/2}(\Gamma)$. Since the single-layer potential is continuous in the whole space $\mathbb{R}^d$, the density is obtained by solving the boundary integral equation

$$(\mathcal{S}u)(\boldsymbol{x}) = \int_\Gamma \frac{u(\boldsymbol{x}')}{4\pi\|\boldsymbol{x} - \boldsymbol{x}'\|_2} \mathrm{d}\sigma_{\boldsymbol{x}'} = f(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Gamma\,, \tag{3}$$

where $\mathcal{S} : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ is the *single-layer operator* and f the Dirichlet data of the Laplace equation. It can be shown that the kernel function $\frac{1}{4\pi\|\boldsymbol{x}-\boldsymbol{x}'\|_2}$ is asymptotically smooth and that $\mathcal{S}$ is continuous and continuously invertible. ▲

2.2 Galerkin BEM discretization

To solve (1) by the boundary element method (BEM), we first bring the equation in its variational form: Find $u \in V(= H^q(\Gamma))$, such that

$$\int_\Gamma \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}') u(\boldsymbol{x}') v(\boldsymbol{x}) \mathrm{d}\sigma_{\boldsymbol{x}'} \mathrm{d}\sigma_{\boldsymbol{x}} = \int_\Gamma f(\boldsymbol{x}) v(\boldsymbol{x}) \mathrm{d}\sigma_{\boldsymbol{x}} \text{ for all } v \in V\,.$$

We discretize it by introducing an approximation of the boundary Γ by surface elements

$$\mathcal{T}_h := \{T_1, \ldots, T_M\}$$

of size $O(h)$. The elements T_i are usually chosen as planar triangles, cf. [16, 46, 43]. Nonetheless, parametric representations of the surface have recently been become quite popular. Then, $\mathcal{T}_h$ would be a structured quadrangulation and T_i a curved quadrangle, cf. [27, 36, 13].

The elements induce a set of nodes

$$X_h := \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}\,.$$

We associate to each node $\boldsymbol{x}_i$ a locally supported piecewise polynomial φ_i of order p leading to a finite-dimensional trial space

$$V_h = \{\varphi_1, \ldots, \varphi_N\} \subset V\,.$$

Then, we look for an approximate solution $u_h \in V_h$, such that

$$\int_\Gamma \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}') u_h(\boldsymbol{x}') v_h(\boldsymbol{x}) \mathrm{d}\sigma_{\boldsymbol{x}'} \mathrm{d}\sigma_{\boldsymbol{x}} = \int_\Gamma f(\boldsymbol{x}) v_h(\boldsymbol{x}) \mathrm{d}\sigma_{\boldsymbol{x}} \text{ for all } v_h \in V_h\,.$$

With $u_h(\boldsymbol{x}) := \sum_{i=1}^N \alpha_i \varphi_i(\boldsymbol{x})$, we finally have to solve the dense linear system

$$\boldsymbol{A}\boldsymbol{\alpha} = \boldsymbol{f} \tag{4}$$

with

$$\boldsymbol{A} = [a_{i,j}]_{i,j=1}^N, \quad a_{i,j} = \int_\Gamma \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}') \varphi_i(\boldsymbol{x}') \varphi_j(\boldsymbol{x}') \mathrm{d}\sigma_{\boldsymbol{x}'} \mathrm{d}\sigma_{\boldsymbol{x}}$$

and

$$\boldsymbol{f} = [f_i]_{i=1}^N, \quad f_i = \int_\Gamma f(\boldsymbol{x}) \varphi_i(\boldsymbol{x}) \mathrm{d}\sigma_{\boldsymbol{x}}\,.$$

2.3 Hierarchical matrices

We aim at solving (4) by an iterative method. To make this tractable for large N, we use an approximate matrix-vector product for the Galerkin system matrix $\boldsymbol{A}$. Our choice is to use the purely algebraic hierarchical matrices [9, 20] with adaptive cross approximation [4, 5], leading to $O(N \log N)$ complexity for the matrix-vector product, if we fix the approximation tolerance.

Let us briefly consider the approximation of the Galerkin system matrix $\boldsymbol{A}$ by hierarchical matrices. We first introduce the concept of index sets $I := \{1, \dots, N\}$, representing the nodes $X_h = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\}$ and basis functions $V_N = \{\varphi_1, \dots, \varphi_N\}$ on Γ. Thereby we can associate an index tuple (i, j) to a geometric location and to each entry $a_{i,j}$ of the system matrix $\boldsymbol{A}$. We will group these index sets into *clusters* $\tau \subset I$ based on geometrical arguments. The product of two clusters (i.e. a *block cluster*), e.g. $\tau \times \sigma \subset I \times I$, can then be translated to a sub-matrix $\boldsymbol{A}|_{\tau\times\sigma}$ of our Galerkin system matrix $\boldsymbol{A}$.

The core idea of hierarchical matrices relies on the fact that for an asymptotically smooth kernel, the evaluation of $\int_\Gamma \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}')\varphi_i(\boldsymbol{x}')\varphi_j(\boldsymbol{x}')\mathrm{d}\sigma_{\boldsymbol{x}'}\mathrm{d}\sigma_{\boldsymbol{x}}$ for two geometrically well separated basis functions φ_i, φ_j can be approximated with a controlled, small error. This can be expanded to the *admissibility*, i.e. the approximability, of a whole block cluster (of nodes). A typical admissibility condition for a block cluster $\tau \times \sigma$ is based on the bounding boxes for clusters τ, σ (compare e.g. [20] for alternatives). The bounding box of cluster $\tau \subset I$ is $Q_\tau := \prod_{i=1}^{3} \left[a_\tau^{(i)}, b_\tau^{(i)}\right]$ with $a_\tau^{(i)} := \min_{j\in\tau} x_j^{(i)}$, $b_\tau^{(i)} := \max_{j\in\tau} x_j^{(i)}$ and $\boldsymbol{x}_j := \left(x_j^{(1)}, x_j^{(2)}, x_j^{(3)}\right)^\top$. Then, we can introduce the admissibility condition

$$\min\{\mathrm{diam}(Q_\tau), \mathrm{diam}(Q_\sigma)\} \leq \eta \mathrm{dist}(Q_\tau, Q_\sigma)\,, \tag{5}$$

where $\eta \in \mathbb{R}^{\geq 0}$ balances convergence and algorithmic complexity and $\mathrm{diam}(Q_\tau)$ and $\mathrm{dist}(Q_\tau, Q_\sigma)$ are the diameter and the distance of bounding boxes, respectively, cf. [9].

Clusters τ shall always collect geometrically close nodes. They are collected in a *cluster tree* $\mathcal{T}_I = (\mathcal{V}_I, \gamma)$, which imposes a spatial data structure with a hierarchy on I (or X_N). With $\mathcal{V}_I \subset \mathcal{P}(I)$ being the set of nodes in the tree, i.e. the clusters, γ a mapping $\gamma : \mathcal{V}_I \to \mathcal{P}(\mathcal{V}_I)$ of each cluster to its hierarchical sub-clusters, a cluster tree is given such that

(C1) $\tau \in \mathcal{P}(I) \setminus \{\emptyset\}$, for all $\tau \in \mathcal{V}_I$,

(C2) $\mathrm{root}(\mathcal{T}) = I$,

(C3) if $\tau \in \mathcal{V}_I$ is a leaf, i.e. $\gamma(\tau) = \emptyset$, then $|\tau| \leq C_{leaf}$ and

(C4) if $\tau \in \mathcal{V}_I$ is no leaf, then it has exactly two children $\gamma(\tau) = \{\tau_1, \tau_2\}$ and $\tau = \tau_1 \dot\cup \tau_2$.

Algorithm 1 Algorithm to build a block cluster tree

procedure BUILD_BLOCK_CLUSTER_TREE($\tau \times \sigma$, C_{leaf})
  if $\tau \times \sigma$ is not admissible and $|\tau| > C_{leaf}$ and $|\sigma| > C_{leaf}$ **then**
    $\gamma(\tau \times \sigma) \leftarrow \emptyset$
    for $\tau' \in \gamma(\tau)$ **do** ▷ Loop over children in cluster trees.
      for $\sigma' \in \gamma(\sigma)$ **do**
        $\gamma(\tau \times \sigma) \leftarrow \gamma(\tau \times \sigma) \cup \{\tau' \times \sigma'\}$ ▷ Add new child.
        BUILD_BLOCK_CLUSTER_TREE($\tau' \times \sigma'$, C_{leaf})
  else
    $\gamma(\tau \times \sigma) \leftarrow \emptyset$ ▷ $\tau \times \sigma$ becomes leaf.

In *cardinality-based clustering* (CBC), which will be use in this work, we further impose $|\tau_1| \approx |\tau_2|$ in **(C4)**.

With a given cluster tree, we can introduce the *block cluster tree* $\mathcal{T}_{I\times I} = (\mathcal{V}_{I\times I}, \gamma, \mu)$, which builds a hierarchy of block clusters out of the given cluster hierarchy. Here, $\mathcal{V}_{I\times I}$ is the set of nodes / block clusters in the tree and γ maps a block cluster to its children. Algorithm 1 recursively defines the block cluster tree and is launched with $\tau \times \sigma = I \times I$.

The corresponding sub-matries $\boldsymbol{A}|_{\tau\times\sigma} \in \mathbb{R}^{|\tau|\times|\sigma|}$ of admissible block clusters in $\mathcal{T}_{I\times I}$ are approximated by an $\mathcal{R}(k)$ *matrix* $\boldsymbol{R}_{\tau\times\sigma} \in \mathbb{R}^{|\tau|\times|\sigma|}$, a matrix of maximum rank k, which is defined as

$$\boldsymbol{R}_{\tau\times\sigma} = \boldsymbol{U}_{\tau\times\sigma}\boldsymbol{V}_{\tau\times\sigma}^{\top}, \quad \boldsymbol{U}_{\tau\times\sigma} \in \mathbb{R}^{|\tau|\times k}, \quad \boldsymbol{V}_{\tau\times\sigma} \in \mathbb{R}^{|\sigma|\times k}.$$

Matrix-vector products with $\mathcal{R}(k)$ matrices $\boldsymbol{R}_{\tau\times\sigma}$ have a computational complexity of $O\left(r \cdot (|\tau| + |\sigma|)\right)$. We will use the algebraic *adaptive cross approximation* (ACA) [5, 4] to approximate sub-matrices $\boldsymbol{A}|_{\tau\times\sigma} \in \mathbb{R}^{|\tau|\times|\sigma|}$. ACA can be seen as a pivoted Gauss elimination and constructs a low-rank approximation by successive rank-one updates.

Given a rank $k \in \mathbb{N}$ and a block cluster tree $\mathcal{T}_{I\times I}$, we introduce an $\mathcal{H}$*-matrix* of block-wise rank k as matrix $\boldsymbol{L} \in \mathbb{R}^{|I|\times|I|}$ such that

$$\operatorname{rank}(\boldsymbol{L}|_{\tau\times\sigma}) \leq k$$

for all admissible $\tau \times \sigma$. The construction of an $\mathcal{H}$-matrix for a dense matrix is known as *truncation*. Matrix-vector products between $\mathcal{H}$-matrices an vectors are realized by a recursive traversal of the block cluster tree. In each non-admissible leaf, the corresponding (precomputed) full sub-matrix is applied, while in admissible leafs the (pre-computed) low-rank approximation is applied. It has been shown, that specific versions of this approach allow to perform an $\mathcal{H}$-matrix-vector product in complexity $O(k \cdot N \log N)$ [20].

3 Scalable parallel $\mathcal{H}$-matrix approach for BEM

Our approximation of the Galerkin matrix by hierarchical matrices is based on the library `hmglib` [48, 49]. The present work aims at extending `hmglib` such that it can be used for the multi-GPU parallel solution of large-scale boundary integral equations discretized by the boundary element method. To this end, an abstract code interface, precomputation of dense matrix blocks and a distributed-memory parallelization had to be introduced. In the following, we give a brief overview of the original implementation [48] and discuss the new techniques that have been added to `hmglib`.

Remark on technical details. Note that an in-depth technical description of state-of-the-art GPU-parallel codes requires a lot of technical details, such as memory hierarchies, parallelization models, scheduling, caching, etc. As in [48], we here stick to a much less technically overwhelming discussion. To this end, we categorize parallel work loads either into work loads that can be handled by the use of a large amount of parallel threads working on independent tasks or into work loads that require the use of more complicated algorithms with complex thread interactions, such as reduction operations. While the first type of work loads can be easily parallelized by standard GPU parallelization techniques, i.e. in CUDA kernels, we use existing GPU libraries for the second type of work loads, whenever this is possible.

3.1 `hmglib` - A many-core parallel $\mathcal{H}$-matrix library

The Open Source GPU library `hmglib` has originally been developed for the approximation of matrices from *kernel interpolation / collocation* or *kernel ridge regression*. It uses a given single GPU for all tasks involved in the construction and application of a hierarchical matrix, i.e. it is not an accelerated but a solely GPU-based software. To be able to get high performance on GPU, the library uses a parallel traversal of the block cluster tree, space-filling curves (to build the clustering) and the concept of batching for many small similarly-sized tasks. In terms of software and hardware, it requires an *Nvidia* GPU and uses the `CUDA Toolkit`, i.e. CUDA kernels [26] for direct GPU programming, the STL-type algorithm library `Thrust` [28] running in parallel on a GPU and the BLAS/LAPACK-type libraries `CUBLAS` and `Magma` [14, 25].

Block cluster tree traversal. To get a high parallel performance on many-core hardware, it is necessary to express an existing algorithm in a very parallel way. In [48], this has been achieved for the construction and traversal of the block cluster by *level-wise* parallelization of the tree traversal. That is, all entries of a given level of a tree are computed in a many-core parallel fashion, while calculations of offsets in the memory are computed

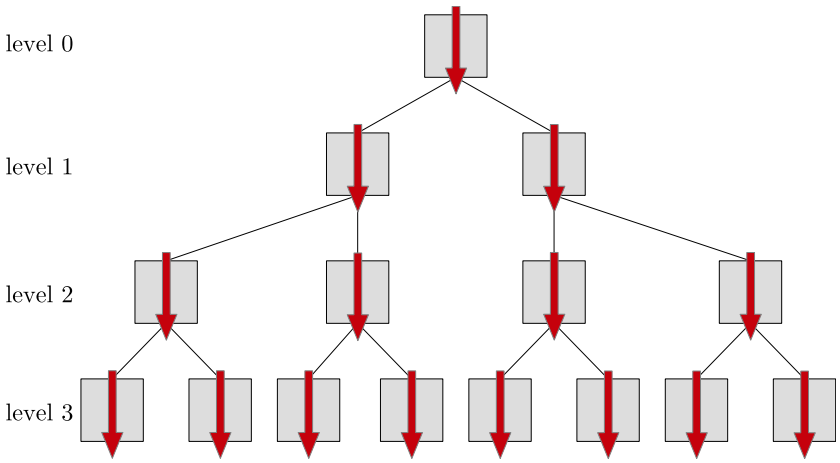

Figure 1: In `hmglib`, tree traversal is parallelized in a level-wise fashion. The red arrows on each level correspond to the executed parallel threads.

by appropriate parallel *scan* operations. Figure 1 outlines this methodology. The red arrows on a given level correspond to the parallel threads that are executed on the GPU. As it becomes obvious, the first few levels of a tree do not lead to a full parallel utilization. However, on higher levels, this limitation is no longer present.

Spatial data structure. While classical implementations of $\mathcal{H}$-matrices use spatial data structures such as *kD-trees* or *quad-/oct-trees*, clustering in `hmglib` is based on *space filling curves* [35, 38, 48]. In particular, a *Morton code* [38] is computed for each node in X_h. This computation is done in a many-core parallel way. After sorting the points in X_h following their Morton codes by a GPU-parallel sorting method, two consecutive nodes in the resulting (sorted) array of nodes are geometrically close. In particular, cardinality-based clustering, cf. Section 2.3, can be reduced to simple array decompositions.

Batching. As shown in [48], the highest impact on the GPU-parallel performance of the `hmglib` code is achieved by *batching* of small similarly sized compute tasks into bigger batches of compute work. This is specifically used in `hmglib` in the context of the computation of dense matrix blocks $\boldsymbol{A}_{\tau\times\sigma}$ and low-rank matrix blocks $\boldsymbol{R}_{\tau\times\sigma}$ and in context of the determination of bounding box sizes for the clusters. Figure 2 outlines the general strategy. Instead of solving (in parallel) several small problems (e.g. the summation of several numbers), all these operations are batched together in one bigger work load. This allows to achieve a much higher utilization of the GPU and, thus, leads to higher performance. While this strategy is supported

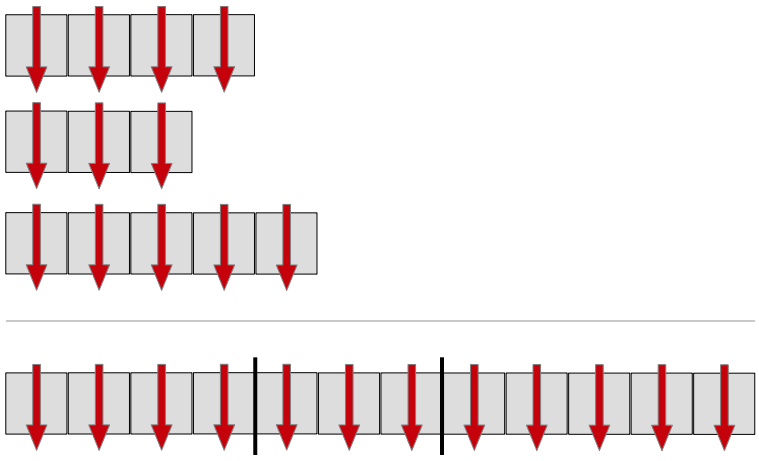

Figure 2: Instead of solving several smaller problems in parallel, *batching*, as used in `hmglib`, aims at combining all problems into a single much bigger work load, leading to a higher utilization and higher performance of a GPU.

in `Thrust` for, e.g., reduction operations by providing index arrays marking the sub-workloads, more complex algorithms such as the adaptive cross approximation in `hmglib` had to be adapted to use this new strategy.

3.2 Abstract program interface.

In [48], `hmglib` was just used for system matrices from kernel interpolation / collocation or kernel ridge regression. Such matrices require the evaluation of a very simple kernel function G, which was hard-coded. The new developments in the context of this work start adding an interface for arbitrary application codes that provide custom matrix entries of system matrices that shall be approximated. Besides of standard configuration options for $\mathcal{H}$-matrices, three general *inputs* have to be provided by an application code:

1. The sets of nodes $X_1 := \{\boldsymbol{x}_1^{(1)}, \ldots, \boldsymbol{x}_{N_1}^{(1)}\}$, $X_2 := \{\boldsymbol{x}_1^{(2)}, \ldots, \boldsymbol{x}_{N_2}^{(2)}\}$. In our BEM application, we have $X_1 = X_2 = X_h$.

2. Functions $\mathrm{idx}_1 : X_1 \to \mathbb{N}$, $\mathrm{idx}_2 : X_2 \to \mathbb{N}$, associating to each node an index. This allows to introduce an `hmglib`- and ordering-independent way to identify the nodes in X_1 and X_2 by the application. In our application, this is simply the index of the nodes.

3. A callback-type function that can be called by `hmglib` in order to evaluate a single entry $a_{i,j}$ of the system matrix $\boldsymbol{A}_{X_1 \times X_2}$ for which the $\mathcal{H}$-matrix shall be constructed. In our application, this is the

quadrature routine that computes the matrix entry

$$a_{i,j} = \int_\Gamma \int_\Gamma G(\boldsymbol{x}, \boldsymbol{x}')\varphi_i(\boldsymbol{x}')\varphi_j(\boldsymbol{x}')\mathrm{d}\sigma_{\boldsymbol{x}'}\mathrm{d}\sigma_{\boldsymbol{x}}\,.$$

Technical challenge. It turned out that developing a generalized way to provide a callback function for the matrix entry evaluation is not easy, at least in connection with the CUDA programming language extension provided by the `CUDA Toolkit` 8.0. While it is technically possible to use function pointers to *device functions*, i.e. functions that are executed on *and* launched from GPU, the practical use of these pointers leads to a strongly reduced performance. In order to work around this, we provide a purely virtual abstract *device* basis class with a function `get_matrix_entry`. This class is overwritten by the application that aims to use `hmglib`. At the same time, `hmglib` uses *dynamic polymorphism* to launch `get_matrix_entry` in an application-independent way.

Nonetheless, this solution introduces two difficulties. The first difficulty is in the memory management of the interface *device* class. It has to be instantiated and destroyed from *device* code. Therefore, its use in a library context, in which the interface code is supposed to run on CPU, tends to be rather involved. The second difficulty shows up in the compile and link process between the library `hmglib` and the application code. Ideally, the aim would be to provide `hmglib` as a shared library. However, in order to be able to use dynamic polymorphism on GPUs in the context of CUDA, it is necessary to put the calling *device* code, the abstract *device* basis class and the overwriting *device* class into the same *compilation unit*. While it is still possible to individually compile the *device* code for the calling code, the basis class and the overwriting class into *device*-only object files, these object files have to be linked by the device code linker before they can be put together with the CPU / *host* code. In practice, this breaks the clear distinction of library and application code. To the best of the authors' knowledge, there is currently no alternative to this approach when using GPUs and *CUDA*.

3.3 Pre-computation of matrix blocks.

In [48], some of us discussed the case of $\mathcal{H}$-matrix approximation for collocation matrices. The computational effort to compute the individual system matrix entries in that case was very low. Therefore, it was only considered to pre-compute the low-rank factors for the $\mathcal{R}(k)$-matrices $\boldsymbol{R}_{\tau\times\sigma} \in \mathbb{R}^{|\tau|\times|\sigma|}$, in order to avoid their re-calculation during each $\mathcal{H}$-matrix vector product.

In contrast, computing the entries in the Galerkin matrix for the boundary element method is very expensive. In this case, it is extremely important to further pre-compute and store the dense blocks $\boldsymbol{A}_{\tau\times\sigma}$. This has been realized in `hmglib`. The evaluation of the matrix entries is done in a batched way and the matrices are stored in GPU memory. To efficiently use the

available memory, we store the system matrices continuously in GPU memory, without any padding. Pointers to the offsets of each matrix are passed to the batched matrix-vector product provided by `Magma`.

3.4 Distributed-memory parallelization

We aim at a distributed-memory, i.e. multi-GPU, parallelization for two reasons. First, we want to be able to solve large problems for which we need a high amount of (GPU) memory. However, GPUs are usually rather limited in terms of the available memory. This is why we need many GPUs (*scale-up*). Moreover, we want to be able to solve BEM problems as fast as possible (*speed-up*).

To fulfill both requirements, we first tried a matrix-based parallelization by dividing the large-scale system matrix into blocks of rows, on which we independently applied the $\mathcal{H}$-matrix approximation. However, this lead to a sub-optimal load balancing and sub-optimal speed-up, since large admissible matrix blocks were cut into smaller pieces. The approach further became prohibitive, as we ran into the situation that admissible blocks were divided into skinny (i.e. wide but thin) sub-blocks. Therefore, in the worst case, one of the low-rank factors in the ACA could become as large as the number of unknowns in the linear system times the required rank, with a strong negative impact on scale-up. By introducing a full (row- and column-wise) block-partitioning of the system matrix, we could remove this second issue. However, we did still cut large admissible blocks into smaller pieces.

Task-based parallelization. We overcome most of the mentioned issues by using a *task-based* parallelization instead of a matrix-based parallelization. In our task-based parallelization, we focus specifically on problem size scale-up and calculation speed-up in the $\mathcal{H}$-matrix construction. This choice is valid, since the $\mathcal{H}$-matrix construction completely dominates the solution process in BEM applications.

Our task-based parallelization first builds on all GPUs the identical global block cluster tree and identifies admissible and non-admissible leafs. These leafs are put into two task lists being identical on each GPU. Note that no system matrix entry has been evaluated at this stage. Then, each of the two task lists is divided into p sub-lists, where p is the number of GPU processors. The very computationally expensive construction and storage of the low-rank or dense blocks in the sub-lists is done in a distributed way on each GPU. Figure 3 illustrates our parallelization concept. At the top part of this figure, we show the decomposed task-lists, which are then associated to one GPU. Our approach allows to fully decouple the construction of the $\mathcal{H}$-matrix. In practice, we use the Message Passing Interface (MPI) and associate one CPU process / thread to one GPU. The only parallelization information, which is required during the $\mathcal{H}$-matrix construction phase, is

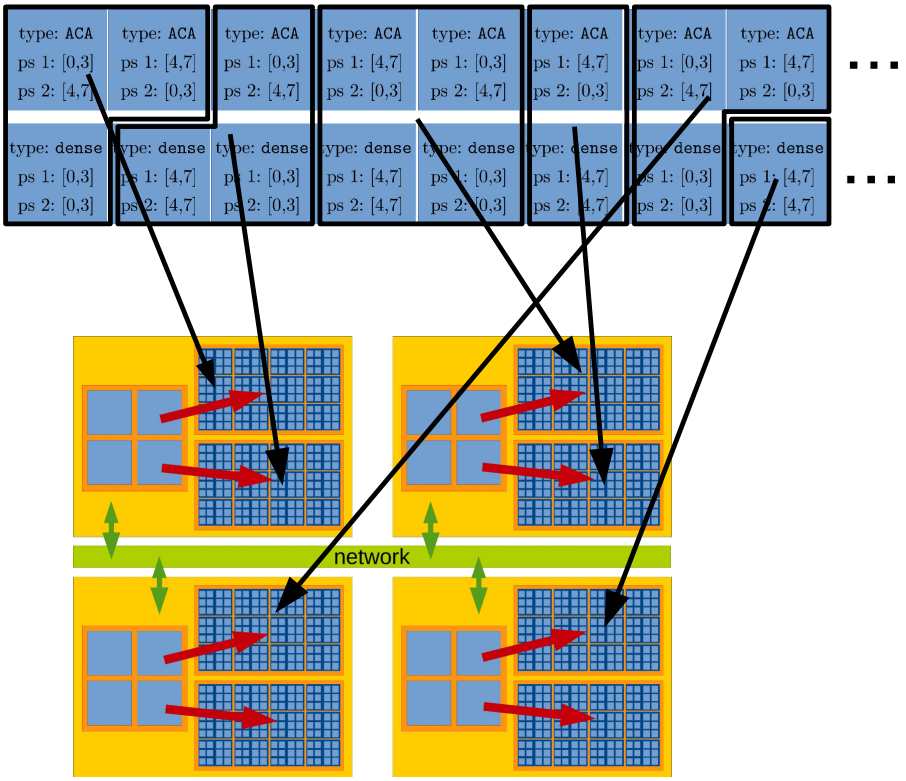

Figure 3: The distributed-memory parallelization of the $\mathcal{H}$-matrix approach distributes similar sized subsets of the dense and low-rank matrix blocks to the different GPUs. The yellow boxes represent hybrid compute nodes in an HPC system that are equipped with a four-core CPU and two GPUs.

the (CPU-)process associated to each GPU. It is needed to distribute the sub-lists. The process number is provided by MPI.

As part of our current parallelization strategy, we store an identical copy of the vector involved in the $\mathcal{H}$-matrix-vector product on each GPU. The $\mathcal{H}$-matrix-vector product is applied individually on each GPU. That is, it is only performed for those admissible and non-admissible blocks that are available on the corresponding processor. To finalize the product, we use a global parallel reduction (summation) provided by MPI. For improved performance, we rely on a CUDA-aware MPI implementation [30], such that we directly pass pointers to GPU memory to the MPI call. Data transfers to and from the network adapter are handled by MPI.

Up to this point, we did not discuss how to partition the task lists into sub-lists. This has a strong impact on load balancing. In our current implementation, we stick to a rather simplistic scheme. It relies on the assumption that the amount of time required for a batched matrix-vector product of sev-

eral dense matrices (or two matrix-vector products of skinny matrices in the ACA case) is proportional to the sum over the number of matrix entries of all matrices that are involved in the batched product. Concerning the dense matrix-vector product task list, we thus balance the storage size of the batched matrices on each GPU. Similarly, we balance the storage size of the batched low-rank factors for the ACA task list, compare Figure 3.

Scalability and load balancing discussion. Independently computing the block cluster tree on each GPU and further storing identical copies of the vector involved in the matrix-vector product requires, with growing problem size, a growing fixed amount of memory. This has a potential impact on the problem size scale-up on GPUs with a small amount of GPU memory. We could partially fix this obstruction by moving the block cluster tree construction to CPU. Nevertheless, our intention is to provide a solely GPU-based implementation. Therefore, we do not use the CPU.

Note, also that the currently required global communication in the $\mathcal{H}$-matrix-vector product might limit the speed-up in the matrix-vector product. We accept this, since we right-now focus on calculation speed-up in the $\mathcal{H}$-matrix construction, as discussed before. Finally, the underlying assumption for our task list partitioning strategy strongly depends on the applied batched matrix-vector product implementation. Here, we see some room for improvements by a more elaborated cost model.

4 Numerical results

In the following, we will first briefly discuss the model problem and the applied GPU-based model BEM solver. This is followed by an overview of the used hard- and software and the definition of two test cases. The first major study of this section is concerned with numerical results that indicate convergence of the implemented method. The remaining part of this section discusses the performance and scalability of the multi-GPU approach.

4.1 Model BEM solver

To test our extended version of the `hmglib` library in the context of boundary element methods, we have implemented a GPU-based model BEM solver. It solves the model problem discussed in Example 2.1, i.e. the Laplace problem reformulated by the single-layer potential ansatz and resulting in the boundary integral equation (3). We stress here that this BEM solver is solely built with the intention to have a test case for the `hmglib` library in the context of BEM. That is, it is not supposed to compete with other BEM libraries.

Our model GPU BEM solver is based on a sequential in-house code. This in-house code gets the boundary Γ by a parametric representation similar to iso-geometric analysis. It discretizes Γ by a quadrangular mesh and

introduces a finite-dimensional trial space with piecewise constant ansatz functions. We identify basis functions with element centers. In the Galerkin matrix assembly, higher-order quadrature and the *Duffy trick* [15, 42] are applied to get an accurate approximation of the integrals. The resulting system of linear equations is solved by a conjugate gradient (CG) solver.

In our GPU version of the CPU code, we re-use the existing sequential CPU code to build the required data structures (mesh, element lists, ...). This data is copied to GPU. Then, the actual matrix assembly is done on GPU. To this end, we parallelize the fully decoupled node-wise assembly operation by appropriate CUDA device functions that overwrite the abstract matrix assembly class of `hmglib`. As e.g. discussed in [8], the complicated, memory-intensive and node-wise sequential quadrature routines easily lead to a limited GPU *utilization*. In fact, we had to limit the size of the so-called *thread blocks*, i.e. the number of threads executed on a symmetric multiprocessor of a GPU, to 128 in order to be able to run the quadrature routines. Typical choices for most other applications are 512 or 1024. The hand-written GPU-based CG solver uses the multi-GPU parallel $\mathcal{H}$-matrix-vector product. The solution of the iterative solver is copied back to CPU, where its error is evaluated by the standard error evaluation routines of the sequential CPU code.

All results of this work are calculated with the $\mathcal{H}$-matrix parameters $\eta = 1.0$ and $C_{leaf} = 32$. The stopping criterion of the CG solver is a relative residual of 10^{-8}.

4.2 Hardware and software setup

To run and benchmark `hmglib` together with the model GPU BEM solver, we use the former Top 1 HPC system *Titan* (27 Peta-FLOPS), located at the Oak Ridge National Lab, US. This is a Cray XK7 cluster equipped with 18688 compute nodes, which are connected by a Gemini interconnect. Each compute node contains a 16-core AMD Opteron processor, 32 GB of (CPU) memory and a Nvidia Tesla K20X GPU (Kepler architecture) with 6 GB of GPU memory.

We use Titan's default `gcc` compiler, Cray's `MPICH` implementation and Titan's (at time of benchmarking) latest `CUDA Toolkit` 7.0 (including the corresponding `Thrust` and `CUBLAS` libraries). In addition, we compile and use `Magma` 2.3.0 and `OpenBLAS` 0.2.20 (as dependency of `Magma`). In all compilations, we use the standard optimization flag `-O3`. All GPU codes are compiled for the best possible *Compute Architecture 3.5*. The version of `hmglib` that is used to create the results in this work is identical to the commit `8b4a4ff` of `hmglib` on Github [49].

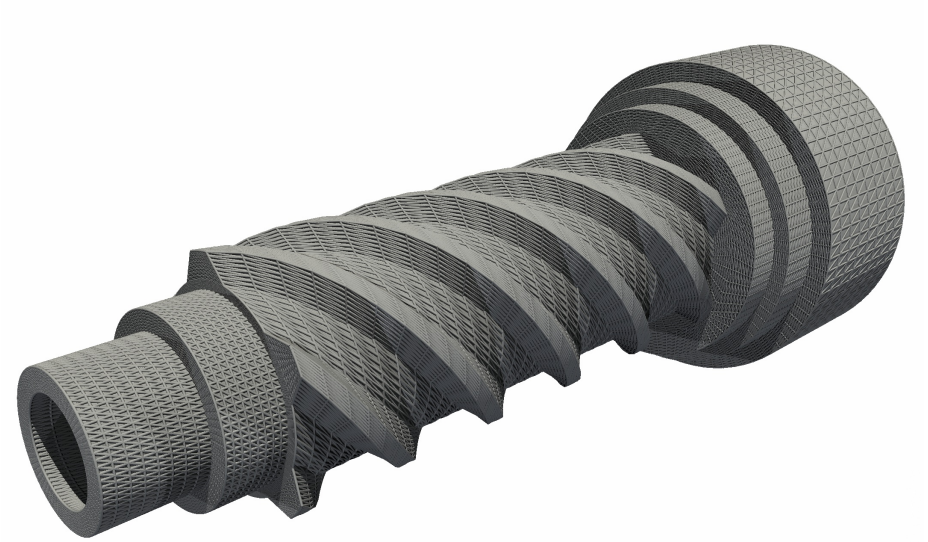

Figure 4: As real-world test case, we solve a boundary integral equation on a complex gearwheel geometry. The overlying mesh corresponds to a discretization with $N = 296960$ boundary elements.

4.3 Test cases

To test our implementation, we first use the model geometry $\Omega := [0, 1]^3$, i.e. Γ is the surface of the unit cube. As real-world test case, we further consider the geometry of a gearwheel with the bounding box $[-4, 4] \times [-4, 4] \times [-11.5, 9.1]$, as shown in Figure 4. In both cases, the right-hand side f in equation (3) is

$$f(\boldsymbol{x}) := 4{x_1}^2 - 3{x_2}^2 - {x_3}^2 \,.$$

Since the right-hand side is the trace of a harmonic function, this choice allows us to compare the numerical solution against the exact solution U_{exact} of the underlying Laplace equation. In particular, we compute the worst-case error

$$\epsilon(h) := \max_{\boldsymbol{x} \in X_{eval}} \left| U_{exact}(\boldsymbol{x}) - \tilde{\mathcal{S}} u_h(\boldsymbol{x}) \right|$$

by evaluating the single-layer potential ansatz from equation (2) for the approximated solution u_h. Here, X_{eval} is a large set of fixed points in the interior of the domain Ω.

4.4 Convergence

In order to verify the correctness of our GPU BEM model implementation and of the distributed-memory parallel $\mathcal{H}$-matrix implementation, we first do a classical convergence study, both with respect to the discretization, i.e. the number of boundary elements, and with respect to the approximation by the $\mathcal{H}$-matrix approach.

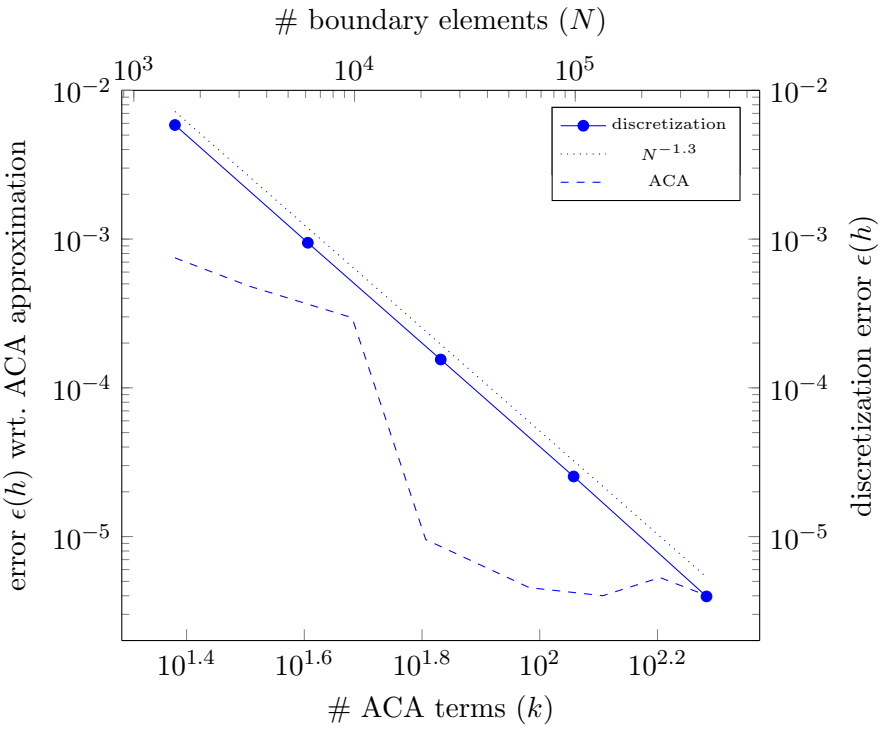

Figure 5: Convergence study for the multi-GPU implementation with model problem on a cube geometry. The discretization error *(solid line)* decays with an appropriate algebraic rate while the error in the ACA *(dashed line)* converges up to exponentially until it hits the discretization error.

Cube geometry. We solve the above discussed model problem on the surface of the cube geometry for a growing number N of boundary elements. In this first experiment, we fix the approximation by the adaptive cross approximation to $k = 128$. The convergence study is done for $N = 1536, 6144, 24576, 98304, 393216$. Note that we compute the first two results on 4 GPUs and the remaining results on 128 GPUs. Although some of the problems would already fit in less GPUs (or even one GPU), we keep a higher number of GPUs to actually check the convergence of the *multi*-GPU code.

Figure 5 shows the convergence results for this first test by the solid blue line. The corresponding axes for this test are on the top and on the right-hand side. We observe an algebraic error decay with a (measured) rate of 1.3. If Γ would be smooth, we could get a rate of 1.5. However, since this is not the case, the observed rate perfectly fits our expectations.

We further check the convergence of the adaptive cross approximation. To this end, we fix the number of boundary elements to $N = 393216$ and gradually increase the number of terms used in the ACA as $k = 24, 32, 48, 64,$

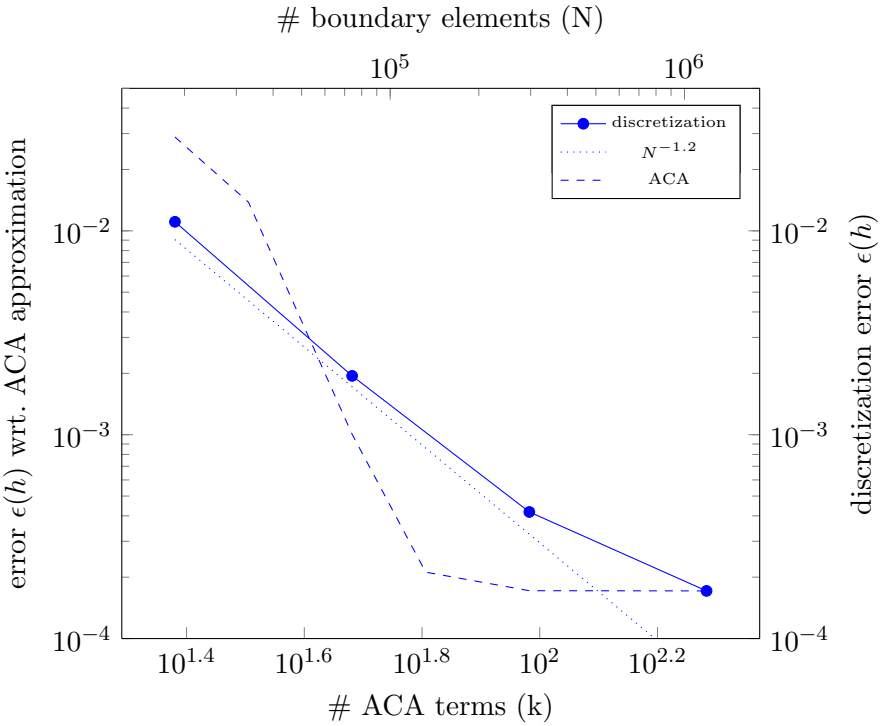

Figure 6: Our multi-GPU parallel BEM solver together with the `hmglib` library is also applied to the very complex gearwheel geometry. Here it shows a similar discretization error *(solid line)* and ACA approximation error *(dashed line)* as for the unit cube geometry.

96, 128, 160, 192. In this second test, we always use 128 GPUs. The results are depicted in Figure 5 as the dashed line. The error in the ACA decays up to exponentially until it hits the discretization error for roughly $k = 128$. Beyond that, it stagnates with small fluctuations.

Gearwheel geometry. Next, we repeat our previous studies with the complex real-world geometry of the gearwheel seen in Figure 4. We again fix the low-rank approximation to $k = 128$ and increase the number of boundary elements in accordance with $N = 18560, 74240, 296960, 1187840$. The first two problem sizes are computed on 256 GPUs and the second two problem sizes are computed on 1024 GPUs.

The results are given in Figure 6 by the solid line. It shows a measured algebraic rate of roughly 1.2, which fits again our expectations since the gearwheel geometry is not smooth. For the largest problem size, we get a slight degradation in this rate. This might be due to our fixed stopping criterion of a relative residual of 10^{-8} in the CG solver and a potentially high condition number of the corresponding Galerkin system matrix.

We also repeat the convergence study for ACA for fixed $N = 1187840$ and growing $k = 24, 32, 48, 64, 96, 128, 160, 192$. As before, we observe an up to exponential convergence until the discretization error is hit. Beyond that, there are again only small variations on the same error level.

To summarize, our distributed-memory multi-GPU parallel model BEM code applied together with `hmglib` perfectly matches our convergence expectations. This holds for the model geometry of a unit cube and for the very complex gearwheel geometry.

4.5 Performance and scalability

To assess the properties of our implementation, we performed a series of benchmarking and scalability studies. All studies were carried out on *Titan*, cf. Section 4.2. Time measurements in our distributed-memory multi-GPU parallelization are wall clock times for the slowest parallel process, i.e. the slowest GPU. If not otherwise stated, these worst-case times were averaged over five runs to reduce the impact of changing loads on the utilized HPC system. Timings for all measurements are collected in Table 1.

Scale-up in problem size. We first discuss the scale-up in terms of problem size, exemplified for the cube geometry. The corresponding timings are given in the first block of Table 1. We are able to solve a problem with up to 24576 boundary elements and $k = 24$ on a single GPU. In this case, the setup time of the hierarchical matrix consumes about 40 seconds. A single iteration of the CG solver requires in average 0.035 seconds with a total of about 100 iterations.

To further increase the problem size, we increased the number of GPUs by a distributed-memory parallelization, cf. Table 1. Using our task-based parallelization, cf. Section 3.4, we are able to treat the cube geometry test case with about 1.5 million boundary elements on 128 GPUs for $k = 48$. The $\mathcal{H}$-matrix setup phase consumes about 31 minutes. Once the setup has been computed, the system of linear equations can be solved with only 0.19 seconds per iteration and a total runtime of about 75 seconds. That is, we do the linear solve with 1.5 million boundary elements in way less than one and a half minutes.

We also tried to further increase the problem size. However, we are not able to accomplish this since the amount of memory consumed on a single GPU (due to our strategy of independently carrying out the data structure setup on all GPUs) becomes too high for the 6 GB of GPU RAM of the Tesla K20X GPUs. The use of a more recent GPU like the Tesla P100 with 16 GB of GPU RAM would of course improve the situation. This GPU would also achieve a much higher performance in the $\mathcal{H}$-matrix setup, since our model BEM code would run much faster than on the rather old Tesla K20X cards. In the future, we also aim at combining a domain-decomposition

				runtime	
geometry	N	k	p	$\mathcal{H}$-setup [s]	CG solver [s/iter.]
cube	1536	24	1	0.86	0.0080
	6144	24	1	5.44	0.0147
	24576	24	1	39.91	0.0350
	98304	48	8	163.15	0.1504
	393216	48	32	698.49	0.0914
	1572864	48	128	1880.26	0.1918
	393216	24	128	46.66	0.0335
		32	128	101.01	0.0284
		48	128	245.35	0.0288
		64	128	342.47	0.0333
		128	128	518.79	0.0342
		160	128	555.77	0.0393
	1572864	48	128	1832.61	0.1124
		48	256	955.92	0.0858
		48	512	543.12	0.0767
		48	1024	338.80	0.0867
gearwheel	1187840	24	1024	497.41	0.0642
		32	1024	509.11	0.0585
		48	1024	534.10	0.0583
		64	1024	684.75	0.0600
		128	1024	864.29	0.0643
		160	1024	877.60	0.0783
	1187840	24	128	2726.23	0.0969
		24	256	1486.17	0.0827
		24	512	937.07	0.0633
		24	1024	497.90	0.0616

Table 1: The above table collects runtime benchmarks done with `hmglib` and our model GPU BEM solver on p GPUs. We are able to e.g. solve a BEM problem on a cube geometry with about 1.5 million boundary elements on GPUs in less than 6 minutes (5.7 minutes for the $\mathcal{H}$-matrix setup, 20 seconds for the CG solver).

parallelization with the task-based parallelization as for example in [3, 31] to solve even much larger problem sizes.

Performance vs. accuracy. As discussed in Section 4.4, the increase in the number of terms utilized in the adaptive cross approximation has an important impact on the accuracy of the approximate solution. Therefore, we analyzed its impact on the performance for the cube geometry and the gearwheel geometry test case. In the first test case, we fix the problem size to about 400000 boundary elements and increase the number of terms k from 24 to up to 160. Note that the timings used in this paragraph correspond to a single test run instead of five test runs. Table 1 collects the runtime results of this test case in the second row block. From a theoretical point of view, we should expect a quadratic increase in the $\mathcal{H}$-matrix setup runtime with respect to k. In practice, this increase is visible in the $\mathcal{H}$-matrix setup (including the ACA approximation of the admissible blocks) for smaller k. However, for larger k, this increase becomes smaller. We assume that this behavior is due to the batching of the linear algebra operations involved in ACA. In fact, the larger the problem, the better it is possible to hide GPU latencies. We observe a much smaller runtime increase in the CG solver.

We tried the same experiment for the real-world gearwheel geometry test case with a problem size of about 1.2 million boundary elements with results given in the fourth row block of Table 1. Here, no quadratic runtime increase is visible. Instead, runtime is increased sub-linearly. In this test case, the runtime for the dense treatment of non-admissible matrix blocks seems to dominate the runtime. To summarize, an increase in the number of ACA terms has only a rather small impact on the overall runtime of our implementation.

Speed-up efficiency One of the main goals of this work is the increase in performance for the *$\mathcal{H}$-matrix setup* by the use of more GPUs. To showcase the achieved efficiency, we perform a parallel speed-up / strong scaling study for the cube and the gearwheel geometry. The results for the cube geometry are given in Figure 7 and in the third row block of Table 1.

In Figure 7, we show the results of a parallel speed-up analysis for problem sizes $N = 393216$ and $N = 1572864$ with $k = 48$. While the smaller problem size does not scale on large GPU counts, we observe a decent result for the larger test case. Here, a relative parallel speed-up efficiency of more than 67 percent is achieved starting from 128 GPUs and going to up to 1024 GPUs. The effective runtime of the $\mathcal{H}$-matrix setup on 1024 GPUs is less than 5.7 minutes, while the solution time by the CG solver is less than 20 seconds. In total, we therefore need on 1024 GPUs less than 6 minutes for the $\mathcal{H}$-matrix setup and solve for a problem size of more than 1.5 million boundary elements.

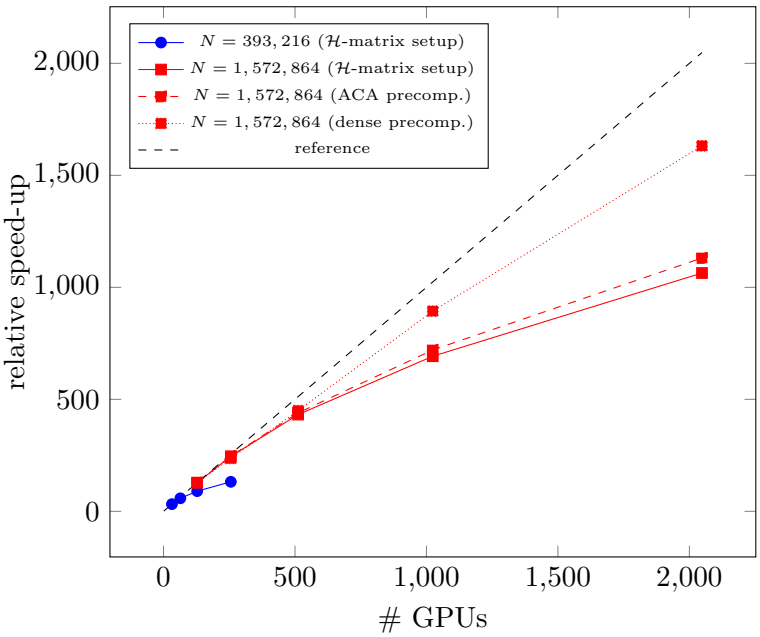

Figure 7: The parallel speed-up efficiency for $\mathcal{H}$-matrix setup of the cube geometry test case *(solid red line)* is above 67 percent on up to 1024 GPU (starting form 128 GPUs) and still reaches above 50 percent on 2048 GPUs. Focussing on the pre-computation of (dense) non-admissible blocks, we even achieve almost 80 percent parallel speed-up efficiency.

When trying to further speed-up the problem, we still achieve an acceptable speed-up efficiency of above 50 percent on 2048 GPUs. In addition, we looked more closely in the contribution to the scalability of the different work loads in the $\mathcal{H}$-matrix setup. As shown in Figure 7, the pre-computation of the (dense) non-admissible blocks scales much better than the approximation of the admissible blocks by ACA. From Table 1, we can depict that the strong scaling efficiency of the dense computations is in the range of 80 percent, even on 2048 GPUs. The scaling of the ACA work load is only a little bit more efficient than the overall $\mathcal{H}$-matrix setup phase. The overall setup phase is always less efficiencient than the dense and ACA work loads, since the GPU-parallel data structure setup, i.e. the block cluster tree traversal, is not parallelized in a distributed-memory manner, compare Section 3.4.

In Section 3.4, we also discussed the issue of load balancing. Effectively, we chose the model assumption that a roughly equal amount of batched matrix entries that are applied in a batched matrix-vector product will result in similar runtime performance. We check the quality of this assumption by examining the distribution of computational runtime over all GPUs for the dense matrix blocks and the ACA matrix approximations. The results of this study for $N = 1572864$, $k = 48$ and $p = 1024, 2048$ are shown in Figure 8

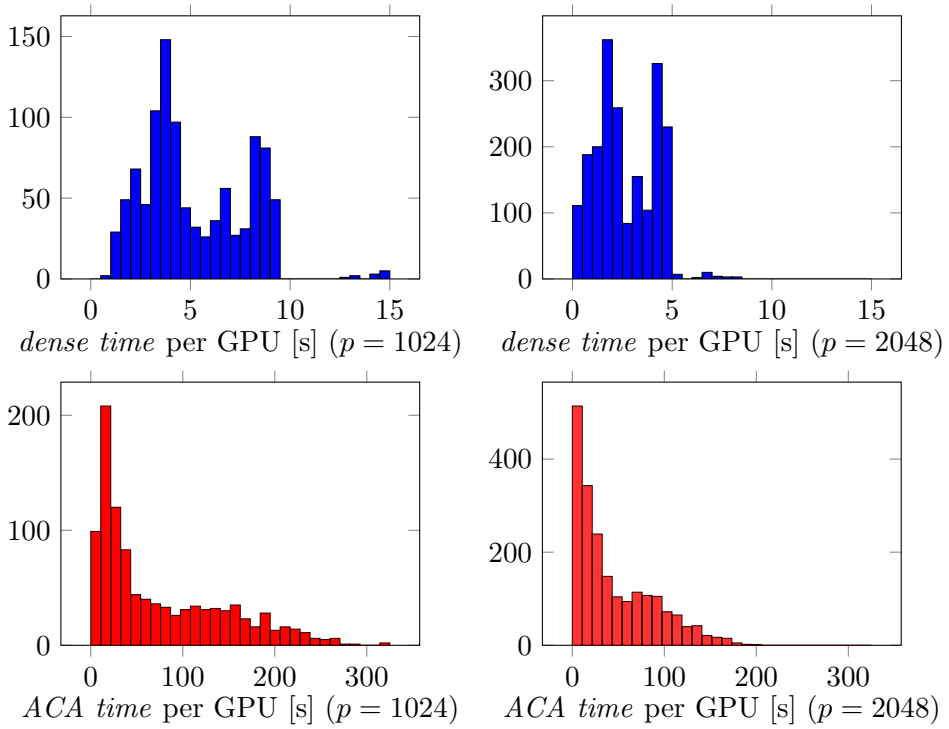

Figure 8: We analyze the load balancing in the speed-up study of the $\mathcal{H}$-matrix setup (cube geometry test case, $N = 1572864$, $k = 48$) with a special focus on the per-GPU runtimes of the pre-computation of the non-admissible blocks *(blue)* and the per-GPU runtimes of the approximation of the admissible blocks *(red)*.

as histogram plots. Qualitatively, the changes between 1024 and 2048 GPUs are only small, that is we only consider $p = 1024$. In practice, our model assumption does not yet lead to an optimal load balancing. While a major part of the timings for the dense matrix operations (see the blue histogram on the left-hand side in Figure 8) is nicely scattered around 5 seconds, we have some non-optimal outliers of up to 15 seconds. Nevertheless, the dense operations have only a moderate influence on the overall scalability due to their small maximum time. In contrast, the matrix block approximations by ACA (see the red histogram on the left-hand side in Figure 8) last up to more than 300 seconds. We especially observe the rather large portion of GPUs that only need a very small amount of time. In the future, we aim at improving the multi-GPU load balancing by techniques proposed e.g. in [3, 31]. However, while these techniques work well in the context of non-batched operations, we assume that their combination with batching will still be sub-optimal on GPUs. Therefore, further research has to be carried out in order to improve the load balancing.

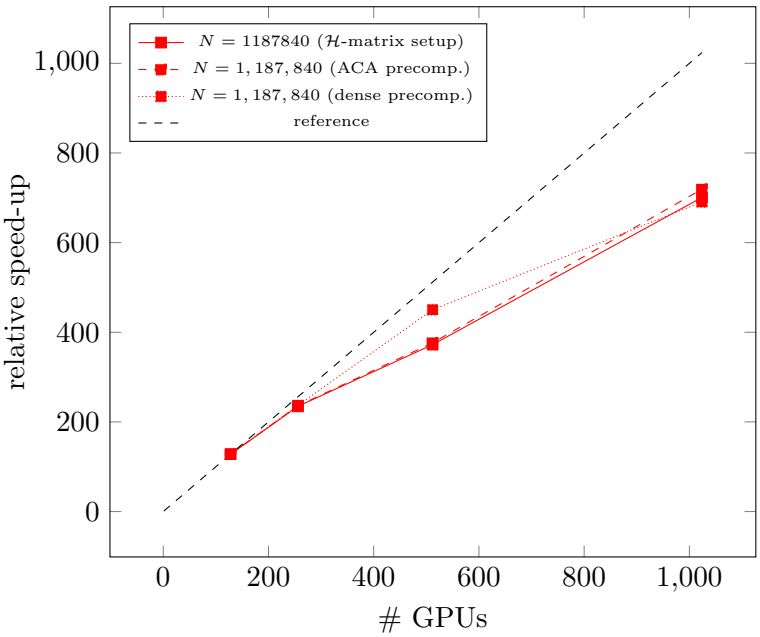

Figure 9: Considering the real-world test case of a gearwheel geometry with $N = 1,187,804$ and $k = 24$, we observe a relative parallel speed-up efficiency of above 68 percent going from 128 to 1024 GPUs. This result is almost identical to the much simpler cube geometry test case.

We finally repeat the parallel speed-up efficiency study for the gearwheel geometry test case with $N = 1187840$ and $k = 24$. The speed-up results are graphically displayed in Figure 9. In this test case, the speed-up for the dense matrix work load and the ACA approximation work load are well aligned to the overall $\mathcal{H}$-matrix setup speed-ups. In total, we achieve above 68 percent of parallel speed-up efficiency going from 128 GPUs to 1024 GPUs. This is a decent result. Moreover, as shown by Table 1 in the last row block, the runtime for the $\mathcal{H}$-matrix setup on 1024 is about 8.3 minutes, while the per-iteration runtime of the CG solver is about 0.06 seconds with a runtime of less than 29 seconds for the full CG solve. That is, a total of 8.8 minutes is needed to solve a BEM problem with 1187840 boundary elements on a real-world geometry on 1024 (rather old) GPUs, recalling that the implemented GPU BEM solver is only intended to be a model application for the underlying `hmglib` library.

5 Conclusions

In this work, we considered the distributed-memory parallel multi-GPU parallel solution of boundary integral equations by the hierarchical matrix li-

brary `hmglib`. The main contribution of this work was the extension and distributed-memory parallelization of `hmglib`, such that Galerkin matrices from boundary element method discretizations given by arbitrary BEM codes can be approximated in a multi-GPU parallel way. Our multi-GPU parallelization of the $\mathcal{H}$-matrix library uses a task-based parallelization. Numerical studies and performance analysis were carried out with a model GPU BEM solver for piecewise constant boundary elements, which is based on an existing in-house CPU solver. This model GPU BEM solver was merely designed for a test of the `hmglib` library, however not with the intention to compete with other BEM solvers in the field. Our two numerical test cases showed, both, roughly 67 percent relative parallel speed-up efficiency going from 128 to 1024 GPUs on the GPU cluster *Titan*. This is a decent speed-up result. A cube geometry test case with about 1.5 million boundary elements could be solved within less than 6 minutes (5.7 minutes for the setup, 20 seconds for the CG solver) on 1024 GPUs. The real-world gearwheel geometry test case with about 1.2 million unknowns could be solved within 8.8 minutes on 1024 GPUs.

As future work, we consider the combination of a domain decomposition parallelization with our task-based parallelization in order to scale to much larger problem sizes. Such an approach should also allow to get a more pronounced speed-up in the CG solver, for which we currently did not aim for. Moreover, we plan to further investigate load balancing techniques, which are assumed to have a strong impact on pre-asymptotic runtimes.

Acknowledgements

This work is funded by the Swiss National Science Foundation (SNF) under project number 407540_167186. Furthermore, code development and benchmarking tasks in this research were done on resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. All funding and support is gratefully acknowledged.

References

[1] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, Ali Charara, and David E. Keyes. Exploiting data sparsity for large-scale matrix computations. Technical report, King Abdullah University of Science and Technology, 2018.

[2] Mario Bebendorf. `AHMED` Another software library on hierarchical matrices for elliptic differential equations. Online. https://github.com/xantares/ahmed (last access: 2018/06/18).

[3] Mario Bebendorf. *Hierarchical Matrices. A Means to Efficiently Solve Elliptic Boundary Value Problems*, volume 63 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008.

[4] Mario Bebendorf and Stefan Kunis. Recompression techniques for adaptive cross approximation. *Journal of Integral Equations and Applications*, 21(3):331–357, 2009.

[5] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2003.

[6] Steffen Börm. $\mathcal{H}^2$-matrices. Multilevel methods for the approximation of integral operators. *Computing and Visualization in Science*, 7(3):173–181, Oct 2004.

[7] Steffen Börm. `H2Lib`, a library for hierarchical matrices. Online, 2017. http://www.h2lib.org (last access: 2018/06/18).

[8] Steffen Börm and Sven Christophersen. Approximation of BEM matrices using gpgpus. *CoRR*, abs/1510.07244, 2015.

[9] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements*, 27(5):405–422, 2003.

[10] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief, and David E. Keyes. Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Computing*, 74:19–33, 2018. Parallel Matrix Algorithms and Applications (PMAA'16).

[11] Ali Charara, David E. Keyes, and Hatem Ltaief. Batched triangular dense linear algebra kernels for very small matrix sizes on gpus. Technical report, King Abdullah University of Science and Technology, 2017.

[12] Ali Charara, David E. Keyes, and Hatem Ltaief. Batched tile low-rank GEMM on GPUs. Technical report, King Abdullah University of Science and Technology, 2018.

[13] Jürgen Dölz, Helmut Harbrecht, Stefan Kurz, Sebastian Schöps, and Fexlix Wolf. A fast isogeometric BEM for the three dimensional Laplace- and Helmholtz problems. *Computer Methods in Applied Mechanics and Engineering*, 330:83–101, 2018.

[14] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with GPUs. *Numerical Computations with GPUs*, pages 1–26, 2014.

[15] Michael G. Duffy. Quadrature over a pyramid or cube of integrands with a singularity at a vertex. *SIAM Journal on Numerical Analysis*, 19(6):1260–1262, 1982.

[16] Lothar Gaul, Martin Kögler, and Marcus Wagner. *Boundary Element Methods for Engineers and Scientists*. Springer, Berlin, 2003.

[17] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):358–384, 2016.

[18] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Parallel black box-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11(4):273–291, 2008.

[19] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica*, 6:229–269, 1997.

[20] Wolfgang Hackbusch. *Hierarchical matrices: Algorithms and Analysis*, volume 49 of *Springer series in computational mathematics*. Springer, Berlin, 2015.

[21] Wolfgang Hackbusch. Survey on the technique of hierarchical matrices. *Vietnam Journal of Mathematics*, 44(1):71–101, 2016.

[22] Wolfgang Hackbusch and Steffen Börm. $\mathcal{H}^2$-matrix approximation of integral operators by interpolation. *Applied Numerical Mathematics*, 43(1-2):129–143, 2002.

[23] Wolfgang Hackbusch, Boris Khoromskij, and Stefan A Sauter. On $\mathcal{H}^2$-matrices. In *Lectures on Applied Mathematics: Proceedings of the Symposium Organized by the Sonderforschungsbereich 438 on the Occasion of Karl-Heinz Hoffmanns 60th Birthday, Munich, June 30–July 1, 1999*, pages 9–29, Berlin-Heidelberg, 2000. Springer.

[24] Wolfgang Hackbusch and Zenon Paul Nowak. On the fast matrix multiplication in the boundary element method by panel clustering. *Numerische Mathematik*, 54(4):463–491, 1989.

[25] Azzam Haidar, Tingxing Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. Framework for batched and GPU-resident factorization algorithms to block Householder transformations. In *ISC High Performance 2015*, pages 31–47, Cham, 2015. Springer International Publishing.

[26] Tom R. Halfhill. Parallel Processing with CUDA. *Microprocessor Report*, January 2008.

[27] Helmut Harbrecht and Maharavo Randrianarivony. From computer aided design to wavelet BEM. *Computing and Visualization in Science*, 13(2):69–82, 2010.

[28] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[29] George C. Hsiao and Wolfgang L. Wendland. *Boundary Integral Equations*, volume 164 of *Applied Mathematical Sciences*. Springer, Berlin, 2008.

[30] Jiri Kraus. An introduction to CUDA-aware MPI. Online, 2013. NVIDIA Developer Blog.

[31] Michal Kravcenko, Lukas Maly, Michal Merta, and Jan Zapletal. Parallel assembly of ACA BEM matrices on Xeon Phi clusters. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 101–110, Cham, 2018. Springer International Publishing.

[32] Ronald Kriemann. Parallel $\mathcal{H}$-matrix arithmetics on shared memory systems. *Computing*, 74(3):273–297, 2005.

[33] Ronald Kriemann. $\mathcal{H}$-LU factorization on many-core systems. *Computing and Visualization in Science*, 16(3):105–117, 2013.

[34] Ronald Kriemann. $\mathcal{H}$-Lib$^{\text{pro}}$. Online, 2017. http://www.hlibpro.com (last access: 2018/06/18).

[35] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[36] Benjamin Marussig, Jürgen Zechner, Gernot Beer, and Thomas-Peter Fries. Fast isogeometric boundary element method based on independent field approximation. *Computer Methods in Applied Mechanics and Engineering*, 284:458–488, 2015.

[37] Michal Merta and Jan Zapletal. A parallel library for boundary element discretization of engineering problems. *Mathematics and Computers in Simulation*, 145:106–113, 2018.

[38] Guy M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, 1966.

[39] Satoshi Ohshima, Ichitaro Yamazaki, Akihiro Ida, and Rio Yokota. Optimization of hierarchical matrix computation on GPU. In Rio Yokota and Weigang Wu, editors, *Supercomputing Frontiers*, pages 274–292, Cham, 2018. Springer International Publishing.

[40] Jack Poulson. `DMHM` - Distributed-Memory Hierarchical Matrices. Online. https://bitbucket.org/poulson/dmhm (last access: 2018/06/18).

[41] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software*, 42(4):27:1–35, 2016.

[42] Stefan A. Sauter and Christoph Schwab. Quadrature for *hp*-Galerkin BEM in $\mathbb{R}^3$. *Numerische Mathematik*, 78(2):211–258, 1997.

[43] Stefan A. Sauter and Christoph Schwab. *Boundary Element Methods*. Springer Series in Computational Mathematics. Springer, Berlin–Heidelberg, 2011.

[44] Zhifeng Sheng, Patrick Dewilde, and Shivkumar Chandrasekaran. Algorithms to solve hierarchically semi-separable systems. In Daniel Alpay and Victor Vinnikov, editors, *System Theory, the Schur Algorithm and Multidimensional Analysis*, pages 255–294, Basel, 2007. Birkhäuser.

[45] Wojciech Śmigaj, Timo Betcke, Simon Arridge, Joel Phillips, and Martin Schweiger. Solving boundary integral problems with BEM++. *ACM Transactions on Mathematical Software*, 41(2):6:1–40, 2015.

[46] Olaf Steinbach. *Numerical Approximation Methods for Elliptic Boundary Value Problems*. Springer, New York, 2008.

[47] Kerstin Vater, Timo Betcke, and Boris Dilba. Simple and efficient GPU parallelization of existing $\mathcal{H}$-matrix accelerated BEM code. *CoRR*, abs/1711.01897, 2017.

[48] Peter Zaspel. Algorithmic patterns for $\mathcal{H}$-matrices on many-core processors. *CoRR*, abs/1708.09707, 2017.

[49] Peter Zaspel. `hmglib` - Simple H matrix library on GPU. Online, 2018. https://github.com/zaspel/hmglib (last access: 2018/06/18).

LATEST PREPRINTS

No.	Author: *Title*
2017-05	**J. Dölz and H. Harbrecht** *Hierarchical Matrix Approximation for the Uncertainty Quantification of Potentials on Random Domains*
2017-06	**P. Zaspel** *Analysis and parallelization strategies for Ruge-Stüben AMG on many-core processor*
2017-07	**H. Harbrecht and M. Schmidlin** *Multilevel Methods for Uncertainty Quantification of Elliptic PDEs with Random Anisotropic Diffusion*
2017-08	**M. Griebel and H. Harbrecht** *Singular value decomposition versus sparse grids: Refined complexity Estimates*
2017-09	**J. Garcke and I. Kalmykov** *Efficient Higher Order Time Discretization Schemes for Hamilton-Jacobi-Bellman Equations Based on Diagonally Implicit Symplectic Runge-Kutta Methods*
2017-10	**M. J. Grote and U. Nahum** *Adaptive Eigenspace Regularization For Inverse Scattering Problems*
2017-11	**J. Dölz, H. Harbrecht, S. Kurz, S. Schöps and F. Wolf** *A Fast Isogeometric BEM for the Three Dimensional Laplace- and Helmholtz Problems*
2017-12	**P. Zaspel** *Algorithmic patterns for $\mathcal{H}$-matrices on many-core processors*
2017-13	**R. Brügger, R. Croce and H. Harbrecht** *Solving a free boundary problem with non-constant coefficients*
2017-14	**M. Dambrine, H. Harbrecht and B. Puig** *Incorporating knowledge on the measurement noise in electrical impedance tomography*
2017-15	**C. Bürli, H. Harbrecht, P. Odermatt, S. Sayasone and N. Chitnis** *Analysis of Interventions against the Liver Fluke, Opisthorchis viverrini*
2017-16	**D. W. Masser** *Abcological anecdotes*

Preprints are available under *https://math.unibas.ch/research/publications*

LATEST PREPRINTS

No.	Author: *Title*
2017-17	**P. Corvaja, D. W. Masser and U. Zannier** *Torsion hypersurfaces on abelian schemes and Betti coordinates*
2017-18	**F. Caubet, M. Dambrine and H. Harbrecht** *A Newton method for the data completion problem and application to obstacle detection in Electrical Impedance Tomography*
2018-01	**H. Harbrecht and P. Zaspel** *On the algebraic construction of sparse multilevel approximations of elliptic tensor product problems*
2018-02	**F. Ghiraldin and X. Lamy** *Optimal Besov differentiability for entropy solutions of the eikonal equation*
2018-03	**H. Harbrecht and M. Schmidlin** *Multilevel quadrature for elliptic problems on random domains by the coupling of FEM and BEM*
2018-04	**M. Bugeanu and H. Harbrecht** *Parametric representation of molecular surfaces*
2018-05	**A. Abdulle, M. J. Grote and O. Jecker** *Finite element heterogeneous multiscale method for Elastic Waves in Heterogeneous Media*
2018-06	**M. J. Grote and J. H. Tang** *On controllability methods for the Helmholtz equation*
2018-07	**H. Harbrecht and M. Moor** *Wavelet boundary element methods — Adaptivity and goal-oriented error estimation*
2018-08	**P. Balazs and H. Harbrecht** *Frames for the solution of operator equations in Hilbert spaces with fixed dual pairing*
2018-09	**R. Brügger, R. Croce and H. Harbrecht** *Solving a Bernoulli type free boundary problem with random diffusion*
2018-10	**J. Dölz, H. Harbrecht and M. D. Multerer** *On the best approximation of the hierarchical matrix product*
2018-11	**H. Harbrecht and P. Zaspel** *A scalable $\mathcal{H}$-matrix approach for the solution of boundary integral equations on multi-GPU clusters*

Preprints are available under *https://math.unibas.ch/research/publications*