Scalable Task-Based Algorithm for Multiplication of Block-Rank-Sparse Matrices

Justus A. Calvin
justusc@vt.edu

Cannada A. Lewis
calewis@vt.edu

Edward F. Valeev*
efv@vt.edu

Department of Chemistry
Virginia Tech
Blacksburg, VA

ABSTRACT

A task-based formulation of Scalable Universal Matrix Multiplication Algorithm (SUMMA), a popular algorithm for matrix multiplication (MM), is applied to the multiplication of hierarchy-free, rank-structured matrices that appear in the domain of quantum chemistry (QC). The novel features of our formulation are: (1) concurrent scheduling of multiple SUMMA iterations, and (2) fine-grained task-based composition. These features make it tolerant of the load imbalance due to the irregular matrix structure and eliminate all artifactual sources of global synchronization. Scalability of iterative computation of square-root inverse of block-rank-sparse QC matrices is demonstrated; for full-rank (dense) matrices the performance of our SUMMA formulation usually exceeds that of the state-of-the-art dense MM implementations (ScaLAPACK and Cyclops Tensor Framework).

Keywords

distributed memory, matrix multiplication, SUMMA, low-rank decomposition, irregular computation, rank-structured, matrix, H matrix, semiseparable matrix, task parallelism, tensor contraction

1. INTRODUCTION

A frontier challenge posed by scientific and engineering applications in areas as distinct as quantum physics and machine learning is dealing with sparse and non-standard tensorial data representations. Such data appears in many forms: sparse tensors, multiresolution spectral element trees, $\mathcal{H}$-matrices, tensor networks, and many others. What they share in common is the reduced number of parameters in terms of which the data is represented, at the cost of more complex data representation and computation relative to the standard/naïve counterpart. The need to deal with irregular data representations conflicts with the evolution of computer hardware and programming models that demand

regular patterns of data access and computation for peak performance. This tension drives the search for algorithms that minimize/avoid communication and/or hide its cost, and are capable of dealing with increasingly irregular numerical data structures.

In this work we explore parallel computation with matrices composed of low-rank blocks that we refer to as Clustered Low-Rank (CLR) matrices[1] Concrete examples of such matrices are taken from quantum chemistry and materials science. Matrices (tensors) in such context represent quantum states (of electrons) and operators represented in some basis. Efficient application of operators to states — represented by matrix multiplication (tensor contraction) — demands taking advantage of the matrix (tensor) structure that take the form of (a) block-sparsity due to the distance decay of the operator kernel and the localized nature of basis functions [20], (b) symmetries under geometric and other transformations [27, 19], and/or (c) block-rank-sparsity due to smoothness of states and operator kernels [24]. Notably, exploiting this matrix structure depends on problem-specific blocking of matrix dimensions that arise due to domain-specific needs and typically cannot be chosen arbitrarily. In other words, the matrices that we encounter are "sparse" in a general sense, which encompasses element-, block-, and block-rank-sparsity; but in a practical sense the matrices are not sparse enough to be a good match for the established sparse MM algorithms.

The key idea is a novel task-based variant of Scalable Universal Matrix Multiplication Algorithm (SUMMA) of van de Geijn and Watts,[36], a well-known algorithm for distributed-memory dense matrix multiplication (MM). The nonuniform blocking and data inhomogeneity of CLR matrices conflict with the *uniform* data distribution exploited by all distributed-memory, dense MM algorithms — including Cannon's [11], SUMMA [36], and others [15, 14, 8, 33, 27]. Task-based formulation allows us to overcome this limitation. Task-based/dataflow programming models are a natural choice for implementation of algorithms with irregular data and computation patterns; such models have already been used successfully for dense matrix algebra applications [21, 7]. Besides handling matrices with structure, the task-based approach provides additional benefits: (a) inter-node commu-

---

[1]Related matrix data structures have appeared under many names (matrices with decay, $\mathcal{H}$-matrices, rank-structured matrices, and mosaic skeleton approximation), but no single globally-accepted terminology exists. For the history of these types of matrices see Ref [37].

nication costs can be partially or fully hidden by overlapping computation and communication, (b) performance should be less sensitive to topology, latency, and CPU clock variations, (c) fine-grained, task-based parallelism is a proven means to attain high intra-node performance by leveraging massively multicore platforms and hiding the costs of memory hierarchy (*e.g.* Intel TBB, Cilk), (d) lack of global synchronization allows the overlap multiple, high-level stages of computation (*e.g.* two or more multiple matrix multiplications contributing to the same expression).

The new formulation was used to implement iterative computation of the square root inverse of a matrix, a prototypical operation in which block ranks of intermediate matrices change dynamically during the iteration. The usual advantage of the task formulation, tolerance of load imbalance and latency, are demonstrated in the regime where matrices approach full rank, by comparison against the state-of-the-art dense MM implementations.

2. STANDARD SUMMA AND VARIANTS

Before describing our algorithm, we start out by recapping the standard SUMMA and its variants for distributed-memory MM. Although there are earlier [11] and asymptotically faster [34] algorithms, SUMMA has become popular due to its relative simplicity and flexibility (it can be easily generalized to rectangular matrices and process grids). From a just as important practical standpoint, SUMMA, like all 2D algorithms, uses memory more economically than its 3D counterparts [5, 1]. SUMMA is also a building block for more complicated 2.5D and 3D MM algorithms [30] and produced a number of variants,[12, 16] including sparse SUMMA (SpSUMMA) [8, 9].

SUMMA implements matrix multiplication $\mathbf{C} = \mathbf{AB}$ as a series of rank-k updates. The input and output matrices are embedded on a rectangular process mesh in an element-cyclic or block-cyclic manner to ensure approximate load balance. In each iteration of the algorithm, a column/row panel of $\mathbf{A}/\mathbf{B}$ is broadcast along rows/columns of the process grid, respectively; matrix $\mathbf{C}$ remains stationary throughout the procedure (variants of SUMMA in which $\mathbf{A}$ or $\mathbf{B}$ are stationary are also possible; transposed multiplies, *e.g.* $\mathbf{C} = \mathbf{AB}^{\dagger}$, are also relatively simple to handle [36, 28]). Each pair of broadcasts is followed by a rank-k update, $C_{ij} \leftarrow A_{ik}B_{kj}+C_{ij}$, (Einstein summation convention is used throughout). Original SUMMA papers by van de Geijn and Watts [36] and by Agarwal *et al.* [2] considered versions of the algorithm that overlapped computation and communication by pipelining and preemptive broadcasts, respectively, and other broadcast variants have been considered [28], including topology-specific broadcasts [29]. For simplicity, we present a SUMMA version with preemptive broadcasts in Figure 1.

DIMMA, an early variation of SUMMA introduced by Choi in 1998, improved performance of synchronous SUMMA by realizing that the order of broadcasts in the original algorithm coupled with communication barriers created significant slack in communication [12, 13]. Iterations were reordered in DIMMA such that each node broadcasts all of its data in succession as opposed to the round-robin approach in SUMMA. Similar improvements can be attained by overlapping communication and computation [2].

SUMMA was recently extended to sparse MM (SpSUMMA) by Buluç and Gilbert [8, 9]. This algorithm is similar to

---

Algorithm 1 SUMMA with non-blocking broadcast
1: BROADCAST($A(*,0), 0, row\_group$)
2: BROADCAST($B(0,*), 0, col\_group$)
3: **for** $k = 0, \ldots, K-1$ **do**
4:     **if** $k+1 < K$ **then**
5:         $row\_root \leftarrow (k+1) \bmod cols$
6:         $col\_root \leftarrow (k+1) \bmod rows$
7:         BROADCAST($A(*,k+1), row\_root, row\_group$)
8:         BROADCAST($B(k+1,*), col\_root, col\_group$)
9:     **end if**
10:   WAIT($A(*,k)$)
11:   WAIT($B(k,*)$)
12:   $C(*,*) \leftarrow \alpha A(*,k) \cdot B(k,*) + C(*,*)$
13: **end for**

dense SUMMA, except matrix sub-blocks are stored in a doubly compressed sparse column (DCSC) format and sparse generalized matrix-multiplication (SpGEMM) is used to compute rank-k updates. The main challenges of all sparse MM algorithms, including SpSUMMA, are the increased relative costs of communication compared to the dense case, load imbalance, and the relatively low intra-node performance of sparse matrix kernels.

The problem of load imbalance does not appear in dense SUMMA implementations as the work load is nearly-optimally balanced. With random sparsity, approximate load balance is achieved in the asymptotic limit, however with structured sparsity (*e.g.* matrices with decay) one does not expect rows/columns to be uniformly filled.

In the regime of high sparsity (low matrix fill-in factors) the communication time dominates the computation time due to the effectively-reduced benefit of blocking. The increasing role of communication in sparse MM can be somewhat alleviated by communication hiding. Buluç and Gilbert discussed potential benefits of communication hiding for sparse MM in Ref. [8] but did not pursue this approach in their experiments due to lack of quality one-sided communication tools [9]. Another possibility is to minimize communication, *e.g.* by switching to 3D MM [3]. Although sparse 2D SUMMA is not strongly scalable, nevertheless good scalability of SpSUMMA was demonstrated in practice [9].

3. TASK-BASED SUMMA FORMULATION

Our work to improve SUMMA is motivated by the needs of computation on matrices/tensors with irregular low-rank structure of their blocks. Some of the challenges of computing with such data are similar to the general challenges of sparse MM: increased communication/computation ratio and lack of load balance. The latter is compounded by the desire to use physics-based blocking of matrix dimensions as well as non-standard data representations. To address these challenges, we set to investigate a task-based formulation of SUMMA algorithm designed to partially offset some communication latency and to alleviate the load imbalance. Prior efforts to reformulate dense matrix multiplication using a tasks-based programming models are known [7, 4]; the novelty of our effort is the focus on dense matrices/tensors with irregular structure (*i.e.* block-rank-sparsity) that cannot be handled straightforwardly using the standard dense-only approaches.

In this section we briefly describe the design of our al-

gorithm, by highlighting the differences with the procedural SUMMA implementations[36]. We first analyze the data dependencies of discrete operations in the procedural SUMMA implementation. We then discuss the task composition and dependencies of our modified implementation. For simplicity, we only consider the 2D SUMMA implementation; though our approach is applicable to 3D and 2.5D variant of SUMMA.

3.1 Control Flow of Standard SUMMA

Like all dense MM algorithms, SUMMA consists of tightly synchronized data movement and computation (see Algorithm 1). Namely, the rank-k update, $C_{ij} \leftarrow A_{ik}B_{kj} + C_{ij}$, of each iteration depends on the data from the broadcasts of the corresponding panels of **A** and **B** as well as the previous iteration's rank-k update. In addition to these *data dependencies*, each broadcast is synchronized with a prior rank-k update since communication operations are initiated at the beginning of each SUMMA iteration, as shown in Figure 1 (we denote such *sequence dependencies* by dashed lines). Such a design ensures that only a minimal memory overhead occurs (although technically, nonblocking broadcasts require more memory than optimal). However, this design also limits the work available to each processor (see the Figure 1), and therefore the amount of parallelism. Specifically, we can parallelize the rank-k update of C as well as the column and row broadcasts of A and B, but SUMMA iterations — although almost independent from one another — are executed serially, one after the other. Furthermore, such design is not tolerant of any source of load imbalance, due to, for example, processor clock variation, network congestion, slack in communication, or — most important for us — due to the inhomogeneity of data from block size or rank variation.
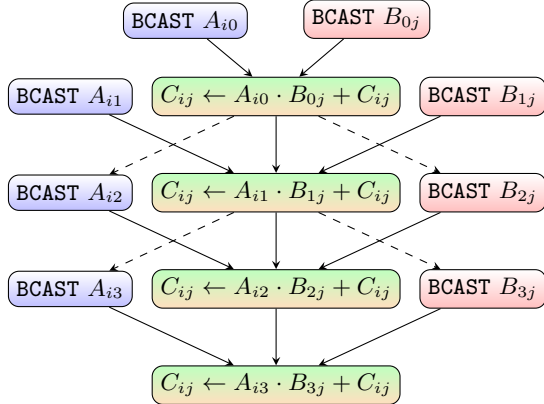

Figure 1: A directed acyclic graph of the procedural SUMMA implementation consisting of broadcast (`BCAST`) and rank-k updates. Solid edges indicate data dependencies, and dashed edges indicate sequence dependencies.

3.2 Task-Based Multiple-Issue SUMMA

To tolerate the data inhomogeneity, whether due to block sparsity or block-rank sparsity and address the above deficiencies of the standard 2D SUMMA, we introduced the following modifications: (1) overdecomposition of data and work, (2) elimination of dependencies between SUMMA iterations, (3) scheduling of multiple iterations of SUMMA at once, and (4) the use of task-decomposed broadcasts.

Although the standard dense SUMMA allows near-perfect load balance of data by using cyclic embedding of matrices onto 2D process grid, it is not feasible to maintain such load balance with irregular data structures throughout the MM computation, even with uniform blocking. In fact the dimension blocking is a central concept in the TILEDARRAY library since it arises naturally in physical problems (hence the need to support arbitrary block sizes, including nonuniform blocking). Thus we support an arbitrary blocking to match the physics of the problem, but assume that the data is overdecomposed (i.e. there are many blocks per processor); this helps to improve load balance of the data.

To improve the load balance of work, the artificial dependencies between SUMMA iterations are eliminated and multiple SUMMA iterations are scheduled at once. This increases the average number of computational tasks per processor, and thus improves the load balance under the assumption that the amount of work performed by each task is randomly distributed. Note that the dependencies between iterations is the result of GEMM operations ($\mathbf{C} = \mathbf{AB} + \mathbf{C}$) via updates of the result matrix, **C**. To decouple this data dependency, we assume that there is enough memory available to split the rank-k update operation into two separate tasks: a matrix-multiplication task producing a temporary block, $C_{ij}^{(k)} = A_{ik} \cdot B_{kj}$, and a reduction of the temporary into the result, $C_{ij} = C_{ij}^{(k)} + C_{ij}$. In our implementation, we mitigate the additional memory cost by automatically switching between GEMM and split matrix-multiply-and-reduces updates based on the data availability. Specifically, the GEMM update is used when only a single thread requires access to a sub-block and the split matrix-multiply and reduces tasks are used when two or more threads must update the same sub-block. This update scheme is essential for high computational throughput and resource management in our task-based SUMMA formulation.

Scheduling multiple SUMMA iterations at once impacts both performance and memory consumption. In that sense the multiple-issue SUMMA is similar to 2.5D and 3D MM algorithms [30] that trade off memory to increase concurrency. The number of SUMMA iterations scheduled concurrently is a user-adjustable parameter and can be tuned to minimize memory consumption (see next Section) or to maximize the computational throughput. In practice, we found that the throughput is maximized when each process initiates I_{opt} SUMMA iterations, where for a process grid with P_r rows and P_c columns $I_{\text{opt}} = \max(2, \min(P_r, P_c))$; this is the default schedule depth, unless specified otherwise. As scheduled SUMMA iterations are retired, additional iterations are scheduled, in a pipelined fashion.

Lastly, data broadcasts are decomposed into several smaller tasks, which allows overlap of communication and computation at the task level. For example, a given process can start the computational work for a given iteration as soon as the minimal amount of data is available, and execute concurrently with the remaining communication tasks. Integration of communication operations allows more efficient work scheduling based on the availability of data in our task approach and improves tolerance of irregular, high-latency communication.

3.3 Memory Overhead of Multiple-Issue SUMMA

The dynamic scheduling of computation and communica-

tion trades off predictable bounds on resource use, in particular memory, for high performance. The maximum memory requirement per process of our multiple-issue SUMMA, for the multiplication of $\mathbb{R}^{M \times K}$ and $\mathbb{R}^{K \times N}$ dense matrices with average block sizes of $m \times k$ and $k \times n$, respectively, is proportional to:

$$I\left(\frac{Mk}{P_\mathrm{r}} + \frac{Nk}{P_\mathrm{c}}\right) + \frac{MN + MK + KN}{P_\mathrm{r}P_\mathrm{c}} \qquad (1)$$

where I is the number of concurrently-scheduled iterations; and P_r and P_c are the number of rows and columns in the process grid, respectively. Eq. (1) correspond to the memory requirements of the 2D, bulk-synchronous SUMMA when $I = 1$. Our implementation also uses additional temporary storage for replicated sub-blocks of the result matrix (not shown in Eq. (1)), but this is a negligible overhead in large problems where the number of blocks is much larger than the number of cores. If, for simplicity, we assume square matrices and process grid ($M = N = K$, $m = n = k$, $P_\mathrm{r} = P_\mathrm{c} = \sqrt{P}$) then Eq. (1) reduces to

$$I\frac{2Nk}{\sqrt{P}} + \frac{3N^2}{P} \qquad (2)$$

where $I(2Nk/\sqrt{P})$ is the memory overhead due to the partial replication of the argument matrices. Under the assumption of overdecomposition, the block size k is much smaller than $N/\sqrt{P}$, hence the first term in Eq. (2) is much smaller than the second, even for $I > 1$. Note that Eq. (1) is derived under the pessimistic assumption that the rate of computation is much lower than the rate of communication. In practice, however, as soon as the data arrives it is consumed by the compute tasks, hence the effective value of I is lower than the actual number of SUMMA iterations scheduled.

The average memory overhead for block-sparse SUMMA can be estimated by scaling each matrix, and corresponding replicated blocks, by the fraction of non-zero elements. In addition, some process may not contain non-zero data for a given iteration. Therefore, the memory overhead of the replicated blocks by $\langle P_\mathrm{r} \rangle / P_\mathrm{r}$ and $\langle P_\mathrm{c} \rangle / P_\mathrm{c}$. With these modifications, Eq. (1) becomes:

$$I\left((1 - z_\mathbf{A})\frac{\langle P_\mathrm{c}\rangle}{P_\mathrm{c}}\frac{Mk}{P_\mathrm{r}} + (1 - z_\mathbf{B})\frac{\langle P_\mathrm{r}\rangle}{P_\mathrm{r}}\frac{Nk}{P_\mathrm{c}}\right) + \\ \frac{(1 - z_\mathbf{C})MN + (1 - z_\mathbf{A})MK + (1 - z_\mathbf{B})KN}{P_\mathrm{r}P_\mathrm{c}} \qquad (3)$$

where $1 - z_\mathbf{X}$ the fraction of non-zero elements for matrix $\mathbf{X}$, and $\langle P_\mathrm{r} \rangle$ and $\langle P_\mathrm{c} \rangle$ are the expectation values for the number of row and column processes, respectively, with non-zero contributions in a single SUMMA iteration.

Note that our implementation may also use less memory than that given in Eqs. (1) and (3). By overdecomposing the rank-k-update and broadcast tasks such that each matrix-multiplication task only computes a small sub-block of the local result matrix. This decomposition is similar to the decomposition of work between nodes within a SUMMA iteration, but without the analogous communication (dependencies) between computation tasks. It allows streaming of data so that processes do not need to hold all data for the block row and column in a given iteration. In addition, higher priority is given to tasks that free resources, which limits the accumulation of temporary storage space. Unfor-

tunately, *a priori* prediction of the actual memory usage is not possible due to the non-deterministic order of execution.

With block-rank-sparse matrices, used in our square root inverse algorithm (see Sections 4.1 and 4.2 for details), the exact memory requirements are similarly non-deterministic as the rank may grow or shrink based on the order of operations and nature of the input data. However, the upper memory bound is equal to that of the full-rank (dense) computation. Unfortunately, the rank of each block cannot easily be determined until runtime, making *a priori* estimations of the memory requirements difficult.

4. RESULTS

In this section, we: (a) discuss the implementation details of tensor arithmetic in low-rank form, (b) evaluate the performance of our methods with this low-rank data structure by computing the inverse square root of matrices used in the quantum chemistry domain, and (c) demonstrate the performance of our implementation for dense matrix multiplication.

Our 2D task-based SUMMA algorithm is implemented in the TILEDARRAY (TA) library [10], which uses the parallel runtime of MADNESS [17] to manage the low-level details of task scheduling and data movement. Unless noted otherwise, we used TA 0.5.0-alpha, and MADNESS dated 08/26/2015. Serial DGEMM from the Intel Math Kernel Library (MKL) 11.2.3 was used as the block multiply-add kernel in our MM tasks. TA and MADNESS were compiled with the Intel Parallel Studio XE 15.3 and Intel MPI 5.0. Both TA and MADNESS can be obtained under the terms of the GNU General Public License.

These tests were performed on a 408-node Cray CS-300 cluster, with two eight-core Intel Xeon E5-2670 CPUs and 64 GB of memory per compute node. In addition, we performed dense MM scaling tests on the IBM BlueGene/Q *Mira* supercomputer at Argonne National Laboratory.

4.1 Clustered Low-Rank Representation

To recover data sparsity in tensors appearing in quantum physics applications we developed the Clustered Low-Rank (CLR) representation [23] that is a general, hierarchy-free compressed tensor format. In this representation, each $\mathbf{M}_{\mathcal{IJ}}$ block of matrix $\mathbf{M}$ is approximated by a low-rank decomposition of the form $\mathbf{M}_{\mathcal{IJ}} \approx \mathbf{XW}^\dagger$, where for a given $\mathbf{M}_{\mathcal{IJ}} \in \mathbb{R}^{m \times n}$ of rank r, $\mathbf{X}$ is $m \times r$ and $\mathbf{W}$ is $n \times r$. $\mathbf{X}$ and $\mathbf{W}$ are constructed from a rank-revealing QR decomposition,[26] $\mathbf{M}_{\mathcal{IJ}}\mathbf{P} = \mathbf{QR}$. Thus $\mathbf{X} \equiv \mathbf{Q}_r$ and $\mathbf{W}^\dagger \equiv \mathbf{R}_r\mathbf{P}^\dagger$ where subscript r indicates that only r most significant columns and rows of $\mathbf{Q}$ and $\mathbf{R}$ were kept; r was determined so that the Frobenius norm of the low-rank approximation error, $\|\mathbf{M}_{\mathcal{IJ}} - \mathbf{XW}^\dagger\|_\mathrm{F}$ is less than or equal to ϵ_lr, a user-defined threshold. For blocks where the rank of $\mathbf{M}_{\mathcal{IJ}}$ is greater than half of full rank, the blocks are stored in their full representation.

To maintain compression, all block operations, such as addition and GEMM, are performed in low-rank form directly whenever possible. Low-rank matrix multiplication of $\mathbf{C} = \mathbf{AB}$, where all matrices appear in low-rank form, uses the following steps. First, create a temporary $\mathbf{Z} = (\mathbf{W}^\mathbf{A})^\dagger\mathbf{X}^\mathbf{B}$, where the superscript letters designate the matrix that $\mathbf{X}$ and $\mathbf{W}$ were constructed from, leaving

$$\mathbf{X}^\mathbf{C}(\mathbf{W}^\mathbf{C})^\dagger = \mathbf{X}^\mathbf{A}\mathbf{Z}(\mathbf{W}^\mathbf{B})^\dagger. \qquad (4)$$
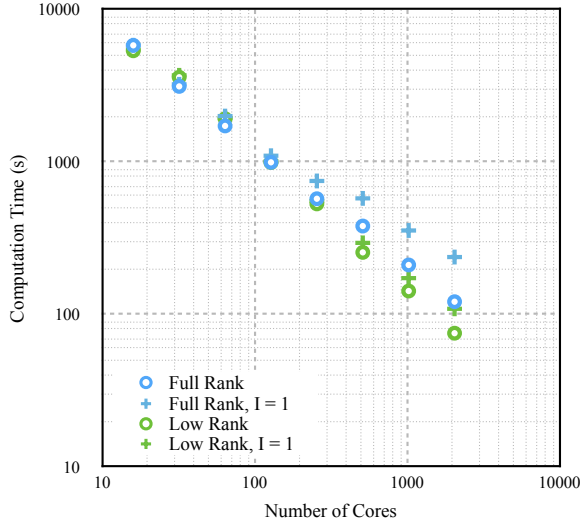
Figure 2: Wall time for computing the square root inverse of the Coulomb matrix. $I = 1$ refers to data obtained with the standard single-issue SUMMA; the rest of data obtained with the multiple-issue SUMMA.
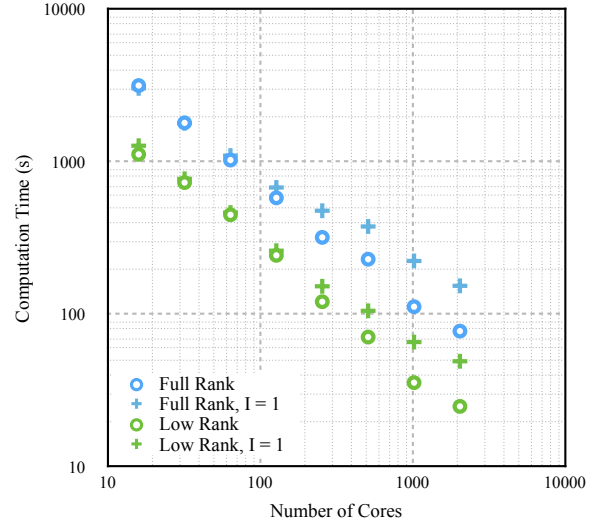

Figure 3: Wall time for computing the square root inverse of the overlap matrix. $I = 1$ refers to data obtained with the standard single-issue SUMMA; the rest of data obtained with the multiple-issue SUMMA.

Then contract $\mathbf{Z}$ with $\mathbf{X^A}$ or $\mathbf{W^B}$ such that the output rank is minimized; contracting $\mathbf{Z}$ with $\mathbf{X^A}$ yields $\mathbf{X^C} = \mathbf{X^A Z}$ and $\mathbf{W^C} = \mathbf{W^B}$.

Low-rank addition of $\mathbf{C} = \mathbf{A} + \mathbf{B}$ may be viewed as a sum of outer products:

$$\mathbf{X^C}(\mathbf{W^C})^\dagger = \sum_{i=1}^{r_A} \mathbf{x}_i^\mathbf{A} * (\mathbf{w}_i^\mathbf{A})^\dagger + \sum_{j=1}^{r_B} \mathbf{x}_j^\mathbf{B} * (\mathbf{w}_j^\mathbf{B})^\dagger, \quad (5)$$

where $\mathbf{x}_i^\mathbf{A}$, $\mathbf{w}_i^\mathbf{A}$, $\mathbf{x}_j^\mathbf{B}$, and $\mathbf{w}_j^\mathbf{B}$ represent column vectors of the corresponding matrix and r_A and r_B are the ranks of $\mathbf{A}$ and $\mathbf{B}$ respectively. From Eq. (5), it is clear that the low-rank matrices representing $\mathbf{C}$ are the union of the input matrices, with $\mathbf{X^C} = \{\mathbf{X^A X^B}\}$ and $\mathbf{W^C} = \{\mathbf{W^A W^B}\}$.

Finally, in order to compress a matrix that is already in a low-rank form $\mathbf{C} = \mathbf{X}(\mathbf{W})^\dagger$ to a more compressed form $\mathbf{C} = \mathbf{X}'(\mathbf{W}')^\dagger$ we first perform a QR decomposition of $\mathbf{X}$ and $\mathbf{W}$

$$\mathbf{X}'(\mathbf{W}')^\dagger = \mathbf{Q^X R^X}(\mathbf{R^W})^\dagger(\mathbf{Q^W})^\dagger \quad (6)$$

and then form a temporary matrix $\mathbf{M} = \mathbf{R^X}(\mathbf{R^W})^\dagger$ giving us

$$\mathbf{X}'(\mathbf{W}')^\dagger = \mathbf{Q^X M}(\mathbf{Q^W})^\dagger. \quad (7)$$

We can perform a low-rank decomposition of $\mathbf{M} = \mathbf{X^M}(\mathbf{W^M})^\dagger$ creating a new compressed representation of the original block where $\mathbf{X}' = \mathbf{Q^X X^M}$ and $\mathbf{W}' = \mathbf{Q^W W^M}$. After compression, the rank of $\mathbf{C}$ has been decreased from $r_A + r_B$ to the rank of $\mathbf{M}$.

Arithmetic on CLR matrices is implemented in terms of the above low-rank block arithmetic. To avoid computing blocks whose norm will be smaller than target precision, block $\mathbf{C}_{\mathcal{IJ}}$ of result matrix $\mathbf{C}$ is only computed if its Frobenius norm *estimate* satisfies $||\mathbf{C}_{\mathcal{IJ}}||_F \leq \epsilon_{\mathrm{sp}} \, \mathrm{area}(\mathbf{C}_{\mathcal{IJ}})$, where $\mathrm{area}(\mathbf{C}_{\mathcal{IJ}})$ is the number of elements in $\mathbf{C}_{\mathcal{IJ}}$, and ϵ_{sp} is a user-defined parameter. Estimated of Frobenius norms of sums and products of blocks are estimated using the upper bounds provided by the triangle inequality and submul-

tiplicativity of the Frobenius norm, e.g. $||\mathbf{A}_{\mathcal{IJ}} + \mathbf{B}_{\mathcal{IJ}}||_F \leq ||\mathbf{A}_{\mathcal{IJ}}||_F + ||\mathbf{B}_{\mathcal{IJ}}||_F$, hence the right-hand side of the inequality is used as the estimate. Complete details of screening the arithmetic operations are provided in Ref. [23].

To summarize: the accuracy of CLR representation and arithmetic is controlled by two user-defined parameters, ϵ_{lr} and ϵ_{sp}. As $\epsilon_{\mathrm{lr}} \to 0$, the CLR representation becomes exact; similarly, when $\epsilon_{\mathrm{sp}} \to 0$ the arithmetic on CLR matrices becomes exact.

4.2 Iterative Square Root Inverse

To compute the inverse square root of a matrix $\mathbf{M}$, where the matrix square root is given by the matrix $\mathbf{M}^{1/2}$ such that $\mathbf{M}^{1/2}\mathbf{M}^{1/2} = \mathbf{M}$ and $\mathbf{M}^{-1/2}\mathbf{M}^{1/2} = \mathbf{I}$, we used the iterative matrix-mulitplication approach in [22] based on the Newton-Schulz method [18, 25] consisting of the following steps in each iteration:

$$\mathbf{X}_n = \alpha \mathbf{Y}_n \mathbf{Z}_n, \quad (8)$$

$$\mathbf{T}_n = \frac{1}{8}\left(-10\mathbf{X}_n + 3\mathbf{X}_n^2 + 15\mathbf{I}\right), \quad (9)$$

$$\mathbf{Z}_{n+1} = \mathbf{Z}_n \mathbf{T}_n, \quad (10)$$

$$\mathbf{Y}_{n+1} = \mathbf{T}_n \mathbf{Y}_n, \quad (11)$$

where $\mathbf{I}$ is the identity matrix, and upon convergence $\sqrt{\alpha}\mathbf{Z} = \mathbf{M}^{-1/2}$ is the sought square root inverse. The starting guesses are $\mathbf{Z}_0 = \mathbf{I}$, $\mathbf{Y}_0 = \mathbf{M}$, and α is chosen to scale $\mathbf{M}$ such that $||\alpha\mathbf{M} - \mathbf{I}||_2 \leq 1$.

Square root inverses of the overlap and Coulomb operator matrices, $\mathbf{S}$ and $\mathbf{V}$ respectively [35], were computed for a cluster of 190 water molecules with the cc-pVTZ-RI Gaussian basis, consisting of 141 basis functions *per* water molecule, for a total basis size of 26,790 basis functions. The matrices use natural blocking, i.e. each 141 by 141 block spans basis functions associated with the corresponding water molecule. $\epsilon_{\mathrm{sp}} = 10^{-13}$ and $\epsilon_{\mathrm{lr}} = 10^{-6}$ were used.

Our performance tests executed 10 iterations of this algorithm; the resulting wall times are reported in Figs. 2 and
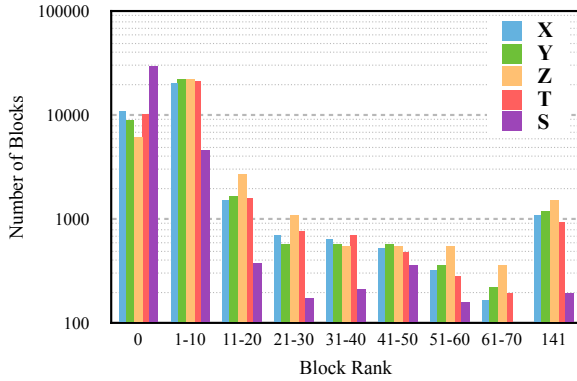
Figure 4: Distribution of block ranks of the intermediate matrices from the 5th iteration of the inverse square root computation of the overlap matrix. A rank of "0" indicates a block contains no data, and "141" indicates full rank.

Figure 5: Wall time of square dense matrix multiplication with TILEDARRAY, CTF, and SCALAPACK. Matrix size = 32,768.

3. Good strong scaling is observed: for example, for the Coulomb matrix in dense and low-rank representations the parallel efficiency is maintained at 43% and 59%, respectively, upon scaling from 1 to 128 nodes. For the overlap matrix, the corresponding figures are 28% and 31%. The observed differences in scaling between overlap and Coulomb are likely due to the much faster (exponential) decay of overlap matrix with distance and hence greatly reduced amount of work due to more efficient screening of matrix operations: without such screening the time to compute the inverses of Coulomb and overlap matrices in full-rank representation would take exactly the same amount of time, whereas in practice the inversion of the overlap is roughly a factor of 2 faster than that of the Coulomb matrix in full-rank form.

Unlike $\mathbf{S}$, $\mathbf{V}$ does not have any block-sparsity, only block-rank-sparsity. Hence computing $\mathbf{V}^{-1/2}$ in the full-rank form essentially performs the same amount of work as would an implementation using a standard dense matrix package. The apparent performance of the full-rank inverse was estimated by counting FLOPs from 4 GEMMs per each iteration; the resulting performance on 1 node for the Coulomb case is 263 GLOPs, or 79% of theoretical peak, which is close to optimal. On 128 nodes the apparent throughput is respectable 99 GLOPs.

The benefit of the low-rank representation is immediately apparent for the overlap matrix on 1 node, whereas for the Coulomb matrix the low-rank implementation outperforms the full-rank counterparts when the number of processors is large, due to the greatly reduced amount of communication in the low-rank case. Also obvious is the importance of scheduling multiple SUMMA iterations for performance in the high-end strong scaling regime for both dense and low-rank matrices. For example, on 128 nodes the use of single-issue SUMMA when computing $\mathbf{S}^{-1/2}$ in low-rank form is 97% slower than multiple-issue SUMMA.

Figure 4 demonstrates the extent of the data inhomogeneity in the matrices involved in these computation. It shows a histogram of block ranks of 5th iteration intermediates $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$, $\mathbf{T}$, as well as the input $\mathbf{S}$ matrix. Most blocks in these matrices have very low ranks. However, a significant number of blocks have close-to-full rank, which is typical behavior for matrices with decay.
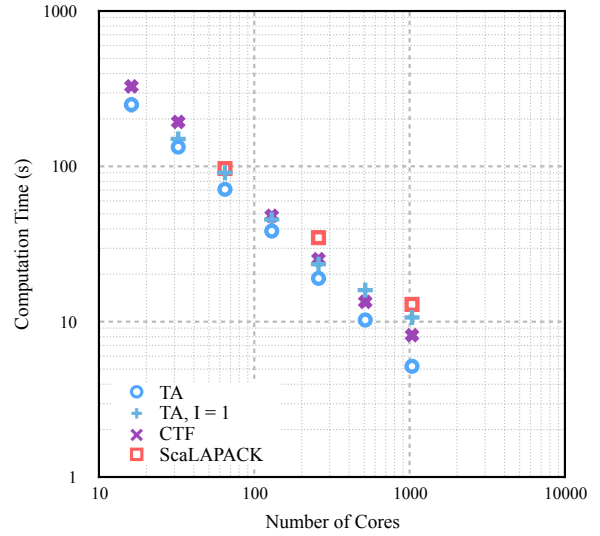
4.3 Dense Matrix Multiplication

In the limit where the low-rank threshold and sparsity thresholds become zero, i.e. $\epsilon_{\mathrm{lr}} \to 0$ and $\epsilon_{\mathrm{sp}} \to 0$, CLR matrices become full-rank (dense) matrices. In addition, the features of our approach that make it appropriate for rank-structured matrix multiplication also lead to excellent performance for dense matrices. Thus, in this section we evaluate the performance of our SUMMA formulation in TILEDARRAY (TA) [10] for the case of dense, full-rank matrices in absolute terms and relative to that of SCALAPACK [6] and CYCLOPS TENSOR FRAMEWORK (CTF) 1.1 [32, 31], a state-of-the-art, dense tensor contraction implementation based on 2.5D SUMMA. Performance was evaluated in two prototypical setups: on a commodity cluster and on a high-end IBM BlueGene/Q *Mira* supercomputer at Argonne National Laboratory.

On the commodity cluster, we used the version of ScaLA-PACK provided by Intel MKL 11.2.3. CTF was compiled with the same Intel compiler described at the beginning of this section. Unlike TA, however, ScaLAPACK and CTF use the threaded version of BLAS DGEMM routine from MKL for their multiply-add kernels.

Due to the constraints of the existing parallel dense MM software and to simplify the performance analysis, we perform multiplications of square, uniformly-blocked, double-precision matrices; our implementation is completely general. The matrix size used in the tests on the commodity cluster was 32,768, with block size of 256 used for TILEDAR-RAY. Reported wall times are averages of 15 repeated multiplications of same input matrices.

In Figure 5, we show the result of our strong scaling tests, which vary from 16 to 1024 cores. Each of the MM software packages shows linear scaling across the range of the tests. However, of the three packages, TA achieved the lowest computational time in all cases. In fact, the performance differences between TA, ScaLAPACK, and CTF are significant. On 64 compute nodes, ScaLAPACK took between $1.36 \sim 2.50$ times longer to complete relative to TA, while

CTF took $1.25 \sim 1.59$ times longer to complete. The parallel efficiency of TA MM relative to 1 node is between 84.3% and 63.7% on 64 nodes.

We also studied the performance of TA with the number of concurrent iterations, I, set to one to demonstrate the performance benefits of multiple-issue SUMMA iterations. As expected, the performance of TA with $I = 1$ was approximately the same on a single node, but performs much more slowly with larger node counts. Scaling with a single-issue (standard) SUMMA is linear, but substantially slower than that of multiple-issue SUMMA, and is consistent with the square root inverse performance tests in Section 4.2.
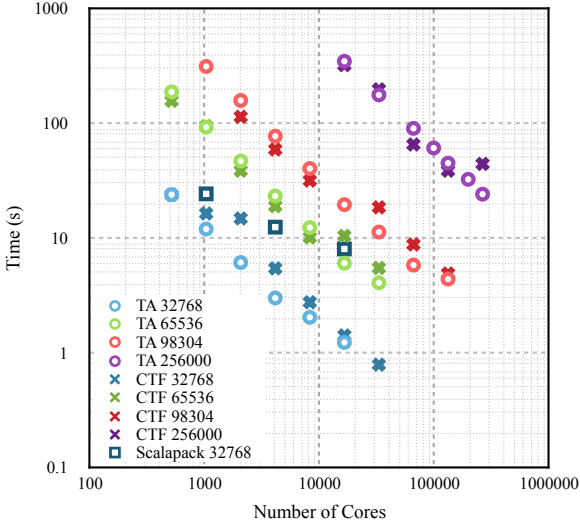

Figure 6: Wall time of square dense matrix multiplication with TILEDARRAY, CTF, and SCALAPACK on IBM Blue-Gene/Q. The matrix sizes included in these tests are 32,768, 65,536, 98,304, and 256,000.

Finally, we evaluated the performance of our SUMMA implementation on the IBM BlueGene/Q *Mira* supercomputer at Argonne National Laboratory. In these performance tests, we used TA 0.4.0-alpha, MADNESS dated 03/02/2015, SCALAPACK 2.0.2, and CTF 1.1. The parallel version of ESSL 5.1 was used for the CTF and ScaLAPACK performance tests, while the serial version is used for TA. TA and MADNESS were compiled with GCC 4.7.2; CTF is compiled with XLC 12.1. Both used the V1R2M2 compiler driver.

In Figure 6, we show the performance of TA, CTF, and SCALAPACK with matrices of various sizes. TILEDARRAY used 128×128 blocks. The number of cores varies from 512 to 262144. Both TA and CTF are found to be competitive with each other, with excellent strong and weak scaling, though each behaves differently across the range of the tests. We found SCALAPACK, on the other hand, to scale very poorly relative to the TA and CTF, with almost an order of magnitude difference in computational time on 16384 cores.

5. CONCLUSIONS

We presented a task-based multiple-issue formulation of the Scalable Universal Matrix Multiplication Algorithm (SUMMA) that performs well for multiplication of matrices with irregular structure, such as block-sparsity and block-rank-

sparsity. An implementation of our algorithm in an open-source library TILEDARRAY was used to compute iteratively square-root inverses of two realistic matrices from the domain of quantum chemistry, using block-sparse and block-rank-sparse representations. Excellent strong scaling from 16 to 2048 cores was observed on a commodity cluster. For dense matrices our implementation is competitive with state-of-the-art dense tensor algebra libraries like CYCLOPS TENSOR FRAMEWORK, both on a commodity cluster using up to 1024 cores as well as on a high-end IBM BG/Q supercomputer using up to 262 thousand cores.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] AGARWAL, R. C., BALLE, S. M., GUSTAVSON, F. G., JOSHI, M., AND PALKAR, P. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. & Dev. 39*, 5 (Sept. 1995), 575–582.

[2] AGARWAL, R. C., GUSTAVSON, F. G., AND ZUBAIR, M. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Res. & Dev. 38*, 6 (Nov. 1995), 673–681.

[3] BALLARD, G., BULUÇ, A., DEMMEL, J., GRIGORI, L., LIPSHITZ, B., SCHWARTZ, O., AND TOLEDO, S. Communication optimal parallel multiplication of sparse random matrices. In *SPAA '13: Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures* (July 2013), ACM Request Permissions.

[4] BARUCH, E. *Designing Communication-efficient Matrix Algorithms in Distributed-memory Cilk*. Tel-Aviv University, 2001.

[5] BERNTSEN, J. Communication efficient matrix multiplication on hypercubes. *Parallel Computing 12*, 3 (Dec. 1989), 335–342.

[6] BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[7] BOSILCA, G., BOUTEILLER, A., DANALIS, A., FAVERGE, M., HAIDAR, A., HERAULT, T., KURZAK, J., LANGOU, J., LEMARINIER, P., LTAIEF, H., LUSZCZEK, P., YARKHAN, A., AND DONGARRA, J. Flexible development of dense linear algebra algorithms on massively parallel architectures with

DPLASMA. In *Proceedings of PDSEC 2011* (Anchorage, United States, may 2011).

[8] Buluç, A., and Gilbert, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Proceedings of the 37th Int'l Conf. on Par. Proc. (ICPP '08)* (2008), pp. 503–510.

[9] Buluç, A., and Gilbert, J. R. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Sci. Comput. 34*, 4 (Jan. 2012), C170–C191.

[10] Calvin, J. A., and Valeev, E. F. Tiledarray: A massively-parallel, block-sparse tensor library written in C++ . https://github.com/valeevgroup/tiledarray/, 2015.

[11] Cannon, L. E. *A cellular computer to implement the kalman filter algorithm.* PhD thesis, Ann Arbor, July 1969.

[12] Choi, J. A fast scalable universal matrix multiplication algorithm on distributed-memory concurrent computers. In *IPPS-97* (Apr. 1997), IEEE Comput. Soc. Press, pp. 310–314.

[13] Choi, J. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. *Concurr. Comput. 10*, 8 (July 1998), 655–670.

[14] Choi, J., Walker, D. W., and Dongarra, J. J. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurr. Comput. 6*, 7 (Oct. 1994), 543–570.

[15] Fox, G. C., Otto, S. W., and Hey, A. J. G. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Comput. 4*, 1 (1987), 17–31.

[16] Gunnels, J., Lin, C., Morrow, G., and van de Geijn, R. A. A flexible class of parallel matrix multiplication algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International … and Symposium on Parallel and Distributed Processing 1998* (1998), pp. 110–116.

[17] Harrison, R., and MADNESS Community. MADNESS project hosted on Github. https://github.com/m-a-d-n-e-s-s/madness, 2015.

[18] Higham, N. J. Stable iterations for the matrix square root. *Numerical Algorithms 15*, 2 (1997), 227–242.

[19] Hirata, S. Tensor Contraction Engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J. Phys. Chem. A 107*, 46 (Nov. 2003), 9887–9897.

[20] Hollman, D. S., Schaefer, H. F., and Valeev, E. F. A tight distance-dependent estimator for screening three-center Coulomb integrals over Gaussian basis functions. *J. Chem. Phys. 142*, 15 (Apr. 2015), 154106, arXiv:chem-ph/1410.4882.

[21] Husbands, P., and Yelick, K. Multi-threading and one-sided communication in parallel LU factorization. In *SC '07* (New York, New York, USA, 2007), ACM Press, p. 1.

[22] Jansík, B., Høst, S., Jorgensen, P., Olsen, J., and Helgaker, T. Linear-scaling symmetric square-root decomposition of the overlap matrix. *J.*

Chem. Phys. 126, 12 (Mar. 2007), 124104.

[23] Lewis, C. A., Calvin, J. A., and Valeev, E. F. Clustered Low-Rank Tensor Format: Introduction and Application to Fast Construction of Hartree-Fock Exchange. arXiv:1510.01156. submitted to *J. Chem. Phys.*

[24] Neese, F., Wennmohs, F., and Hansen, A. Efficient and accurate local approximations to coupled-electron pair approaches: An attempt to revive the pair natural orbital method. *J. Chem. Phys. 130*, 11 (2009), 1–18.

[25] Niklasson, A. Iterative refinement method for the approximate factorization of a matrix inverse. *Physical Review B 70*, 19 (2004), 193102.

[26] Quintana-Ortí, G., Sun, X., and Bischof, C. H. A BLAS-3 version of the QR factorization with column pivoting. *SIAM J. Sci. Comput. 19* (1998), 1486–1494.

[27] Rajbhandari, S., Nikam, A., Lai, P.-W., Stock, K., Krishnamoorthy, S., and Sadayappan, P. CAST: Contraction algorithm for symmetric tensors. In *Parallel Processing (ICPP), 2014 43rd International Conference on* (2014), IEEE, pp. 261–272.

[28] Schatz, M. D., Poulson, J., and van de Geijn, R. A. Scalable universal matrix multiplication algorithms: 2D and 3D variations on a theme. *submitted to ACM Transactions on Mathematical Software* (2012), 1–30.

[29] Solomonik, E., Bhatele, A., and Demmel, J. Improving communication performance in dense linear algebra via topology aware collectives. In *SC '11* (New York, New York, USA, 2011), ACM Press, p. 1.

[30] Solomonik, E., and Demmel, J. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing.* Springer Berlin Heidelberg, 2011, pp. 90–109.

[31] Solomonik, E., Devin, M., Hammond, J. R., and Demmel, J. Cyclops Tensor Framework: Reducing communication and eliminating load imbalance in massively parallel contractions. Tech. Rep. UCB/EECS-2012-210, Nov. 2012.

[32] Solomonik, E., Hammond, J. R., and Demmel, J. A preliminary analysis of Cyclops Tensor Framework. Tech. Rep. UCB/EECS-2012-29, Mar. 2012.

[33] Solomonik, E., Matthews, D., Hammond, J. R., Stanton, J. F., and Demmel, J. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel Distrib. Comput. 74*, 12 (Dec. 2014), 3176–3190.

[34] Strassen, V. Gaussian elimination is not optimal. *Numer. Math. 13*, 4 (Aug. 1969), 354–356.

[35] Szabo, A., and Ostlund, N. S. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory.* McGraw-Hill, New York, 1989.

[36] van de Geijn, R. A., and Watts, J. SUMMA: Scalable universal matrix multiplication algorithm. *Concurr. Comput. 9*, 4 (1997), 255–274.

[37] Vandebril, R., Van Barel, M., Golub, G., and Mastronardi, N. A bibliography on semiseparable matrices*. *Calcolo 42*, 3-4 (2005), 249–270.