ACCELERATING REDISTRIBUTION OF TENSORS

Lundwall Marc, Hui Zijun, Ma Yitian, Wei Yu-Shan, Zeng Peiyu

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The performance of high-dimensional multilinear algebra kernels on massively parallel system is often dominated by data movement. Recent studies on improving the kernel execution performance introduced a data movement-optimal distribution schedule. After the computation of intermediaries, the distribution often changes, making redistribution of tensors necessary. This work focuses on improving the performance of tensor redistribution through optimising data communication. In this project, we propose and compare various methods of accelerating the redistribution of multi-dimensional tensors with MPI. The methods experimented with different MPI communication modes, acceleration with multithreading, and pipelining. In addition, a C++ library for point-to-point tensor transmission and redistribution was implemented. When compared to MPI's inbuilt custom datatypes, our proposed method achieved an improvement of up to 1.98x.

1. INTRODUCTION

Motivation. Linear algebra kernels are a ubiquitous component of scientific computing, with uses in machine learning [1, 2], physics [3], and numerical modelling [4]. On modern massively-parallel systems, the performance of these kernels are often dominated by data movement rather than the floating-point operations per second (FLOPs) performance of the processors [5]. Though linear algebra kernels have been extensively studied, culminating in libraries such as BLAS [6] and LAPACK [7], there has been little investigation in its generalisation to higher dimensions, multilinear algebra. Multilinear algebra involves higher-order tensors and is used in critical computational kernels for data analysis, such as CANDECOMP/PARAFAC (CP) [8] and Tucker decompositions [9].

Recently, Ziogas et al. introduced Deinsum [10], which is a communication-optimal schedule for multilinear algebra kernels. This framework provides a redistribution schedule of tensors needed in the execution of kernels, but little investigation has been done in the performance of the redistribution itself. This project investigates various methods of communications in a point-to-point and redistribution setting and proposes a redistribution method that is up to 1.98x faster than an implementation using MPI custom datatypes.

Related Work. In the implementation of Deinsum, MPI custom data types were used to transfer the tensor blocks between different nodes [10, 11]. Custom datatypes are provided by MPI for non-contiguous data access and involve a declaration step and a commit step. In the commit step, a formal description of the data layout is generated. However, the MPI standard does not provide any performance guarantees, thus it is up to implementations themselves to optimise the commit step [11].

Schneider et al. introduced libpack [12] to use runtime compilation to generate optimised packing code for MPI custom datatypes at commit time. They observed that the packing code is typically generated by interpretation at run, thus preventing optimisation techniques such as loop unrolling, vectorisation, and constant propagation from being used. Runtime compilation trades a small compilation overhead to enable such optimisation techniques.

Instead of using custom datatypes, explicit packing by programmers have also been investigated in the past [13]. This involves looping over the data and manually copying them into a contiguous temporary buffer. By using manual packing, the overhead during communication is reduced up to a factor of 2.

Pipelining is a common method to improve the performance in distributed algorithms. The code is segmented to overlap computation and communication. There has been many successful applications of pipelining to accelerate MPI communication in other domains of application [14, 15].

The previously described methods have not been tested in the context of tensors, where the high dimensionality introduces uncommon memory layouts and access patterns. Furthermore, given improvements in bandwidth and computing power, it is unclear whether the above results still hold. Our most optimal redistribution method is a variant of manual packing, where multithreading and pipelining were used to increase performance.

2. BACKGROUND

Tensor redistribution. Tensors are generalisation of matrices in high dimensions. In common multilinear algebra kernels, intermediate tensors have a dimensionality of less than or equal to 5. In massively parallel systems, the tensors are initially distributed across many different nodes. During computation, intermediate data that are the output of one group of operations could be the input to another. These groups generally differ and are executed on different nodes, thus redistribution is needed to shuffle the intermediaries.

During redistribution, different blocks of the original tensor on a given node may be distributed to different nodes. Redistribution of the blocks are divided into two types: self-copying and point-to-point communication. In the former, the previous and next operation on the data are performed on the same node, thus no inter-node communication is needed. In the latter, the block is sent to a different node according to the redistribution schedule. An example is shown in Fig. 1. The top left and bottom right blocks undergo self-copying, while the remaining two blocks are communicated from node 1 to node 2 and vice versa.

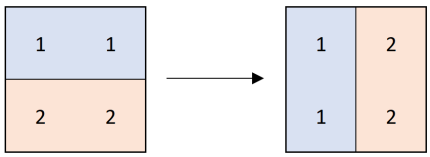

Fig. 1. Example of redistribution of a 2D tensor. The number marked on the block indicate the node it belongs to. Colored regions indicate the block stored on each node. Each black box indicates a tensor.

MPI communication. MPI is a standardised message passing interface used to communicate data on parallel computing architectures [11]. The interface provides many ways to transfer data point-to-point, e.g. standard send, one-sided communication. Standard send can either be non-blocking, where a request for communication is created to get back a handle and the function returns, or can be blocking, where the sender waits until the data is copied to a system buffer or the receiver is ready to receive. The method selected is implementation and data dependent. For non-blocking send, it is necessary to either test for completeness of the send or wait for the completion. Note that this does not guarantee asynchronous communication, and often communication is deferred until the send is waited. One-sided communication used a different paradigm, where one processes specifies a region of memory that other processors can freely modify. Compared to the previously described two-sided communications, delays in the receiving process does not affect send. Furthermore, it reduces the number of synchronisation primitives and improves data movement, resulting in a higher performance [16].

OpenMP. OpenMP is an API interface that enable simple shared-memory parallel programming in C++ [17]. It forks a specific number of threads and divides a job between them. The threads run concurrently and the operating system may allocate the threads to different cores. OpenMP is most commonly used to accelerate `for` loops, where each iteration is assigned to a different thread. Nested `for` loops can be collapsed into one large iteration space. To distribute the work, OpenMP statically assigns each thread a single contiguous chunk of the iteration space. Alternatively, OpenMP provides a task API, where tasks are first generated and then executed concurrently.

OpenMP can be used in the same application as MPI. Different settings are available, depending on whether the MPI sends are called in only the main, all threads but not concurrently, or all threads. Relaxing the MPI calling restrictions introduce additional overhead, since more synchronisation is required.

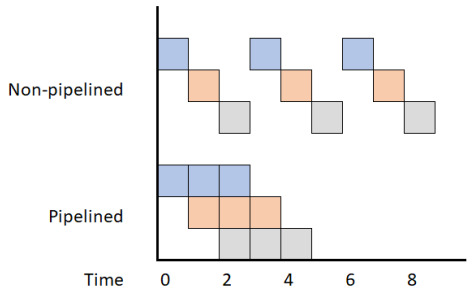

Fig. 2. Comparison of pipelined with non-pipelined operations. The different colors represent different stages. Each row indicate different hardware that the stages run on.

Pipelining. Pipelining is a technique where each iteration of a loop is segmented into stages such that the output of one is the input of the next. The execution of the stages can then overlap and execute simultaneously. Compared to non-pipelined tasks where the stages are executed serially, pipelining introduces parallelism that increases the performance of the program. In the case of data transmission, the iteration is often split into computation, memory communication, and internode communication. The stages are bottlenecked by the processor, memory bandwidth, and internode bandwidth respectively. Pipelining thus aims to saturate them simultaneously. An example of pipelining is seen in Fig. 2.

3. PROPOSED METHOD

During the redistribution, the data that needs to be communicated between nodes is a block, which is a subset/subarray

partitioned from a multi-dimensional tensor/array. In memory, the tensor is collapsed to one dimension, and block is stored in a strided pattern in memory. We propose and investigate several methods to transmit blocks between different MPI processes.

MPI custom datatypes (baseline). The baseline implementation mirrors the method used in the Deinsum and uses MPI's custom datatype. `MPI_Type_create_subarray` function declares a custom MPI datatype for a block. The subarray datatype records the dimensionality, the size of the full multi-dimensional array, the start coordinate in the full array, and the size of the block. The datatype was committed using `MPI_Type_commit`. The custom datatype was sent using the nonblocking `MPI_ISend` and received serially using `MPI_Recv`.

Manual packing. Instead of using the custom datatype interface, the non-consecutive multi-dimensional block is manually packed into a consecutive buffer. This avoids generating packing code at runtime based on interpreted datatypes [12]. To send and receive the data, the basic datatypes (i.e. the datatype of the tensor element) are used, since there is no longer discontinuity. Similar to the baseline, sending and received were serialised and performed using the standard functions.

In more detail, the blocks were packed by computing its target address based on its index and then copying them into the send buffer using `memcpy`. The unpacking operation is conceptually symmetric to the packing. After the receiver receives the data, manual unpacking restores the desired strided data layout from the consecutive buffer.

Manual packing with thread acceleration. Manual packing performs a large number of `memcpy`s of continuous chunks of block elements. Since `memcpy` is clearly a memory-bound operation, where the performance is limited by the memory bandwidth, one would wish to increase the memory bandwidth used by the program. One way is to use multiple threads, since a single thread may not fully saturate the memory channel. In addition, machines with multiple sockets - commonly used in high performance computing - have a non-uniform memory access (NUMA) pattern. Different sockets are connected to different memory channels, thus to saturate all memory channels simultaneously, multiple threads are required [18]. Since the `memcpy` of different chunks are independent, each `memcpy` was distributed to a different thread using OpenMP. However, without exclusive access to the Euler nodes, we could not obtain exploitable results from using threads on different nodes.

Manual packing with pipelining. To further speed up manual packing, pipelining was was used. Instead of only using one MPI send for each tensor block, the send is broken up into multiple smaller chunks of even size. Each chunk is copied into a smaller buffer, and after the copying is completed, an MPI send is dispatched. An analogous method was used for receiving. In order to enable parallelism and avoid unnecessary copies, non-blocking communication was used. Before the current chunk sends its data, it waits for the previous data to communicate. This avoids oversaturation of the communication bandwidth. In more details, `MPI_Isend` and `MPI_IRecv` were used and `MPI_Request` were stored and waited for with `MPI_Wait`. An example is seen in Fig. 3.

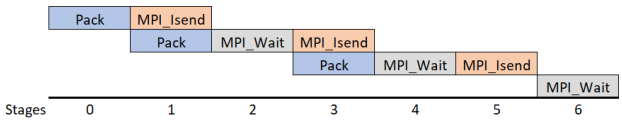

Fig. 3. An example of the sending component of manual packing with pipelining with 3 chunks. The different boxes indicate different stages. `MPI_Wait` matches the `MPI_ISend` of the previous chunk. The overlapping of the packing and sending enables parallelism.

MPI one-sided. Given the potential advantages of MPI one-sided, we attempted to combine it with manual packing. Using `MPI_Win_create`, each process specifies a window of existing memory that it exposes to other processes in the same communicator via Remote Memory Access (RMA). Given the benefits of manual packing, it was used to pack data. After a given block is packed into a buffer, `MPI_Put` transfers the data from one process to the RMA window in the target process. `MPI_Win_fence` ensures the completion of `MPI_Puts` that were previously issued. After the fence, the receiver can start the unpacking as usual.

Redistribution. Based on the work of Deinsum [10], the schedule of redistribution is determined by the previous and new operation groups defined on Cartesian process grids. The inputs of the scheduling algorithm are the total tensor shape involved in the computation along with the previous and new process grid. The algorithm outputs for each process which block of the tensor should be sent to which target process or be self-copied. Also the algorithm determines how the target process places the received block in their new local tensor buffer.

During the redistribution process, the tensor owned by each processor is divided into different blocks based on the schedule. Each of them could be either sent to other processes or be kept in the original process (self-copying). Thus, there are three steps to perform for each process: send blocks to other processes, receive blocks send by other processes, and perform self-copying. Blocks are sent / received with our implementations for different proposed point-to-point communicating methods.

4. EXPERIMENTAL RESULTS

For each proposed method in Sec.3, two benchmarks are provided. One is a micro-benchmark, where single-block transmission between two MPI processes on different nodes are considered. In this benchmark, rank 0 sends a block to rank 1 and receives a block from rank 1, and vice versa. The second benchmark is the full redistribution. Comparing different redistribution schedules is not the aim of this paper, so a brute-force schedule was used. All nodes simultaneously send their blocks in a non-blocking manner using `MPI_Isend` and then start to receive the blocks destined to them. To facilitate easy replacement of the block transmission methods, we wrote a C++ redistribution library based on the DaCe [19] code.

LibSciBench [20] was used to record the runtime of our benchmarks. For MPI custom datatype, only the time of data transmission is recorded. For the methods related to manual packing, we measure the time of both data packing and transmission. Before the experiment proceeds, 10 warmup runs were ran by repeating the redistribution process 10 times to warmup the cache. An additional 10 runs follows to satisfy the need for synchronisation by LibSciBench. Only after these 20 runs are data collected. In each run, the maximum time among all processes is collected. To avoid erroneous values, the processors wait until all processors are ready. The redistribution is repeated until the 95% confidence interval is within 5% of the median of all runs. For each method, MPI custom datatype with one thread is used as the baseline.

Our measurements focus on 5D tensors, where the performance improvement is most needed among 2D-5D (see Appendix A). The full redistribution operates on an integer tensor of varying dimension sizes, but for majority of the experiments, a 5D tensor of size $(48, 24, 24, 72, 288)$ was used. Originally, it is split equally among 12 nodes following the processor grid $(2, 1, 1, 3, 2)$ (i.e., 2 processors split dimension 1 and 5, 3 processors split dimension 4). The tensor was then redistributed to follow the processor grid $(1, 1, 1, 1, 12)$. Each integer is 4 bytes, making the tensor around 2.29GB. Each node stores around 191MB and every sent block is approximately 32MB. With different node sizes, the size of each dimension changes equally.

Experimental setup. Experiments were conducted on ETH Euler Clusters, specifically Euler VII phase 2. Each node has 2 AMD EPYC 7763 processors (64 cores, 2.45 GHz nominal, 3.5 GHz peak, 256 GB of DDR4 memory clocked at 3200 MHz). All the nodes are connected in a fat tree fashion via a dedicated 100 Gb/s InfiniBand HDR network. We use GCC version 8.2.0 and compile the benchmarks with `-O3 -march =native` flags. The MPI library used is OpenMPI 4.1.4 and the OpenMP conforms to standard 4.5. Exclusive ownership of each node was obtained during full redistribution experiments.

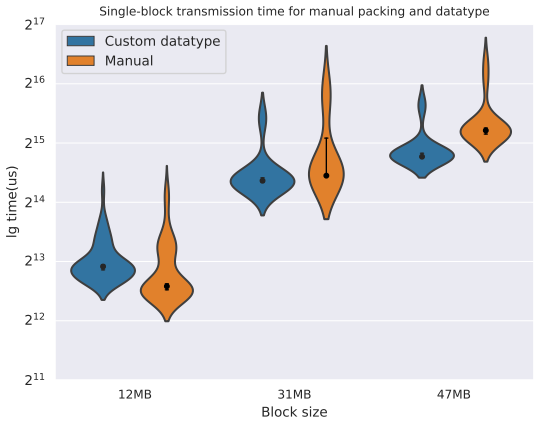

Fig. 4. Comparison of mutual block sending between 2 MPI processes using (1) datatype (2) manual packing. Black dots represent the median and error bars are the 95% confidence intervals.

Micro-benchmark. The comparison of manual packing against the baseline, custom datatypes, in the micro-benchmark is shown in Fig. 4. It performs better at small block sizes with statistical significance, but performance scales worse as the block size increases.

Manual packing.

Redistribution. To test different block sizes in redistribution, tensors are proportionally scaled so that the mean sending block sizes are similar to that of the micro-benchmarks. The results are shown in Fig. 5. In the full redistribution, manual packing is significantly faster than custom datatypes. This effect is the most present when the data size per node is intermediate.

Manual packing with thread acceleration.

Micro-benchmark. Unfortunately, due to the limitation of Euler guest accounts, it was extremely difficult to obtain cores allocated on different sockets, since it prioritises reducing NUMA overhead. Thus, all experiments were ran with the threads being on the same socket. The result of using multithreading during manual packing is shown in Fig. 6. The execution time is shorter at when an additional thread was used, but it plateaus after. This is likely because the given memory bandwidth saturates with only two threads, thus adding more threads doesn't help with our memory-bounded workloads. Greater performance improvements are likely present if the threads run on different sockets.

Redistribution. The results with threading and redistribution The results are shown in Fig. 7. While there was a statistically significant improvement in the micro-benchmarks, no statistically significant speedup is observed in redistribution, since the 95th confidence intervals for the medians

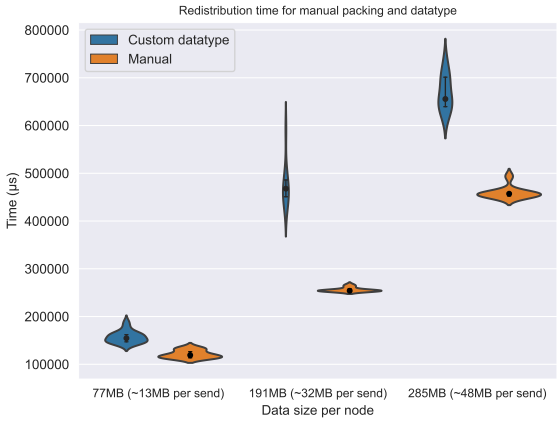

Fig. 5. Comparison of redistribution between 12 MPI processes using (1) datatype (2) manual packing.

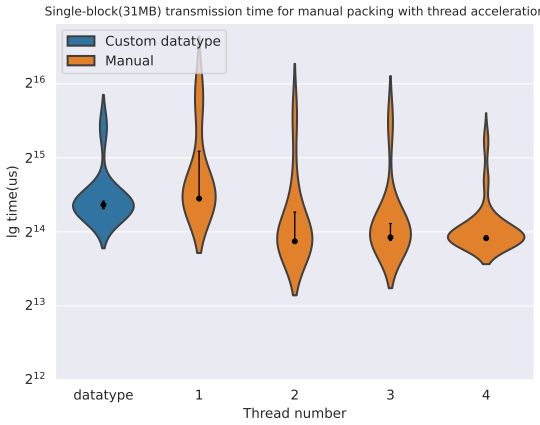

Fig. 6. The runtime plot when using different threads. Block size is 31 MB.

overlap in all cases. A plausible reason is MPI buffering the received data. This may occur when the machine receives the data but the program itself is not ready to receive. This results in background activity saturating the memory bandwidth earlier and could be caused by one node still performing the send while it incoming data has already arrived.

Manual packing with pipelining.

Micro-benchmark. In Fig. 8, the effect of pipelining the sends / receives is shown. A speedup is observed when the data is split and sent in two separate chunks. Similar to threading, the performance plateaus and using more chunks does not significantly improve the performance. Given the available data, no suitable number of chunks could be deduced and pipeling may not help at all. For example, when the block grows from 31 MB to 47 MB large, pipelining doesn't bring a speedup.

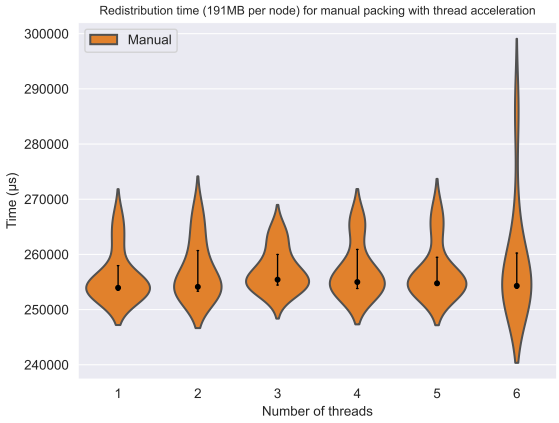

Fig. 7. The runtime plot of redistribution when using different threads. Tensor size is 191 MB per node. Custom datatype time is omitted since it already is significantly worse, as can be seen in Fig. 5.

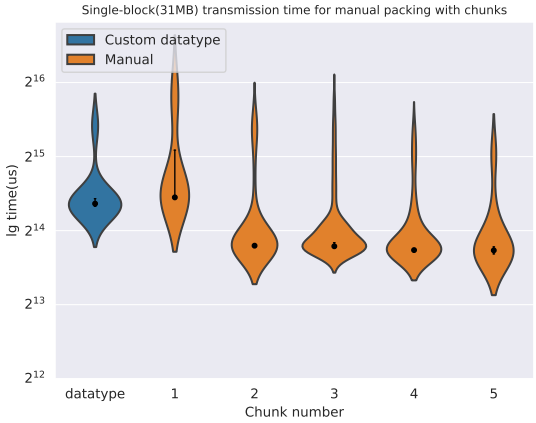

Fig. 8. The runtime plot when using different number of chunks. Block size is 31 MB.

Redistribution. The same experiment was performed on the full redistribution to investigate whether the same improvement holds. The results presented in Fig. 9 do portray a statistically significant speedup: the redistribution time drops from 255ms with no pipelining, to 237ms at 4 chunks, plateauing after. At 4 chunks, the chunk size is approximately 8MB.

MPI one sided.

Micro-benchmark. A three way comparison of custom datatype, manual packing, and manual packing with one-sided is presented in Fig. 10. While one-sided communication is not initially faster than the other two methods, as the block size increases, one-sided communication starts to outperform the other methods. This indicates one-sided

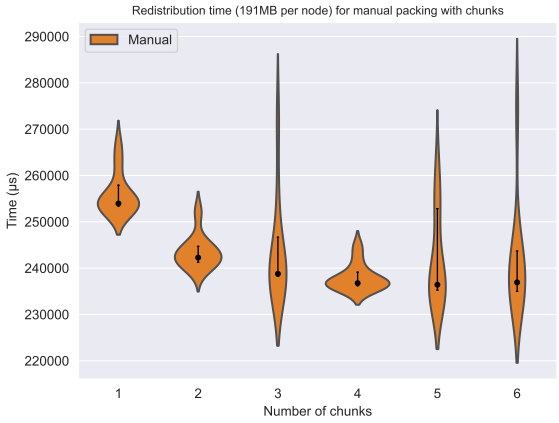

Fig. 9. The runtime plot when using different numbers of chunks. Tensor block size is 191MB per node and average sent block size is 32MB. Custom datatype time is omitted since it already is significantly worse, as seen in Fig. 5.

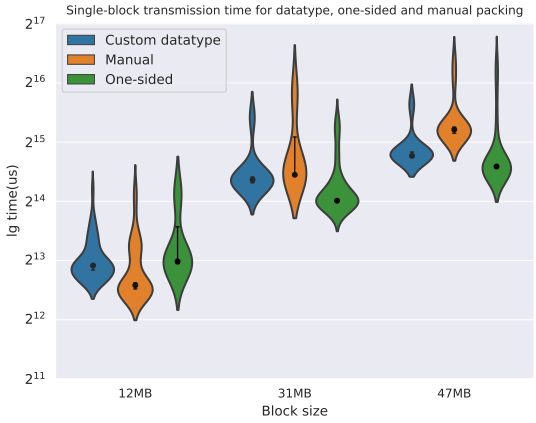

Fig. 10. The runtime plot for the block of different sizes using different methods.

communication scales better than other transmission methods for point-to-point communication.

We also test pipelining with MPI one-sided communication. In the single block transmission, increasing the number of chunks helps further reduce the runtime of manual packing with MPI one-sided functions. There is no significant improvement after 2 chunks. Unlike manual packing with standard two-sided communication, we observe that the speedup holds for larger blocks in one-sided case.

Redistribution. Although the results in the one-sided micro-benchmark are encouraging, its effects on full redistribution were not measured. With the chosen brute-force redistribution schedule, the running order of the nodes are random. However, the creation of the MPI windows for the

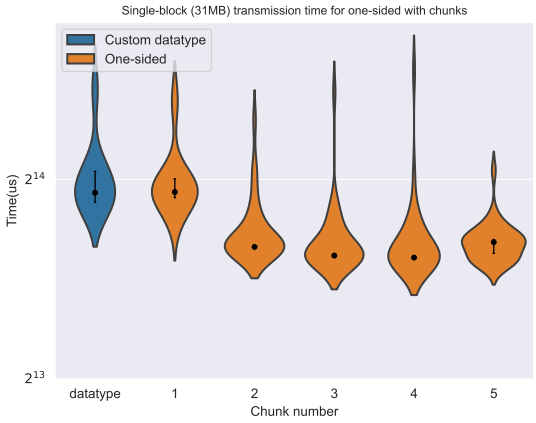

Fig. 11. The runtime plot for one-sided communication using different numbers of chunks. Pipelining helps the manual packing with one-sided becomes faster.

pair-wise data transmissions is a blocking collective operation, thus it may easily deadlock. A more sophisticated redistribution schedule where the nodes agree on an optimal ordering would allow the nodes to more easily establish their pairwise windows for every one-sided data transfer.

5. CONCLUSIONS

This project has explored the performance improvements of using manual packing over the default MPI custom datatypes, as well as a few methods to further improve our results. We have found that multithreading the memory copying in the context of redistribution was not useful, as the memory bandwidth was already saturated with one thread. Overlapping memory-intensive operations with communication through pipelining, on the other hand, led to an important decrease in the redistribution time. Given the current results, the optimal communication method during redistribution is to use only one thread and select a chunk size of 8MB in pipelining. This gave our best result of 236.760 ms, compared to our baseline of 467.761 ms when using the custom MPI datatypes, which is an 1.98 times improvement. Further exploration is needed to investigate whether multithreading helps when the threads are spread over different sockets while having exclusive access of the nodes. These experiments may also be carried out on different redistribution scheduling algorithms. This measures the effect of less simultaneous communication through a more efficient message schedule, and can test the one-sided window communication through `MPI_Put`, which was promising in the micro-benchmarks.

6. REFERENCES

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, Curran Associates Inc., Red Hook, NY, USA, 2019.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, USA, 2016, OSDI'16, p. 265–283, USENIX Association.

[3] William Tang, Bei Wang, Stephane Ethier, Grzegorz Kwasniewski, Torsten Hoefler, Khaled Z. Ibrahim, Kamesh Madduri, Samuel Williams, Leonid Oliker, Carlos Rosales-Fernandez, and Tim Williams, "Extreme scale plasma turbulence simulations on top supercomputers worldwide," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 502–513.

[4] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt, "Operational convective-scale numerical weather prediction with the cosmo model: Description and sensitivities," *Monthly Weather Review*, vol. 139, no. 12, pp. 3887 – 3905, 2011.

[5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, oct 2009.

[6] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, jun 2002.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[8] Muthu Baskaran, Tom Henretty, Benoit Pradelle, M. Harper Langston, David Bruns-Smith, James Ezick, and Richard Lethin, "Memory-efficient parallel tensor decompositions," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[9] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, and Dheeraj Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1038–1047.

[10] Alexandros Nikolaos Ziogas, Grzegorz Kwasniewski, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler, "Deinsum: Practically I/O Optimal Multilinear Algebra," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*, 11 2022.

[11] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 3.1*, June 2015.

[12] Timo Schneider, F. Kjolstad, and Torsten Hoefler, "MPI Datatype Processing using Runtime Compilation," in *Proceedings of the 20th European MPI Users' Group Meeting*. 09 2013, pp. 19–24, ACM.

[13] Timo Schneider, Robert Gerstenberger, and Torsten Hoefler, "Micro-applications for communication data access patterns and mpi datatypes," in *Recent Advances in the Message Passing Interface*, Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, Eds., Berlin, Heidelberg, 2012, pp. 121–131, Springer Berlin Heidelberg.

[14] J. Worringen, "Pipelining and overlapping for mpi collective operations," in *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN '03. Proceedings.*, 2003, pp. 548–557.

[15] Bin Bao, Chen Ding, Yaoqing Gao, and Roch Archambault, "Delta send-recv for dynamic pipelining in mpi programs," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 384–392.

[16] Oracle, *Sun HPC ClusterTools 6 Software Performance Guide*, 2010, Chapter 5, One sided communication.

[17] Leonardo Dagum and Ramesh Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[18] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang, "PETSc/TAO users manual," Tech. Rep. ANL-21/39 - Revision 3.18, Argonne National Laboratory, 2022, Section 4.4.1, Maximizing Memory Bandwidth.

[19] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler, "Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, SC '19.

[20] Torsten Hoefler and Roberto Belli, "Scientific Benchmarking of Parallel Computing Systems," Nov. 2015, pp. 73:1–73:12, ACM, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

A. RUNTIME FOR DIFFERENT DIMENSIONS

In our experiment section, we focus on 5d tensors. We choose 5d because we observe that custom datatype works slower at higher dimension with same size of data being transmitted. An example of single-block transmission is shown in Fig. 12.

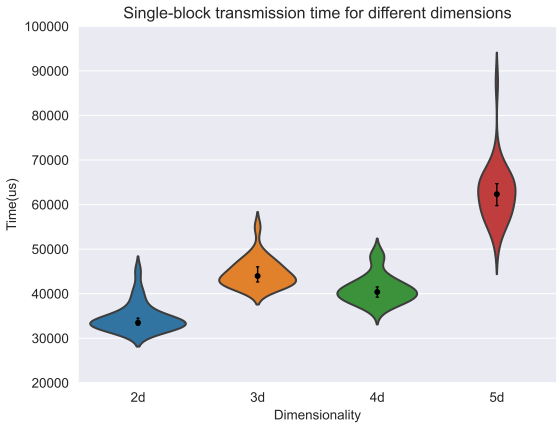

Fig. 12. The transmission time for different dimensions. Block size is 93 MB.