NEAR-OPTIMAL PARALLEL TENSOR CONTRACTIONS

Mihai Zorca, Daniel Trujillo, Berke Egeli
Supervised by *Grzegorz Kwasniewski*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We present a near communication optimal tensor contraction framework for parallel systems. Previous frameworks primarily focus on computational optimizations. Our contribution is to make use of a graph based analytical approach to perform I/O optimal contractions. Our framework is capable of performing general tensor contractions, meaning that it can perform contractions with arbitrary number of tensors with arbitrary modes and dimensions. For several operations, we communicate significantly less than the state-of-the-art contraction library Cyclops Tensor Framework: up to $4\times$ fewer bytes sent for Matrix-matrix multiplications (MMM) and $6\times$ fewer bytes sent for Matricized Tensor times kathri-rhao product (MTT-KRP).

1. INTRODUCTION

Motivation. Tensor contractions are an important class of operations in fields like machine learning [1] and quantum chemistry [2, 3]. Despite their similarity to matrix-matrix multiplications (MMM), they are more difficult to optimize due to strided access patterns. To address this issue, multiple tensor contraction frameworks have been developed in the past [3, 4, 5, 6, 7]. However, most of these frameworks focus on optimizing the computational performance of the contractions. Since I/O operations can be an important source of overhead for tensor contractions, particularly in parallel systems, optimal communication between parallel processors can lead to significant performance gains. Therefore, in this paper we introduce a communication optimal tensor contraction framework for parallel systems.

Related work. Tensor contraction frameworks have been developed mostly for the quantum chemistry field due to a popular family of methods, Coupled Clusters, requiring efficient tensor contractions.

NWChem [6] and Cyclops Tensor Framework (CTF) [3] are tensor contraction frameworks developed for massively parallel systems. NWChem modularizes its parallel tensor contraction framework. It uses Tensor Contraction Engine (TCE) [8] for local tensor contractions and Global Arrays (GA) framework [9] for its communication layer. TCE generates sequences of tensor contractions and minimizes the memory used by intermediate steps. GA allows processors to access data which is physically stored in different processors. It avoids synchronization between different nodes by performing one sided data communication. The main drawback of GA is that communication patterns can be irregular and therefore unbalanced.

CTF cyclically subdivides tensors to create regular communication patterns, thus achieving better load balance than NWChem. It uses a mapping framework to determine the optimal mapping for tensor contractions at runtime based on the memory usage of nodes and the communication volume. By exploiting index transpositions and partial unpacking, CTF transforms sequential and symmetric tensor contractions into MMM, enjoying increased performance benefits.

More recently, Springer and Bientinesi developed a new framework called Tensor Contraction Code Generator (TCCG) [7]. As the name suggests, TCCG focuses on generating high performance tensor contraction code rather than dynamically executing contractions. It leverages techniques used by previous frameworks as well as the novel so-called *Gemm-like Tensor Tensor Contraction* (GETT). GETT's primary objective is to minimize the runtime overhead of rearranging input tensors for GEMM calls. TCCG generates multiple candidate codes, scores each of them based on a predetermined metric and selects the code with the best score.

The majority of the tensor contraction frameworks focus on optimizing the performance of the computations. Our framework, on the other hand, aims to optimize the I/O operations. Our framework is conceptually similar to NWChem. We also modularize the communication and computation. The primary difference is that our framework makes use of a graph based analytical approach [10] to determine I/O optimal communication patterns.

2. BACKGROUND

This section introduces the definition of tensor contractions and terminology relevant to this problem. We then discuss the theoretical communication lower bounds discovered for tensor contractions and other related linear algebra operations. Finally, we introduce a class of programs and a method to analyze these programs. This method can be used to derive communication lower bounds and schedules for tensor contraction operations.

Tensor. Tensors are N-mode arrays [11]. The number of modes defines the number of integer values we need to index an element of the tensor. Dimension of a mode is defined as the length or size of a mode [11].

Iteration Variable. Iteration variables are iterators that get their values from the iteration domain during the program execution [10]. They provide a way to access the elements of tensors across a specific tensor mode.

Iteration Vector. Iteration vectors are a vector of all iteration variables [10].

Iteration Domain. Iteration domain is the set of all possible values an iteration vector can assume during the program execution [10].

Tensor Contractions. Tensor contractions, for our purposes, are a class of operations that multiply elements of input tensors determined by the corresponding iteration variables of an iteration domain and accumulate these multiplications in the elements of an output tensor.

Tensor contraction operations are the main focus of our framework.

I/O Lower Bounds. Tensor contractions are similar to MMM and other linear algebra operations in terms of the computations they perform. Thus, understanding the I/O lower bounds for these operations can provide important insights into how well our framework can perform.

Hong and Kung showed that to perform MMM on 2 n-by-n matrices using the conventional $\Theta(n^3)$ algorithm, $\Omega(n^3/\sqrt{M})$ words of data must be communicated from slow memory to fast memory of size M [12]. Later, Irony, Toledo and Tiskin showed that, in the parallel case, the lower bound for communication was similar. For the parallel case with P processors, the lower bound can be generalized as the following expression: $\Omega((n^3/P)/(\sqrt{n^2/P})) = \Omega(n^2/\sqrt{P})$ [13]. The lower bound is in the order of number of arithmetic operations per processor divided by the square root of memory per processor. Ballad et al. showed similar results for a wide variety of linear algebra operations including LU, Cholesky, LDL^T, QR factorizations, Gram-Schmidt algorithm, and algorithms for eigenvalues and singular values [14].

Amongst all of the tensor contraction frameworks we have introduced in the introduction, CTF is the framework with the most amount of focus on optimal communications. The authors claim that CTF does no more communication than the following lower bound: $\Omega(\frac{F}{\sqrt{P.c.S}} - \frac{S}{P})$ [3]. Here, F is the total number of operations, $c \in [1, P^{1/3}]$, M is the size of memory on each processor, P is the number of processors and $S = \Theta(M.P/c)$ is the size of an operand tensor.

Simple Overlap Access Programs (SOAP). SOAP are a class of programs which can be analyzed to obtain precise I/O lower bounds for the said program. For a program to be considered as SOAP it needs to have a specific access pattern called simple overlap [10]. All of the data accessed in the simple overlap pattern comes from disjoint tensors and different accesses to the same tensor are offset by constant strides.

```
for i in range(1, N):
    for j in range(1, N):
        A[i,j+1] = A[i-1,j] + B[i]
```

Listing 1. An example of simple overlap access pattern. A and B are disjoint tensors. Array A is accessed twice through [i,j+1] and [i-1,j]. References to A is offset by constant strides, [0,1] and [-1,0].

It has been shown by Kwasniewski et al. that the I/O cost of a program is bounded by the size of the largest subcomputation subgraph H in the Computation Directed Acyclic Graph (CDAG) of a program [10]. SOAP concept is particularly useful because it provides a way to find a tighter upper bound on the size of H.

What is of interest to us, is that Tensor contractions can be transformed to SOAP. In our tensor contraction description we mentioned the multiplications are accumulated in an output tensor. This usually translates to the fact that there is one tensor which is both the input and the output which violate the simple access pattern. Kwasniewski et al. show that this issue can be solved by adding an additional tensor mode to the output then reducing the elements across that mode [10].

Symbolic Directed Graph (SDG) for SOAP. A natural way to model program executions is to build a CDAG $G = (V, E)$. The size of a CDAG depends on the size of our iteration domain [10]. However, multi-statement SOAP can be modelled by a slightly different abstraction, SDGs. This is because, it has been shown that the I/O lower bound for SOAP depends on computational intensity instead of the size of G [10].

The main objective in representing SOAP as an SDG is to find subgraph partitions of the SDG, such that we maximize the input/output tensor reuse and minimize the data recomputation. In other words, we only need to "model dependencies between arrays and how they are accessed." [10] Therefore, reuse of data and increasing data locality is crucial.

Symbolic directed graph can be denoted as $G_s = (V_s, E_s)$. The vertices represent the tensors accessed by SOAP and the edges represent the data dependencies between the tensors [10]. There is an edge between the tensors A_s and A_d if A_s is used in the computation of A_d [10].

Let $A \in V_s$, $|A|$ be the number of vertices that belong to tensor A in CDAG G, $S(A)$ be the set of all subgraphs in G_s that contain A and ρ_H be the computational intesity of the subgraph H. Kwasniewski et al. proved that the I/O cost for a SOAP is lower bounded by $\sum_{i \in V_s} \frac{|A|}{max_{H \in S(A)} \rho_H}$ [10].

3. DISTRIBUTED TENSOR CONTRACTION

In this section, we describe our implementation of a general-purpose tensor contraction library, which aims to minimize communication for all tensor ranks.

Our implementation is capable of running fully-general distributed tensor contractions: an arbitrary number of ranks to run on, an arbitrary number of input tensors of arbitrary order, and any contraction operation given by an einsum string. Additionally, a binding to Python is provided. Using this binding, it is possible to perform the distributed contraction on Python lists or NumPy arrays.

Similar to COSMA [15], the contraction consists of three main steps: input tensor distribution, the local contractions and the reduction of output tensors. The following sections describe each of these steps in greater detail.

3.1. Tensor Distribution

To minimize communication, the *distribution schedule*, i.e. the allocation of tensor slices to (compute) ranks, has to be carefully considered. A sub-optimal distribution schedule will lead to increased communication, as the ranks will have to send more data than strictly necessary.

Further, a distribution model has to be chosen. In our case, the data is assumed to be distributed in a non-redundant fashion. If two or more ranks share the same slice of a tensor then only one of them will store the data of this slice. This is in contrast to other models, such as [3], which allow replication of tensor data if sufficient extra memory is available. Non-redundancy requires the least amount of space, and thus scales best to larger data sets and number of cores. We make an additional assumption to simplify the implementation and testing: initially, all data is assumed to only be present in rank 0. This does not affect the communication volume, but could pose a significant performance bottleneck.

Distribution Schedule. Given are the contraction operation on k inputs (as einsum string), the number of ranks n and the input dimensions $I_1, I_2, \ldots, I_k$. The goal is to assign, for each input $i \in [k]$ and to every rank j, a slice S_{ij} of the i-th input. If this i-th input is a tensor of order m, then the slice is of the form: $S_{ij} = ((x_1, y_1), \ldots, (x_m, y_m))$. The pair (x_i, y_i) indicates that along mode i, the rank is assigned all values starting from the x_i'th until and including the $(y_j - 1)$'st element.

We compute the distribution schedule using the SOAP analysis algorithm described and implemented in [10]. It aims to give I/O efficient schedules that minimize communication.

Distributing the Tensors. Given a distribution schedule, we consider the set of chosen slices $\{S_{i1}, \ldots, S_{in}\}$ for each input i. For any set of ranks $x_1, \ldots, x_g$ such that $S_{ix_1} = \ldots = S_{ix_g}$, we create an MPI group (by calling `MPI_Group_incl()`) containing the ranks $x_1, \ldots, x_g$, as well as rank 0 (irrespective of its slice). Inside this group, by having root 0 issue a single call to `MPI_Distribute()`, the data is distributed to all ranks inside the group. Using `MPI_Type_create_subarray()`, we can directly access the slice of the tensor at the sender rank 0. All receiving ranks store the received data as a contiguous 1D-array, so local contractions can treat data like a regular tensor. This procedure is executed in parallel (limited in our case by the bottleneck that rank 0 poses) for all groups of ranks sharing the same slice.

After all input tensors have been (sequentially) iterated over, every rank has received exactly the slice, of each input, that it was assigned by the distribution schedule.

3.2. Local Contractions

Multi-index. Multi-index is a tool that allows us to iterate over any iteration domain given certain parameters. It is essentially a linearized abstraction of nested for loops with an arbitrary number of for statements. It takes a vector of limits as input, whose elements represent the maximum value an iteration variable can take in the iteration domain.

Multi-index keeps a vector of iteration variables. Each element of this vector keeps the current value of the iteration variable. The iteration variable vector is ordered. The last element of the iteration variables vector represents the iteration variable of the innermost loop and the first element of the vector represents the outermost loop's iteration variable. The iteration variables are incremented from innermost variable to outermost variable. An iteration variable is incremented only after the iteration variable in the next vector element reaches its limit with the exception of the innermost loop variable.

Using an extensible data structure like a vector is what enables us to perform general tensor contractions. The multi-index structure can represent an arbitrarily large iteration domain with arbitrary number of iteration variables. Thus, we don't have to know the total number of distinct tensor modes that come from the input tensors at compile time.

Partial Tensor Contractions. To perform local contractions, each rank receives slices of input tensors. These slices of the inputs might not necessarily be continuous in the original tensors. However, in the local nodes they are stored in continuous 1D arrays. This allows us to treat each local contraction as a single, independent contraction. Also, storing the inputs in a 1D array is necessary for general tensor contractions as we abstract away the number of modes each input tensor has.

To perform local contractions we require specific metadata about the input and output tensors. These are the dimensions of the tensor modes, the iteration variables of tensors and their order for each tensor. Using this metadata and the current state of the multi-index, we are able to access the correct elements of the input/output tensor for each time we iterate over the iteration domain. After accessing the correct elements of the input/output tensors, computation of correct result reduces to multiplication of input elements and accumulation of these multiplications in the output elements.

3.3. Reduction of Output Tensors

After the local contractions have been computed, the results have to be reduced. The reductions are performed similar to the tensor distribution described in section 3.1. In the previously computed distribution schedule (given by the SOAP analysis algorithm), each rank is assigned a slice of the output tensor. All ranks sharing a slice are grouped (together with rank 0), and within each group, a simple `MPI_Reduce()` call is run. After rank 0 receives the result of the reduction (in contiguous form), it then copies this data into the corresponding slice of the output tensor. This copy is also performed via in-place `MPI_Sendrecv()`.

After this is repeated for all groups of ranks sharing a slice, all values of the final result tensor are on rank 0.

4. EXPERIMENTAL RESULTS

To evaluate our framework, we perform several measurements to quantify the communication that occurs during the tensor contractions. As our framework is not built to achieve top performance, but rather to be generic and use as little I/O as possible, we do not measure execution time of tensor contractions. Instead, we compare the communication used by our framework with the corresponding theoretical lower bound - which is given to us by the SOAP analysis algorithm. As a next step, we compare our results with the state-of-the-art tensor contraction library Cyclops Tensor Framework (CTF), given that this framework has the most focus on optimal communication among other already-published tensor libraries.

Experimental setup. We run our experiments on ETH's Euler cluster, using up to a 1000 Intel Xeon E3-1585Lv5 cores with a 3.0 - 3.7 GHz frequency and a 8MB cache. For each experiment, we allocate 2048MB of RAM per core that we use. For compiling and running CTF, we rely on gcc (version 4.8.5), openblas (version 0.3.15), net-lib-lapack (version 3.9.1), cmake (version 3.20.3) and openMPI (version 4.0.2), which are all available by default on Euler. Our code is compiled with gcc (version 11.1.0) and MPICH (version 3.3.2).

Profiling. For our library, we implemented manual counters on each broadcast call. They confirm that our library communicates exactly as much as is expected from the given distribution schedule. Specifically, if a rank is assigned a slice $((x_1, y_1), \dots, (x_m, y_m))$ of some input, then we expect to communicate $4\prod_{i=1}^{m}(y_i - x_i)$ bytes (4 byte `float`).

We also profile both our library and Cyclops using a profiler called mpiP [16]. It hooks into any MPI call and, among other data, keeps track of all bytes sent. Profiling with mpiP matches the manual counters on our library.

Correctness. To ensure correctness of our implementation, we compared the results produced by our framework with the output of NumPy and opt-einsum, using the same tensor contraction. For a diverse set of inputs, we manually verified that our outputs match perfectly, disregarding small deviations due to floating point arithmetic.

Comparison with theory. As described above, we first compare our results with the theoretical lower bound, given to us by the SOAP algorithm. Specifically, we measure the influence of *strong scaling* and *weak scaling* on the amount of bytes communicated between the compute ranks.

For our benchmarks, we consider two types of tensor contractions: MMM ($ij, jk \rightarrow ik$) and *matricized-tensor times Khatri-Rao product* (MTT-KRP) ($ijk, kl, jl \rightarrow il$).

We use the following setups:

1. *Strong scaling*: When performing MMM, we consider a *square case*, where we set all dimensions to 1024 ($i = 1024$, $j = 1024$ and $k = 1024$), and we consider a *skinny case* ($i = 1$, $j = 16777216$ and $k = 1$).
   For performing MTT-KRP, we also consider a *square case* ($i = 1024$, $j = 1024$, $k = 1024$ and $l = 1024$) and a *skinny case* ($i = 2048$, $j = 10$, $k = 10$ and $l = 2048$). We then vary the number of cores, ranging from 50 to 1000 (with steps of 50).

2. *Weak scaling*: We fix the number of cores to 1000 and vary the dimensions, ranging from 500 to 10,000 (with steps of 500).

We compare its communication volume with that of CTF. CTF distinguishes itself by putting emphasis on minimizing communication, making it the perfect candidate for comparison with our framework. Although CTF allows data replication of tensor data, thereby requiring less communication than minimally necessary in our distribution model, we still communicate less than CTF in some of the cases.

Note: for both cyclops and our code, the communication volume is deterministic – unchanging across runs and pre-determined by the (deterministic) distribution schedule. In our case, the schedule is the one given by the SOAP analysis, and for cyclops it is computed internally.

Strong Scaling Results. Figure 1 and 2 present the results for MMM, varying the number of cores for the *square* and *skinny* case respectively. The results for the *square* case clearly show that we exactly communicate as much as the theoretical lower bound, for any of the tested number of cores. The plot for the *skinny* case shows that we communicate more than the theoretical minimum. However, as the number of cores increases, we converge towards the lower bound.

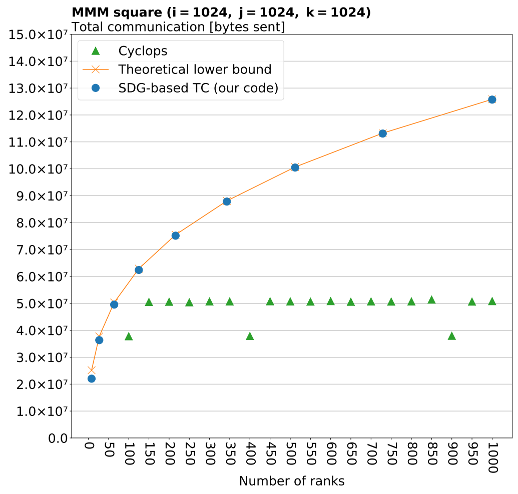

Fig. 1. *Strong scaling*: Comparison of square MMM communication with the theoretical lower bound and CTF.

Similarly, Figure 3 and Figure 4 present the amount of bytes communicated when performing MTT-KRP. Again, our plots show that we communicate more than the theoretical lower bound. We clearly converge to the lower bound as the number of ranks increase: we only communicate approximately $1.4\times$ (*square*) and $1.15\times$ (*skinny*) more when the number of cores is set to 1000.

The deviations from the optimum when using a low number of ranks may be explained by considering the SOAP analysis algorithm. It has to decompose the input tensors into slices. In doing so, the algorithm is constrained by integer assignments. The more cores are available to decompose over, the closer the optimal solution can be approximated. It may again be worth noting that we match the communication of the computed schedule exactly.

Next we compare our results to that of CTF. Figure 1 and 2, the *square* and *skinny* cases respectively, present a comparison between our framework and CTF for the MMM tensor contraction. In the case of *square* MMM, CTF communicates less than our code – or the theoretical lower bound given by the SOAP analysis. Here we may have to consider the distribution model in two ways. On the one hand, our code avoids replication of tensor data. On the other hand, the lower bound given by SOAP is *memory-independent*, whereas cyclops considers the available per-core fast memory. In the case of *skinny* MMM, the new tensor distribution shines. CTF communicates up to $4\times$ as much as our library, consistently for any of the tested number of ranks.

Figure 3 (*square*) and Figure 4 (*skinny*) present the comparison for MTT-KRP. The *square* result shows clearly that

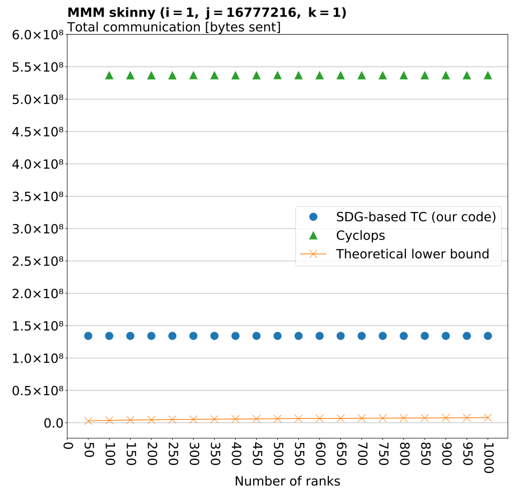

Fig. 2. *Strong scaling*: Comparison of skinny MMM communication with the theoretical lower bound and CTF.

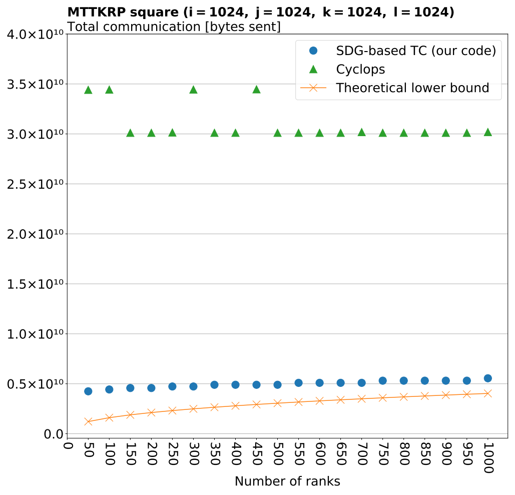

Fig. 3. *Strong scaling*: Comparison of square MTT-KRP communication with the theoretical lower bound and CTF.

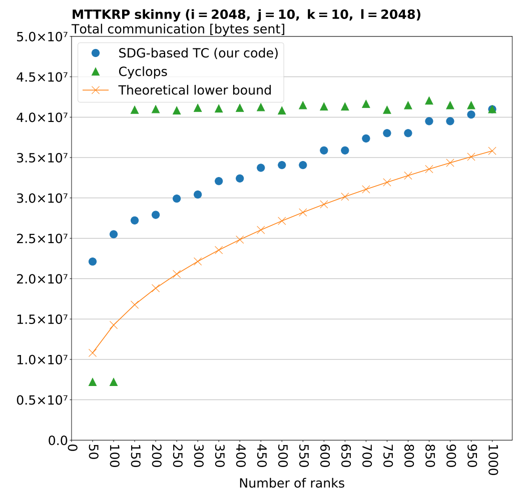

Fig. 4. *Strong scaling*: Comparison of skinny MTT-KRP communication with the theoretical lower bound and CTF.

CTF communicates significantly more than our framework – up to 6 times more. The *skinny* result shows that our framework communicates less than CTF, although our communication volume approaches that of CTF as the number of ranks increase. Unfortunately, due to resource constraints (on the Euler cluster), we were unable to run CTF with more than 1000 cores.

Weak Scaling Results. For weak scaling, we keep the number of cores fixed at 1000 and increase the input size. Specifically, for a size S, we set for MMM $(ij, jk \to k)$ that $i = j = k = S$ and for MTTKRP $(ijk, kl, jl \to il)$ that $i = j = k = l = S$. In the case of MMM, Figure 5 shows that as we increase the tensor dimensions, we consistently match the theoretical lower bound exactly. Together with the strong scaling we see our code performs MMM optimally in terms of communication.

For MTT-KRP we don't match the lower bound exactly. Figure 6 shows that as the dimensions increase, the deviation with the given theoretical lower bound increases, when fixing the number of cores to 1000. Our communication matches exactly the amount expected by the distribution schedule computed by the SOAP analysis algorithm. As discussed before, the SOAP analysis algorithm is constrained by integer assignments, and therefore the tensor slices communicated might be slightly larger than theoretically needed. As we increase the tensor size, the volume of this too large tensor slice increases at a super-linear rate. We suspect this is the main reason for the growing deviation of the communication volume.

5. CONCLUSION

We have presented a communication-optimized distributed tensor contraction framework. Utilizing a distribution schedule computed via SOAP SDG graph analysis, we converge towards the theoretical minimum, communicating up to 6× less than the state-of-the-art contraction library cyclops for MMM and MTT-KRP with ideal dimensions. In the case of skinny tensors, the slightly sub-optimal distribution caused deviation from the lower bound – although still outperforming CTF in most cases except square MMM.

Future work could focus on performance optimizations of the local contractions. Multi-Index based strided accesses to the tensors could be replaced by loop-structuring via code-generation like in [7] and specialized kernels using BLAS calls like d_gemm, such as [3]. Combining high-performance contractions with minimal communication may outperform the state-of-the-art on all fronts.

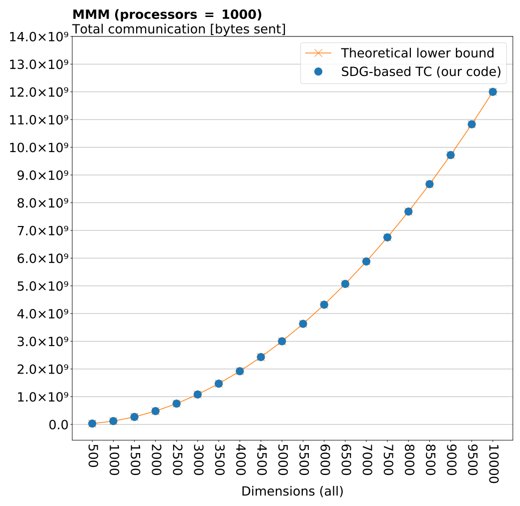

Fig. 5. *Weak scaling*: Comparison of MMM comm. with the theoretical lower bound, varying dimensions.

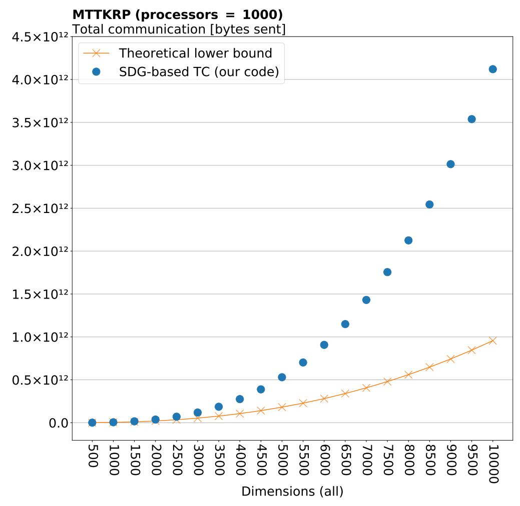

Fig. 6. *Weak scaling*: Comparison of MTTKRP comm. with the theoretical lower bound, varying dimensions.

6. REFERENCES

[1] E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin M. Abadi, A. Agarwal, P. Barham, "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015, Retrieved from https://www.tensorflow.org.

[2] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill et al., "Madness: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. 123–142, October 2016.

[3] E. Solomonik, D. Matthews, J. R. Hammon and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 1730 Massachusetts Ave., NW Washington, DC United States, May 2013, pp. 813–824, IEEE Computer Society.

[4] D. Mester Z. Rolik G. Samu J. Csontos J. Csóka P. B. Szabó L. Gyevi-Nagy B. Hégely I. Ladjánszki L. Szegedyb B. Ladóczkic K. Petrov M. Farkasd P. D. Mezeie M. Kállaya, P. R. Nagy and Á. Ganyecz, "The mrcc program system: Accurate quantum chemistry from water to proteins," *The Journal of Chemical Physics*, vol. 152, no. 7, 2020.

[5] V. Lotrich, N. Flocke, M. Ponton, B. A. Sanders, E. Deumens, R. J. Bartlett and A. Perera, "An infrastructure for scalable and portable parallel programs for computational chemistry," *Proceedings of the 23rd international conference on Supercomputing*, p. 523–524, June 2009.

[6] E. J. Bylaska et. al., "Nwchem, a computational chemistry package for parallel computers, version 6.1.1," 2012.

[7] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software*, vol. 44, no. 3, pp. 1–29, September 2018.

[8] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.

[9] J. Nieplocha, R. J. Harrison and R. J. Littlefield, "An infrastructure for scalable and portable parallel programs for computational chemistry," *The Journal of Supercomputing*, vol. 10, pp. 169–189, 1996.

[10] G. Kwasniewski, T. Ben-Nun, L. Gianinazzi, A. Calotoiu and T. Schneider, "Pebbles, graphs, and a pinch of combinatorics: Towards tight i/o lower bounds for statically analyzable programs," online: https://arxiv.org/abs/2105.07203, May 2021.

[11] M. D. Schatz, *Distributed tensor computations: formalizing distributions, redistributions, and algorithm derivation*, Ph.D. thesis, The University of Texas at Austin, December 2015.

[12] J. Hong and H. Kung, "I/o complexity: The red-blue pebble game," *in STOC*, pp. 326–333, 1981.

[13] D. Irony, S. Toledo and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[14] G. Ballard, J. Demmel, O. Holtz and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, September 2011.

[15] G. Kwasniewski, M. Kabic, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[16] J. Vetter, C. Chambreau, "mpip - a lightweight mpi profiler," https://github.com/LLNL/mpiP, 2006.