Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems

by

Ronald Kriemann

Preprint no.: 29 2004

Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems

R. Kriemann*

May 12, 2004

Abstract

$\mathcal{H}$-matrices, as they were introduced in previous papers, allow the usage of the common matrix arithmetic in an efficient, almost optimal way. This article is concerned with the parallelisation of this arithmetics, in particular matrix building, matrix-vector multiplication, matrix multiplication and matrix inversion.

Of special interest is the design of algorithms, which reuse as much as possible of the corresponding sequential methods, thereby keeping the effort to update an existing implementation at a minimum. This could be achieved by making use of the properties of shared memory systems as they are widely available in the form of workstations or compute servers. These systems provide a simple and commonly supported programming interface in the form of POSIX-Threads.

The theoretical results for the parallel algorithms are tested with numerical examples from BEM and FEM applications.

AMS Subject Classification: 65F05, 65F30, 65Y05, 65Y10, 68W10
Keywords: Hierarchical matrices, parallel algorithms, shared memory systems

1 Introduction

Working with matrices of size $n \times n$ often involves a complexity of $\mathcal{O}\left(n^2\right)$ or even $\mathcal{O}\left(n^3\right)$ which restricts the applicability of these algorithms. However, for some problems defined by boundary element methods or elliptic partial differential equations, this complexity can be reduced to an almost linear behaviour. For this, in [13] a special matrix format was introduced, the so-called hierarchical matrices or $\mathcal{H}$-matrices, for short.

Unfortunately, the complexity estimates of $\mathcal{H}$-matrix algorithms often include large constants (see [11]), thereby often reducing the competitiveness of $\mathcal{H}$-matrices. Hence, the wish arises to increase the efficiency of these methods, for which one way is in the form of parallelisation.

In this paper parallel algorithms for some of the most frequent tasks in the context of $\mathcal{H}$-matrices are presented, namely matrix building, matrix-vector multiplication, matrix multiplication and matrix inversion.

Since often a sequential implementation of the $\mathcal{H}$-matrix arithmetic is available, the presented algorithms are designed with a minimal effort of changing these programs in mind. For this, the properties of shared memory systems, i.e. computers with a global memory concurrently accessibly by all processors, play a crucial role, for they allow the usage of simple, but efficient load balancing algorithms. Together with the complexity of the $\mathcal{H}$-matrix arithmetics, this leads to almost optimal parallel algorithms.

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

The remainder of this paper is organised as follows. In the next section, $\mathcal{H}$-matrices and the corresponding basic concepts are defined. It also contains the model problems which are used for the numerical experiments. Section 3 describes shared memory systems and how to program these computers. The load balancing methods used in this article are introduced in Section 4. Finally, in Section 5 the parallel $\mathcal{H}$-matrix algorithms are discussed.

2 Hierarchical Matrices

In this section the basic definitions and notions in the context of $\mathcal{H}$-matrices are recalled. For a more, in-depth introduction please refer to [4] or [11]. Furthermore two model problems are defined, which are used in the numerical examples in the next sections.

Definition 2.1 (Cluster tree) *Let $I = \{0, \dots, n-1\}$ be an index set and $T(I) = (V, E)$ a tree with vertices (nodes) V and edges $E \subseteq V \times V$. For a vertex $v \in V$ the set $\mathcal{S}(v) = \{w \mid (v, w) \in E\}$ defines all* sons *of v. $T(I)$ is called a* cluster tree *over I if*

1. *I is the root of $T(I)$ and*
2. *a vertex $v \in V$ is either a* leaf*, i.e., $\mathcal{S}(v) = \emptyset$, or $v = \dot{\cup}_{w \in \mathcal{S}(v)} w$.*

Starting with the root of $T = T(I)$, level sets $T^{(\ell)}$ can be recursively defined by

$$T^{(0)} = \{I\} \quad \text{and} \quad T^{(\ell)} = \{v \mid \exists\, w \in T^{(\ell-1)} : v \in \mathcal{S}(w)\}.$$

All nodes $v \in T^{(\ell)}$ are said to be on *level* ℓ. The *depth* of the cluster tree is define by the maximal level $\max_{\ell \in \mathbb{N}} T^{(\ell)} \neq \emptyset$. The set of leaves of a cluster tree is denoted by $\mathcal{L}(T)$.

In practice the number of sons is usually restricted to 2 resulting in a binary tree. For simplicity, this case is also assumed in this article. Such a tree is also the output of a typical algorithm used to construct a cluster tree: *binary space partitioning* or BSP (see [3]). In a (cardinality balanced) BSP algorithm geometrical data corresponding to a cluster τ is divided along an adaptively chosen hyperplane into two subsets τ', τ'', such that $||\tau'| - |\tau|| \leq 1$. The BSP method is afterwards applied recursively to the sons τ' and τ'' until a minimal cluster size $n_{\min} \geq 1$ is reached. Here, a value of about $30 - 60$ proved efficient in most practical computations.

Definition 2.2 (Block cluster tree) *Let $T(I)$ be a cluster tree over the index set I and $T(I \times I)$ a cluster tree over the product index set $I \times I$. If for all $v \in T^{(\ell)}(I \times I)$ vertices $w, u \in T(I)^{(\ell)}$ exist, such that $v = (w \times u)$ then $T(I \times I)$ is called a* block cluster tree *based on $T(I)$.*

Since the cluster tree $T(I)$ is assumed to be a binary tree, a corresponding block cluster tree $T(I \times I)$ is a quad-tree.

To restrict the theoretical limit of n^2 for the number of leaves in a block cluster tree $T(I \times I) = (V, E)$ an *admissibility condition* $z : V \to \mathbb{B}$ is introduced, which determines whether a block cluster is a leaf or not. By a recursive algorithm (see [3]), a minimal block cluster tree w.r.t. to z, i.e., one with a minimal number of nodes, can then be constructed.

A crucial part in most complexity estimates in the context of $\mathcal{H}$-matrices is played by the sparsity of the block cluster tree, which can be expressed by:

Definition 2.3 (Sparsity constant) *For a block cluster tree $T = T(I \times I)$ based on $T(I)$ the* sparsity constant $c_{\mathrm{sp}} = c_{\mathrm{sp}}(T)$ *is defined as:*

$$c_{\mathrm{sp}} := \max_{\tau \in T(I)} |\{\sigma \,|\, (\tau, \sigma) \in \mathcal{L}(T)\}|. \tag{2.1}$$

The transition from a (minimal) block cluster tree to an $\mathcal{H}$-matrix can be done by creating low-rank matrices for all admissible block clusters and dense matrices for all inadmissible blocks.

Definition 2.4 ($\mathcal{H}$-matrices) *Let $T = T(I \times I)$ be a (minimal) block cluster w.r.t. to an admissibility condition z. The matrix $A \in \mathbb{R}^{I \times I}$ is called an* $\mathcal{H}$-matrix *with local rank $k > 0$, if for all admissible leaves $v \in \mathcal{L}(T)$ the matrix block $A|_v$ has at most rank k, i.e., rank$(A|_v) \leq k$. The set of all $\mathcal{H}$-matrices induced by T, z and k is denoted by $\mathcal{H}(T, k, z)$.*

The low rank of all matrices $A \in \mathbb{R}^{\tau \times \sigma}$ corresponding to admissible blocks $(\tau, \sigma) \in \mathcal{L}(T)$ allows a representation as an R-matrix:

$$A = \sum_{i=0}^{k-1} a_i b_i^T \quad \text{with} \quad a_i \in \mathbb{R}^{\tau} \text{ and } b_i \in \mathbb{R}^{\sigma}.$$

In the following this representation is used for all low-rank matrix blocks of an $\mathcal{H}$-matrix.

Instead of choosing a fixed rank in the definition of $\mathcal{H}$-matrices, a fixed precision $\varepsilon \geq 0$ is also usable. In this case the rank of all R-matrices M' is defined by controlling the relative error w.r.t. to the exact matrix M: $\|M - M'\| \leq \varepsilon \|M\|$.

2.1 Examples

The first example is based on the Fredholm integral equation

$$\lambda u(y) + \int_{\Gamma} \kappa(x, y) u(x) \, \mathrm{d}s_x = f(y), \quad y \in \Gamma, \tag{2.2}$$

with a given right-hand side f and the domain of integration $\Gamma \subset \mathbb{R}^3$ being a 2-dimensional manifold (see Figure 1). This kind of problem arises for instance from the boundary element method (BEM) and is therefore in the following referred to as the *BEM-example*. The kernel function considered is defined by the single layer potential

$$k(x, y) = \frac{1}{4\pi} \frac{1}{|x - y|}.$$

In order to solve (2.2) numerically, the domain Γ is divided into triangles: $\Gamma = \cup_{i \in I} \pi_i, I = \{0, \ldots, n-1\}$. Since the *Galerkin method* is applied in this example, this triangulation is used to define a finite dimensional trial space V_h with piecewise constant basis functions $\{\varphi_i\}_{i \in I}$ in which the solution u is approximated, e.g., $u_h = \sum_{i \in I} u_i \varphi_i$ is looked for. The modified equation leads to a linear equation system (see [12]).

Due to the *asymptotic smoothness* of k (see [6]) the matrix blocks $M \in \mathbb{R}^{\tau \times \sigma}$ can be approximated, if

$$\min\{\mathrm{diam}(X_\tau), \mathrm{diam}(X_\sigma)\} \leq \eta \, \mathrm{dist}(X_\tau, X_\sigma), \tag{2.3}$$

with $X_\tau = \cup_{i \in \tau} \mathrm{supp}\, \varphi_i$ and $\eta \geq 0$. Equation (2.3) can therefore be used as the admissibility condition in this example. Applying *adaptive cross approximation* (ACA) (see [1] or [2]), a low-rank matrix

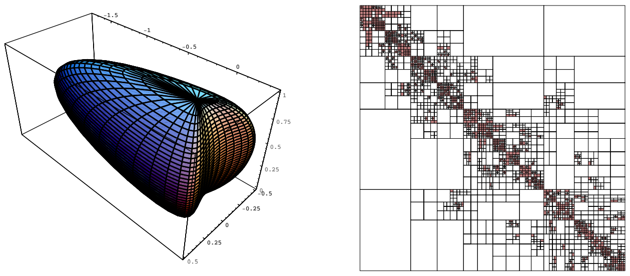

Figure 1: Integration domain and block cluster tree of the BEM-example

$M' \in \mathbb{R}^{\tau \times \sigma}$ with a sufficient approximation for these admissible matrix blocks can be computed in time $\mathcal{O}\left(k^2(|\tau| + |\sigma|)\right)$. ACA is also used in this example to build all R-matrices. The final rank of M' is either fixed or variable with a fixed precision.

Since the construction of the BEM matrix can be quite time consuming, the BEM-example is used to demonstrate matrix building. Furthermore, since Krylov methods are often applied to solve these kind of problems, numerical experiments for the matrix-vector multiplication are also done with the BEM-example. The computer system which was used to run both experiments was a HP9000 Superdome with 875 MHz PA-RISC 8700 processors.

For the second example the Poisson equation is chosen, defined by

$$-\Delta u = f \text{ in } \Omega = [0,1]^2 \quad \text{and} \quad u = 0 \text{ on } \partial\Omega. \tag{2.4}$$

Again the Galerkin method with a finite dimensional ansatz space is used to compute an approximated solution, which leads to the *finite element method* or FEM (see [5]). Therefore, this example is afterwards referred to as the *FEM-example*. In contrast to the previous ansatz, the basis functions are piecewise linear. For the construction of the block cluster tree, the same admissibility condition is used as in the BEM case. An example of the resulting block cluster tree is shown in Figure 2. The linear equation system defined by this ansatz is represented as an $\mathcal{H}$-matrix. Due to the local support of the basis functions and the separation of clusters in admissible blocks all R-matrices have a zero rank. Therefore, no approximation is necessary to represent the system matrix in the $\mathcal{H}$-matrix format.

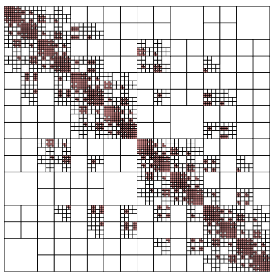

Figure 2: Block cluster tree of the FEM-example

The FEM-example is used to demonstrate the multiplication and inversion of an $\mathcal{H}$-matrix, as it occurs in solving problem (2.4) or in the computation of matrix equations. All numerical experiments for the FEM-example were computed on a SUN Sunfire 6800 with 900 MHz UltrasparcIII processors.

3 Shared Memory Systems

Necessary for the design and implementation of parallel algorithms is a model of the underlying computer system. For sequential computers the *von Neumann* model, which can be generalised in the *random access machine* or *RAM* (see [8]), proved successful and allowed to design algorithms with predictable execution times on nearly all computers.

For parallel systems many models exist which try to approximate the different aspects of real computer systems. One of the simplest model is the *parallel RAM* or *PRAM* (see [9]) which is the analog to the RAM with more than one processor. The PRAM consists of p processors and a global memory which can be accessed by all processors simultaneously as is shown in Figure 3. All data transfers between different processors are handled by this memory system. This avoids communication which is therefore neglected in the analysis of parallel algorithms. Hence, all complexity estimates contain only terms related to computation.

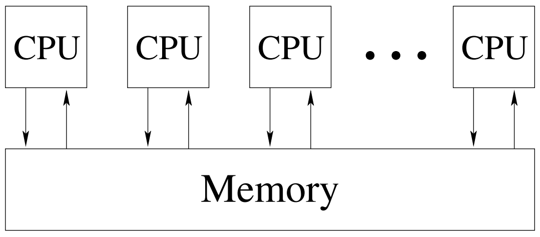

Figure 3: Parallel Random Access Machine

Although not suited to describe all parallel systems, there exists a class of computers which follow the principles of the PRAM, namely *shared memory systems*. As the name indicates, the main property of such computers is a global memory, accessible by all processors as in the PRAM. The concurrent access of this memory is usually implemented by a very fast communication network between the processors (see [15]). Unfortunately such a fast memory interconnect can only be build with justifiable costs for a small number of processors, typically in the range of 2 up to 64 processors. Therefore, the algorithms presented in this paper are restricted to these values of p.

Since most computer languages lack the support for parallel programming, a special interface is needed to use the shared memory with more than one processor. On many systems this can be accomplished by using *threads*. Threads are parallel execution paths in a single process, thereby sharing all resources of this process, e.g., the address space or memory. A common interface for threads is defined by the *POSIX threads* or *Pthreads* (see [7]).

Unfortunately the Pthread interface can be quite complex and often distracts from the actual problem to solve. Therefore a different approach in the form of a *thread pool*, which consists of a set of p fixed threads, is used. Any job, which is supplied to the thread pool is assigned to an idle thread and afterwards executed. After finishing the job, the thread is set free and can be reused by the thread pool. The interface for such a thread pool can be restricted to the following functions:

- `init(`p`)`: initialises the thread pool with p threads,
- `run(`j`)`: associate job j with an idle thread and execute it; if no unused thread is available, the function blocks, i.e., halts further execution, until a thread finishes,
- `sync_all()`: synchronise with the termination of all jobs handled by the thread pool.

Another function which allows the synchronisation with a specific job could be added to the list but since it is not used in this paper, it is neglected. A complete implementation of such a thread pool in the C++ computer language can be found in [16].

Beside the obvious advantage of having a shared memory, this also leads to problems, if two or more processors want to change the same data. Since the result is not predictable, it has to be avoided. Hence, some kind of protection has to be introduced, which is supplied in the PThread interface in the form of *mutices.* A mutex is a variable which is either *locked* or *unlocked.* Locking an already locked mutex blocks the computation until another processor unlocks the mutex. By this, one can ensure that only one processor at a time is inside a critical section of an algorithm.

4 Load Balancing

In this section the algorithms for distributing the load of the $\mathcal{H}$-matrix algorithms on p processors are introduced. At this, two different types of load balancing are used: *online* and *offline* scheduling. An online scheduling method assigns a job to a specific processor *during* the execution of the algorithm. In contrast to this, an offline scheduling algorithm computes the distribution a priori, i.e., before the computation starts.

The main advantage of online scheduling is that no knowledge about the costs of the individual task which are executed is required. On the other hand, often offline scheduling theoretically results in a better approximation of an optimal scheduling, i.e., one with a minimal execution time.

4.1 List-Scheduling

List-Scheduling is one of the oldest online load balancing algorithms. The basic idea behind it is to assign the next not yet executed job to the first idle processor. For a set $J = \{J_0, \ldots, J_{m-1}\}$ of m tasks which should be scheduled to p identical processors, the scheduling algorithm can be formulated as:

```
for all $j \in J$ do
    run( $j$ );
sync_all();
```

Here, the interface from the thread pool introduced in the previous section was used. Since the function `run(`j`)` blocks until an idle thread is available, the next job is executed on the first free processor.

In [10] a discussion of the worst case behaviour of list scheduling can be found, from which the result is shown in the following lemma.

Lemma 4.1 *For a set J of jobs let $t_{\min}(J,p)$ be the minimal time needed to execute all jobs on a parallel machine with p identical processors. Furthermore, let $t_{LS}(J,p)$ be the time needed by list scheduling. Then the following holds:*

$$t_{LS}(J,p) \le \left(2 - \frac{1}{p}\right) t_{\min}(J,p). \tag{4.1}$$

The upper limit in (4.1) is reached, if very costly jobs are executed last. By ordering the jobs according to their costs, the factor in (4.1) can be reduced to $\frac{4}{3} - \frac{1}{3p}$. The resulting algorithm is known as *Longest Process Time* scheduling or *LPT* scheduling. Unfortunately this requires the knowledge of the costs and can therefore only be used in offline load balancing.

4.2 Sequence Partitioning

In contrast to list scheduling where a set of jobs was considered, *sequence partitioning* aims at distributing a sequence, i.e., an ordered set of jobs $J = \{j_0, \ldots, j_{m-1}\}$ with costs $\{c_0, \ldots, c_{m-1}\}, c_i \geq 0$. The sequence partitioning problem to solve is then defined by:

> Find delimiters $r_0 = 0, \ldots, r_p = m-1$ such that the costs $\max_{q=0}^{p-1} \left(\sum_{i=r_q}^{r_{q+1}} c_i \right)$ of the most expensive subsequence is minimal.

In [18] an algorithm is described which solves this problem in time $\mathcal{O}(mp)$.

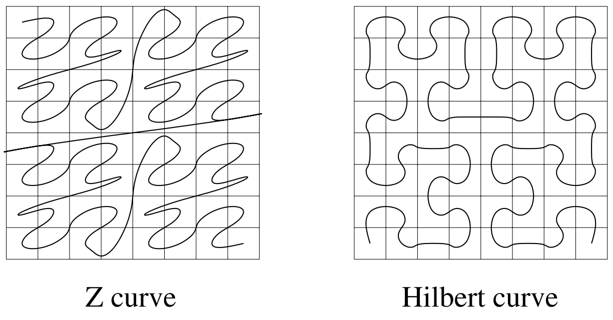

Figure 4: Examples for space filling curves

Sequence partitioning is used in Section 5.2 to distribute the $\mathcal{H}$-matrix. For this an order has to be defined for the matrix blocks or leaves of the block cluster tree T. One way to define such an ordering is by mapping T to the unit-square and applying *space-filling curves* (see [19]). These curves describe a surjective mapping from the unit interval $[0,1]$ to the unit square $[0,1]^2$. In Figure 4 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 hence through all leaves of T. The order in which each node is reached defines the sequence used in the sequence partitioning algorithm (see Figure 5).

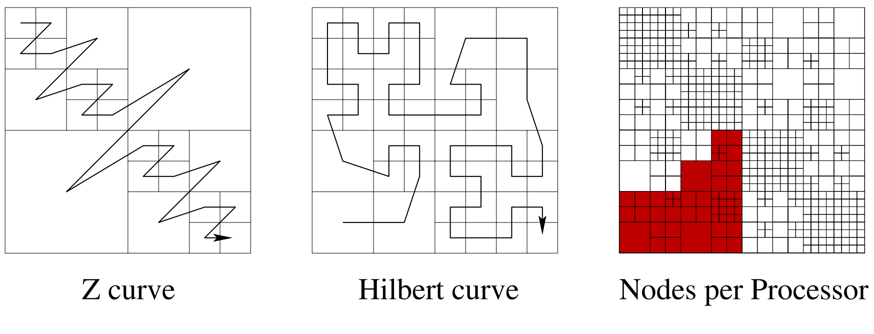

Figure 5: Space filling curves applied to $\mathcal{H}$-matrices

For quad-trees, as they are considered in this article, this ordering can be computed by using a depth first search (see [20]) on T. The order in which the sons of a node are traversed defines the type of the space-filling curve. For this, each node is associated with a *mark*. Depending on this mark, the ordering and the corresponding marks of the sons are set. Figure 6 shows this construction for the Z and the Hilbert curve. The root of the block cluster tree is marked with an “A” in both cases.

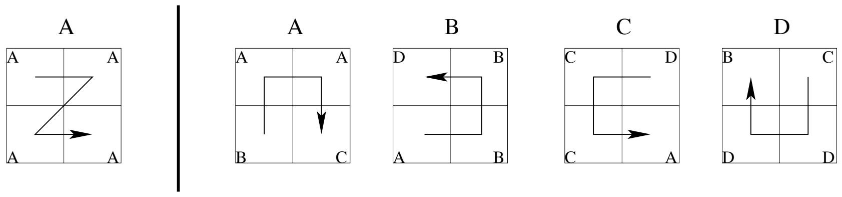

Figure 6: Construction of space filling curves: Z (left) and Hilbert (right)

An interesting property of this kind of distribution is the "compactness" of the set of leaves, scheduled to a specific processor (see Figure 5, right). This behaviour is due to the neighbourhood relationship of adjacent subintervals of space-filling curves, especially the Hilbert-curve. This property will be of importance for the complexity of the parallel matrix-vector multiplication in Section 5.2.

4.3 Quality of the Scheduling

Let t_{seq} be the time needed by a sequential algorithm to solve a specific problem. The quality of the distribution computed by list scheduling is measured w.r.t. to the best possible scheduling. Unfortunately this optimal distribution is not necessarily in the order of t_{seq}/p, i.e., resulting in an optimal speedup of a parallel algorithm. Consider for example a set of jobs, with m tasks of cost 1 and one task of cost $m' \gg m$. Obviously the total time is dominated by this large job, independently of the number of processors.

The same is valid for sequence partitioning. The optimal partition does not guarantee a perfect speedup if the costs of the jobs differ too much.

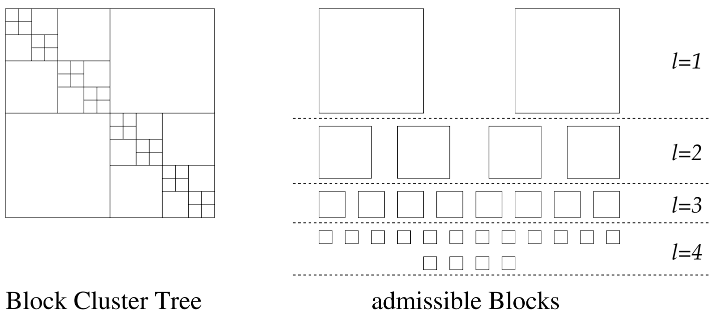

Figure 7: Cost distribution for $\mathcal{H}$-matrices

Fortunately in the context of $\mathcal{H}$-matrices the distribution of the costs involved is not arbitrary but dependent on the hierarchical decomposition of the index set. Consider for example the format depicted in Figure 7 where all block clusters with different index sets are admissible. Since $T(I)$ is a cardinality balanced cluster tree, the resulting $\mathcal{H}$-matrix contains the same amount of data on each level, save the first one (see [13]). Hence, for $p = l - 1$ a perfect load balancing is possible if for instance matrix-vector multiplication is considered. Since the number of processors is limited in this paper (see Section 3), this case occurs even for small problem sizes.

This observation gives rise to the conjecture that almost all practical algorithms in the context of $\mathcal{H}$-matrices can be perfectly distributed on shared memory systems. Therefore in the following sections for an algorithm $\mathfrak{A}$ with a sequential complexity $\mathcal{W}_{\mathfrak{A}}$ it is assumed that the optimal parallel complexity $\mathcal{W}_{\mathfrak{A},\mathrm{opt}}(p)$ is given by

$$\mathcal{W}_{\mathfrak{A},\mathrm{opt}}(p) = \frac{\mathcal{W}_{\mathfrak{A}}}{p}. \tag{4.2}$$

5 Parallel Arithmetics

Parallelising a sequential algorithm often results in a complete rewrite to achieve an optimal performance. In contrast to this, the following parallel algorithms were designed with a special emphasis on reusing as much as possible from an existing sequential implementation. Of course, an adequate *parallel efficiency* was also sought.

Definition 5.1 *Let $t(p)$ denote the time needed by a parallel algorithm $\mathfrak{A}$ 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 $\mathfrak{A}$.*

For the remainder of this section, let $I = \{0, \ldots, n-1\}$ be an index set and $T(I)$ a cluster tree over I. Furthermore let $T = T(I \times I)$ be a block cluster tree based on $T(I)$. The $\mathcal{H}$-matrices which are considered in the following are not restricted to a fixed rank, but might also be defined by a fixed precision (see Section 2). Therefore, the complexity estimates are only given w.r.t. corresponding sequential formulae. For the exact estimates in the case of a constant rank, please refer to [11].

5.1 Matrix Construction

Building the $\mathcal{H}$-matrix $A \in \mathcal{H}(T)$ is usually accomplished by constructing the hierarchy which is associated to inner nodes of T, i.e., nodes which are not a leaf, and building R-matrices for all admissible leaves and dense matrices for all non-admissible leaves, respectively. This is shown in the following recursive sequential algorithm:

```
procedure build_matrix( $b, T$ )
    if $b \in \mathcal{L}(T)$ then
        if $b$ is admissible then build R-matrix;
        else build dense matrix;
    else
        build block matrix;
        for all $v' \in \mathcal{S}(v)$ do build_matrix( $v', T$ );
endif; end;
```

As the computationally intensive part of this computation is concentrated in the matrix blocks corresponding to leaves, it suffices to distribute this part of the algorithm. Furthermore, since the cost for computing a matrix block is not necessarily known, e.g., as in the BEM example considered in this paper, online scheduling techniques are preferred. The list scheduling algorithm in Section 4.1 can be used by moving the leaf-related computation into a special function which is then executed in a thread of the thread pool:

```
procedure build_leaf( $b$ )
    if $b$ is admissible then build R-matrix;
    else build dense matrix;

procedure build_matrix( $b, T$ )
    if $b \in \mathcal{L}(T)$ then run( build_leaf( $b$ );
    else build block matrix;
        for all $v' \in \mathcal{S}(v)$ do build_matrix( $v', T$ );
endif; end;
```

Algorithm 5.1: Parallel Matrix-Construction using List Scheduling

Assuming a constant time for the construction of a block matrix and by using (4.1), the following result for the execution time of Algorithm 5.1 is obtained.

Lemma 5.2 *The building of an $\mathcal{H}$-matrix $A \in \mathcal{H}(T)$ using Algorithm 5.1 with p processors has complexity of*

$$\mathcal{W}_{MB}(n,p) = \mathcal{O}\left(|V(T) \setminus \mathcal{L}(T)| + \frac{\mathcal{W}_{MB}(n,1)}{p} \right). \tag{5.1}$$

Proof: Block matrices are built in Algorithm 5.1 for all inner nodes of the block cluster tree, which results in $\mathcal{O}(|V(T) \setminus \mathcal{L}(T)|)$. Together with the complexity of computing the matrix blocks corresponding to leaves (5.1) is obtained. $\square$

If only the matrix blocks corresponding to leaves are needed, e.g., for the matrix-vector multiplication, the building of block-matrices can be skipped in Algorithm 5.1. This also leads to an optimal parallel complexity of

$$\mathcal{O}\left(\frac{\mathcal{W}_{\text{MB}}(n,1)}{p} \right)$$

for the resulting algorithm.

The following two tables prove this theoretical result. In the first table, the $\mathcal{H}$-matrix defined by the BEM example described in Section 2.1 is build with a constant rank $k = 10$.

Matrix building, fixed rank $k = 10$										
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	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

As one can see, the parallel efficiency is almost perfect. Even for small problem sizes, an optimal speedup is achieved indicating a small overhead induced by the thread pool.

In the second table the same example is used to built an $\mathcal{H}$-matrix with a fixed precision of $\varepsilon = 10^{-4}$.

Matrix building, fixed precision $\varepsilon = 10^{-4}$										
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 results are similar to those in the case of a constant rank. Hence, list scheduling provides a very efficient method to parallelise the construction of $\mathcal{H}$-matrices.

5.2 Matrix-Vector Multiplication

In the sequential case the matrix-vector multiplication

$$y := \alpha A x + \beta y \tag{5.2}$$

with an $\mathcal{H}$-matrix $A \in \mathcal{H}(T)$, vectors $x, y \in \mathbb{R}^I$ and $\alpha, \beta \in \mathbb{R}$ is performed by looping over the set of matrix blocks M corresponding to a leaf (τ, σ) in T and computing the local product[1] $y|_\tau := \alpha M x|_\sigma + y|_\tau$.

Unfortunately, applying list scheduling to this algorithm implies some kind of write-protection of y (or parts of it), e.g., in the form of mutices, since more than one processor could access the same part of y while applying a processor-local result. This potentially leads to a blocking of processors, which should be avoided to maintain a sufficient parallel efficiency.

Alternatively the multiplication could be split into two phases. At first each processor computes all local products and collects them in a temporary vector $y'_q := \alpha A_q x$ with $y'_q \in \mathbb{R}^I, 0 \leq q < p$ and A_q denoting the part of A containing all matrices assigned to processor q. This computation is completely independent resulting in an optimal complexity of

$$\frac{\mathcal{W}_{\mathrm{MV}}(n, 1)}{p} \quad .$$

Afterwards the sum $y := y + \sum_{q=0}^{p-1} y'_q$ is computed in parallel. At this, the vectors x and y are assumed to be distributed among the processors, each holding a sub-vector x_i and y_i of size n/p. Unfortunately, all processors could contribute to every coefficient of y and hence, the complexity of the summation alone is $\mathcal{O}((pn)/p) = \mathcal{O}(n)$, spoiling the parallel efficiency.

This property of the list scheduling distribution is due to the unpredictable assignment of matrix blocks to processors. That way, for a specific processor q the matrices in A_q can stem from all parts of the block index set $I \times I$ as shown in Figure 8 (left). What is sought instead, is a more "compact" distribution of the matrix blocks, which can be obtained by using sequence partitioning and space-filling curves as described in Section 4.2 (Figure 8, right).

As explained above, the crucial part of a load balancing of the matrix blocks or block clusters, respectively, is the number of processors contributing to a specific coefficient of y. This is expressed in the *sharing degree* of a distribution:

[1]The scaling of y by β is assumed to be done beforehand

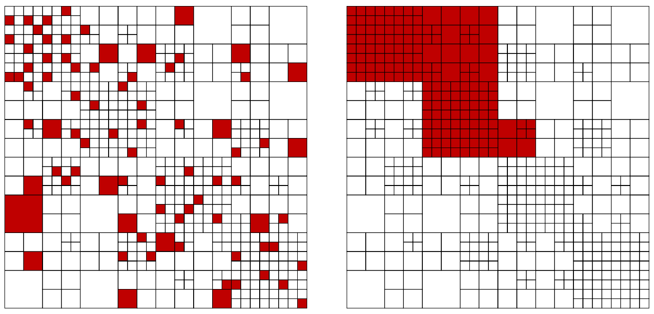

Figure 8: Matrices scheduled to the same processor with List scheduling (left) and sequence partitioning (right)

Definition 5.3 *Let $\mathcal{L}_q \subseteq \mathcal{L}(T)$ denote the block clusters assigned to processor q. For $i \in I$ define the* sharing degree $d_{\rm sh}(i)$ *of i as*

$$d_{\rm sh}(i) = \max\{d^r_{\rm sh}(i), d^c_{\rm sh}(i)\}, \tag{5.3}$$

where $d^r_{\rm sh}(i) = |\{q \,|\, \exists(\tau,\sigma) \in \mathcal{L}_q : i \in \tau\}|$ and $d^c_{\rm sh}(i) = |\{q \,|\, \exists(\tau,\sigma) \in \mathcal{L}_q : i \in \sigma\}|$. Furthermore, let $d_{\rm sh} = \max_{i \in I} d_{\rm sh}(i)$ be the sharing constant of I.

For dense matrices the sharing degree is $\sqrt{p}$ (see [17]), whereas for typical sparse matrices, defined by a FEM discretisation, $d_{\rm sh}$ equals the number of entries per row, which is usually bounded by a constant.

In Figure 9 the value of $d_{\rm sh}$ for list scheduling and sequence partitioning with space-filling curves is shown for a fixed problem size and an increasing number of processors. In the case of list scheduling, for $p \le 50$ the sharing degree equals p. For larger values of p, the number of blocks per processor gets smaller, thereby reducing the probability of contributing to an index. Hence, the dependence of $d_{\rm sh}$ is more of order $\sqrt{p}$.

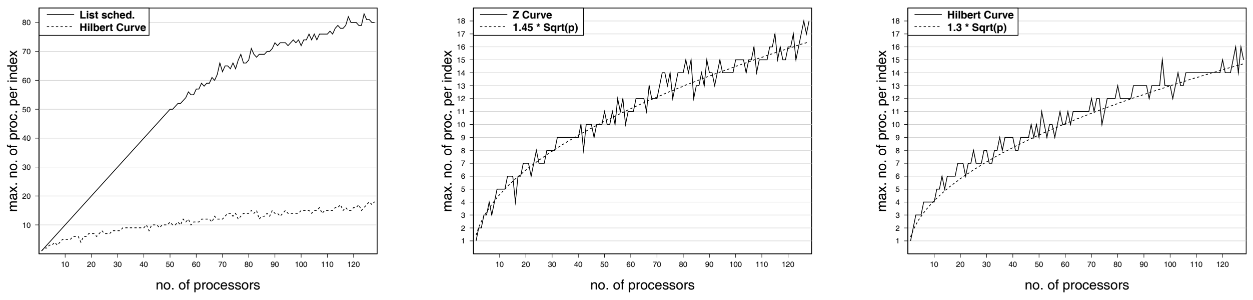

Figure 9: Value of $d_{\rm sh}$ for List scheduling and seq. partitioning with Z and Hilbert curve

The latter behaviour of $d_{\rm sh}$ is visible from the beginning when using sequence partitioning with space-filling curves. In this case, distributing the block cluster tree using the Hilbert curve has a slight advantage compared to the Z curve.

Due to this results, the parallel matrix-vector multiplication seems to be more efficient when using sequence partitioning, i.e., with an offline load balancing algorithm. The costs per matrix or block cluster, respectively, which are needed for this type of scheduling, are defined by the representation of the matrix block $M \in \mathbb{R}^{\tau \times \sigma}$:

$$c_M = \begin{cases} k(M)(|\tau| + |\sigma|), & (\tau \times \sigma) \text{ is admissible} \\ |\tau| \cdot |\sigma|, & \text{else} \end{cases},$$

where $k(M)$ is the rank of M if using a R-representation.

For the final, parallel algorithm let $I_{q,\text{vec}}$ denote the part of the index set associated to each processor $0 \le q < p$, as it is defined by the distribution of the vector y and $I_{q,\text{mat}} = \cup_{(\tau,\sigma)\in\mathcal{L}_q} \tau$ the indices to which A_q contributes. The algorithm makes use of the two-step method: first the local result is computed and afterwards these results are summed up in parallel.

```
procedure mv_mul( $i, \alpha, A, x, \beta, y$ )
    procedure mat_vec ( $i, \beta, y_i, A_i, x$ )
        $y_i := \beta \cdot y_i$; $y'_i := \alpha A_i x$;
    end;

    procedure sum ( $i, y_i$ )
        $Y_i := \{y'_j \mid I_{i,\text{vec}} \cap I_{j,\text{mat}} \neq \emptyset\}$;
        $y_i := y_i + \sum_{y'_j \in Y_i} y'_j|_{I_{i,\text{vec}}}$;
    end;

    for $0 \le i < p$ do run( mat_vec( $i, \beta, y, A, x$ ) ); endfor sync_all();
    for $0 \le i < p$ do run( sum( $i, y_i$ ) ); endfor sync_all();
end;
```

Algorithm 5.2: Parallel matrix-vector multiplication

The complexity of the parallel matrix-vector multiplication can be estimated as:

Lemma 5.4 *Computing the matrix-vector product (5.2) with an $\mathcal{H}$-matrix $A \in \mathcal{H}(T)$ with Algorithm 5.2 on p processors has a complexity of:*

$$\mathcal{W}_{MV}(n,p) = \mathcal{O}\left(\frac{\mathcal{W}_{MV}(n,1)}{p} + \frac{d_{\text{sh}} \cdot n}{p} \right). \tag{5.4}$$

Using the $\sqrt{p}$ behaviour of d_{sh} in the case of sequence partitioning, (5.4) becomes

$$\mathcal{W}_{\text{MV}}(n,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\text{MV}}(n,1)}{p} + \frac{n}{\sqrt{p}} \right).$$

For the limited range of processors considered in this paper, the first term in (5.4) should dominate the overall complexity of the matrix-vector multiplication. This is also visible in the following numerical examples. In the first table the time for 100 multiplications of an $\mathcal{H}$-matrix with a fixed rank $k = 10$ defined by the BEM example is presented.

Matrix-Vector multiplication, fixed rank $k = 10$									
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 low efficiency for small problem sizes is not due to the summation but comes from other sequential parts in the algorithm, e.g., management overhead. Since that part remains constant if n is increased, the parallel efficiency grows and stabilises at about 80-90 %.

The same experiment is done with an $\mathcal{H}$-matrix of a fixed precision ($\varepsilon = 10^{-4}$). The results are shown in the second table.

Matrix-Vector multiplication, fixed precision $\varepsilon = 10^{-4}$										
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

Again, the parallel performance of the matrix-vector multiplication algorithm increases if an $\mathcal{H}$-matrix corresponding to a larger problem size is used.

These results demonstrate the applicability of Algorithm 5.2 for the matrix-vector multiplication on shared memory systems. Furthermore, the same algorithm can also be used in a distributed memory environment, e.g., a cluster, as is shown in [2].

5.3 Matrix-Multiplication

In this section the multiplication of two $\mathcal{H}$-matrices $A, B \in \mathcal{H}(T)$ is considered, but instead of restricting the discussion to this basic operation, the more general update of a matrix $C \in \mathcal{H}(T)$

$$C := \alpha AB + \beta C \tag{5.5}$$

with $\alpha, \beta \in \mathbb{R}$ is used.

The sequential matrix multiplication can be implemented by using a recursive algorithm based on the multiplication of block-matrices as is shown in the following procedure:

```
procedure mul ( $\alpha, A, B, \beta, C$ )
    procedure rec_mul ( $\alpha, A, B, C$ )
        if $A, B$ and $C$ are block-matrices then
            for $i, j, l \in \{0, 1\}$ do rec_mul( $\alpha, A_{il}, B_{lj}, C_{ij}$ );
        else
6:          $C := C + \alpha AB$;
    end;

    $C := \beta C$; rec_mul( $\alpha, A, B, C$ );
end;
```

Algorithm 5.3: Sequential matrix multiplication

The precise definition of the multiplication in line 6 is not considered. This can be done by computing the exact product or by truncating the result to a fixed rank or a fixed precision. The recursion is stopped if either a factor or the destination matrix is a leaf. It should be noted, that since admissible blocks occur at more than one level in the block cluster tree, this stopping criterion leads to a different depth of the recursion in different parts of the $\mathcal{H}$-matrix.

A straight-forward modification of Algorithm 5.3 using list scheduling is to execute the multiplications in line 6 on a free processor. Unfortunately this potentially leads to a lot of collisions if different processors attempt to write to the same matrix-block. Consider for example the following multiplication:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} := \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} + \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

For the matrix C_{00} one gets the following two operations

$$C_{00} := C_{00} + A_{00}B_{00} \quad \text{and } C_{00} := C_{00} + A_{01}B_{10}.$$

which might be executed on different processors resulting in a concurrent write to C_{00}, which therefore has to be protected by a mutex. But this potentially leads to a sub-optimal parallel efficiency of the multiplication algorithm.

If, on the other hand, all multiplications with the same destination, e.g., C_{00}, are scheduled to the same processor, no collision can occur. This idea is used in the following to distribute the complete multiplication. At first all products contributing to a specific matrix block have to be identified. This can be done by a simulation process which is almost identical to the recursive multiplication Algorithm 5.3 itself, only instead of computing the product, the involved matrices are stored:

```
procedure sim ( $A, B, C$ )
    if $A, B$ and $C$ are block-matrices then
        for $i, j, l \in \{0, 1\}$ do sim( $A_{il}, B_{lj}, C_{ij}$ );
    else
        $P_C := P_C \cup \{(A, B)\}$; $\mathcal{L}_{\mathrm{MM}} := \mathcal{L}_{\mathrm{MM}} \cup \{C\}$;
    endif;
end;
```

Algorithm 5.4: Identify all products for a matrix block

After completing the simulation, the set P_C contains all pairs of factors contributing to the matrix C. Furthermore, all blocks which occur as destination matrices during the multiplication are stored in the set $\mathcal{L}_{\mathrm{MM}}$. Due to the stopping criterion in Algorithm 5.3, this set is not identical to $\mathcal{L}(T)$ but a superset.

The actual matrix multiplication is afterwards accomplished by using list scheduling on $\mathcal{L}_{\mathrm{MM}}$.

```
procedure mul_tp( $\alpha, \beta, T, \mathcal{L}_{\mathrm{MM}}$ )
    procedure scale ( $\beta, C$ ) $C := \beta C$;

    procedure mul_block( $\alpha, C$ )
        for all $(A, B) \in P_C$ do
            $C := C + \alpha AB$

    for all $C' \in \mathcal{L}(T)$ do run( scale( $\beta, C'$ ) ); endfor; sync_all();
    for all $C' \in \mathcal{L}_{\mathrm{MM}}$ do run( mul_block( $\alpha, C'$ ) ); endfor sync_all();
end;
```

Algorithm 5.5: Parallel matrix multiplication

Unfortunately, due to the stopping criterion in Algorithm 5.3 this parallel multiplication method still can produce concurrent access to the data of a matrix block, as is shown in the example in Figure 10. For the matrix C_{00} one obtains the set $P_{C_{00}} = \{(A_{01}, B_{10})\}$. In the same way,

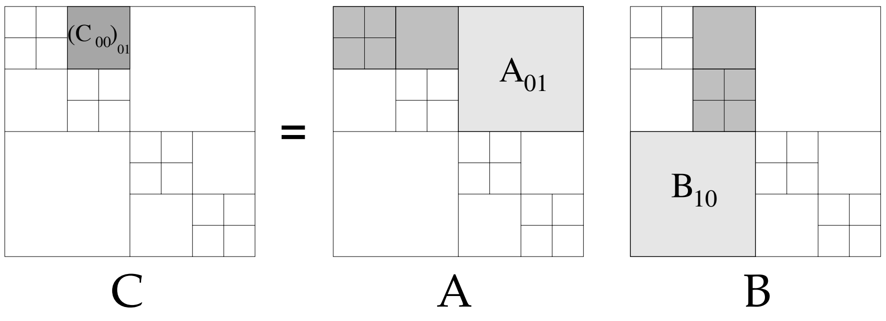

Figure 10: Concurrent access of matrix data

$P_{(C_{00})_{01}} = \{((A_{00})_{00}, (B_{00})_{01}), ((A_{00})_{01}, (B_{00})_{10})\}$ is determined for the matrix $(C_{00})_{01}$. Since $(C_{00})_{01}$ is a subblock of C_{00}, computing the products for both matrices in parallel, therefore potentially leads to a concurrent write-access to $(C_{00})_{01}$. Since this situation was rarely observed in practical computations, a mutex suffices to ensure correct data without spoiling the parallel efficiency of the matrix multiplication.

Lemma 5.5 *The parallel computation of the product (5.5) by Algorithm 5.4 and Algorithm 5.5 on a parallel computer with p processors has a complexity of:*

$$\mathcal{W}_{MM}(n,p) = \mathcal{O}\left(c_{\mathrm{sp}}(T)|V(T)| + \frac{\mathcal{W}_{MM}(n,1)}{p}\right) \tag{5.6}$$

Proof: Since only nodes of T can occur in the simulation algorithm, the size of the set $\mathcal{L}_{\mathrm{MM}}$ is bounded by the size of the block cluster tree: $|\mathcal{L}_{\mathrm{MM}}| \leq |V(T)|$.

Let $b(M) = (\tau, \sigma)$ denote the corresponding block cluster in T of a matrix $M \in \mathbb{R}^{\tau\times\sigma}$. For a matrix C the stopping criterion in Algorithm 5.4 ensures

$$\forall\, (A,B) \in P_C : b(C) \in T^{(i)} \implies (b(A) \in T^{(i)} \wedge b(B) \in T^{(i)}),$$

i.e., all factors are on the same level of T. Hence, $|P_C| \leq c_{\mathrm{sp}}(T)$ holds for all C and the execution time of Algorithm 5.4 is bounded by $\mathcal{O}\left(c_{\mathrm{sp}}(T)|V(T)|\right)$. By using (4.1) the final result (5.6) is obtained. $\square$

Although (5.6) still has a sequential part due to the simulation of the matrix multiplication, this part is associated with a very small constant. Hence for the considered range of processors this term can be neglected, leading to an almost perfect parallel efficiency of the multiplication of $\mathcal{H}$-matrices.

Matrix multiplication, fixed rank $k = 7$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	72.9	18.6	98.0	9.8	93.0	8.1	75.0	5.1	90.0
16 384	564.3	142.5	98.7	73.7	95.6	58.8	80.0	38.2	92.3
65 536	3775.1	949.0	99.3	496.9	95.0	345.4	91.1	257.7	91.6
262 144	23062.1	5878.2	98.1	3048.3	94.6	2084.6	92.2	1591.2	90.6
1 048 576	132373.7	33357.5	99.2	17372.7	95.2	12029.9	91.7	9213.7	89.8

In the first example the time for multiplying matrices from the FEM example with a constant rank of $k = 7$ is measured. One can see a slight drop of the parallel efficiency with a higher processor number, although it stabilises at about 90 %. This behaviour is not due the determination of multiplication factors for each matrix block which is done sequentially since running Algorithm 5.4 takes less then 5 seconds even for the largest problem size.

Instead, the characteristics of the computer system have a large influence on the results. Especially since the computation of the matrix product involves many accesses to the global memory, a non-optimal implementation of a PRAM leads to the results in the table. In particular, the low efficiency in the case of $p = 12$ for smaller problem sizes seems to be directly related to the computer hardware (see also [15]).

The same behaviour is visible in the second example with an $\mathcal{H}$-matrix of a fixed precision of $\varepsilon = 10^{-6}$.

Matrix multiplication, fixed precision $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	44.8	12.4	90.3	6.2	90.3	4.9	76.2	3.3	84.8
16 384	376.1	101.8	92.1	51.5	90.9	41.8	74.6	26.3	89.4
65 536	2526.0	682.9	92.5	344.6	91.6	232.4	90.6	172.2	91.7
262 144	15089.6	4096.2	92.1	2057.5	91.7	1415.7	88.8	1041.2	90.6
1 048 576	84737.3	22617.4	93.7	11320.8	93.6	7706.1	91.6	5774.2	91.7

The dependency of the execution time from the number of processors is not as large as in the previous case, especially since the drop of the parallel efficiency is already visible for $p = 4$.

Nevertheless, both experiments show a nearly optimal speedup of Algorithm 5.5 for the parallel matrix multiplication.

5.4 Matrix-Inversion

For a 2×2 block matrix

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

the inverse $C = A^{-1}$, if it exists, can be written as

$$C = \begin{pmatrix} A_{00}^{-1} + A_{00}^{-1} A_{01} S^{-1} A_{10} A_{00}^{-1} & -A_{00}^{-1} A_{01} S^{-1} \\ -S^{-1} A_{10} A_{00}^{-1} & S^{-1} \end{pmatrix},$$

with the *Schur complement*

$$S = A_{11} - A_{10} A_{00}^{-1} A_{01}.$$

This representation results from a Gaussian elimination applied to A and leads to the well known (see [13] or [11]) algorithm

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

Algorithm 5.6: Computation of the inverse of an $\mathcal{H}$-matrix

As one can see, it only consists of recursive calls and matrix multiplications in the form of (5.5). Hence, a parallelisation ansatz is to distribute these multiplications as was described in the previous section, albeit this cannot lead to an optimal parallel efficiency. This is due to the intrinsic, sequential nature of the inversion algorithm where the computation of the diagonal element c_{ii} of the matrix C depends on the coefficients c_{jj} for $j < i$.

On the other hand, it has to be recalled, that the number of processors is of a limited range. So the hope is, that at least for this range, the sequential part of the inversion does not dominate the computation.

Nevertheless, the complexity of the hereby defined parallel algorithm can be analysed without this discussion. For that only the quasi-optimal result (5.6) is necessary.

Lemma 5.6 *The parallel computation of the inverse of an $\mathcal{H}$-matrix $A \in \mathcal{H}(T)$ on p processors using Algorithm 5.6 has a complexity of*

$$\mathcal{W}_{inv}(n,p) = \mathcal{O}\left(n_{\min}^2 n + \frac{\mathcal{W}_{inv}(n,1)}{p}\right) \tag{5.7}$$

Proof: Since only diagonal blocks are inverted in Algorithm 5.6, the recursion stops if A is of size $n_{\min} \times n_{\min}$. The costs for inverting a matrix block of this size can be estimated by $\mathcal{O}\left(n_{\min}^3\right)$. Together with $n/n_{\min}$, the number of matrices of this size, the first term in (5.7) is gained.

The remaining complexity is defined by the multiplications in Algorithm 5.6 and can be estimated by $\mathcal{W}_{\text{inv}}(n,1)$. Since these multiplications have a perfect parallel efficiency (see (5.6)), (5.7) is completed. □

In practice however, not all multiplications can be parallelised with optimal parallel efficiency as is assumed so far. Especially the multiplication of small blocks, or even leaves, cannot be distributed to p processors. Therefore, a minimal size n_0 of the matrices is introduced, which defines a switching point from working in parallel and doing all computations sequentially. The complexity of Algorithm 5.6 has to be modified in this case resulting in

$$\mathcal{W}_{\text{inv}}(n,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\text{inv}}(n_0,1)\cdot n}{n_0} + \frac{\mathcal{W}_{\text{inv}}(n,1)}{p}\right).$$

Still open is the question of the influence of the sequential part in the parallel inversion algorithm. This is best answered by numerical experiments for which the results of the FEM model problem are presented below. At first an $\mathcal{H}$-matrix with a fixed rank of $k = 7$ was inverted.

Matrix inversion, fixed rank $k = 7$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	27.3	7.6	89.8	4.4	78.4	3.1	73.4	2.6	65.6
16 384	197.3	53.8	91.7	28.8	85.6	20.9	78.7	17.3	71.3
65 536	1237.2	329.2	94.0	179.4	86.2	132.9	77.6	110.8	69.8
262 144	7264.0	1937.0	93.8	1041.6	87.2	757.0	80.0	640.7	70.9
1 048 576	40316.4	10674.4	94.4	5849.4	86.2	4328.8	77.6	3619.3	69.6

As the results show, the parallel efficiency is dependent on the number of processors. The results are also consistent throughout all level numbers, save the first. For $p = 16$ an efficiency of about 70 % is achieved, which demonstrates the usability of the parallel Gaussian elimination.

In the next table the results for inverting an $\mathcal{H}$-matrix with a fixed precision of $\varepsilon = 10^{-6}$ are presented.

Matrix inversion, fixed precision $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	12.8	3.6	88.5	2.1	76.3	1.6	65.9	1.4	56.4
16 384	114.5	30.8	92.9	17.5	81.8	13.0	73.4	11.5	62.2
65 536	806.0	213.2	94.5	119.8	84.1	91.7	73.2	78.9	63.8
262 144	4993.3	1330.5	93.8	745.1	83.8	562.0	74.0	487.1	64.1
1 048 576	28653.5	7610.3	94.1	4298.4	83.3	3275.0	72.9	2795.8	64.1

The results are only slightly worse than in the previous case with a fixed rank. Again, the parallel efficiency for $p = 16$ indicates, that the simple parallelisation method used in Algorithm 5.6 leads to an applicable parallel inversion algorithm.

Remark 5.7 *By using the Newton iteration as it was described in [14] to invert an $\mathcal{H}$-matrix, a parallel algorithm with an optimal parallel efficiency can be obtained since only matrix multiplications are involved. In practice however, the Gaussian elimination is more efficient, i.e., less time consuming, which can also be seen by comparing the numerical results in the tables above with the corresponding results in the previous subsection.*

References

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

[2] M. Bebendorf and R. Kriemann: *Fast Parallel Solution of Boundary Integral Equations and Related Problems.* Preprint 10/2004, MPI Leipzig.

[3] S. Börm and L. Grasedyck and W. Hackbusch: *Introduction to Hierarchical Matrices with Applications.* Engineering Analysis with Boundary Elements 27, pp. 405–422, 2003.

[4] S. Börm and L. Grasedyck and W. Hackbusch: *Hierarchical Matrices.* Lecture note 21, MPI Leipzig, 2003.

[5] D. Braess: *Finite Elements. Theory, Fast Solvers and Applications in Solid Mechanics*, Cambridge University Press, 2001.

[6] A. Brandt: *Multilevel computations of integral transforms and particle interactions with oscillatory kernels.* Comput. Phys. Comm., 65(1-3), pp. 24–38, 1991.

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

[8] S. Cook and R. Reckhow: *Time Bounded Random Access Machines.* Journal of Computer and Systems Sciences, Vol. 7, pp. 354–375, 1973.

[9] S. Fortune and J. Wyllie: *Parallelism in random access machines.* Proceedings of the tenth annual ACM symposium on Theory of Computing (San Diego, CA), pp. 114–118, 1978, ACM Press.

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

[11] L. Grasedyck and W. Hackbusch: *Construction and Arithmetics of $\mathcal{H}$-Matrices.* Computing 70, pp. 295–334, 2003.

[12] W. Hackbusch: *Integral equations: Theory and numerical treatment.* Vol. 128 of ISNM, Birkhäuser, Basel, 1995.

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

[14] W. Hackbusch and B.N. Khoromskij: *A sparse $\mathcal{H}$-matrix arithmetic: general complexity estimates.* J. Comput. Appl. Math. 125, 479–501, 2000.

[15] R. Kriemann, J. Burmeister and R. Kleinrensing: *Benchmarking a Shared Memory System.* Technical Report and Documentations 1/2003, MPI Leipzig.

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

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

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

[19] H. Sagan: *Space-Filling Curves.* Springer-Verlag, 1994.

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