Max-Planck-Institut
für Mathematik
in den Naturwissenschaften
Leipzig

Fast Parallel Solution of Boundary Integral Equations and Related Problems

by

Mario Bebendorf and Ronald Kriemann

Preprint no.: 10 2004

Fast Parallel Solution of Boundary Integral Equations and Related Problems

M. Bebendorf* and R. Kriemann†

March 5, 2004

This article is concerned with the efficient numerical solution of Fredholm integral equations on a parallel computer with shared or distributed memory. Parallel algorithms for both, the approximation of the discrete operator by hierarchical matrices and the parallel matrix-vector multiplication of such matrices by a vector, are presented. The first algorithm has a complexity of order $\frac{1}{p}N\log^{2d-1}N$, while the latter is of order $\frac{1}{p}N\log^d N$, where N and p are the number of unknowns and the number of processors, respectively. The $\mathcal{H}$-approximant needs $\mathcal{O}\left(\frac{1}{p}N\log^d N\right)$ units of storage on each processor.

AMS Subject Classification: 65D05, 65D15, 65F05, 65F30
Keywords: Integral equations, hierarchical matrices, parallel solvers

1 Introduction

This article deals with the efficient solution of Fredholm integral equations

$$\lambda u(y) + \int_\Gamma \kappa(x,y)u(x)\,\mathrm{d}s_x = f(y), \quad y \in \Gamma, \tag{1.1}$$

with given right-hand side f on a parallel computer. The domain of integration $\Gamma \subset \mathbb{R}^d$ is a $(d-1)$-dimensional manifold. This kind of integral equation arises for instance from the boundary element method. The results of this article can however be easily extended to volume integral equations in $\mathbb{R}^d$. In this article the kernel function $\kappa : \Gamma \times \mathbb{R}^d \to \mathbb{R}$ in (1.1) is assumed to be *asymptotically smooth* with respect to y, i.e., $\kappa(x,\cdot) \in C^\infty(\mathbb{R}^d \setminus \{x\})$ for almost all $x \in \Gamma$, and there is a constant $g < 0$ such that for all multiindices $\alpha \in \mathbb{N}_0^d$ it holds that

$$|D_y^\alpha \kappa(x,y)| \le c_1^{\mathrm{as}} r!(c_2^{\mathrm{as}}|x-y|)^{g-r}, \quad r = |\alpha|, \tag{1.2}$$

where c_1 and c_2 are two constants. As usual we denote by D_y^α the partial derivative

$$D_y^\alpha = \left(\frac{\partial}{\partial y_1}\right)^{\alpha_1} \cdots \left(\frac{\partial}{\partial y_d}\right)^{\alpha_d}.$$

*Fakultät für Mathematik und Informatik, University of Leipzig, Augustusplatz 10/11, 04109 Leipzig, Germany, bebendorf@math.uni-leipzig.de
†Max-Planck-Institute for Mathematics in the Sciences, Inselstr. 22–26, 04103 Leipzig, Germany, rok@mis.mpg.de

Strongly singular kernels are not excluded. However, then the integral in (1.1) has to be defined by an appropriate regularisation. E.g., the kernels $\kappa(x,y) = |x-y|^g$, $g < 0$, as well as their partial derivatives are asymptotically smooth. Furthermore, the kernel of the double-layer potential operator for the three-dimensional Laplace equation

$$\kappa(x,y) = -\frac{1}{4\pi}\frac{(x-y,n_x)}{|x-y|^3}$$

is asymptotically smooth with respect to y. Here n_x, $x \in \Gamma$, denotes the outer normal unit vector to the surface Γ at x.

In order to solve equation (1.1) numerically, the domain of integration Γ is divided

$$\Gamma = \bigcup_{i\in\mathcal{I}} \pi_i$$

into triangles $\Pi_h = \{\pi_i : i \in \mathcal{I}\}$, where $\mathcal{I}$ is an index set. Besides the *Galerkin method*, the *collocation method* and the *Nyström method* are commonly used. In the Galerkin and the collocation method the solution u is approximated from a finite dimensional ansatz space V_h, i.e., the approximant $u_h \in V_h$ to u is sought of the form

$$u_h = \sum_{j=1}^{N} u_j \varphi_j,$$

where φ_j, $j \in I := \{1,\dots,N\}$, is a basis of V_h. Especially for complicated domains in three dimensions the number of degrees of freedom N has to be chosen quite large to guarantee a reasonable accuracy of u_h. The supports $X_j := \operatorname{supp}\varphi_j \subset \Gamma$ of the basis functions φ_j are assembled from the sets π_i, i.e., there is $\nu \in \mathbb{N}$ independently of N and for each X_j an index set $\mathcal{I}_j \subset \mathcal{I}$, $\#\mathcal{I}_j \le \nu$, exists, so that

$$X_j = \bigcup_{i\in\mathcal{I}_j} \pi_i.$$

All three methods reduce (1.1) to a linear system of the form

$$(\lambda B + A)x = b, \quad A, B \in \mathbb{R}^{N\times N},\ b \in \mathbb{R}^N, \tag{1.3}$$

where B is a sparse matrix and produces no numerical difficulties. In this article B will hence be neglected. The entries of A in the case of the Galerkin method are

$$a_{ij} = \int_\Gamma \int_\Gamma \varphi_j(x)\kappa(x,y)\varphi_i(y)\,\mathrm{d}s_x\,\mathrm{d}s_y, \tag{1.4}$$

in the case of the collocation method

$$a_{ij} = \int_\Gamma \kappa(x,y_i)\varphi_j(x)\,\mathrm{d}s_x \tag{1.5}$$

with *collocation points* $y_i \in X_i$ and in the case of the Nyström method

$$a_{ij} = \kappa(y_j,y_i) \quad \text{for } i \ne j \quad \text{and} \quad a_{ii} = c_i \tag{1.6}$$

with N pairwise distinct points $y_i \in \Gamma$ and N numbers c_i. Integral equations are just one example of problems which lead to coefficient matrices of one of the types (1.4) through (1.6). The evaluation of coulomb and gravitational potentials of particle systems are further examples.

In any case A is dense and therefore needs $\mathcal{O}(N^2)$ units of storage. Already this gives limitations on the computability of problems even on modern computer systems. Additionally, the usual matrix-vector multiplication requires $\mathcal{O}(N^2)$ arithmetical operations. The latter is of particular importance if a Krylov subspace method is used for the solution of (1.3). A direct method like the LU-decomposition would even lead to $\mathcal{O}(N^3)$ complexity. Hence, the main aim of this article is to reduce the complexity of storing and muliplying A (or an approximant of it) by a vector.

Modern numerical methods for the solution of (1.3) provide an approximation $\tilde{x}$ to the solution vector x in almost linear complexity by solving a perturbed linear system in which A is easier to handle. The accuracy of $\tilde{x}$ is chosen so that the additional error for u is of the same size as the consistency error of the discretisation method. The starting point of these modern numerical methods was the fast multipole method (FMM) published by Rokhlin [19], which concentrated on the fast multiplication of matrices of type (1.6) by a vector. The proposed method was tightly adapted to the single layer potential

$$\kappa(x, y) = \log |x - y|$$

of the Laplacian in two dimensions. In the meanwhile the FMM has been shown to be applicable to many other kernel functions. The panel clustering method proposed by Hackbusch and Nowak [12] was designed for the fast matrix-vector multiplication for collocation matrices (1.5) and general kernels κ which allow a *degenerate* approximation $\tilde{\kappa}$, i.e.,

$$\kappa(x, y) \approx \tilde{\kappa}(x, y) := \sum_{i=1}^{k} u_i(x) v_i(y), \quad x \in D_1,\, y \in D_2, \tag{1.7}$$

on a pair of domains D_1 and D_2 satisfying $\eta \operatorname{dist}(D_1, D_2) > \operatorname{diam} D_2$, where $\eta > 0$ is a parameter and k a small number compared with N. It is not difficult to see that condition (1.2) implies (1.7). Note that for this neither the smoothness of κ with respect to x nor smoothness properties of the surface Γ are necessary. A difficulty of the FMM and the panel clustering method is that functions u_i and v_i in (1.7) have to be known explicitly, and if they are known, they are usually not adapted to the geometry.

If one is looking at (1.7) from an algebraic point of view, this means nothing but a low-rank approximation of appropriate matrix blocks. Then the optimal choice of vectors corresponding to the functions u_i and v_i are scaled major singular vectors. This led to algebraic methods like the mosaic skeleton method [21] and hierarchical ($\mathcal{H}$-) matrices [10, 11]. An interesting observation due to Tyrtyshnikov et al. [7] was the existence of low-rank approximants generated from few of the original matrix entries. If such an approximant can be computed with reasonable effort, this is of great practical importance since existing computer codes can easily be modified, whereas the FMM and panel clustering need a complete recoding of the matrix-vector multiplication including computation of coefficients. Stimulated by this existence result one of the authors published [2, 3] the so-called ACA algorithm for the computation of these low-rank approximants. Although an approximant of A can be generated and stored by this algorithm with almost linear complexity, especially the time for building the approximant is still significant if N is large. Therefore, it may be useful to reduce the execution time by parallelisation.

The aim of this article is to present algorithms to approximate A by a matrix $\tilde{A}$ with $\mathcal{O}(\frac{1}{p} N \log^{2d-1} N)$ complexity, where p is the number of processors used. The structure we are using to approximate A are hierarchical ($\mathcal{H}$-) matrices. In this matrix format $\mathcal{O}(\frac{1}{p} N \log^{d} N)$ units of storage on each processor are sufficient to hold $\tilde{A}$. Basically, $\mathcal{H}$-matrices consist of a hierarchical block partitioning, where each block is a matrix of low rank. In Section 2 a more precise review on $\mathcal{H}$-matrices can be found. Section 3 contains the approximation algorithm

by which on each block a low-rank matrix approximant of the original block can be computed. Since the blocks do not depend on each other, the computation of all low-rank approximants can easily be done in parallel. However, the sizes of the blocks may differ significantly. This problem is addressed by appropriate scheduling algorithms. In Section 4 the parallelisation of the matrix-vector multiplication of an $\mathcal{H}$-matrix by a vector is treated. Here, the scheduling is based on space-filling curves, which provide a means to reduce the computation and communication costs in the case of distributed memory systems. In the last section numerical results are presented demonstrating the efficiency of the proposed algorithms.

2 Hierarchical matrices

This section gives a brief overview over the structure of $\mathcal{H}$-matrices originally introduced by Hackbusch et al. [10, 11]. Let us assume that $M \in \mathbb{R}^{N\times N}$ is of one of the types (1.4) through (1.6). In this section we will show how to generate a partition P of the set of matrix indices $I \times I$ such that on each block $b = \tau \times \sigma$, $\tau, \sigma \subset I$, M can be approximated by a matrix of low-rank, i.e.,

$$M_b \approx UV^T, \quad U \in \mathbb{R}^{\tau\times k}, V \in \mathbb{R}^{\sigma\times k},$$

where k is small compared with $|\tau|$ and $|\sigma|$. Obviously, by M_b we denote the subblock in the intersection of the rows τ and columns σ of M. From (1.7) we already know that in the case of asymptotically smooth kernel functions κ the matrix block M_b can be approximated, if for a given real number $0 < \eta < 1$ it holds that

$$\operatorname{diam} X_\sigma < \eta \operatorname{dist}(X_\tau, X_\sigma). \tag{2.1}$$

A block satisfying the previous condition is called *admissible*. Note that for $\tau = \sigma$, i.e., a block on the diagonal, condition (2.1) cannot be fulfilled. Therefore, in order to guarantee that M_b has low rank, one of the dimensions of M_b has to be small.

2.1 Cluster tree

Our aim is to find a partition P from the set of partitions of $I \times I$ such that most of the blocks are admissible. This set however is too large to be searched for. A possibility to significantly decrease the size of possible partitions is to restrict P to those partitions which consist of blocks $b = \tau \times \sigma$, whose index sets τ and σ stem from a *cluster tree* T_I, i.e., a tree satisfying the following conditions:

(i) I is the root of T_I

(ii) if $\tau \in T_I$ is not a leaf, then τ has sons $\tau_1, \tau_2 \in T_I$, so that $\tau = \tau_1 \cup \tau_2$.

The set of sons of τ is denoted by $S(\tau)$, while $\mathcal{L}(T_I)$ stands for the set of leaves of the tree T_I.

Remark 2.1 *Sometimes the number of sons in the previous definition of a cluster tree is not restricted to two. However, this generalisation has not proved useful in practice.*

A cluster tree is usually generated by recursive subdivision of I so as to minimise the diameter of each part. For practical purposes the recursion should be stopped if a certain cardinality $n_{\min}$ of the clusters is reached, rather than subdividing the clusters until only one index is left. The depth of T_I will be denoted by L, which for reasonable cluster trees one would always expect to be of the order $\log N$. A strategy based on the principle component analysis can be found in [1]. The complexity of building the cluster tree in the case of quasi-uniform grids can be estimated as $\mathcal{O}(N \log N)$.

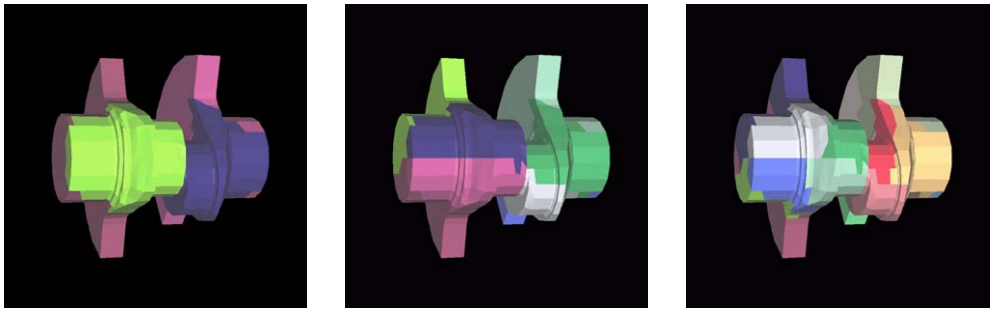

Figure 1: Three subsequent levels of T_I

2.2 Block cluster tree

Based on a cluster tree T_I which contains a hierarchy of partitions of I, we are able to construct the so called *block cluster tree* $T_{I\times I}$ describing a hierarchy of partitions of $I \times I$ by the following algorithm:

procedure build_block_cluster_tree(τ, σ)
  if (τ, σ) is not admissible and $\tau, \sigma \notin \mathcal{L}(T_I)$ **then**
    $S(\tau, \sigma) := \{(\tau', \sigma') : \tau' \in S(\tau),\ \sigma' \in S(\sigma)\}$;
    for $(\tau', \sigma') \in S(\tau, \sigma)$ **do** build_block_cluster_tree(τ', σ');
  else $S(\tau, \sigma) := \emptyset$;
end;

Applying *build_block_cluster_tree* to $I \times I$ we obtain a cluster tree for the index set $I \times I$. The set of leaves $P := \mathcal{L}(T_{I\times I})$ is a partition of $I \times I$ with admissbile blocks $b = \tau \times \sigma \in P$ or blocks made of clusters τ and σ one of which is a leaf in T_I. The complexity of building the block cluster tree in the case of quasi-uniform grids can be estimated as $\mathcal{O}(\eta^{-(d-1)} N \log N)$, cf. [1].

Remark 2.2 *In the case of unstructured grids the computation of the distance in* (2.1) *between two supports X_τ and X_σ is too costly. Therefore, for practical purposes the supports are enclosed into sets of a simpler structure, e.g. boxes or spheres.*

We are now in a position to define the set of $\mathcal{H}$-matrices for a partition P with blockwise rank $k \geq n_{\min}$

$$\mathcal{H}(P, k) := \{M \in \mathbb{R}^{I\times I} : \operatorname{rank} M_b \leq k \text{ for all } b \in P\}.$$

Note that $\mathcal{H}(P, k)$ is not a linear space, since in general the sum of two rank-k matrices exceeds rank k.

Remark 2.3 *For a block $M_b \in \mathbb{R}^{\tau\times\sigma}$ the low-rank representation $M_b = UV^T$, $U \in \mathbb{R}^{\tau\times k}$, $V \in \mathbb{R}^{\sigma\times k}$, is only advantageous compared with the entrywise representation, if $k(|\tau| + |\sigma|) \leq |\tau|\,|\sigma|$. Hence, we will employ the entrywise representation for blocks that do not satisfy the last estimate.*

Since each block $b = \tau \times \sigma$ has the representation $M_b = UV^T$, $U \in \mathbb{R}^{\tau\times k}$, $V \in \mathbb{R}^{\sigma\times k}$, $\mathcal{O}(k(|\tau| + |\sigma|))$ units of memory are needed to store M_b. Exploiting the hierarchical structure of M it can therefore be shown that the overall complexity of storing M is $\mathcal{O}(\eta^{-(d-1)} kN \log N)$, see [1] for a rigorous analysis.

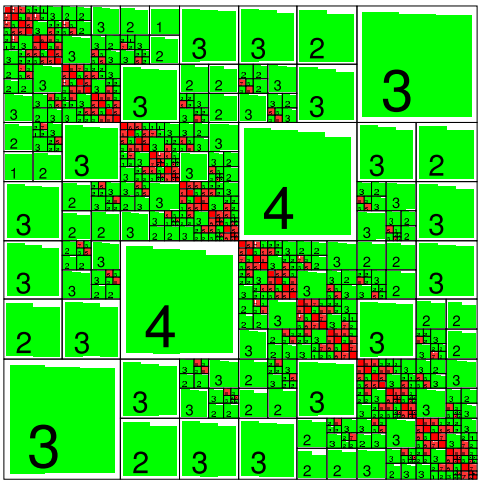

Figure 2: An $\mathcal{H}$-matrix ($N = 17000$) with its rank distribution

3 Building the $\mathcal{H}$-Matrix Approximant

In the first part of this section we present the ACA algorithm which generates a low-rank approximant for each admissbile block of the original matrix. The theory in the case of Nyström matrices (1.6) was treated in [2], collocation matrices (1.5) were addressed in [3]. For applications of this algorithm see for example [14], [15]. In contrast to other fast methods the low-rank approximant is not generated by replacing the kernel function of the integral operator. The algorithm uses few of the original matrix entries to compute the low-rank matrix. Note that it is not necessary to compute the whole original matrix in advance. The proposed algorithm will specify which entries have to be computed. Obviously, this has the advantage that this algorihm can be easily built on top of any existing code, whereas methods like multipole require a complete recoding. In the second part of this section the ACA algorithm is then used on each processor to generate a low-rank approximant of the blocks that are assigned to it.

The idea of the algorithm is as follows. Starting from an initial admissible matrix block $R_0 \in \mathbb{R}^{\tau\times\sigma}$, $\tau, \sigma \subset I$, in the k-th step find a nonzero pivot in R_k, say entry (i_k, j_k), and subtract a scaled outer product of the i_k-th and the j_k-th column:

$$R_{k+1} = R_k - [(R_k)_{i_k j_k}]^{-1}(R_k)_{\tau,j_k}(R_k)_{i_k,\sigma}. \tag{3.1}$$

Example 3.1 *We apply equation* (3.1) *to the following matrix* R_0. *The bold entries are the chosen pivots.*

$$R_0 = \begin{bmatrix} 0.431 & 0.354 & \mathbf{0.582} & 0.417 & 0.455 \\ 0.491 & 0.396 & 0.674 & 0.449 & 0.427 \\ 0.446 & 0.358 & 0.583 & 0.413 & 0.441 \\ 0.380 & 0.328 & 0.557 & 0.372 & 0.349 \\ 0.412 & 0.340 & 0.516 & 0.375 & 0.370 \end{bmatrix} \overset{\substack{i_1=1\\ j_1=3}}{\longrightarrow} \frac{1}{0.582} \begin{bmatrix} 0.582 \\ 0.674 \\ 0.583 \\ 0.557 \\ 0.516 \end{bmatrix} \begin{bmatrix} 0.431 \\ 0.354 \\ 0.582 \\ 0.417 \\ 0.455 \end{bmatrix}^T$$

$$R_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ -0.008 & -0.014 & 0 & -0.033 & \mathbf{-0.100} \\ 0.014 & 0.003 & 0 & -0.004 & -0.014 \\ -0.032 & -0.011 & 0 & -0.026 & -0.087 \\ 0.029 & 0.025 & 0 & 0.005 & -0.034 \end{bmatrix} \overset{\substack{i_2=2\\ j_2=5}}{\longrightarrow} \frac{1}{-0.1} \begin{bmatrix} 0 \\ -0.100 \\ -0.014 \\ -0.087 \\ -0.034 \end{bmatrix} \begin{bmatrix} -0.008 \\ -0.014 \\ 0 \\ -0.033 \\ -0.100 \end{bmatrix}^T$$

$$R_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \mathbf{0.016} & 0.005 & 0 & 0.000 & 0 \\ -0.02 & 0.001 & 0 & 0.002 & 0 \\ 0.032 & 0.030 & 0 & 0.017 & 0 \end{bmatrix} \overset{\substack{i_3=3 \\ j_3=1}}{\longrightarrow} \frac{1}{0.016} \begin{bmatrix} 0 \\ 0 \\ 0.016 \\ -0.02 \\ 0.032 \end{bmatrix} \begin{bmatrix} 0.016 \\ 0.005 \\ 0 \\ 0.000 \\ 0 \end{bmatrix}^T$$

Obviously, the size of the entries decreases from step to step. Since in the k-th step only the entries in the j_k-th column and the i_k-th row of R_k are used to compute R_{k+1}, there is no need to calculate the whole matrix R_k. Taking advantage of this, the following algorithm is an efficient reformulation of (3.1). Note that the vectors u_k and $\tilde{v}_k$ coincide with $(R_{k-1})_{\tau,j_k}$ and $(R_{k-1})_{i_k,\sigma}$, respectively.

Let $k = 1$; $Z = \varnothing$ { *in Z the vanishing rows of R_k are collected* }
repeat
  if $k > 1$ **then** $i_k := \operatorname{argmax}_{i \notin Z} |(u_{k-1})_i|$
  else $i_k := \min \tau \setminus Z$
  $\tilde{v}_k := a_{i_k,\sigma}$
  for $\ell = 1, \ldots, k-1$ **do** $\tilde{v}_k := \tilde{v}_k - (u_\ell)_{i_k} v_\ell$
  $Z := Z \cup \{i_k\}$
  if $\tilde{v}_k$ does not vanish **then**
    $j_k := \operatorname{argmax}_{j=1,\ldots,n} |(\tilde{v}_k)_j|$; $v_k := (\tilde{v}_k)_{j_k}^{-1} \tilde{v}_k$
    $u_k := a_{\tau,j_k}$
    for $\ell = 1, \ldots, k-1$ **do** $u_k := u_k - (v_\ell)_{j_k} u_\ell$.
    $k := k + 1$
  endif
until the stopping criterion (3.3) is fulfilled or $Z = \tau$

Algorithm 3.1: Adaptive Cross Approximation

Obviously, the rank of

$$S_k := \sum_{\ell=1}^{k} u_\ell v_\ell^T$$

is bounded by k. Furthermore, it can be seen that S_k approximates A. For simplicity we assume that for the pivotal indices i_ℓ and j_ℓ it holds that $i_\ell = j_\ell = \ell$, $\ell = 1, \ldots, k$. Then A has the decomposition

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad A_{11} \in \mathbb{R}^{k \times k}, \tag{3.2}$$

where only the matrix blocks A_{11}, A_{12} and A_{21} have been used by Algorithm 3.1. Note that the (large) block $A_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$ has never been touched. Since the determinant of A_{11} is the product of the pivots, A_{11} is invertible and we can express the remainder R_k of the approximation in terms of the original matrix A. For a proof of the following lemma see [3].

Lemma 3.2 *For R_k it holds that*

$$R_k = A - \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} A_{11}^{-1} \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & C_k \end{bmatrix},$$

where $C_k := A_{22} - A_{21} A_{11}^{-1} A_{12}$ is the Schur complement of A_{11} in A.

From Algorithm 3.1 it can be seen that the number of zero rows in R_k may be larger than k, i.e. $k' := |Z| > k$. This happens if in the ℓ-th step the algorithm comes across a zero vector $\tilde{v}_\ell$ and has to continue with another row. It is remarkable that not k but k' will determine the

accuracy of the approximation, as can be seen from the following theorem, which is proved in [4] and estimates the norm of the Schur complement. Similar estimates for the Nyström- and collocation matrices (1.6) and (1.5) may be proven with the same arguments, see [2, 3].

Let $q \in \mathbb{N}$ and $n_q = \sum_{j=0}^{q} \binom{d+j-1}{j}$ be the dimension of the space of polynomials in d variables of degree at most q.

Theorem 3.3 *Let (X_τ, X_σ) fulfil condition (2.1) and let κ be an asymptotically smooth kernel. In the case of Galerkin matrices*

$$a_{ij} = \int_\Gamma \int_\Gamma \varphi_j(x)\kappa(x,y)\varphi_i(y)\,\mathrm{d}s_x\,\mathrm{d}s_y, \quad i \in \tau,\ j \in \sigma$$

it holds that

$$|(C_{n_q})_{ij}| \leq c\,\mathrm{dist}^g(X_\tau, X_\sigma)\,\mathrm{area}(X_\tau)\,\mathrm{area}(X_\sigma)\,\eta^q, \quad 0 < \eta < \frac{1}{4\sqrt{d}}.$$

The constant c in the last estimate contains a growth factor as it appears in the column-pivoted LU-decomposition, which theoretically behaves like 2^{n_q}. In practice this behaviour is rarely observed and we may assume that c is bounded independently of q.

Rather than prescribing the number of approximation steps n_q a priori, the iteration should be stopped if a given accuracy $\varepsilon > 0$ has been reached. Since we want to avoid computing the entries of R_{n_q}, we have to find a condition on q implying $\|R_{n_q}\|_F \leq \varepsilon \|A\|_F$, where $\|\cdot\|_F$ denotes the Frobenius norm. Assume that $\|R_{n_{q+1}}\|_F \leq \eta\,\|R_{n_q}\|_F$, with η from (2.1), then

$$\|\sum_{\ell=n_q+1}^{n_{q+1}} u_\ell v_\ell^T\|_F \leq \frac{\varepsilon(1-\eta)}{1+\varepsilon}\,\|S_{n_q}\|_F \tag{3.3}$$

can be used as a stopping criterion. This can be seen from

$$\|R_{n_q}\|_F \leq \|R_{n_{q+1}}\|_F + \|\sum_{\ell=n_q+1}^{n_{q+1}} u_\ell v_\ell^T\|_F \leq \eta\,\|R_{n_q}\|_F + \|\sum_{\ell=n_q+1}^{n_{q+1}} u_\ell v_\ell^T\|_F.$$

and

$$\|R_{n_q}\|_F \leq \frac{1}{1-\eta}\,\|\sum_{\ell=n_q+1}^{n_{q+1}} u_\ell v_\ell^T\|_F \leq \frac{\varepsilon}{1+\varepsilon}\,\|S_{n_q}\|_F \leq \frac{\varepsilon}{1+\varepsilon}\,(\|A\|_F + \|R_{n_q}\|_F).$$

The Frobenius norm of S_{n_q} an be computed with $\mathcal{O}(n_q(|\tau| + |\sigma|))$ operations. Therefore, the amount of numerical work required by Algorithm 3.1 is of the order $n_q^2(|\tau| + |\sigma|)$.

Remark 3.4 *If in practise the costs for generating the matrix entries dominate the algebraic transformations of Algorithm 3.1 the complexity behaves like $n_q(|\tau| + |\sigma|)$.*

The computation of the low-rank approximants can easily be computed in parallel since there is no dependence between the blocks. The only problem that has to be cared for are the different computational costs of the blocks. In contrast to building the $\mathcal{H}$-matrix approximant, a parallelisation of the $\mathcal{H}$-matrix-vector multiplication is not as obvious.

3.1 Parallel Computation of the $\mathcal{H}$-Matrix Approximant

In this section the parallel computation of the $\mathcal{H}$-matrix approximant is discussed. Algorithms for shared and for distributed memory are introduced which show an optimal *parallel speedup*. The basis for these parallel algorithms is the following sequential algorithm for the approximation of $M \in \mathbb{R}^{N\times N}$.

```
for all b ∈ P do
    if b is admissible then
        create a low-rank approximant of M_b using ACA;
    else
        create the dense matrix M_b;
endfor;
```

The result is either a rank-k matrix if the block satisfies (2.1) or a dense matrix if it does not satisfy the admissibility criterion. In both cases the computation on each block is fully independent from the others and can therefore be easily done in parallel.

Definition 3.5 *Let $t(p)$ denote the time needed by a parallel algorithm $\mathcal{A}$ with p processors. Then $S(p) := \frac{t(1)}{t(p)}$ denotes the* parallel speedup *and $E(p) := \frac{S(p)}{p} = \frac{t(1)}{p\cdot t(p)}$ the* parallel efficiency *of $\mathcal{A}$.*

To achieve a good parallel speedup, usually prior knowledge of the amount of work per matrix block is needed for balancing the load among the processors. Unfortunately this information is only accessible if the low-rank matrix blocks are computed with a fixed rank. If however the stopping criterion (3.3), which guarantees a prescribed accuracy, is used during the ACA algorithm, the rank is chosen adaptively and may hence vary from block to block. Therefore, the actual costs per matrix block are unknown.

An alternative to a cost-related load balancing algorithm is *list scheduling*. This scheduling algorithm works by assigning the next not yet executed job to the first idle processor, independently of the amount of work associated with the job. A version of the above described $\mathcal{H}$-matrix computation algorithm utilising list scheduling is

```
for all b ∈ P do
    let 0 ≤ i < p be the number of the first idle processor;
    if b is admissible then
        create a low-rank approximant of M_b using ACA on processor i;
    else
        create the dense matrix M_b on processor i;
endfor;
```

Algorithm 3.2: List scheduling

List scheduling does not produce an optimal scheduling, but one can proof the following estimate, see [8].

Remark 3.6 *Let $t(p)$ be the time for executing n jobs on p processors using list scheduling and let $t_{\min}(p)$ be the minimal time needed for running n jobs on p processors. Then the following holds:*

$$t(p) \leq \left(2 - \frac{1}{p}\right) t_{\min}(p). \tag{3.4}$$

If the costs are known a priori, e.g., if a constant rank approximation is generated, one is able to use a refined version of list scheduling, namely *longest process time* (LPT) scheduling. Instead of assigning the matrix blocks randomly to the processors, they are ordered according to their costs, starting from the most expensive one.

```
V := P;
while V ≠ ∅ do
    choose v ∈ V with maximum costs; V := V \ {v};
    let 0 ≤ i < p be the first idle processor;
    if v is admissible then
        create a rank-k approximant of M_b using ACA on processor i;
    else
        create the dense matrix M_b on processor i;
endfor;
```

Algorithm 3.3: LPT scheduling

Compared with list scheduling, the result of this ordering is an improved approximation of an optimal scheduling, see [8].

Remark 3.7 *Let $t(p)$ be the time for executing n jobs on p processors using LPT scheduling and let $t_{\min}(p)$ be as in Remark 3.6. Then the following holds:*

$$t(p) \leq \left(\frac{4}{3} - \frac{1}{3p}\right) t_{\min}(p). \tag{3.5}$$

Since the costs differ from an optimal scheduling only by a constant factor (see Remark 3.6 and Remark 3.7), the costs for building the approximant are of order $\mathcal{O}\left(\frac{1}{p} N \log^{2d-1} N\right)$.

3.1.1 Shared Memory

For the problem of building an $\mathcal{H}$-matrix on a shared memory machine list scheduling turns out to be sufficient to provide an optimal parallel speedup independently of the type of approximation (fixed rank or adaptive rank). Therefore, in this section we will restrict ourselves to this scheduling algorithm.

A widely used mechanism for parallel computations on a shared memory system are *threads* which are independent, parallel execution paths in a single task. Because all threads are part of the same task, they share the same address space which eliminates the need for communication between individual processors.

A standard interface to threads on most computer systems are *POSIX*-threads or *Pthreads* (see [5]). Unfortunately, the usage of the Pthread interface can be complicated and often distracts from the real problem to solve. Furthermore, the creation of Pthreads comes with some costs, especially if several hundreds of threads are used. The latter situation occurs in a modified version of Algorithm 3.2, where each submatrix is computed in a new thread:

```
for all b ∈ P do
    create a new thread t;
    if b is admissible then
        build a rank-k approximant of M_b using ACA in thread t;
    else
        build the dense matrix M_b in thread t;
endfor;
```

An alternative to this approach is the usage of a *threadpool* which consists of a fixed set of working threads. These threads are created once during the initialisation of the threadpool and stay active until the program terminates. Hence, no further costs for creating threads occur. Each thread in the threadpool takes a given job, executes it and remains idle until the next job is given to the pool. The following pseudo-code shows a basic interface to a threadpool:

```
class ThreadPool {
    init( p ∈ ℕ );
    run( Job j );
    sync_all();
}
```

The function `init` initialises the threadpool with the given number of threads and hence sets the maximal degree of parallelism. A job is given to the pool and associated with a thread in the pool by the function `run`. The function `sync_all` blocks until all threads have finished, thus providing a synchronisation mechanism. See [13] for a complete implementation of a threadpool in the programming language C++ with a detailed discussion of the advantages.

Utilising this interface the matrix computation can be rewritten and we end up with the following algorithm.

```
procedure build_matrix( b )
    if b is admissible then
        build a rank-k approximant of M_b using ACA;
    else
        build the dense matrix M_b;
end;

for all b ∈ P do run( build_matrix( b ) );
sync_all();
```

Algorithm 3.4: List scheduling using a threadpool

In [3] it has been shown that the generated $\mathcal{H}$-matrix approximant can be stored using $\mathcal{O}\left(N \log^d N\right)$ units of memory.

3.1.2 Distributed Memory

In the first part of this section we consider the case of a constant rank approximation. To be able to apply LPT scheduling the costs per matrix block must be known. For this let P_{adm} and P_{nadm} be the set of admissible and non-admissible blocks in P, respectively. The costs per matrix block $b = \tau \times \sigma$ are then defined by

$$c_{\text{MB},k}(\tau,\sigma) = \begin{cases} |\tau| \cdot |\sigma|, & (\tau,\sigma) \in P_{\text{nadm}} \\ k^2 \cdot (|\tau| + |\sigma|), & (\tau,\sigma) \in P_{\text{adm}} \end{cases} . \tag{3.6}$$

Instead of using *online* scheduling as in Algorithm 3.3, where a block is associated to a processor during the computation, in the case of a distributed memory an *offline* version is necessary, i.e., the load balancing has to be done in advance. Hence, no communication is involved during the matrix building process. The offline version of Algorithm 3.3 is shown in Algorithm 3.5.

```
procedure LPT( P )
    V := P; C_i := 0 for 0 ≤ i < p;
    while V ≠ ∅ do
        let v ∈ V, such that c_{MB,k}(v) = max_{b∈V} c_{MB,k}(b);
        choose i ∈ {0, ..., p − 1} such that C_i = min_{0≤j<p} C_j;
        assign v to processor i;
        C_i := C_i + c_{MB,k}(v); V := V \ {v};
    endwhile;
end;
```

Algorithm 3.5: Offline LPT scheduling

In C_i the sum of the costs associated to each processor $0 \leq i < p$ are stored. Hence, processor i for which $C_i = \min_{0 \leq j < p} C_j$ is the first idle processor to receive the next block.

As mentioned above in the case of an adaptive rank approximation a cost function for load balancing cannot be defined. Although an online scheduling algorithm like it was used for a shared memory system is possible to be implemented, e.g., by a master-slave model, this involves communication which should be avoided.

In order to be able to still use offline scheduling, we assume that the ranks of the low-rank matrices are bounded by some $k_{\max}$. Then an average rank k_{avg} for these submatrices exists. With k_{avg} one can use the same load balancing algorithm as for a constant rank approximation. Because only an approximation of the real costs per matrix block are used, estimate (3.5) will not hold in general. Instead the algorithm can be considered as list scheduling and hence (3.4) is valid. Therefore, the resulting distribution still approximates an optimal load balancing up to a constant factor.

Since the costs for building and for storing each block have the same complexity (see Remark 3.4), the amount of storage needed on each processor to hold the $\mathcal{H}$-matrix approximant is of order $\frac{1}{p} N \log^d N$.

In Section 4.2.1 a scheduling is presented to balance the $\mathcal{H}$-matrix-vector multiplication among the processors. Although not being optimal, this load balancing can also be used to distribute the blocks when building the $\mathcal{H}$-matrix approximant. For more details see Section 4.2.2.

4 Parallel Matrix-Vector Multiplication

Instead of a pure matrix-vector multiplication, in this section we examine the more general update of a vector y by the operation

$$y := \alpha A x + \beta y,$$

with an $\mathcal{H}$-matrix $A \in \mathcal{H}(P)$, vectors $x, y \in \mathbb{R}^N$ and $\alpha, \beta \in \mathbb{R}$. This follows the definition of the function `GEMV` in the *LAPACK* software library (see [6]).

Two algorithms for the matrix-vector multiplication are presented: one for distributed and the other for shared memory systems. Although both algorithms are formulated for a general parallel computer, namely a BSP machine (see Section 4.1), they take advantage of the different properties of the memory subsystem leading to different parallel characteristics.

We assume that both vectors x and y are distributed uniformly across p processors, each holding N/p entries of x and y, respectively. By x_i and y_i we denote the part of x and y on processor $0 \leq i < p$. This distribution ensures optimal complexity of all vector operations.

4.1 Bulk Synchronous Parallel Machines

Both algorithms for the matrix-vector multiplication are usually described in the context of a *Bulk Synchronous Parallel* or *BSP*-computer (see [22]). This model for a parallel system is based on three parameters: the number of processors p, the number of time steps for a global synchronisation l and the ratio g of the total number of operations performed on all processors and the total number of words delivered by the communication network per second. All parameters are normalised with respect to the number of time steps per second.

A BSP computation consists of single *supersteps*. Each superstep has an *input phase*, a *local computation phase* and an *output phase* (see Figure 3). During the input phase, each processor receives data sent during the output phase of the previous superstep. While all processors are synchronised after each superstep, all computations within each superstep are asynchronous.

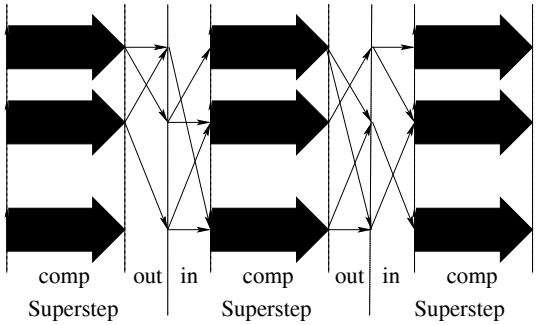

Figure 3: BSP computation

The complexity of a BSP computation can be described by the parameters l, g, the number of operations done on each processor and the amount of data sent between the processors. For a single superstep, the amount of work can be expressed by $w + (h_{\text{in}} + h_{\text{out}}) \cdot g + l$, where w is the maximum number of operations performed and h_{in}, h_{out} are the maximum numbers of data units received and sent by each processor, respectively. The total work of the BSP computation is the sum of the costs in each superstep, yielding an expression of the form $W + H \cdot g + S \cdot l$, where S is the number of supersteps.

4.2 Distributed Memory

For the $\mathcal{H}$-matrix-vector multiplication on a distributed memory machine the ideas of the BSP algorithm for dense matrices described in [17] can be used. The fundamental difference between both algorithms is the way the matrix blocks are distributed among the processors. In the case of a dense matrix $A \in \mathbb{R}^{N \times N}$ each of the p processors holds a block A_i of size $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$[1].

The BSP algorithm is split into three steps. In the first step each processor has to receive $\frac{N}{\sqrt{p}}$ entries of x needed for the local matrix-vector multiplication, which is done in the second superstep. The resulting entries of y are afterwards sent to the corresponding processors. In the third step, all local coefficients of y are summed up for the final result.

[1]For simplicity we assume that N is a multiple of p and that p is a perfect square

procedure dense_mul($\alpha, A_i, x, \beta, y, i$)
{ *first step* }
$y_i := \beta \cdot y_i$;
send x_i to all processors sharing it;
sync();
{ *second step* }
$y_i' := \alpha A_i x_i$;
send respective parts of y_i' to all processors sharing it;
sync();
{ *third step* }
$Y_i := \{\text{received vectors of local results}\}$;
$y_i := y_i + \sum_{y_j' \in Y_i} y_j'$;
sync();
end;

Algorithm 4.1: Dense Matrix-Vector Multiplication

The costs of Algorithm 4.1 are $\frac{N}{p} + g \cdot \frac{N}{\sqrt{p}} + l$ for scaling y and sending x in the first step, $\frac{N^2}{p} + g \cdot \frac{N}{\sqrt{p}} + l$ for the local matrix-vector product in the second step and $\frac{N}{\sqrt{p}} + l$ for the summation in the last superstep. Therefore, the total work for the dense matrix-vector multiplication is

$$\mathcal{O}\left(\frac{N^2}{p} + \frac{N}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\frac{N}{\sqrt{p}}\right) + 3 \cdot l, \tag{4.1}$$

which can be shown (see [17]) to be optimal with respect to computation and communication costs.

This regular block distribution is not suitable for $\mathcal{H}$-matrices, since the costs of a matrix-vector product vary between the matrix blocks within an $\mathcal{H}$-matrix, e.g., dense matrix blocks require more work than rank-k blocks of the same size. Unfortunately, a different distribution pattern might result in communication costs which are no longer optimal. For example, applying list scheduling as described in Section 3.1 leads to a random distribution (see Figure 4 (left)). Although such a load balancing is efficient with respect to the local multiplication phase in Step 2 of Algorithm 4.1, due to the scattering of the matrix blocks across the whole $\mathcal{H}$-matrix, the vector x has to be sent to all processors with communication costs of $\mathcal{O}(N)$. Furthermore, p vectors have to be summed up in the third step with computation costs $\mathcal{O}(N)$. Such a situation should therefore be avoided.

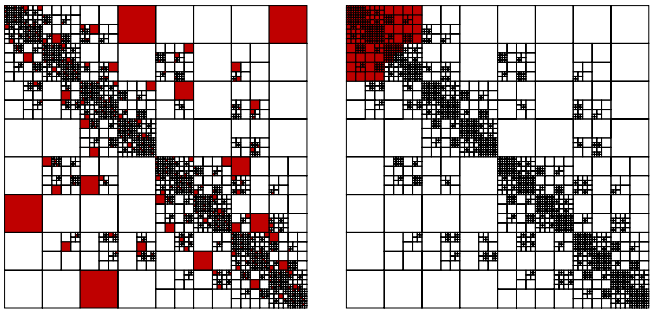

Figure 4: List scheduling (left) versus sequence partitioning (right)

In order to be able to measure the communication and computation costs with respect to the vectors x and y we introduce the *sharing constant* of an index set, i.e., the maximum number of processors sharing one row or one column.

Definition 4.1 *Let $A \in \mathcal{H}(P)$ and let P_q denote the blocks in P assigned to processor q. For $i \in I$ define the* sharing constant $c_{\text{sh}}(i)$ *of i as*

$$c_{\text{sh}}(i) = \max\{c^r_{\text{sh}}(i), c^c_{\text{sh}}(i)\}, \tag{4.2}$$

where $c^r_{\text{sh}}(i) = |\{q \,|\, \exists(\tau,\sigma) \in P_q : i \in \tau\}|$ and $c^c_{\text{sh}}(i) = |\{q \,|\, \exists(\tau,\sigma) \in P_q : i \in \sigma\}|$. Furthermore, let $c_{\text{sh}} = \max_{i\in I} c_{\text{sh}}(i)$ be the sharing constant of I.

We can use c_{sh} to express the costs for sending x in the first step and for summing up all local vectors y_i in the last step of Algorithm 4.1. The following definition allows us to define the costs for receiving the vector x and sending the local result y_i for general matrices.

Definition 4.2 *For a partition P and $\tau \subset I$ let the sets $I^c(q)$ and $I^r(q)$ be defined as*

$$I^c(q) = \{j \in I \,|\, \exists(\tau,\sigma) \in P_q : j \in \sigma\} \quad \text{and} \quad I^r(q) = \{i \in I \,|\, \exists(\tau,\sigma) \in P_q : i \in \tau\}.$$

Furthermore, let $\mu = \max_{0\le q<p}\{|I^c(q)|, |I^r(q)|\}$.

For dense matrices and the above regular block distribution c_{sh} equals $\sqrt{p}$, whereas the random distribution induced by list scheduling results in a c_{sh} of order p. Similarly, we find $\mu = n/\sqrt{p}$ in the case of dense matrices and $\mu = \mathcal{O}(n)$ for list scheduling. Using c_{sh} and μ we can rewrite (4.1) and obtain the complexity of the matrix-vector multiplication for $\mathcal{H}$-matrices

$$\mathcal{O}\left(\frac{kN\log N}{p} + c_{\text{sh}}\frac{N}{p}\right) + g \cdot \mathcal{O}\left(c_{\text{sh}}\frac{N}{p} + \mu\right) + 3 \cdot l \tag{4.3}$$

One way to decrease c_{sh} and μ are *space filling curves* and *sequence partitioning*. These methods produce a distribution of P with a much higher locality of the blocks associated to a specific processor q (see Figure 4 (right)). Due to the "compactness" of the sets P_q the probability of sharing an index with another processor is reduced.

4.2.1 Load Balancing with Sequence Partitioning

The first ingredient for the load balancing algorithm is a cost function for the matrix-vector multiplication. This function is defined by the representation of dense and low-rank matrix blocks:

$$c_{\text{MV},k}(\tau,\sigma) = \begin{cases} |\tau| \cdot |\sigma|, & (\tau,\sigma) \in P_{\text{nadm}} \\ k \cdot (|\tau| + |\sigma|), & (\tau,\sigma) \in P_{\text{adm}} \end{cases}. \tag{4.4}$$

The set $\{c_{\text{MV},k}(v) \,|\, v \in P\}$ together with an ordering is the input for the sequence partitioning problem.

Definition 4.3 (Sequence Partitioning) *Let $C = \{c_0, c_1, \ldots, c_{n-1}\}$ be a sequence with costs $c_i > 0$. Furthermore, let $R = \{r_0, \ldots, r_p\}$ with $0 = r_0 \le r_1 \le \ldots \le r_p = n$, $r_i \in \mathbb{N}$, $0 \le i \le p$. Then R is called a* sequence partition *of (C,p). R is* optimal *with respect to (C,p), if there is no partition $R' = \{r'_0, \ldots, r'_p\}$ of C such that*

$$\max_{0\le i<p} \sum_{j=r'_i}^{r'_{i+1}-1} c_j < \max_{0\le i<p} \sum_{j=r_i}^{r_{i+1}-1} c_j =: c_{\max}(C).$$

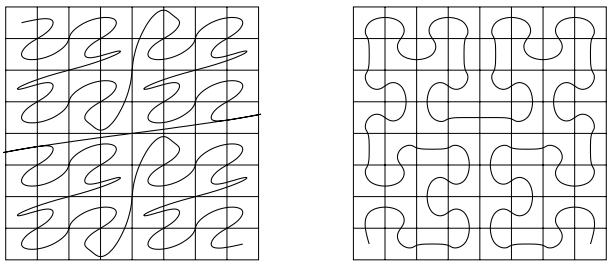

Figure 5: Space filling curves

For the computation of an optimal partition of a sequence C the knowledge of $c_{\max}(C)$, the costs of the most expensive interval in an optimal partition, is sufficient. In [18] an algorithm is presented which computes $c_{\max}(C)$ with complexity $\mathcal{O}(N \cdot p)$. The optimal partition can then be obtained by summing up the costs for each element of the list and starting a new subsequence whenever the costs exceeds $c_{\max}(C)$.

The required ordering of the blocks in P can be defined by using space filling curves. These curves describe a surjective mapping from the unit interval $[0, 1]$ to the unit square $[0, 1]^2$. In Figure 5 two examples of such curves are presented, the *Z*-curve and the *Hilbert*-curve. Space filling curves pass through all points of $[0, 1]^2$ and because partitions of $I \times I$ can be mapped to the unit square, the order in which a leaf is reached by the curve defines a sequence usable for sequence partitioning. The neighbourhood relationship of adjacent subintervals of the space-filling curve guarantees a high "compactness" of the corresponding sets P_q. The result is shown in Figure 6, where the Z- and the Hilbert-curve are applied to a block partition.

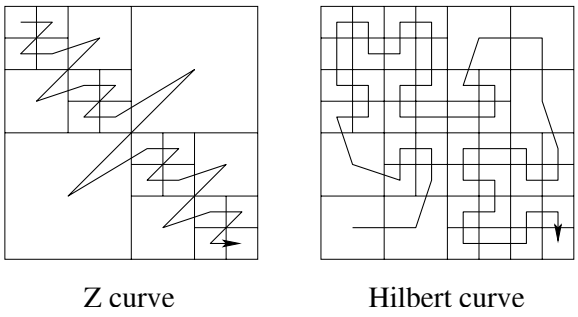

Figure 6: Space filling curves applied to $\mathcal{H}$-matrices

The restriction to quad trees in the definition of block cluster trees allows a simple computation of the ordering induced by space filling curves. The basic algorithm is a depth first search (DFS, see [20]) in $T_{I\times I}$. In contrast to the usual DFS algorithm, the order in which the sons $S(v)$ of a node v are accessed is defined by a *mark* associated to each node. In Figure 7 the marks and the corresponding order of the sons is presented for the Z- and the Hilbert-curve, respectively. Here, the root of the block cluster tree always has the mark "A".

The motivation of the load balancing with sequence partitioning was the reduction of c_{sh} compared to a random distribution generated by list scheduling. In Figure 8 (left) the value of c_{sh} obtained by using the Z- and the Hilbert-curve for different numbers of processors is presented. Independently of the space filling curve, one can observe a behaviour of the kind

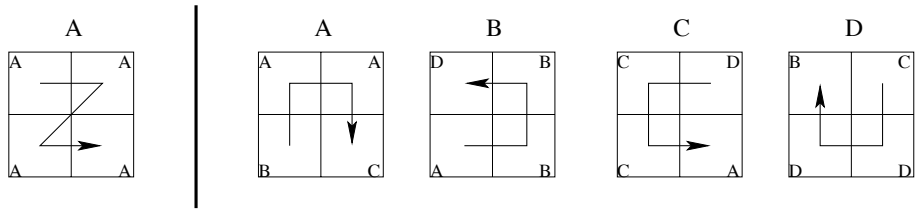

Figure 7: Construction of space filling curves: Z (left) and Hilbert (right)

$c_{\text{sh}} \sim \sqrt{p}$, which is equal to the uniform distribution in the case of dense matrices. This shows the reduction of c_{sh} compared to a random distribution.

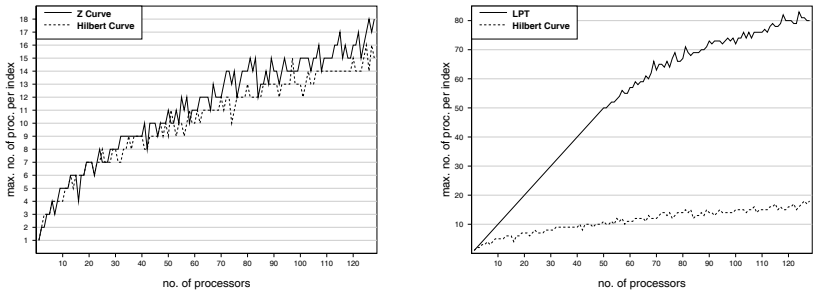

Figure 8: Value of c_{sh} for space-filling curves and LPT-scheduling

For comparison in Figure 8 (right) c_{sh} is shown for LPT scheduling without space filling curves. Especially for a small p the $\mathcal{O}(p)$ behaviour of c_{sh} is visible. If $p > 50$ the number of blocks per processor gets smaller, therefore less processors share the same index.

However, μ is not reduced by using space filling curves and sequence partitioning. The main reason for this is that large admissible blocks have a size of $\mathcal{O}(N)$. Therefore, for processors q with these blocks values of $|I^c(q)|$ and $|I^r(q)|$ are also of order N.

4.2.2 Building the $\mathcal{H}$-matrix

In Section 3.1.2 a load balancing algorithm based on LPT scheduling for efficiently building an $\mathcal{H}$-matrix was introduced. As discussed in the beginning of this section this distribution is not optimal for the matrix-vector multiplication since $c_{\text{sh}} \sim p$. If one uses LPT scheduling for building the matrix and sequence partitioning for the matrix-vector multiplication a communication step is necessary between both operations to transfer the matrix blocks according to the changed distribution of the matrix.

Due to the similarity of the cost functions (3.6) and (4.4) (see Remark 3.4), in practise sequence partitioning also produces a scheduling which can be used for building the $\mathcal{H}$-matrix approximant.

4.3 Shared Memory

In the case of a shared memory architecture we propose using the same algorithm as in the case of a distributed memory machine. This can be justified by examining the (hidden) constants in the part of (4.3) describing the computational work. By assuming $c_{\text{sh}} \sim \sqrt{p}$ we can rewrite this

equation as

$$\mathcal{O}\left(c_1 \frac{kN \log N}{p} + c_2 \frac{N}{\sqrt{p}}\right) \tag{4.5}$$

with $c_1, c_2 > 0$. In practise $c_1 \gg c_2$ holds. On shared memory system, usual values for p range from 1 to 128. Since the influence of the second term can only be seen for large p, the first term dominates the computational work.

Furthermore, we can neglect communication on a shared memory computer. Hence, the complexity of the matrix-vector multiplication is completely described by (4.5).

Algorithm 4.1 can be simplified by using the interface for a thread pool described in Section 3.1.1. For this, the first two steps of the BSP-algorithm, namely scaling y and the matrix-vector multiplication are combined because the vector x can be accessed by all processors. The summation of the final result is done in a second step, after all threads have computed the corresponding local results y_i'.

```
procedure step_1 ( i, β, y_i, A_i, x )
    y_i := β · y_i;
    y_i' := α A_i x;
end;

procedure step_2 ( i, y_i )
    Y_i := {y_j' | I_τ(j) ∩ I_τ(i) ≠ ∅};
    y_i := y_i + Σ_{y_j' ∈ Y_i} y_j'|_{I_τ(i)};
end;

procedure tp_mv_mul( i, α, A, x, β, y )
    for 0 ≤ i < p do run( step_1( i, β, y, A, x ) );
    sync_all();
    for 0 ≤ i < p do run( step_2( i, y_i ) );
    sync_all();
end;
```

Algorithm 4.2: Matrix-vector multiplication using threads

Another advantage of this algorithm is that for the implementation only minor modifications of an existing sequential version are necessary, e.g., the computation of the matrix-vector product in the first step differs only by the involved set of matrix blocks.

5 Numerical experiments

As an academic example the single layer potential operator $\mathcal{V} : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ defined by

$$(\mathcal{V}u)(x) = \frac{1}{4\pi} \int_\Gamma \frac{u(y)}{|x-y|} \, \mathrm{d}s_y$$

is used to test the proposed algorithms. In the following experiments Γ is the surface from Figure 9.

A Galerkin discretization with piecewise constants φ_i, $i = 1, \ldots, N$, leads to the matrix $V \in \mathbb{R}^{N \times N}$ with entries

$$v_{ij} = (\mathcal{V}\varphi_i, \varphi_j), \quad i, j = 1, \ldots, N,$$

which is symmetric, since $\mathcal{V}$ is selfadjoint with respect to $(\cdot, \cdot)_{L^2(\Gamma)}$. Therefore, it is only necessary to approximate the upper triangular part of V.

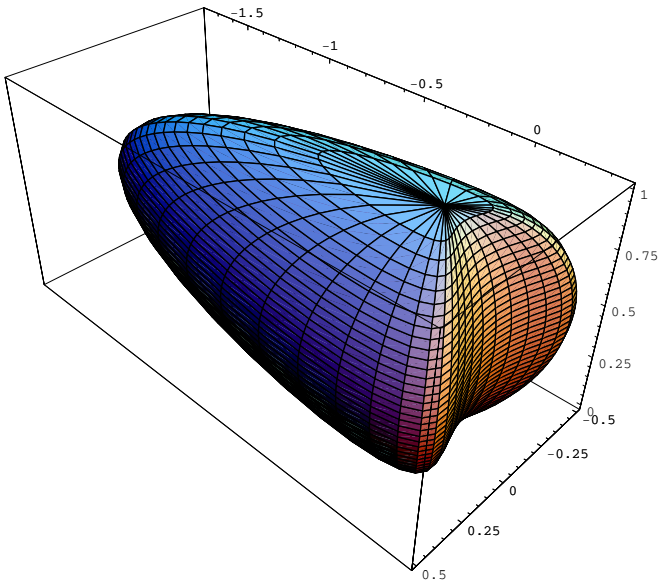

Figure 9: The test surface

For the numerical tests of the shared memory algorithms an HP9000, PA-RISC with 875 Mhz was used, whereas the tests of the distributed memory algorithms were carried out on an AMD 900 MHz cluster. In the first part of this section numerical results for building the $\mathcal{H}$-matrix approximant are presented. The second part concentrates on the parallel matrix-vector multiplication.

5.1 Building the $\mathcal{H}$-matrix

5.1.1 Shared Memory

In the case of a shared memory Algorithm 3.4 based on online scheduling was used. The first table shows the time and the corresponding parallel efficiency E (see Definition 3.5) in % for building an $\mathcal{H}$-matrix using a fixed rank of $k = 10$ for different numbers of processors.

N	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$		Storage
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E	in MB
3 968	117.3	29.4	99.9	14.7	99.7	9.8	99.6	7.4	99.3	33
7 920	320.2	80.1	99.8	40.2	99.6	26.8	99.7	20.1	99.4	83
19 320	1042.6	261.1	99.8	130.6	99.8	87.2	99.7	65.5	99.5	258
43 680	2904.2	727.2	99.9	364.4	99.6	243.1	99.6	182.4	99.5	706
89 400	6964.6	1747.2	99.7	875.2	99.5	582.1	99.7	437.4	99.5	1670
184 040	16694.9	4179.0	99.9	2081.1	100.0	1391.4	99.9	1049.9	99.4	3980

The second table contains the corresponding times for building the $\mathcal{H}$-matrix using a fixed approximation accuracy $\varepsilon = 10^{-4}$.

N	$p=1$	$p=4$		$p=8$		$p=12$		$p=16$		Storage
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E	in MB
3 968	101.9	25.6	99.9	12.8	99.6	8.6	99.6	6.4	99.4	46
7 920	269.3	67.4	99.9	33.8	99.7	22.5	99.7	16.9	99.5	88
19 320	826.0	206.9	99.8	103.6	99.7	69.1	99.7	51.9	99.4	226
43 680	2258.9	566.2	99.8	283.7	99.6	192.2	99.5	142.0	99.4	577
89 400	5362.4	1347.8	99.5	675.3	99.3	448.7	99.6	337.2	99.4	1326
184 040	12741.6	3185.6	100.0	1596.3	99.8	1059.5	100.0	796.0	100.0	3079

The algorithm shows a very high parallel efficiency even for small problem sizes indicating the low overhead resulting from the threadpool and the almost optimal load balancing produced by list scheduling.

5.1.2 Distributed Memory

Due to memory restrictions problems for large N could not be computed with a small number of processors p. The corresponding parallel efficiency for these problem sizes is therefore computed with respect to the smallest p available, i.e.,

$$E(p) := \frac{p' \cdot t(p')}{p \cdot t(p)}$$

with p' being the smallest number of processors able to compute the problem. The presented storage size in these cases is approximated by $p'\mathcal{N}$, where $\mathcal{N}$ denotes the memory consumption per processor on a p'-CPU system.

Numerical experiments have shown that when building the matrix the advantage of LPT scheduling over sequence partitioning with space-filling curves can be neglected. Especially, if also the matrix-vector multiplication is to be executed in parallel, sequence partitioning should be prefered, since then the redistribution of blocks is avoided (see also Section 4.2.2).

The first table shows again the time for buildung the $\mathcal{H}$-matrix with a fixed rank of $k = 10$.

N	$p=1$	$p=4$		$p=8$		$p=12$		$p=16$		Storage
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E	in MB
3 968	147.9	37.4	98.9	18.9	97.8	13.1	94.0	10.0	92.6	29
7 920	404.2	102.0	99.0	51.7	97.7	35.6	94.4	27.3	92.7	77
19 320	1317.0	334.7	98.4	166.9	98.7	119.2	92.1	87.9	93.6	245
43 680		929.9		466.8	*99.6*	330.3	*93.9*	248.0	*93.7*	*712*
89 400				877.4		605.1	*96.5*	467.9	*94.8*	*1784*
184 040						1430.6		1075.1	*98.9*	*4892*

The results show that sequence partitioning in the case of a constant rank approximation is highly efficient and also works very well for small problem sizes.

In the next table results for a fixed accuracy of $\varepsilon = 10^{-4}$ are presented. The average rank which is needed in this case was set to $k_{\text{avg}} = 10$ with a maximal rank of $k_{\max} = 100$. Again, for the larger problem sizes the $\mathcal{H}$-matrix could not be built for small p.

N	$p=1$	$p=4$		$p=8$		$p=12$		$p=16$		Storage
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E	in MB
3 968	128.1	33.7	94.9	18.0	89.0	12.2	87.7	9.4	85.3	26
7 920	338.5	90.3	93.8	48.1	88.1	32.5	86.9	25.1	84.4	66
19 320	1038.4	272.2	95.4	143.8	90.3	96.6	89.6	74.1	87.6	197
43 680	2845.5	739.4	96.2	395.3	90.0	258.6	91.7	202.7	87.7	530
89 400				941.7		628.1	*100.0*	480.2	*98.1*	*1528*
184 040						1450.5		1107.0	*98.3*	*3864*

Because the real costs per matrix block are only approximated, the parallel efficiency in the case of fixed accuracy is not as high as the efficiency in the case of approximations of fixed rank. Nevertheless, the scheduling still produces reasonable speedups.

5.2 Matrix-Vector Multiplication

In this section we examine the performance of the parallel matrix-vector multiplication. For simplicity the factors α and β in the product $y := \beta y + \alpha Ax$ are choosen to be 1. In all examples the time for 100 matrix-vector multiplications was measured.

5.2.1 Shared Memory

Again at first an $\mathcal{H}$-matrix with a fixed rank of $k = 10$ is used. The times resulting from using Algorithm 4.2 and the parallel efficiency are presented in the following table.

N	$p=1$	$p=4$		$p=8$		$p=12$		$p=16$	
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	11.7	3.3	88.9	1.8	80.0	1.3	73.2	1.3	56.6
7 920	30.9	8.5	90.8	4.6	83.5	3.5	74.3	2.9	66.2
19 320	94.9	25.9	91.7	13.5	87.8	9.5	83.1	7.5	79.2
43 680	251.7	70.9	88.7	36.0	87.4	23.9	87.9	18.9	83.8
89 400	556.4	152.2	91.4	80.0	87.0	53.4	86.9	41.1	84.7
184 040	1277.5	347.7	91.9	186.0	85.9	120.1	88.7	97.5	81.9

The weak parallel performance on the lower levels is probably due to sequential parts in the algorithm, e.g., management overhead. Since that part remains constant independently of the problem size, the parallel efficiency grows with N and stabilises at about 80-90 %.

In the following table the results for the same operation but with an $\mathcal{H}$-matrix obtained from an approximation with fixed accuracy $\varepsilon = 10^{-4}$ are shown.

N	$p=1$	$p=4$		$p=8$		$p=12$		$p=16$	
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	9.6	2.6	93.2	1.6	73.3	1.3	61.2	1.3	47.7
7 920	23.8	6.3	94.8	3.7	79.7	3.1	65.1	2.4	63.1
19 320	66.7	17.2	96.9	9.6	87.2	7.0	79.6	5.6	74.2
43 680	169.6	44.6	95.1	22.8	93.2	15.7	90.0	13.0	81.7
89 400	346.2	91.1	95.0	47.1	91.9	32.8	87.9	25.7	84.2
184 040	780.5	202.6	96.3	107.1	91.1	69.8	93.2	55.0	88.6

The same behaviour as in the previous table is visible: the parallel efficiency grows with N and reaches an almost optimal value of about 90 %.

5.2.2 Distributed Memory

The same restrictions with respect to memory as when building the $\mathcal{H}$-matrix on a distributed memory machine apply for the matrix-vector multiplication. Therefore, the times for larger problems sizes N and small p are missing.

The results for a fixed rank ($k = 10$) are as follows:

N	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	16.2	4.8	85.1	2.8	72.1	2.1	64.6	1.6	64.9
7 920	43.8	12.1	90.7	6.7	81.3	4.8	76.3	4.0	69.3
19 320	141.1	39.5	89.2	20.2	87.1	14.7	80.1	11.1	79.6
43 680		107.9		57.0	*94.75*	42.0	*85.6*	32.1	*84.2*
89 400				129.9		90.8	*95.3*	69.7	*93.2*
184 040						209.4		157.4	*99.8*

One observes the same behaviour for the parallel performance as in the case of a shared memory system: a better efficiency is obtained for larger N. The same behaviour is also visible for a fixed accuracy ($\varepsilon = 10^{-4}$) as the results in the following table show. The average and the maximal rank were chosen as when building the $\mathcal{H}$-matrix ($k_{\text{avg}} = 10$ and $k_{\max} = 100$).

N	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	12.0	3.6	82.9	2.1	70.8	2.0	49.8	1.3	56.8
7 920	30.0	8.6	87.4	5.0	75.6	3.6	69.3	3.0	63.6
19 320	84.6	27.0	78.5	13.2	80.1	9.5	74.3	7.5	70.2
43 680	221.0	64.0	86.4	34.5	80.1	23.2	79.6	25.5	54.2
89 400				74.1		53.6	*92.2*	42.4	*87.4*
184 040						119.6		90.8	*98.8*

Again, due to the approximation of the actual costs, the parallel efficiency is not as high as for an $\mathcal{H}$-matrix with a fixed rank.

References

[1] M. Bebendorf: *Effiziente numerische Lösung von Randintegralgleichungen unter Verwendung von Niedrigrang-Matrizen.* dissertation.de, Verlag im Internet, 2001. ISBN 3-89825-183-7.

[2] M. Bebendorf: *Approximation of boundary element matrices.* Numer. Math. 86, 565–589, 2000.

[3] M. Bebendorf and S. Rjasanow: *Adaptive Low-Rank Approximation of Collocation Matrices.* Computing 70(1), 1–24, 2003.

[4] M. Bebendorf: *Efficient Galerkin BEM using ACA.* in preparation.

[5] D.R. Butenhof: *Programming with POSIX Threads.* Addison-Wesley, 1997.

[6] J. Dongarra and J. Demmel: *LAPACK: a portable high-performance numerical library for linear algebra.* Supercomputer, 8, 33–38, 1991.

[7] S. A. Goreinov, E. E. Tyrtyshnikov and N. L. Zamarashkin: *A theory of pseudoskeleton approximations.* Linear Algebra Appl. 261, 1–21, 1997.

[8] R. L. Graham: *Bounds on Multiprocessing Timing Anomalies.* SIAM Journal of Applied Mathematics 17(2), 416–429, 1969.

[9] L. Greengard and V. Rokhlin: *A new version of the fast multipole method for the Laplace equation in three dimensions.* Acta numerica, 229–269. Cambridge Univ. Press, Cambridge, 1997.

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

[11] W. Hackbusch and B. N. Khoromskij: *A sparse $\mathcal{H}$-matrix arithmetic. II. Application to multi-dimensional problems.* Computing 64(1), 21–47, 2000.

[12] W. Hackbusch and Z. P. Nowak: *On the fast matrix multiplication in the boundary element method by panel clustering.* Numer. Math. 54(4), 463–491, 1989.

[13] R. Kriemann: *Implementation and Usage of a Thread Pool based on POSIX Threads.* Technical Report and Documentations 2/2003, MPI Leipzig.

[14] S. Kurz, O. Rain and S. Rjasanow: *The Adaptive Cross Approximation Technique for the 3D Boundary Element Method.* IEEE Transaction on Magnetics 38(2), 421–424, 2002.

[15] U. Langer and D. Pusch: *Algebraic Multigrid Preconditioners for Large Scale Boundary Element Equations.* IMACS 'Applied Numerical Mathematics', Special Issue 2004.

U. Langer, D. Pusch. Sparse Algebraic Multigrid Methods for

[16] F. Manne, and T. Sørevik: *Optimal Partitioning of Sequences.* J. Algorithms 19, 235–249, 1995.

[17] W. F. McColl: *Scalable Computing.*, Computer Science Today: Recent Trends and Developments 1000, 46–61, Springer-Verlag, 1995.

[18] B. Olstad and F. Manne: *Efficient partitioning of sequences.* IEEE Trans. Comput. 44, 1322–1326, 1995.

[19] V. Rokhlin: *Rapid solution of integral equations of classical potential theory.* J. Comput. Phys. 60(2), 187–207, 1985.

[20] R.E. Tarjan: *Depth-first search and linear graph algorithms.* SIAM Journal on Computing 1(2), 146–160, 1972.

[21] E. Tyrtyshnikov: *Mosaic-skeleton approximations.* Calcolo 33(1-2), 47–57, 1998.

[22] Leslie G. Valiant: *A bridging model for parallel computation.* Communications of the ACM 33(8), 103–111, 1990.