Abstract. In this paper we consider parallel implementations of approximate multiplication of large matrices with exponential decay of elements. Such matrices arise in computations related to electronic structure calculations and some other fields of science. Commonly, sparsity is introduced by truncation of input matrices. In turn, the sparse approximate multiplication algorithm [M. Chalacombe and N. Bock, arXiv preprint 1011.3534, 2010] performs truncation of sub-matrix products. We consider these two methods and their combination, i.e. truncation of both input matrices and sub-matrix products. Implementations done using the Chunks and Tasks programming model and library [E. H. Rubensson and E. Rudberg, Parallel Comput., 40:328343, 2014] are presented and discussed. The absolute error asymptotic behavior is derived. A comparison between the three methods in terms of performance is done on a model problem. The algorithms are also applied to matrices coming from large chemical systems with \(\sim 10^6\) atoms.

1. Introduction. Although dense matrix-matrix multiplication is a well-known procedure, its cubic complexity makes it very resource-demanding or even infeasible to use for large matrices. Much effort has been invested in finding better algorithms with reduced complexities, see for instance [45], [19]. However, when it comes to sparse matrices, the methods used in the community are very similar in terms of performance due to data formats.

As a rule, in sparse matrix algorithms, matrices are stored in special formats, such as compressed sparse rows (CSR) or compressed sparse columns (CSC) [44]. The drawback of these formats is that once one of them is chosen, operations with matrix transpose become difficult to perform in parallel. Using the idea of blocking, Buluç et al. [15] suggested compressed sparse blocks (CSB) format, which solves this problem, at least partially. Another application of blocking technique is the blocked CSR format (BCSR), which works better for matrices with zeros occurring in blocks [30]. All these storage formats and sparse matrix algorithms are designed for common-sense sparse matrices, i.e. such that have a few non-zero elements per row.

The problem of our interest falls somewhere in between dense and sparse matrix multiplication. In electronic structure calculations based on the Hartree–Fock method or the Kohn–Sham density functional theory, a key component is computation of a density matrix \(D\) for a given Hamiltonian \(\mathcal{F}\). A popular method is to perform a polynomial approximation of the function \(D = \theta(\mu I - \mathcal{F})\), where \(\theta\) is the Heaviside step function and \(\mu\) is the chemical potential. This can be done in different ways. The two commonly used approaches are recursive application of low-order polynomials (density matrix purification, [34, 29]) and construction of Chebyshev polynomials [26, 5, 33]. Depending on the implementation, the core operation is often a matrix-matrix multiplication, thus it is the crucial component for overall performance and scaling of the code.

Matrices in electronic structure calculations have an important property of decay of elements. A matrix \(A\) is said to obey an exponential decay with respect to \(d(i,j)\) if \(|a_{i,j}| \leq c \lambda^{-\alpha d(i,j)}\) for some constants \(c > 0\) and \(\alpha > 0\) or an algebraic decay if \(|a_{i,j}| \leq c (d(i,j)^\lambda + 1)^{-1}\), where \(d(i,j)\) is a distance function defined on the index set.
of the matrix. The distance function corresponds to physical distance between some a priori chosen basis functions. It is convenient to look at such chemical system as a spatial graph, where distance function is defined on the edges. Under certain assumptions small matrix elements can be ignored, as usually done to matrices arising in linearly scaling electronic structure calculations [32, 16, 40, 12]. In this context linear scaling means that the computational time is directly proportional to the size of the considered chemical system. Such matrices are not sparse in the common sense, since they may have from several to several thousands non-zero elements per row, depending on the type of basis functions used, and thus standard sparse matrix algorithms are not applicable here. However, the special structure of matrices with decay allows to achieve linear scaling, hence the name for the group of methods. Ideally, with linearly increasing computational power, it should give constant computational time. However, this is difficult to achieve due to communication costs, which start to play a significant role at some point.

The number of non-zero elements per row and decay properties dictate also that the storage formats listed above are unlikely to be usable in this context, because their strong point, i.e. compression, does not work well enough in this case. One alternative is introduced by Mohr et al. [33], where they use the so-called segment storage format (SSF). This format is based on the idea of grouping together consecutive non-zero entries in segments. Another approach is to exploit the hierarchical structure of a matrix, i.e. treat it as a matrix of matrices [38]. Similar ideas are used in [9, 37]. This representation is natural not only for the matrices which arise in electronic structure calculations, but in a wider class of problems including, but not limited to integral equations, partial differential equations, matrix equations and many more, see [7, 28] for extensive study. However, the notion of a hierarchical matrix used by Hackbusch [28] is different, since it is a data-sparse approximation of matrices, which are not necessarily sparse by construction.

Multiplication of matrices, which have more non-zero elements per row, than standard sparse matrices, or, in other words, nearly sparse matrices, is a challenging problem, especially since in general the sparsity pattern is not known in advance. In this case, load balancing becomes a substantial difficulty.

One of the first parallel implementations is done by Challacombe [16]. The author uses conventional MPI and a special data format with distributed blocked compressed sparse rows. The load balancing and locality issues are addressed by introducing a space filling curve, which is a heuristic solution for the travelling salesman problem. The efficiency is demonstrate up to 128 cores.

Better results are achieved by using a space filling curve ordering with two-dimensional matrix decompositions, for instance VandeVondele et al. [46] report scalability of \( \sim 10^6 \) atoms on 46556 cores.

An alternative approach is suggested by Buluç and Gilbert [14]. They address load balancing issues by randomly permuting rows and columns, which leads to high efficiencies in the general case. This approach is successfully applied in electronic structure calculations by Borštnik et al. [11].

Dawson and Nakajima [20] employ communication-avoiding 3D multiplication algorithm by Ballard et al. [6] in their NTPoly code. The load balancing is addressed in the same manner using permutations as in [14].

Rubensson and Rudberg [37] present a way of multiplying sparse matrices in parallel, preserving an important property of locality. The approach is based on a hierarchical representation of matrices and a task-based parallel programming model called Chunks and Tasks [36]. The load balancing issue is addressed by a scheduler,
which utilizes a work stealing concept. A more detailed description of the model is presented in Section 4.1.

Commonly, in electronic structure calculations, sparsity is maintained by truncation of small elements before and sometimes after multiplication. To this end, different strategies of choosing those elements exist, see for example the discussions in [25, 39]. Unlike the majority of methods, the algorithm considered in this article performs truncation of sub-matrix products to reduce the complexity of operations.

In this article we consider different aspects of parallel implementations of the regular multiplication of truncated matrices, sparse approximate matrix multiplication (SpAMM) and their combination in a fully recursive task-based parallel environment called Chunks and Tasks [36]. The main contributions of the article are the error analysis and estimations and the combined technique, which has reduced communication, but preserves the accuracy of the two original methods.

The article is organized as follows. Section 2 contains a brief description of the sparse approximate matrix multiplication algorithm (SpAMM), existing implementations. Section 3 contains derivations related to the error control. In Section 4 we describe our implementations within the Chunks and Tasks programming model, briefly discuss the programming model itself and the leaf level library. Section 5 contains description of the experimental set-up, benchmarks and their results discussion. Some conclusions are given in Section 6.

2. The SpAMM algorithm. The sparse approximate matrix multiplication algorithm introduced in [17, 9] belongs to a family of N-body solvers, which use hierarchical approximations of sub-problems to reduce complexity, similarly to the famous fast multipole method by Greengard and Rokhlin [27]. The algorithm exploits a hierarchical representation of the matrix and the decay property, which in this case is preserved for all levels in the hierarchy, to skip multiplication of small sub-matrices. The decay property is preserved from level to level because the distance function does not change. Assuming that the matrices are represented as quad-trees such that

\[
A^t = \begin{pmatrix}
A^t_{0,0} & A^t_{0,1} \\
A^t_{1,0} & A^t_{1,1}
\end{pmatrix},
\]

where \( t \) is the level in the hierarchy and

\[
\|A^t\|_F = \sqrt{\sum_{i,j=0}^{1} \|A^t_{i,j}\|_F^2},
\]

the SpAMM algorithm can be written recursively as in Algorithm 1. The squared Frobenius norm is kept along with each matrix, and, since it is additive, it can be conveniently computed recursively starting from the lowest level. The parameter \( \tau \) determines truncation of sub-matrix products, i.e. in the product space, whereas commonly truncation is applied on the input matrices, i.e. in the vector space. (see for example [35]). In [18] a modified version of SpAMM is introduced, which adjusts the truncation threshold according to the norms of multiplicands.

2.1. Serial implementations of SpAMM. Following the naive serial recursive implementation in [17], the highly optimized serial version of the algorithm is implemented by the authors of the algorithm in [9] using the technique of symbolic
Algorithm 1 SpAMM

Input: $A$, $B$, $\tau$
Output: $C$

1: if lowest level then
2: return $C = AB$
3: end if
4: for $i = 0$ to 1 do
5: for $j = 0$ to 1 do
6: if $\|A_{i,0}\|_F \|B_{0,j}\|_F \geq \tau$ then
7: $T_0 = \text{SpAMM}(A_{i,0}, B_{0,j}, \tau)$
8: else
9: $T_0 = 0$
10: end if
11: if $\|A_{i,1}\|_F \|B_{1,j}\|_F \geq \tau$ then
12: $T_1 = \text{SpAMM}(A_{i,1}, B_{1,j}, \tau)$
13: else
14: $T_1 = 0$
15: end if
16: $C_{i,j} = T_0 + T_1$
17: end for
18: end for
19: return $C$

multiplication with linkless trees and SSE intrinsics. This implementation easily outperformed vendor-provided \texttt{sgemm} (single precision \texttt{gemm} routine of BLAS [23]) routines both from the Intel Math Kernel Library and AMD’s Core Math Library and even demonstrated lower error with $\tau = 0$ than \texttt{sgemm} had. The authors noted that \texttt{sgemm} has stagnating performance with increasing matrix size due to memory bandwidth saturation, whereas SpAMM has increasing performance.

2.2. Parallel implementations of SpAMM. Bock et al. [10] present two parallel implementations of the SpAMM algorithm. The first one, which uses the OpenMP application programming interface, exploits parallel quad-tree traversal using untied \texttt{task}, i.e. a task that could be executed by more than one thread simultaneously. The implementation demonstrates good parallel scaling behavior with efficiency up to 80\%. However, the authors notice that at some point load balancing becomes an issue with decreasing the quad-tree height.

The second implementation by the same authors targets distributed memory machines and is done using the Charm++ parallel programming model [31]. The implementation is tested on a large cluster with more than 24000 cores. A presence of modest serial component is noted, which is likely to be attributed to the Charm++ runtime. The load balancing problem is addressed by Charm++ built-in load balancer, which migrates chares (actors in the Charm++ terminology) between the nodes taking into account communication costs.

3. Error estimations. As it was noted earlier, the SpAMM algorithm performs truncation of sub-matrix products, i.e. operates in the product space. Truncation inevitably brings some error in the product matrix. The authors of the original algorithm note that the dependency between the $\tau$ parameter in the algorithm and resulting error in the product matrix is not known, see [9, p. C84]. With the modified
version of SpAMM [18], it is possible to derive the relative error bound and to show stability of the process in the sense of [21]. However, it is hardly convenient for practical use, since the bound depends on \( n^2 \), where \( n \) is the matrix size.

In Section 1 we mentioned the decay property of matrices, which are common in electronic structure calculations. This property and ignoring matrix elements smaller than a certain threshold value is exploited to derive the error estimates. In this section we show the asymptotic behavior of the absolute error. We limit ourselves to the case of matrices with exponential decay and the original version of the SpAMM algorithm [17, 9].

Define first a distance function. We use the same framework as in [41]. The function \( d(i, j) \) is a distance function on the index set \( I = \{1, \ldots, n\} \) if for any \( i, j, k \in I \)

1. \( d(i, j) \geq 0 \);
2. \( d(i, j) = d(j, i) \);
3. \( d(i, i) = 0 \);
4. \( d(i, j) \leq d(i, k) + d(k, j) \).

In other words, \( d(i, j) \) is a pseudo-metric defined on \( I \).

Rubensson et al. [41] consider a sequence of matrices \( \{A_n\} \) with exponential decay with the same constants \( \gamma > 0 \) and \( \alpha > 0 \) and associated distance functions \( d_n(i, j) \).

Under the assumption \(|N_{d_n}(i, R)| \leq \gamma R^\beta, \forall R > 0, \forall i, \) where \( N_{d_n}(i, R) \) is the set of all vertices within a distance \( R \) away from the \( i \)-th vertex and \( |N_{d_n}(i, R)| \) is its cardinality, \( \gamma > 0 \) and \( \beta > 0 \) are some constants independent of \( n \), they show that for any \( \varepsilon > 0 \), and for any \( n > 0 \), the matrix \( A_n \) contains at most \( O(n) \) elements greater than \( \varepsilon \) in magnitude and that each row and column of each \( A_n \) has the number of entries greater than \( \varepsilon \) in magnitude bounded by a constant independent of \( n \), see Theorem 4 in [41]. The assumption means that the number of vertices situated within a distance \( R \) away from a vertex is finite, which holds for the underlying physical systems. The theorem states that each row and column of such a matrix has at most \( \kappa \) significant elements, i.e. elements with magnitude greater than \( \varepsilon \), where \( \kappa \) is a constant independent of \( n \). The rest of the elements are not significant, but not necessarily zeros. We directly exploit these results to derive our estimations.

First we consider the error introduced by truncation of the input matrices, i.e. truncation in the vector space.

**Lemma 1.** Let \( \{A_n\} \) and \( \{B_n\} \) be sequences of \( n \times n \) matrices with a common associated distance function \( d_n(i, j) \) for each \( n \). Assume

\[
\|[A_n]_{i,j}\| \leq c_1 e^{-\alpha d_n(i,j)}, \|[B_n]_{i,j}\| \leq c_2 e^{-\alpha d_n(i,j)}
\]

for all \( i, j = 1, \ldots, n \), where \( c_1 \) and \( c_2 \) are positive constants independent of \( n \). Assume also that there exist constants \( \gamma > 0 \) and \( \beta > 0 \) such that

\[
|N_{d_n}(i, R)| \leq \gamma R^\beta, \forall i = 1, \ldots, n, \forall R > 0.
\]

Let \( C_n = A_n B_n \) and \( \bar{C}_n = \bar{A}_n \bar{B}_n \), where

\[
\bar{A}_n = \begin{cases} 0, & \text{if } \|[A_n]_{i,j}\| < \tau, \\ [A_n]_{i,j}, & \text{otherwise,} \end{cases}
\]

and \( \bar{B}_n = \begin{cases} 0, & \text{if } \|[B_n]_{i,j}\| < \tau, \\ [B_n]_{i,j}, & \text{otherwise,} \end{cases} \)

are the truncated matrices, \( 1 > \tau > 0 \) is some given parameter. Then, the error matrix \( E_n = C_n - \bar{C}_n \) has the following element-wise property:

\[
\|[E_n]_{i,j}\| = O(\tau), \forall i, j = 1, \ldots, n.
\]
Proof. Any element of a matrix product is computed as follows:

\[
(C_n)_{i,j} = \sum_{k=1}^{n} [A_n]_{i,k} [B_n]_{k,j},
\]

which is a scalar product of the \(i\)-th row of \(A_n\) and the \(j\)-th column of \(B_n\). At the same time, any element of the product of the truncated matrices \(\tilde{A}_n\) and \(\tilde{B}_n\) is computed as follows:

\[
(\tilde{C}_n)_{i,j} = \sum_{k=1}^{n} [\phi_n]_{i,k} [\phi_n]_{k,j} = \begin{cases} 0, & \text{if } |[A_n]_{i,k}| < \tau \text{ or } |[B_n]_{k,j}| < \tau, \\ \text{otherwise.} \end{cases}
\]

Then any element of the error matrix \(E_n = C_n - \tilde{C}_n\) has an absolute value

\[
|E_n|_{i,j} = |\sum_{k=1}^{n} ([A_n]_{i,k} [B_n]_{k,j} - [\phi_n]_{i,k} [\phi_n]_{k,j})|, \forall i, j = 1, \ldots, n.
\]

This quantity attains zero if no information is lost when computing a given element, i.e. no elements truncated to zero, and attains its maximum when all information is lost when performing multiplication, i.e. \([\phi_n]_{i,k} = 0\), \(k = 1, \ldots, n\). The worst-case scenario is when every element in the error matrix attains its maximum, i.e. \(\tilde{C}_n = 0\) and thus \(E_n = C_n\).

Let us consider the matrix \(C_n = A_n B_n\). We have

\[
(C_n)_{i,j} = |\sum_{k=1}^{n} [A_n]_{i,k} [B_n]_{k,j}| \leq \sum_{k=1}^{n} |[A_n]_{i,k}||[B_n]_{k,j}|.
\]

Since both \(A_n\) and \(B_n\) possess on the exponential decay property with respect to a common distance function \(d_n(i, j)\), by Theorem 4 in [41] for any \(\varepsilon > 0\) both matrices have at most \(\kappa\) significant elements (i.e. greater than \(\varepsilon\) in magnitude) in every row or column, where \(\kappa\) is a constant independent of \(n\). Let us use \(\varepsilon = \tau\).

When estimating the absolute value in (5), three possible types of summands are to be taken into account:

1. \(|[A_n]_{i,k}| \geq \tau, \quad |[B_n]_{k,j}| < \tau \Rightarrow 0 \leq |[A_n]_{i,k}||[B_n]_{k,j}| \leq c\tau; \]
2. \(|[A_n]_{i,k}| < \tau, \quad |[B_n]_{k,j}| \geq \tau \Rightarrow 0 \leq |[A_n]_{i,k}||[B_n]_{k,j}| \leq c\tau; \]
3. \(|[A_n]_{i,k}| < \tau, \quad |[B_n]_{k,j}| < \tau \Rightarrow 0 \leq |[A_n]_{i,k}||[B_n]_{k,j}| \leq \tau^2. \]

The case where both multipliers are greater than \(\tau\) in magnitude is not included since it falls outside the worst-case scenario. The estimation of the product of the absolute values for the first two types is due to exponential decay property, \(c = \max(c_1, c_2)\).

There are not more than \(2\kappa\) summands of type 1 and 2 for any element (Theorem 4 from [41]). The rest summands are of type 3. We split the index set \(I_n = \{1, \ldots, n\}\) into two sets, \(I_{n2} = \{k : |[A_n]_{i,k}||[B_n]_{i,k}| < \tau^2\}\) and \(I_{n1} = I_n \setminus I_{n2}\). Without loss of generality, we assume that \(I_{n1} = \{1, \ldots, 2\kappa\}\) and \(I_{n2} = \{2\kappa + 1, \ldots, n\}\). Then

\[
|E_n|_{i,j} = |C_n|_{i,j} \leq \sum_{k \in I_{n1}} |[A_n]_{i,k}||[B_n]_{k,j}| + \sum_{k \in I_{n2}} |[A_n]_{i,k}||[B_n]_{k,j}|
\]

\[= \sum_{k=1}^{2\kappa} |[A_n]_{i,k}||[B_n]_{k,j}| + \sum_{k=2\kappa+1}^{n} |[A_n]_{i,k}||[B_n]_{k,j}|.
\]

\[= S_1 + S_2.
\]
We consider next the partial sums $S_1$ and $S_2$ separately, starting with $S_1$:

$$S_1 = \sum_{k=1}^{2\kappa} |[A_n]_{i,k}| |[B_n]_{k,j}| \leq 2\kappa c \tau.$$  

To estimate $S_2$, one should take into account that all summands in this expression are of type 3, i.e. both $|[A_n]_{i,k}| < \tau$, $|[B_n]_{k,j}| < \tau \forall k \in I_{n_2}$:

$$S_2 = \sum_{k=2\kappa+1}^{n} |[A_n]_{i,k}| |[B_n]_{k,j}| < \sum_{k=2\kappa+1}^{n} |[B_n]_{k,j}| \leq c_2 \tau \sum_{k=2\kappa+1}^{n} e^{-\alpha d_n(k,j)}.$$  

It can be shown that the last sum in (7) is bounded by a constant independent of $n$:

$$\sum_{k=2\kappa+1}^{n} e^{-\alpha d_n(k,j)} \leq \sum_{k=1}^{n} e^{-\alpha d_n(k,j)} \leq \sum_{r=1}^{\infty} (|N_{d_n}(j,r)| - |N_{d_n}(j,r-1)|) e^{-\alpha(r-1)}$$

$$\leq \sum_{r=1}^{\infty} (|N_{d_n}(j,r)|) e^{-\alpha(r-1)} \leq \sum_{r=1}^{\infty} \gamma r^\beta e^{-\alpha(r-1)},$$

where the last series is a convergent one, which can be demonstrated with d’Alembert test [43]. Let $\sigma$ denote its sum, then

$$S_2 < c_2 \tau \sigma = \delta_2 \tau.$$  

By combining (6) and (8), we obtain

$$|[E_n]_{i,j}| < \delta_1 \tau + \delta_2 \tau = (\delta_1 + \delta_2) \tau = \delta \tau, \forall i,j = 1,\ldots,n,$$

$$|[E_n]_{i,j}| = O(\tau) \forall i,j = 1,\ldots,n.$$  

The same machinery can be applied to the SpAMM algorithm. It is straightforward to see that the only difference is the cases to be considered when computing an element of the product matrix.

**Lemma 2.** Let $\{A_n\}$ and $\{B_n\}$ be sequences of $n \times n$ matrices with a common associated distance function $d_n(i,j)$ for each $n$. Assume that each $A_n$ and each $B_n$ satisfy the exponential decay property (1) with constants $\gamma_1 > 0, \gamma_2 > 0$ and $\alpha > 0$ and (2) holds for some constants $\gamma > 0$ and $\beta > 0$. Let $C_n = A_n B_n$ and $C_n = \text{SpAMM}(A_n, B_n, \tau)$, where $1 > \tau > 0$ is some given parameter. Then, the error matrix $E_n = C_n - C_n$ has the following element-wise property:

$$|[E_n]_{i,j}| = O(\tau), \forall i,j = 1,\ldots,n.$$  

**Proof.** Any element of a matrix product is computed as:

$$[C_n]_{i,j} = \sum_{k=1}^{n} [A_n]_{i,k} [B_n]_{k,j},$$  

$$\sum_{k=1}^{n} [A_n]_{i,k} [B_n]_{k,j} \leq 2\kappa c \tau.$$  

By combining (6) and (8), we obtain

$$S_2 = \sum_{k=1}^{n} |[A_n]_{i,k}| |[B_n]_{k,j}| < \tau \sum_{k=1}^{n} |[B_n]_{k,j}| \leq c_2 \tau \sum_{k=1}^{n} e^{-\alpha d_n(k,j)}.$$  

It can be shown that the last sum in (7) is bounded by a constant independent of $n$:

$$\sum_{k=1}^{n} e^{-\alpha d_n(k,j)} \leq \sum_{k=1}^{n} e^{-\alpha d_n(k,j)} \leq \sum_{r=1}^{\infty} (|N_{d_n}(j,r)| - |N_{d_n}(j,r-1)|) e^{-\alpha(r-1)}$$

$$\leq \sum_{r=1}^{\infty} (|N_{d_n}(j,r)|) e^{-\alpha(r-1)} \leq \sum_{r=1}^{\infty} \gamma r^\beta e^{-\alpha(r-1)},$$

where the last series is a convergent one, which can be demonstrated with d’Alembert test [43]. Let $\sigma$ denote its sum, then

$$S_2 < c_2 \tau \sigma = \delta_2 \tau.$$  

By combining (6) and (8), we obtain

$$|[E_n]_{i,j}| < \delta_1 \tau + \delta_2 \tau = (\delta_1 + \delta_2) \tau = \delta \tau, \forall i,j = 1,\ldots,n,$$

$$|[E_n]_{i,j}| = O(\tau) \forall i,j = 1,\ldots,n.$$  

The same machinery can be applied to the SpAMM algorithm. It is straightforward to see that the only difference is the cases to be considered when computing an element of the product matrix.
which is nothing but a scalar product of the $i$-th row of $A_n$ and the $j$-th column of $B_n$. At the same time, any element of a SpAMM product is computed as follows:

$$[C_n]_{i,j} = \sum_{k=1}^{n} [\chi_n]_{ik} [B_n]_{k,j}$$

if $|[A_n]_{i,k}||[B_n]_{k,j}| < \tau$, otherwise.

Then any element of the error matrix $E_n = C_n - \bar{C}_n$ has an absolute value

$$|[E_n]_{i,j}| = \left| \sum_{k=1}^{n} ([A_n]_{i,k} [B_n]_{k,j} - [\chi_n]_{ik}) \right|, \forall i,j = 1,\ldots,n.$$  

This quantity attains zero if no SpAMM condition has been met when computing a given element, i.e. the information is preserved, and attains its maximum when all information is lost when performing SpAMM, i.e. $[\chi_n]_{ik} = 0, k = 1,\ldots,n$. The worst-case scenario is when every element in the error matrix attains its maximum, i.e. $\bar{C}_n = 0$ and thus $E_n = C_n$.

Let us consider the matrix $C_n = A_n B_n$. We have

$$[C_n]_{i,j} = \left| \sum_{k=1}^{n} ([A_n]_{i,k} [B_n]_{k,j} - [\chi_n]_{ik}) \right|.$$  

Since both $A_n$ and $B_n$ have exponential decay with respect to a common distance function $d_n(i,j)$, by Theorem 4 in [41] for any $\varepsilon > 0$ both matrices have at most $\kappa$ significant elements (i.e. greater than $\varepsilon$ in magnitude) in every row or column, where $\kappa$ is a constant independent of $n$. Let us use $\varepsilon = \tau$, where $\tau$ is a SpAMM parameter. Note that $|[A_n]_{i,k}||[B_n]_{k,j}| < \tau \forall i,k,j = 1,\ldots,n$ since we are at the worst case scenario.

When estimating the absolute value in (9), three possible types of summands are to be taken into account:

1. Both $|[A_n]_{i,k}| \geq \tau$ and $|[B_n]_{k,j}| \geq \tau \Rightarrow \tau^2 \leq |[A_n]_{i,k}||[B_n]_{k,j}| < \tau$;
2. Either $|[A_n]_{i,k}| \geq \tau$ or $|[B_n]_{k,j}| \geq \tau \Rightarrow 0 \leq |[A_n]_{i,k}||[B_n]_{k,j}| < \tau$;
3. Both $|[A_n]_{i,k}| < \tau$ and $|[B_n]_{k,j}| < \tau \Rightarrow 0 \leq |[A_n]_{i,k}||[B_n]_{k,j}| < \tau^2$.

There are not more than $2\kappa$ summands of type 1 and 2 for any element due to Theorem 4 from [41]. The rest summands are of type 3. We split the index set $I_n = \{1,\ldots,n\}$ into two sets, $I_{n,2} = \{ k : |[A_n]_{i,k}||[B_n]_{k,j}| < \tau^2 \}$ and $I_{n,1} = I_n \setminus I_{n,2}$. Without loss of generality, we assume that $I_{n,1} = \{1,\ldots,2\kappa\}$ and $I_{n,2} = \{2\kappa+1,\ldots,n\}$. The further proof is technical and very similar to the proof of Lemma 1, therefore it is omitted.

One can also combine the two approaches, i.e. perform the SpAMM algorithm on truncated matrices. We assume that the truncation parameter and the SpAMM $\tau$ are the same. In this case, we can again apply the machinery used in Lemmas 1 and 2.

**Lemma 3.** Let $\{A_n\}$ and $\{B_n\}$ be sequences of $n \times n$ matrices with a common associated distance function $d_n(i,j)$ for each $n$. Assume that each $A_n$ and each $B_n$ satisfy the exponential decay property (1) with constants $c_1 > 0, c_2 > 0$ and $\alpha > 0$ and (2) holds for some constants $\gamma > 0$ and $\beta > 0$. Let $C_n = A_n B_n$ and $\bar{C}_n = \text{SpAMM}(\hat{A}_n, \hat{B}_n, \tau)$, where $\hat{A}_n$ and $\hat{B}_n$ are the truncated matrices (3) and $1 > \tau > 0$ is some given parameter. Then, the error matrix $E_n = C_n - \bar{C}_n$ has the following
element-wise property:

$$\|E_n\|_{i,j} = O(\tau), \ \forall i, j = 1, \ldots, n.$$  

Proof. Let $C_n$ be the true product of $A_n$ and $B_n$ and $\tilde{C}_n$ be the result of the SpAMM algorithm applied on matrices $[A_n]$ and $[B_n]$ truncated with the same $\tau > 0$ as used by SpAMM. Then, the error matrix $E_n = C_n - \tilde{C}_n$ satisfies

$$(10) \quad \|E_n\|_{i,j} = \|C_n\|_{i,j} - |\tilde{C}_n|_{i,j} = \|C_n\|_{i,j} - |\tilde{C}_n|_{i,j} + |\tilde{C}_n|_{i,j} - |\tilde{C}_n|_{i,j}, \ \forall i, j = 1, \ldots, n.$$  

The first expression in (10) is the element-wise error introduced by truncation of matrices before multiplication, and its bound is derived in Lemma 1. The second expression in (10) is the error introduced by the SpAMM algorithm assuming that the matrices have been already truncated, and its bound is derived in Lemma 2. Combination of those results gives us

$$\|E_n\|_{i,j} = O(\tau).$$

In the proofs of Lemmas 1, 2 and 3 we considered the worst case scenarios, where the true product matrix becomes the error matrix. The exponential decay property of the product matrix $[41]$ can be exploited further to estimate the sum of the squared absolute values of all elements below certain threshold in magnitude.

**Lemma 4.** Let $C_n$ be a sequence of $n \times n$ matrices with associated distance functions $\{d_n(i, j)\}$ and assume that every matrix $C_n$ satisfies (1) with constants $c > 0$ and $\alpha > 0$. Assume also that (2) holds for some constants $\gamma > 0$ and $\beta > 0$. Then, for any $\varepsilon > 0$ and every $C_n$,

$$\sum_{i,j: |C_n|_{i,j} \leq \varepsilon} |C_n|_{i,j}^2 = O(n) \cdot O(\varepsilon^p), \ p < 2.$$  

Proof. One should consider two cases, in which insignificant elements can exist: $d(i, j) < r_{\varepsilon} \equiv \frac{1}{\alpha} \ln \frac{c}{\varepsilon}$ and $d(i, j) \geq r_{\varepsilon}$. This is due to inequality sign in (1). Note that it is implicitly assumed that $\varepsilon \leq c$.

Let us first consider the latter case, i.e. $d_n(i, j) \geq r_{\varepsilon}$. We directly use the exponential decay property (1), here. Then

$$\sum_{i,j: d_n(i, j) \geq r_{\varepsilon}} |C_n|_{i,j}^2 \leq c^2 \sum_{i,j: d_n(i, j) \leq r_{\varepsilon}} e^{-2\alpha d_n(i, j)} = c^2 \sum_{j} \sum_{r=r_{\varepsilon}+1}^\infty e^{-2\alpha d_n(i, j)}$$

$$\leq c^2 \sum_{j} \sum_{r=r_{\varepsilon}+1}^\infty (|N_{d_n}(j, r)| - |N_{d_n}(j, r-1)|) e^{-2\alpha (r-1)}$$

$$\leq c^2 \sum_{j} \sum_{r=r_{\varepsilon}+1}^\infty |N_{d_n}(j, r)| e^{-2\alpha (r-1)}$$

$$\leq c^2 \sum_{j} \sum_{r=r_{\varepsilon}+1}^\infty \gamma r^\beta e^{-2\alpha (r-1)} \leq c^2 n \sum_{r=r_{\varepsilon}+1}^\infty \gamma r^\beta e^{-2\alpha (r-1)},$$
One can set \( p = r - r_e - 1 \), then

\[
e^2 n \sum_{r=r_e+1}^{\infty} \gamma r^\beta e^{-2\alpha(r-1)} = e^2 n \sum_{p=0}^{\infty} \gamma (p + r_e + 1)^\beta e^{-2\alpha(p+r_e)}
\]

(11)

\[
= e^2 n e^{-2\alpha r_e} \sum_{p=0}^{\infty} \gamma (p + r_e + 1)^\beta e^{-2\alpha p}
\]

The series in (11) converges due to d’Alembert test [43]. Let \( \nu \) be the sum of that series. It is a constant independent of \( n \). Note that \( e^{-2\alpha r_e} = e^{-2\alpha \frac{1}{\alpha} \ln \frac{2}{\epsilon}} = \frac{2^\alpha}{\epsilon^2} \), thus

\[
\sum_{i,j: d_n(i,j) \geq 2} |[C_n]_{i,j}|^2 \leq n \epsilon^2 \nu.
\]

Since \( \nu \) is independent of \( n \), \( n = O(n) \) and

\[
\lim_{\epsilon \to 0} \frac{\epsilon^2}{\epsilon^2} = 1,
\]

then

(12)

\[
\sum_{i,j: d_n(i,j) \geq 2} |[C_n]_{i,j}|^2 = O(n) \cdot O(\epsilon^2).
\]

Now one has to treat the other part, i.e. the elements, for which the distance functions is less than \( r_e \), using the fact that the elements are smaller than \( \epsilon \) in magnitude. This can be done as follows:

\[
\sum_{i,j: d_n(i,j) < r_e \atop |[C_n]_{i,j}| \leq \epsilon} |[C_n]_{i,j}|^2 \leq \sum_{i,j: d_n(i,j) \in [0,r_e)} \epsilon^2 = \epsilon^2 \sum_{j} \sum_{i: d_n(i,j) \in [0,r_e)} 1
\]

\[
= \epsilon^2 \sum_{j} |N_{d_n}(j,r_e)| \leq \epsilon^2 \sum_{j} \gamma (r_e)^\beta
\]

\[
\leq \epsilon^2 n \gamma (r_e)^\beta = \epsilon^2 n \gamma \left( \frac{1}{\alpha} \ln \frac{2}{\epsilon} \right)^\beta.
\]

Since \( \alpha, \beta, \gamma \) and \( c \) are constants independent of \( n \), \( o(x) = O(x) \) for any \( x \) and

\[
\lim_{\epsilon \to 0} \frac{\epsilon^2 \gamma \left( \frac{1}{\alpha} \ln \frac{2}{\epsilon} \right)^\beta}{\epsilon^p} = 0, \forall p < 2,
\]

one concludes that

(13)

\[
\sum_{i,j: d_n(i,j) < r_e} |[C_n]_{i,j}|^2 = O(n) \cdot o(\epsilon^p) = O(n) \cdot O(\epsilon^p), \ p < 2.
\]

Combining together (12) and (13), we obtain

\[
\sum_{i,j: |[C_n]_{i,j}| \leq \epsilon} |[C_n]_{i,j}|^2 = O(n) \cdot \left(O(\epsilon^2) + O(\epsilon^p)\right) = O(n) \cdot O(\epsilon^p), \ p < 2.
\]
The result of Lemma 4 should be interpreted as follows: the sum of all elements smaller than $\varepsilon$ in magnitude is proportional to $n$ when $\varepsilon$ is fixed and $n$ grows, but decays slower than $\varepsilon^2$ when $n$ is fixed.

Now we combine Lemmas 1, 2, 3 and 4 to show the behavior of the error matrix Frobenius norm for all the methods.

**Theorem 5.** Let $\{A_n\}$ and $\{B_n\}$ be sequences of $n \times n$ matrices with a common associated distance function $d_n(i,j)$ for each $n$. Assume that each $A_n$ and each $B_n$ satisfy the exponential decay property (1) with constants $c_1>0,c_2>0$ and $\alpha>0$ and (2) holds for some constants $\gamma>0$ and $\beta>0$. Let $C_n = A_nB_n$ and $\tilde{C}_n = \tilde{A}_n\tilde{B}_n$, where $\tilde{A}_n$, and $\tilde{B}_n$ are the truncated matrices (3). Then, the error matrix $E_n = C_n - \tilde{C}_n$ has the following property:

$$\|E_n\|_F = O \left( n^{1/2} \right) \cdot O \left( \tau^{p/2} \right), \quad p < 2.$$

**Proof.** In the worst-case scenario, $\tilde{C}_n = 0$ and thus $E_n = C_n$. We know by theorem 5 from [41] that $C_n$ satisfies the exponential decay property with respect to the same distance function as $A_n$ and $B_n$. Thus by theorem 4 from [41] $C_n$ has at most $O(n)$ significant elements $\forall \varepsilon > 0$. In order to estimate the Frobenius norm of the error matrix, we use lemma 4 to compute partial sum for all insignificant elements with $\varepsilon = \tau$ and lemma 1 to the rest of elements, i.e. significant ones:

$$\|E_n\|_F^2 = \|C_n\|_F^2 = \sum_{i,j:|C_{n,i,j}| \leq \tau} |C_{n,i,j}|^2 + \sum_{i,j:|C_{n,i,j}| > \tau} |C_{n,i,j}|^2$$

$$= O(n) \cdot O(\tau^p) + O(n) \cdot (O(\tau))^2$$

$$= O(n) \cdot (O(\tau^p) + O(\tau^2)) = O(n) \cdot O(\tau^p), \quad p < 2.$$

By taking the square root from both sides of the previous relation, we arrive at

$$\|E_n\|_F = \sqrt{O(n)} \cdot O(\tau^p) = O \left( n^{1/2} \right) \cdot O \left( \tau^{p/2} \right), \quad p < 2.$$
Theorem 7. Let \( \{A_n\} \) and \( \{B_n\} \) be sequences of \( n \times n \) matrices with a common associated distance function \( d_n(i, j) \) for each \( n \). Assume that each \( A_n \) and each \( B_n \) satisfy the exponential decay property (1) with constants \( c_1 > 0, c_2 > 0 \) and \( \alpha > 0 \) and (2) holds for some constants \( \gamma > 0 \) and \( \beta > 0 \). Let \( C_n = A_nB_n \) and \( \hat{C}_n = \text{SpAMM}(\tilde{A}_n, \tilde{B}_n, \tau) \), where \( \tilde{A}_n, \tilde{B}_n \) are the truncated matrices (3) and \( 1 > \tau > 0 \) is some given parameter. Then, the error matrix \( E_n = C_n - \hat{C}_n \) has the following property:

\[
\|E_n\|_F = O \left( n^{1/2} \right) \cdot O \left( \frac{\tau^{p/2}}{\beta} \right), \quad p < 2.
\]

The proof is very similar to the proof of Theorems 5 and 6 and is based on combination of Lemmas 3 and 4 and therefore is omitted.

Based on the Theorems 5 and 6 we see the error matrix norm for both regular multiplication of truncated matrices and the SpAMM algorithm has a similar behavior. Even if one applies SpAMM on truncated matrices, the behavior of the error matrix norm does not become qualitatively worse. We show in Section 5 that in a distributed environment it is profitable to combine the two algorithms into a new technique, referred to as hybrid.

4. Implementation details.

4.1. The Chunks and Tasks programming model. The Chunks and Tasks model, developed by Rubensson and Rudberg [36], is defined by a C++ API. The basis of the model comprises two basic abstractions - a chunk and a task. A chunk represents a piece of data whereas a task represents a piece of work. The parallelism is exploited in both work and data dimensions.

The user prepares a chunk and registers it to the library. Then, a chunk identifier is obtained and the modification of the chunk is no longer possible (similarly to the Concurrent Collections model [13]). This identifier can be provided as input to a task or be a part of another chunk. This allows to build hierarchies of chunks and the runtime takes responsibility of managing such hierarchies.

The work is expressed in terms of tasks. A task has one or more input chunk identifiers and a single output chunk identifier. Access to chunk data is possible only as input to a task, thus one cannot execute a task without having all input chunks available. The user is free to start other tasks inside a task, giving rise to dynamic parallelism.

A proof-of-the-concept runtime library [36] is implemented in C++ and internally parallelized with MPI and Pthreads. A Chunks and Tasks program is executed by a group of worker processes. To address load balancing, the implementation uses a concept of work-stealing, similarly to XKaapi [24], Cilk [8] and StarPU [4].

The set of rules introduced by the model is intended to simplify parallel programming for distributed memory machines by reducing the chance of getting a deadlock to zero. Moreover, immutable data makes it impossible to get a race condition of any kind. However, it takes a certain effort for a conventional MPI programmer to shift the programming paradigm.

4.2. Matrix representation. Similarly to [38] we employ a hierarchical matrix representation. Not only is it naturally suitable for recursive algorithms like SpAMM or inverse factorization [3], but it is easily mapped onto the Chunks and Tasks model. Rubensson and Rudberg [37] present a matrix library implemented using the model (CHTML). A matrix is viewed as a quadtree, and all non-leaf nodes are chunks which contain identifiers of child nodes. Matrices containing no non-zero elements are not
kept in the memory, they are represented by a special identifier value. Matrix elements are kept at the leaf level only. Actual computations take place only if the leaf level is reached. This approach allows to skip computations on zero sub-matrices and thus exploit sparsity patterns dynamically. Figure 4.2 illustrates the hierarchical approach.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{fig1.png}
\caption{Matrix representation as a quad-tree.}
\end{figure}

The leaf level matrices can be organized in different ways. One could simply keep them dense, or as a more advanced approach, use a block-sparse representation, where non-zero blocks are kept in a 2-D array, see [37] for details and discussion. In any case, when implementing the leaf-level functionality, one can assume that the matrix fits into a single computational node. For efficient computations, leaf level routines should rely on BLAS (Basic Linear Algebra Subroutines [23]).

In this work we continue developing the Chunks and Tasks matrix library by adding new functionality.

4.3. Matrix chunk type. In the previous works [37, 3] we use a chunk type which directly corresponds to the hierarchical matrix representation described in above. Now it has to be modified in order to be able to keep the Frobenius norm of the corresponding matrix as internal information. We choose to make updates of the internal information only explicitly, otherwise it is set to zero by default and never touched. For instance, the internal information should be updated before registering the SpAMM task or any other task, which uses the information. However, one arrives at a contradiction with the rules of the Chunks and Tasks programming model described in Section 4.1: once a chunk is registered to the library, it cannot be changed, while the update operation implies changing the chunk.

To overcome this difficulty, we implement a special task type for updating internal information. In our case it is a simple double floating point number, but in general, it might be a more complex data structure. The task builds a copy of the chunk hierarchy, starting from the leaves, and registers parent chunks only after the children chunks provide their Frobenius norms. After that the original chunks can be deleted. Another option is to update the internal information when constructing the chunk or the hierarchy.

4.4. SpAMM task type. The SpAMM Algorithm 1 is expressed in terms of the Chunks and Tasks matrix library. The main difference from regular matrix-
matrix multiplication introduced by Rubensson and Rudberg [37] is that it accepts input chunks of another type, checks Frobenius norms of multipliers and if their norm product is small enough, the output chunk is set to null. Otherwise the same task is registered for sub-matrices and the process continues recursively until reaching the leaf level. The $\tau$ parameter is provided as one of the input chunks.

4.5. Leaf level library. As we discussed in Section 4.2, CHTML can be used together with different leaf level representations. The previous works [37, 3] use a block-sparse leaf level library. However, it does not suite the SpAMM algorithm, since it is not consistent with the hierarchical structure used in the algorithm.

In order to be fully consistent, we develop a new leaf level library, namely hierarchical blocks-sparse library, which relies on a hierarchical matrix representation as the name suggests. At each level a matrix contains four C++ smart pointers to point at children matrices, one regular pointer to point at the parent, 1-D array where actual numbers are kept and own sizes. A null pointer represents a zero block of a matrix. Matrices containing no non-zero elements are only allowed at the top level, in the intermediate levels such matrices are represented as null pointers. A matrix is not allowed to have a child of equal or larger size. The lowest level matrices do not have children, they only keep actual numbers and have a parent pointer.

The choice of smart pointers is done to simplify routines where removal of sub-matrices is necessary. A parent pointer is chosen to be a regular one to avoid a state of a cyclic reference, which may lead to memory leaks.

The parent pointer is utilized as follows: using this pointer, for any matrix one can determine which of the parent’s children the current matrix is. By going up in the hierarchy using a sequence of parent pointers, it is possible to construct a sequence of digits from 0 to 3, which uniquely identifies a matrix in the hierarchy at any level. Using this code, it is easy to get the coordinates of the top left corner of the current matrix in the "big picture", which might be useful for some algorithms.

The hierarchy does not go down to single matrix elements. It stops when reaching a certain predefined block size. A block is kept if it contains at least one non-zero value. So, in some sense it exploits blocking ideas similar to block-sparse leaf matrix in [37] and [14].

The most important advantage of the hierarchical block-sparse representation over the previously used block-sparse format is that is handles "not so dense" (nearly sparse) matrices more efficiently than the previously used format, since there is no need to iterate and look for a non-zero block when accessing it. This is confirmed by experimental results in Section 5.

The list of operations currently supported by the hierarchical block-sparse matrix library includes:

1. $C = A \times B$, $C = A^T \times B$, $C = A \times B^T$, $C = A^T \times B^T$;
2. $C = A + B$;
3. $C = A^2$, where $A = A^T$, in this case, only the upper triangle of the matrix is stored and utilized;
4. $C = A \times B$, where either $A$ or $B$ is symmetric;
5. $C = A \times A^T$, $C = A^T \times A$, where $A$ is symmetric;
6. $C = \alpha A$;
7. $C = A + \alpha I$;
8. $C = Z$, $ZZ^T = A^{-1}$, where $Z$ is the inverse Cholesky factor, $A$ is a symmetric positive-definite matrix;
Algorithm 2 MatrixMultiply

Input: $A, B$
Output: $C$

1. if $A$ not NULL and $B$ not NULL then
2. if lowest level then
3. $X = \text{leafMatrixMultiply}(A, B)$;
4. $C = \text{registerChunk}(X)$;
5. else
6. for $m = 1$ to 2 do
7. for $n = 1$ to 2 do
8. $Y_1 = \text{registerTask}(\text{multiply}, A_{m1}, B_{1n})$;
9. $Y_2 = \text{registerTask}(\text{multiply}, A_{m2}, B_{2n})$;
10. $C_{mn} = \text{registerTask}(\text{add}, Y_1, Y_2)$;
11. end for
12. end for
13. $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$;
14. end if
15. else
16. $C = \text{NULL}$;
17. end if
18. return $C$

9. $C = \text{SpAMM}(A, B, \tau)$.

Corresponding BLAS operations are called at the lowest level of the hierarchy. As one can see, the list of operations in consistent with the functionality of the Chunks and Tasks matrix library, which can be found in [37], except the last one. This is done for compatibility purposes. The last operation corresponds to the SpAMM task type.

Basically, the hierarchical block-sparse leaf matrix library is a small-scale recreation of the Chunks and Tasks matrix library (to a certain extent). It exploits similar structure, but, since the computation happens within a single computational node, several operations can be improved.

Let us consider how the regular multiplication is done in the Chunks and Tasks matrix library. The reader can find description in Algorithm 2. At line 1 the input is checked and if any of input matrices are zero (represented as special NULL identifier), then the output matrix identifier is also set to NULL. Then, lines 2–4 describe leaf-level operations, lines 5–14 describe higher level operations. The code is organized in this manner due to one of imposed limitations: a chunk content can be accessed only as an input to a task, i.e. only a single level of the hierarchy is visible at a time. In other words, if both of the two input matrices are not NULL, then 13 tasks (8 multiplications, 4 additions, creating a big matrix out of child identifiers) will be anyway registered, unless the lowest level is involved. Registration of tasks definitely leads to overhead.

In a shared memory environment multiplication of hierarchical matrices can be optimized. Consider an illustrative example of such multiplication in (14). As one can see, although both input matrices are not NULL, the resulting matrix is NULL. So, it is not worth to multiply those matrices at all. However, we could judge this only when seeing the whole hierarchy, but not the two matrix identifiers.
Since the leaf level library works in a shared memory, it is indeed possible to see the whole hierarchy at once and predict if it is worth to start multiplication of two matrices. This check should be done recursively, since, it might happen that the resulting matrix $C$ does have a non-zero entry, see (15). In this case, prediction also should be done when multiplying $A_1$ by $B_3$. If those matrices give non-zero product, then the procedure is worth to perform.

In our implementation, such prediction is utilized not only for multiplication routine, but also for the others, including the SpAMM routine, where the conclusion is based not only on the layout of children and recursive checks, but also on the norms of corresponding matrices. This allows our library to be competitive in terms of performance with the block-sparse library used in the previous works. More details can be found in Section 5.

Though predictor routines are called before every call to multiplication and thus extra traversals of the quad-tree are done, the overhead of such prediction is small compared to the overhead of memory allocation for new matrix objects, which anyway would not be used. Prediction is cheap since it involves only pointer arithmetic and logical operations. Note that these predictor routines allow us to avoid the prohibited states in matrices discussed above.

\[
\begin{bmatrix}
A_0 & \text{NULL} \\
\text{NULL} & \text{NULL}
\end{bmatrix}
\times
\begin{bmatrix}
\text{NULL} & \text{NULL} \\
\text{NULL} & B_3
\end{bmatrix}
= \begin{bmatrix}
\text{NULL} & \text{NULL}
\end{bmatrix}
\]

\[
\begin{bmatrix}
A_0 & A_1 \\
\text{NULL} & \text{NULL}
\end{bmatrix}
\times
\begin{bmatrix}
\text{NULL} & \text{NULL} \\
\text{NULL} & B_3
\end{bmatrix}
= \begin{bmatrix}
\text{NULL} & A_1 \times B_3
\end{bmatrix}
\]

4.6. Two-level algorithm. The full implementation then looks as follows: at the level of the Chunks and Tasks library, the task-based version of Algorithm 1 is utilized. The matrices are split into relatively large blocks so that one task could occupy a whole computational node. Inside those blocks the matrices are kept in hierarchies as well using the hierarchical blocks-sparse leaf level library. When the task code eventually arrives at the leaf level, a corresponding SpAMM method of the leaf library is called. Actual computations are performed there using the prediction routines and the resulting block products are computed. Then, a chunk hierarchy of the resulting product is built from bottom.

5. Results.

5.1. Experimental setup. In this section we describe the set of experiments which are carried out in order to determine performance of the leaf level library and the approximate multiplications algorithms.

All experiments are done at PDC high performance computing center located at KTH Royal Institute of Technology in Stockholm. The target machine is the Beskow cluster, which is a Cray machine equipped with 2060 nodes each containing 2 16-core Intel Xeon E5-2698v3 CPUs combined with 64 gigabytes of RAM. The CPUs operate at 2.3 GHz frequency. The nodes are connected in Dragonfly topology using the Cray Aries high speed network. The code is compiled with GCC 7.3.0 g++ compiler, Cray MPICH 7.7.0 and OpenBLAS 0.2.20 [2]. The OpenBLAS library is built from the source code with disabled multi-threading. This is highly recommended by the developers in case an application uses multi-threading somewhere else, which is the case with the Chunks and Tasks library.
5.2. Performance of the leaf level library. In this benchmark the aim is to investigate the performance of the new hierarchical block-sparse matrix library and compare it with the existing block-sparse library. For this, we perform matrix-matrix multiplication at the leaf level, and the Chunks and Tasks library is not used at all. Two matrices of size $2048 \times 2048$ of varying fill-in factor are multiplied. The non-zero blocks of the matrices are distributed randomly uniformly over the matrix. All computations are performed in double precision. The total number of multiplications is computed by counting calls to BLAS $\text{gemm}$ routine. As a reference when reporting the performance of our implementation, we use a theoretical peak performance model based on [22]. It takes into account a variety of instruction sets, also known as intrinsics. Modern compilers automatically generate codes which use intrinsics. The target CPU belongs to the Intel Haswell family, thus it has four different instruction sets: SSE, AVX, AVX2 and FMA. The last two are often referred as the same, but technically they are distinct, since FMA instructions have their own flag. According to the performance model, the peak performance of a single core can be computed as follows (case AVX+FMA (DP) of Table 4 in [22]):

\begin{equation}
P = F \cdot \frac{\text{flop}}{\text{operation}} \cdot \frac{\text{operation}}{\text{instruction}} \cdot \frac{\text{instructions}}{\text{cycle}},
\end{equation}

where $F = 2.3 \cdot 10^9$ is the base CPUs frequency in Hz, $\frac{\text{flop}}{\text{operation}} = 2$ is the number of floating point operation included in the instruction’s semantic (commonly, fused operation like multiply-add are used, hence the value), $\frac{\text{operation}}{\text{instruction}} = 4$ is the number of operands in the SIMD operation, which is $256/64 = 4$ for AVX registers operating with doubles, $\frac{\text{instruction}}{\text{cycle}} = 2$ is the number of simultaneously executed instruction in a single cycle. The latter quantity is rather complicated to compute, but, as mentioned in [22], it can be approximated well enough by the presented value. The formula (16) gives the peak performance of a single CPU core equal to $36.8 \text{ Gflop/s}$, thus the overall performance of the CPU is $16 \cdot 36.8 \text{ Gflop/s} = 588.8 \text{ Gflop/s}$, which is consistent with CPU benchmark results available on the Internet [1].

The benchmark results are presented in Figure 2. As one can observe, the hierarchical block-sparse library performs slightly worse, than the previously used block-sparse library. However, there is a cross-point, which moves to the right with increasing block-size. This indicates that the hierarchical library handles matrices with fill factor about 0.2 and less better, than the other one. The theoretical peak performance is plotted in magenta for reference. For $bs = 256$ the hierarchical block-sparse library attains almost 76% of theoretical peak performance for any fill factor, while block-sparse library attains 73–88% depending on the fill factor.

5.3. Performance on a model problem. The aim of this benchmark is to investigate the performance of the Chunks and Tasks implementations of the three method considered in Section 3. When working on leaf level in a shared memory environment, it is relatively easy to count the total number of operations for random sparse matrix multiplication. However, in distributed environment, there is no convenient tool for that. That is why the performance should be evaluated on a model problem, for which one knows exactly how many operations are to be done.

For the model problem, we create a matrix with exponential decay away from the main diagonal. Elements smaller than $10^{-16}$ in magnitude are set to zero by construction. The matrix is then truncated with a certain truncation threshold $\varepsilon$ and multiplied by itself.
Truncation gives us a banded matrix. Assuming that the matrix has exponential decay away from the main diagonal, i.e. $a_{i,j} = e^{-\alpha d}, d = |i - j|$, the bandwidth $d$ of the truncated matrix is derived from the following relation:

\begin{equation}
    e^{-\alpha d} = \frac{\varepsilon}{b_s},
\end{equation}

where $b_s$ is the block-size on the lowest level of the leaf matrix library, or, in other words, blocks of size $b_s$ are given to BLAS routines, $\varepsilon$ is the threshold value.

Once the bandwidth of the matrix is known, there is a closed formula for the number of floating point operations (FLOPs) which are to be performed in order to multiply the matrix by itself, assuming that the matrix is not symmetric:

\begin{equation}
    N_{\text{ops}} = 2 \left( N (2d + 1)^2 - \frac{5}{3} d(d + 1)(d + 1) \right),
\end{equation}

where $d$ is the bandwidth, $N$ is the matrix size $[37]$. This number of operations divided by the time needed to multiply the banded matrix by itself gives us a reference performance of a regular multiplication routine.

Here comes the question how to measure the performance of the algorithms. We propose the following: for a fixed target accuracy $\sigma$ a set of tests for varying $\tau$ is performed. In each test, the truncation threshold for the regular algorithm is set to
For the regular multiplication we seek the largest \( \tau \) such that the error does not exceed \( \sigma \). This \( \tau \) gives us a number of FLOPs according to formulas (17) and (18) and we can compute the performance of the regular multiplication. Then, a set of tests for the same \( \sigma \) and range of \( \tau \) is repeated for the SpAMM algorithm, which is applied to original matrix. Again, the largest \( \tau \) is found such that the error in SpAMM does not exceed \( \sigma \). This \( \tau \) gives us the timing. Then, we assume that the number of FLOPs is exactly the same as in regular multiplication after truncation and divide the number of FLOPs obtained for the regular case by the timing for SpAMM. This gives us the performance of SpAMM. The performance of the hybrid algorithm, i.e. SpAMM applied to truncated matrices, is computed the same way.

Single node performance. For this benchmark, we employ CHT-MPI Chunks and Tasks library implementation and the corresponding matrix library. We set a single worker process to occupy the whole computational node and to perform all the tasks, thus no communication is involved. Within the worker, the tasks are executed in parallel, if possible, by 31 available threads, whereas a single thread is left for communication just to be consistent with multiple node case. We set \( N = 5 \times 10^4 \), \( \alpha = 0.005 \), the task size, i.e. the size where a piece of matrix processed by the leaf level library, is 2048, \( bs = 32, 64, 128, 256 \). The parameters are chosen so that the computation fits the amount of RAM available in a single node. We apply seven different values of \( \tau \), namely \( 10^{-4}, 10^{-5}, \ldots, 10^{-10} \). The two target accuracies are \( \sigma_1 = 10^{-4}, \sigma_2 = 10^{-6} \).

![Graph](image.png)

**Fig. 3.** Single node performance on a model problem for target accuracies \( \sigma_1 = 10^{-4}, \sigma_2 = 10^{-6} \). A matrix of size \( N = 5 \times 10^4 \) is multiplied by itself. Block-sizes \( bs = 32, 64, 128, 256 \) is used and truncation is performed with \( \tau = 10^{-4}, 10^{-5}, \ldots, 10^{-10} \).

The results of this benchmark test are presented in Figure 3. This is the average of four runs. As one can observe, the SpAMM and the hybrid algorithm perform better than the regular multiplication competing with each other. The performance gain varies from around 25–30\% for \( bs = 32 \) to about 5–6\% for the largest block-size for target accuracy \( \sigma_1 = 10^{-4} \), whereas for \( \sigma_2 = 10^{-6} \) it ranges from approximately 7–13\% for \( bs = 32 \) to about 5–7\% for the largest block-size.

In the shared memory regime one can’t see how communication affects the performance of the algorithms. This becomes clear in the next test case.

Performance on multiple nodes. In this benchmark we keep the computational load per node approximately the same, i.e. when doubling the matrix size, we also double the number of nodes. The system size is \( N = 5 \times 10^4 \times N_{\text{nodes}} \), \( \alpha = 0.005 \), task size is 2048. The number of nodes ranges from 1 to 64, the two target accuracies, \( \sigma_1 = 10^{-4}, \sigma_2 = 10^{-6} \). The leaf blocksize \( bs \) is set to 64. We perform the tests for
each target accuracy twice and then use the average values for visualizations.

The results for target accuracy $\sigma_1 = 10^{-4}$ are presented in Figure 4. One can observe that when communication is involved, the excellence of the SpAMM algorithm does not disappear. However, the hybrid algorithm outperforms SpAMM. The average performance gain for the SpAMM is about 13% compared to the regular multiplication, whereas the hybrid version gives about 17%. The rightmost point on the plot for SpAMM corresponds to $\approx 11.8\%$ of the overall peak performance, whereas the hybrid algorithm gives 14% and the regular algorithm gives about 10.5%. The Chunks and Tasks library provides auxiliary information about the execution including number of executed tasks of both leaf and non-leaf (i.e. higher in the hierarchy) types for each worker. It turns out that SpAMM requires fewer tasks of both classes to be executed by each worker, see the right panel of Figure 4. On average, the SpAMM algorithm generates about 4%-5% fewer tasks of both classes, than the regular algorithm after truncation. Although it does not look impressive, it does reduce the administrative overhead related to task registration, load balancing etc. Not only generates the SpAMM algorithm fewer tasks, the average execution time for a task is also lower, than for the regular algorithm. This can be clearly seen on a subplot of the right panel of Figure 4. Note that the hybrid algorithm demonstrates similar properties. This is achieved by skipping computations of products of small sub-matrices.

![Figure 4](image_url)

**Fig. 4.** Left panel: multiple node performance on the model problem for target accuracy $\sigma_1 = 10^{-4}$. Computational load is kept the same by doubling the matrix size simultaneously with the number of nodes, $N = 5 \times 10^4 \times \text{Nodes}$. Right panel: average number of executed tasks by a worker (main plot), average execution time of a task (subplot).

We also perform the same experiment for target accuracy $\sigma_2 = 10^{-6}$, its results are presented in Figure 5. In general, we observe a very similar behavior to the previous case, but the advantages of SpAMM and the hybrid algorithm become more visible at larger sizes. The rightmost point at the performance plot for SpAMM corresponds to approximately 14.6% of the total peak performance of all involved nodes, whereas the hybrid version gives 15.2% and the regular algorithm reaches about 12% of the peak value, which is somehow similar to the previous case. This means that for the chosen target accuracy performance of the SpAMM algorithm in the current implementation is about 19% better than performance of the regular multiplication after truncation for the largest tested case. At the same time, the hybrid version is about 24% better in performance. The average number of executed tasks has similar character to the previous case. The difference between the average execution times of a task is reduced compared to the previous case, but relative relation still holds. The performance of the hybrid version is very close to the SpAMM and it is difficult to name the winner.
However, the performance curve for the hybrid method is steeper.

Fig. 5. Left panel: multiple node performance on the model problem for target accuracy $\sigma_1 = 10^{-6}$. Computational load is kept the same by doubling the matrix size simultaneously with the number of nodes, $N = 5 \times 10^4 \times N_{\text{nodes}}$. Right panel: average number of executed tasks by a worker (main plot), average execution time of a task (subplot).

To verify error estimates derived in Section 3 we plot the experimental data in Figure 6. The errors for all three algorithms behave the same way: it grows proportionally to the square root of the system size when it grows and decreases proportionally to $\tau$ when this parameter is decreased. The lines on the left panel are not straight because at some point the truncation does not happen at all, since the threshold value is too small for the given decay properties, and thus the error tends to zero a bit faster than expected.

Fig. 6. Errors in the model problem. Left panel: the Frobenius norm of the error matrix as a function of $\tau$ parameter for $N = 32 \times 10^5$. All three lines have approximate slope of one. Right panel: the Frobenius norm of the error matrix as a function of $N$ for fixed $\tau = 10^{-6}$. All lines have approximate slope of 0.5.

5.4. Performance of the SpAMM algorithm on a real problem. In this benchmark we use basis set overlap matrices originating from space discretization of water clusters of increasing size using a Gaussian basis set. In other contexts such matrices are known as mass matrices or Gramm matrices. The overlap matrices obey the exponential decay property with respect to Euclidean distance between the centres of Gaussian basis functions, which are placed at nuclei positions. The corresponding xyz coordinates can be downloaded from http://ergoscf.org/. The same coordinate files were also used in [3]. The software used to generate the matrices is the Ergo open-source program for linear-scaling electronic structure calculations [42], distributed
under the GNU Public Licence v3 and freely obtainable at http://ergoscf.org/. The basis set used for this particular set of experiments is STO-3G.

The main difference from the model problem is that we do not know the exact number of operations to be performed. This could be counted if we were in a shared memory environment, but in the distributed setting there is no convenient way for that. Thus the performance cannot be easily expressed in FLOPs and, therefore, we use execution time to compare the algorithms.

**Approximate weak scaling.** This computational experiment is performed as follows: an overlap matrix \( S \) is computed using the Ergo code for the given .xyz file and its square is used as a reference to measure errors. The matrix is multiplied by itself assuming that it is not symmetric by all algorithms for the same value of truncation parameter. Then, the errors and their norms are computed. We find out that in case of complicated systems, such as water clusters, block-size should not be large in order to efficiently catch the patterns in the matrix, so we used block-size 16. We also find out that the range of \( \tau \) should be a bit different, i.e. from \( 10^{-4} \) to \( 10^{-12} \). The system size is approximately doubled each time as well as the number of worker processes involved, and each process handles about 9000 atoms, i.e. the test shows an approximate weak scaling. We set 31 thread per process to perform tasks and left a single thread for communication, thus single process occupies the whole node. The task size is 2048. We perform four runs of each test and collect some statistics. The results for two particular values of \( \tau \) are presented in Figure 7. As one can observe, the SpAMM algorithm can only compete with the other only for relatively small systems and a small value of \( \tau \). Otherwise, it demonstrates worse timings.

![Fig. 7](image)

**Average serial execution time of tasks.** The CHT-MPI library conveniently provides auxiliary information about the execution of the program. One can for instance look how much time each of the worker processes spends doing the actual computations, but not sending and receiving the data. This quantity does not take into account threading. We computed the averages of those values and the corresponding plots are presented in Figure 8. As one can see, the SpAMM tasks require much less time in average than the tasks of regular multiplication after truncation. The hybrid version demonstrates about the same behavior. In general, however, it contradicts the weak scaling results observed above.
Average serial execution time of all tasks by a single worker, threading disabled, weak scaling regime, the number of atoms per process is approximately fixed to $9 \times 10^3$, so that the system size is scaled up together with the number of working processes involved. Left panel: $\tau = 10^{-6}$. Right panel: $\tau = 10^{-12}$.

Data movement. The main reason why the SpAMM algorithm behaves badly on a real problem is the amount of data to be moved. Since no initial truncation is applied, it has to manipulate very many small sub-matrices. The other two algorithms do employ the truncation step before starting the main procedure, thus they do not have to move so much data. We collect statistics after four sets of tests and the plots representing amount of data sent by a worker are presented in Figure 9. There are two main observation one can do when looking at the plots: 1) on average, the SpAMM algorithms moves more data than its competitors and 2) the gap between the average amount and the maximal amount for the data sent for the SpAMM algorithm is about 300%. The regular multiplication of truncated matrices and the hybrid algorithm demonstrate better results. Note that even in case $\tau = 10^{-12}$, where more information is preserved during the truncation step and thus more data is to be sent, the regular algorithm and the SpAMM demonstrate quite similar behavior quantitatively, whereas the hybrid version still keeps the average low.

Average amount of data sent by a worker when executing the corresponding algorithm for water clusters of increasing size, $\tau = 10^{-6}$. The number of atoms per process is approximately fixed to $9 \times 10^3$, so that the system size is scaled up together with the number of working processes involved. Right panel: average amount of data sent by a worker when executing the corresponding algorithm for water clusters of increasing size, $\tau = 10^{-12}$. Points represent maximal and minimal amounts over all the workers.
Errors. In order to verify the error estimates obtained in Section 3, we plot the Frobenius norms of the error matrices as functions of the system size $N$ (or, equivalently, as functions of the number of worker processes, since we are in a weak scaling setting) and then we fix the number of processes and plot the norms as functions of $\tau$. The corresponding plots are presented in Figure 10. The left panel shows the case when the $\tau$ parameter is fixed. In this case, the error grows as a square root of $N$, since the slopes of all the three lines are 0.5. The right panel demonstrates the second setting, then the systems size is fixed, but $\tau$ varies. In this case one can observe that the error matrix norm depends almost linearly on $\tau$ parameter, as predicted by the error analysis in Section 3. The slopes of the lines are $\approx 0.99$ for the multiplication of truncated matrices, $\approx 0.96$ for the SpAMM algorithm and $\approx 0.97$ for the hybrid approach. In practice, for a sufficiently large matrix, the dependency between the error matrix norm and the $\tau$ parameter becomes virtually linear.

6. Discussion and concluding remarks. We have developed a two-level fully-recursive implementation of the SpAMM algorithm using the Chunks and Tasks matrix library (CHTML). The algorithm requires a new leaf-level library. One could use either the block-sparse library, or some other implementation, but we are not aware of such implementation that would be able to utilize the approach of the SpAMM algorithm. Otherwise, the SpAMM would work only high in the matrix hierarchy. As demonstrated by the tests, the new leaf level library performs at least not worse than the block-sparse library, especially for nearly sparse matrices with low fill-in, which are common in electronic structure calculations.

We also derived the error estimations and demonstrated that if SpAMM is applied to truncated matrices with exponential decay, then this extra truncation step does not qualitatively change the behavior of the error matrix Frobenius norm. However, it reduces the amount of data to be sent over the communication network in a distributed environment significantly.

When testing on matrices coming from quantum chemistry problems, a drawback of the SpAMM algorithm shows up. It performs poorly due to large amounts of data to be moved. The proposed hybrid approach performs better, especially for small values of the truncation parameter $\tau$. 
Our conclusion is that the SpAMM algorithm for multiplication of nearly sparse matrices with exponential decay with respect to a distance function can be successfully used in a shared memory environment, but it is not the best choice for a distributed environment. Truncation step added before application of SpAMM improves the behavior of the algorithm drastically. This hybrid approach does have the same error behavior as the original SpAMM algorithm and the regular multiplication of truncated matrices, but performs better due to reduced communication.

7. Acknowledgement. Support from the Swedish national strategic e-science research program (eSSENCE) is gratefully acknowledged. Computational resources were provided by the Swedish National Infrastructure for Computing (SNIC) at the PDC Center for High Performance Computing at the KTH Royal Institute of Technology in Stockholm.

We thank Assoc. Prof. Emanuel H. Rubensson and Prof. Maya Neytcheva for comments that greatly improved the manuscript.

References.
[1] Detailed Specifications of the Intel Xeon E5-2600v3 Haswell-EP Processors. https://tinyurl.com/zzqzh52h. [Online; accessed June 18, 2019].
[2] An optimized BLAS library. http://www.openblas.net/. [Online; accessed June 18, 2019].
[3] Anton G. Artemov, Elias Rudberg, and Emanuel H. Rubensson. Parallelization and scalability analysis of inverse factorization using the Chunks and Tasks programming model. arXiv e-prints, art. arXiv:1901.07993, January 2019.
[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comp.-Pract. E., 23(2):187–198, 2011.
[5] Roi Baer and Martin Head-Gordon. Sparsity of the density matrix in Kohn-Sham density functional theory and an assessment of linear system-size scaling methods. Phys. Rev. Lett., 79(20):3962, 1997.
[6] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In Proceedings of the 25th annual ACM symposium on Parallelism in algorithms and architectures, pages 222–231. ACM, 2013.
[7] Mario Bebendorf. Hierarchical matrices. Springer, 2008.
[8] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. J. Parallel Distr. Com., 37(1):55–69, 1996.
[9] Nicolas Bock and Matt Challacombe. An optimized sparse approximate matrix multiply for matrices with decay. SIAM J. Sci. Comput., 35(1):C72–C98, 2013.
[10] Nicolas Bock, Matt Challacombe, and Laxmi Kant V Kalé. Solvers for O(N) electronic structure in the strong scaling limit. SIAM J. Sci. Comput., 38(1):C1–C21, 2016.
[11] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. Sparse matrix multiplication: the distributed block-compressed sparse row library. Parallel Comput., 40(5-6):47–58, 2014.
[12] David R Bowler and T Miyazaki. Calculations for millions of atoms with density functional theory: linear scaling shows its potential. J. Phys.-Condens. Mat., 22(7):074207, 2010.
[13] Zoran Budimić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney,
Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Sci. Programming-Neth.*, 18(3-4):203–217, 2010.

[14] Aydin Buluç and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments. *SIAM J. Sci. Comput.*, 34(4):C170–C191, 2012.

[15] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[16] Matt Challacombe. A general parallel sparse-blocked matrix multiply for linear scaling SCF theory. *Comput. Phys. Commun.*, 128(1-2):93–107, 2000.

[17] Matt Challacombe and Nicolas Bock. Fast multiplication of matrices with decay. *arXiv e-prints*, art. arXiv:1011.3534, Nov 2010.

[18] Matt Challacombe, Terry Haut, and Nicolas Bock. A N-Body Solver for Square Root Iteration. *arXiv e-prints*, art. arXiv:1508.05856, Aug 2015.

[19] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[20] William Dawson and Takahito Nakajima. Massively parallel sparse matrix function calculations with NTpoly. *Comput. Phys. Commun.*, 225:154–165, 2018.

[21] James Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg. Fast matrix multiplication is stable. *Numerische Mathematik*, 106(2):199–224, 2007.

[22] Romain Dolbeau. Theoretical peak FLOPS per instruction set: a tutorial. *J. Supercomput.*, 74(3):1341–1377, 2018.

[23] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM T. Math. Software*, 16(1):1–17, 1990.

[24] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1299–1308. IEEE, 2013.

[25] Stefan Goedecker. Linear scaling electronic structure methods. *Rev. Mod. Phys.*, 71(4):1085, 1999.

[26] Stefan Goedecker and Luciano Colombo. Efficient linear scaling algorithm for tight-binding molecular dynamics. *Phys. Rev. Lett.*, 73(1):122, 1994.

[27] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[28] Wolfgang Hackbusch. *Hierarchical matrices: algorithms and analysis*, volume 49. Springer, 2015.

[29] A Holas. Transforms for idempotency purification of density matrices in linear-scaling electronic-structure calculations. *Chem. Phys. Lett.*, 340(5-6):552–558, 2001.

[30] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, pages 127–136. Springer, 2001.

[31] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[32] X-P Li, RW Nunes, and David Vanderbilt. Density-matrix electronic-structure method with linear system-size scaling. *Phys. Rev. B*, 47(16):10891, 1993.
[33] Stephan Mohr, William Dawson, Michael Wagner, Damien Caliste, Takahito Nakajima, and Luigi Genovese. Efficient computation of sparse matrix functions for large-scale electronic structure calculations: the CheSS library. J. Chem. Theory Comput., 13(10):4684–4698, 2017.

[34] Anders MN Niklasson. Expansion algorithm for the density matrix. Phys. Rev. B, 66(15):155115, 2002.

[35] Emanuel H Rubensson and Elias Rudberg. Bringing about matrix sparsity in linear-scaling electronic structure calculations. J. Comput. Chem., 32(7):1411–1423, 2011.

[36] Emanuel H Rubensson and Elias Rudberg. Chunks and Tasks: a programming model for parallelization of dynamic algorithms. Parallel Comput., 40:328–343, 2014. doi: 10.1016/j.parco.2013.09.006.

[37] Emanuel H Rubensson and Elias Rudberg. Locality-aware parallel block-sparse matrix-matrix multiplication using the Chunks and Tasks programming model. Parallel Comput., 57:87–106, 2016.

[38] Emanuel H Rubensson, Elias Rudberg, and Paweł Salek. A hierarchic sparse matrix data structure for large-scale Hartree-Fock/Kohn-Sham calculations. J. Comput. Chem., 28(16):2531–2537, 2007.

[39] Emanuel H Rubensson, Elias Rudberg, and Paweł Salek. Truncation of small matrix elements based on the Euclidean norm for blocked data structures. J. Comput. Chem., 30(6):974–977, 2009.

[40] Emanuel H Rubensson, Elias Rudberg, and Paweł Salek. Methods for hartree-fock and density functional theory electronic structure calculations with linearly scaling processor time and memory usage. In Linear-Scaling Techniques in Computational Chemistry and Physics, pages 263–300. Springer, 2011.

[41] Emanuel H. Rubensson, Anton G. Artemov, Anastasia Kruchinina, and Elias Rudberg. Localized inverse factorization. arXiv e-prints, art. arXiv:1812.04919, December 2018.

[42] Elias Rudberg, Emanuel H. Rubensson, Paweł Salek, and Anastasia Kruchinina. Ergo: an open-source program for linear-scaling electronic structure calculations. SoftwareX, 7:107–111, 2018.

[43] Walter Rudin et al. Principles of mathematical analysis, volume 3. McGraw-hill New York, 1976.

[44] Yousef Saad. Iterative methods for sparse linear systems, volume 82. SIAM, 2003.

[45] Volker Strassen. Gaussian elimination is not optimal. Numer. Math., 13(4):354–356, 1969.

[46] Joost VandeVondele, Urban Borstnik, and Jurg Hutter. Linear scaling self-consistent field calculations with millions of atoms in the condensed phase. J. Chem. Theory Comput., 8(10):3565–3573, 2012.