Computing Sparse Tensor Decompositions via Chapel and C++/MPI Interoperability without Intermediate I/O

S. Isaac Geronimo Anderson
University of Oregon, Sandia National Laboratories
USA
igeroni3@uoregon.edu,sgeroni@sandia.gov

Daniel M. Dunlavy
Sandia National Laboratories
USA
dmdunla@sandia.gov

ABSTRACT

We extend an existing approach for efficient use of shared mapped memory across Chapel and C++ for graph data stored as 1-D arrays to sparse tensor data stored using a combination of 2-D and 1-D arrays. We describe the specific extensions that provide use of shared mapped memory tensor data for a particular C++ tensor decomposition tool called GentenMPI. We then demonstrate our approach on several real-world datasets, providing timing results that illustrate minimal overhead incurred using this approach. Finally, we provide a roadmap for extending our work to improve memory usage and provide efficient random access to sparse shared mapped memory tensor elements in Chapel, while still leveraging high performance implementations of tensor algorithms in C++.

KEYWORDS

Chapel, C++, MPI, interoperability, sparse tensor decomposition

ACM Reference Format:
S. Isaac Geronimo Anderson and Daniel M. Dunlavy. 2023. Computing Sparse Tensor Decompositions via Chapel and C++/MPI Interoperability without Intermediate I/O. In *Proceedings of The 10th Annual Chapel Implementers and Users Workshop (CHIUW '23).* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

We present several challenges and successes of sharing sparse tensor (i.e., multi-dimensional array) data, with no data duplication, between Chapel [2] and a parallel program for computing sparse tensor decompositions that is written in C++ and uses the Message Passing Interface (MPI) [8] library to exchange data. Tensors are the higher-order generalization of matrices, and low-rank tensor decompositions (or factorizations) provide a useful tool for analyzing latent relationships in tensor data [5]. We focus here on computing low-rank canonical polyadic (CP) decompositions, for which several high-performance C++ implementations have been developed over the past decade [3, 9, 11, 12].

While Chapel is portable and easy to use, we believe there is great benefit in leveraging existing high performance C++ libraries and applications from within a Chapel program. One way to share data between Chapel programs and C++ applications is to write data to a file in Chapel, read the file into the C++ application, perform some computations, write the results to file in the C++ application, and then read the file into Chapel. This incurs significant overhead which increases with the size of the data, in terms of file I/O and of loading a copy of the file into memory via the C++ application.

CHIUW '23, June 1-2, 2023, Virtual Format
.

Previous work demonstrated sharing 1-D data between a Chapel program and a C++/MPI program, without writing the data to an intermediate file [7]. The approach in that work effectively maintained data layout and locality, memory usage efficiency, and data-parallel computation capability. Here, we present extensions of the previous 1-D work to multiple dimensions.

2 BACKGROUND

Previous work enabled Chapel/C++ distributed-memory interoperability for graph algorithms whose graphs are represented using multiple 1-D arrays (edge lists and optional weights). McCrary *et al.* demonstrated that it is possible to share data between Chapel programs and a C++/MPI graph analysis application, Grafiki, without duplicating the data on disk or in memory [7]. They extended the Chapel `BlockDist` distribution to use shared mapped memory, via the low-level `shm_open` and `mmap` Linux functions, to enable read/write access to distributed Chapel arrays by Grafiki MPI ranks. Two one-dimensional (1-D) Chapel arrays store the pairs of vertices associated with each edge in the graph. These arrays use the same Chapel domain constructed with the extended `BlockDist` distribution, hence automatically distributing the shared mapped memory arrays across the available Chapel locales (and, thus, corresponding MPI ranks). Once the arrays are created and populated, the Chapel program writes a small metadata file including references to the shared mapped memory arrays. The Chapel program then signals Grafiki using a flag passed via shared mapped memory that the arrays are ready for use in Grafiki. The metadata file containing the shared mapped memory references is read from a single process on the MPI rank-0 node and distributed to the other nodes. Each MPI rank opens its respective shared mapped memory array reference and wraps the corresponding memory in a `Kokkos::View` [4], which is the array abstraction format used by Grafiki. Grafiki then performs the distributed graph analysis computations and signals Chapel via the shared mapped memory flag that the operation is finished. The Chapel program can then perform further operations on the graph and/or the results of the Grafiki computation.

3 METHODOLOGY

The goal of this work is to extend the existing Chapel/C++ distributed-memory interoperability for sharing 1-D arrays between Chapel and C++/MPI programs to algorithms and programs that use sparse d-D tensors. We focus here on specific cases where $d \in \{3, 4, 5\}$, but our work covers the general case of $d \geq 1$.

GentenMPI stores sparse tensors in a coordinate, or COO, format that extends the standard Matrix Market format [1] used for sparse matrices. The GentenMPI COO format stores non-zero sparse tensor elements using a 2-D array for the coordinates (i.e., indices) and

a 1-D array for the values. That is, for a d-dimensional sparse tensor with nnz non-zero elements, the coordinates are stored in a 2-D array of size $d \times$ nnz, and the values are stored in a 1-D array of size nnz. The `BlockDist` distribution allows 1-D and 2-D arrays, but the semantics of the 2-D coordinates array requires re-shaping the Chapel `Locales` array for proper distribution of the coordinates. The semantics of the $d \times$ nnz 2-D coordinates array are such that the first dimension of the array—corresponding to the coordinates—should be considered indivisible, because all d coordinates for a given non-zero element should be distributed together. That is, if there is a non-zero element with coordinates (i, j, k), it should not be the case that coordinates i and j belong to locale A while coordinate k belongs to locale B, when $B \neq A$. The default behavior of `BlockDist` is to partition a 2-D array uniformly across both dimensions, and to distribute the partitions across the number of locales (`numLocales`). Such behavior breaks the semantic indivisibility of the first dimension of the coordinates array. But by reshaping the `Locales` array into a 2-D, $1 \times$ `numLocales` array, this leads to `BlockDist` partitioning the second dimension of the coordinates array—corresponding to the non-zero elements—as desired.

GentenMPI also imposes further constraints on the partitioning of the sparse tensor. Specifically, each Chapel array partition must only contain elements belonging to a non-overlapping bounding box (sub-tensor) within the coordinate space of the sparse tensor. For example, suppose a 3-D sparse tensor with dimensions $8 \times 6 \times 7$ using 1-based coordinates comprises two bounding boxes A and B, where A contains elements whose coordinates range from (1, 1, 1) to (4, 3, 3) and B contains elements whose coordinates range from (5, 4, 4) to (8, 6, 7). This scenario would require sparse tensor elements associated with A to be stored together on one Chapel locale, and would require the same for elements associated with B. GentenMPI requires the use of non-overlapping bounding boxes to minimize expensive inter-processor communication, thus maximizing overall computational performance. This constraint poses an issue for partitioning the coordinates array in Chapel, because the coordinates stored in the array determine how the array should be partitioned, but Chapel partitions uniformly based on the dimensions of the 2-D coordinates array (i.e., $d \times$ nnz), not the dimensions of the sparse tensor (e.g., $8 \times 6 \times 7$, as in the example above). We address this constraint by performing a sparse tensor file analysis for determining partitioning information (a one-time cost), then setting the size of the coordinates array such that Chapel's uniform partitioning is compatible with the GentenMPI bounding box constraint.

4 EXPERIMENTS

We demonstrate the use of our method for sparse tensor interoperability between Chapel and C++/MPI using several real-world sparse tensor datasets from the FROSTT [10] tensor benchmark data repository. Table 1 shows the details of the three tensor datasets used in the experiments presented here.

We use GentenMPI's implementation of GCP-ADAM [6] for computing the low-rank CP tensor decompositions, which is well suited for the FROSTT data, as it can compute decompositions on sparse count tensor data. We use GentenMPI's partitioning scheme, provided by the SPLATT tensor package [11], for computing the one-time cost of identifying non-overlapping bounding boxes as a function of the dataset and number of Chapel locales/MPI ranks. We vary the number of locales/ranks (4, 8, 16, 32, 64) used in the experiments to illustrate the performance characteristics over a range of computational resources. In each experiment, we run 5 iterations of the GCP-ADAM algorithm to compute an approximate rank-16 CP decomposition. We repeated each experiment five times, and recorded the mean and standard deviation for the runtime. All experiments were run using Chapel 1.24 on a Cray XC40 system, using up to 64 (of the 100 available) compute nodes, where each node has 32 cores 128GB memory.

Table 1: Datasets from the FROSTT tensor benchmark repository [10] used in experiments.

Sparse Tensor	Tensor Dimensions	nnz
chicago-crime-comm	$6.2K \times 24 \times 77 \times 32$	5.3 M
lbnl-network	$1.6K \times 4.2K \times 1.6K \times 4.2K \times 868K$	1.7 M
nell-2	$12K \times 9K \times 29K$	77 M

Note that we do not report timing results for file I/O at this time, as the methods in Chapel and GentenMPI are very different and thus not comparable. Instead, we focus on just the timing of the computations associated with shared mapped memory setup and computation across Chapel and C++/MPI versus C++/MPI alone.

Table 2 presents timing results of these experiments. Column 3 (Chapel → C++ `shm_open`/`mmap`) shows the time taken in allocating the shared mapped memory in Chapel and signaling GentenMPI to start. As shown, these times are quite small and appear to scale as the logarithm of the node count. Column 4 (Chapel → C++ GentenMPI call) shows the time taken to initialize the shared mapped memory in GentenMPI. These times are similarly small, except for an unusual outlier with chicago-crime-comm on 16 nodes—experiments are underway to determine why the GentenMPI call timing for this configuration is longer than for the others. Finally, Columns 5 and 6 show the times for running GCP-ADAM using our Chapel → C++ interoperability and in C++ only (i.e., GentenMPI as a standalone program), respectively. These times are very similar to each other, suggesting non-significant amounts of overhead for our shared mapped memory approach. From a computational performance perspective, these are promising results.

5 FUTURE DIRECTIONS

There are several research opportunities moving forward, including improved memory efficiency and general access to sparse tensor elements and operations from Chapel.

Our current approach suffers from three challenges associated with memory efficiency:

(1) Chapel's `BlockDist` distribution partitions arrays uniformly, so the coordinates and values arrays are distributed in asymptotically equal portions across the locales.
(2) Sparse tensors generally exhibit irregular non-zero element patterns, which means that uniform bounding boxes for a sparse tensor generally will not contain equal portions of non-zero elements.

Table 2: Timing results of Chapel and C++/MPI interoperability for computing low-rank sparse tensor decompositions over a range of number of Chapel locales/MPI ranks. Each column shows the mean (with standard deviations) for the runtime over five repeated experiments in the given configuration.

Sparse tensor	Nodes	Chapel → C++ shm_open/mmap time (s) (Stdev)	Chapel → C++ GentenMPI call time (s) (Stdev)	GCP-ADAM Chapel → C++ time (s) (Stdev)	GCP-ADAM C++ only time (s) (Stdev)
chicago-crime-comm	4	1.50e-03 (2e-05)	5.76e-03 (2e-03)	7.36e+00 (4e-02)	7.20e+00 (3e-02)
chicago-crime-comm	8	2.63e-03 (8e-04)	5.06e-03 (5e-04)	5.99e+00 (4e-02)	5.86e+00 (5e-02)
chicago-crime-comm	16	3.06e-03 (9e-05)	1.74e-01 (2e-01)	6.74e+00 (2e-02)	6.62e+00 (3e-02)
chicago-crime-comm	32	4.41e-03 (3e-05)	6.70e-03 (6e-04)	9.39e+00 (6e-02)	9.41e+00 (3e-02)
chicago-crime-comm	64	5.49e-03 (5e-05)	1.04e-02 (4e-04)	1.23e+01 (2e-02)	1.22e+01 (5e-02)
lbnl-network	4	1.76e-03 (5e-04)	5.43e-03 (9e-04)	1.04e+03 (2e+01)	1.07e+03 (9e+00)
lbnl-network	8	2.17e-03 (3e-04)	4.48e-03 (4e-04)	5.21e+02 (5e+00)	5.26e+02 (8e-01)
lbnl-network	16	3.06e-03 (1e-04)	1.02e-02 (8e-03)	2.67e+02 (4e+00)	2.69e+02 (6e-01)
lbnl-network	32	4.41e-03 (7e-05)	6.36e-03 (3e-04)	1.31e+02 (1e+00)	1.31e+02 (6e-01)
lbnl-network	64	5.51e-03 (2e-05)	1.31e-02 (5e-03)	8.74e+01 (2e-01)	8.71e+01 (5e-01)
nell-2	4	1.49e-03 (7e-06)	8.05e-03 (6e-04)	5.06e+01 (2e-01)	4.99e+01 (2e-01)
nell-2	8	2.04e-03 (2e-05)	1.30e-02 (8e-03)	3.22e+01 (9e-02)	3.20e+01 (1e-01)
nell-2	16	2.91e-03 (4e-05)	8.83e-03 (9e-04)	2.03e+01 (1e-01)	2.03e+01 (4e-02)
nell-2	32	4.38e-03 (6e-05)	9.76e-03 (2e-03)	1.49e+01 (2e-01)	1.49e+01 (3e-02)
nell-2	64	5.49e-03 (3e-05)	1.17e-02 (2e-03)	1.19e+01 (2e-01)	1.21e+01 (2e-01)

(3) The COO sparse tensor format (as used by sparse tensor decomposition libraries like GentenMPI) stores non-zero elements in dense arrays which will be uniformly partitioned by `BlockDist` based on their dense dimensions (i.e., $d \times \texttt{nnz}$).

These three conditions lead to a situation where we must pad the dense arrays in order to guarantee that each locale has enough storage for its (generally irregularly large) number of bounding box non-zero elements. This padding clearly leads to inefficient use of memory. In extreme cases, even for the datasets used in our experiments here with poorly chosen bounding boxes, we could have some bounding boxes containing $O(1) - O(10)$ elements while others containing as much as $O(10^7)$ elements, leading to memory allocations of $O(10^7)$ across all locales. Hence, improving the memory efficiency is a clear first research opportunity.

A second research opportunity regards the COO format itself, which does not allow for efficient random access to sparse tensor values. This is because the coordinates of non-zero elements are stored (in any order), and thus access would require a search over the entire coordinates array. Chapel provides a *sparse domain* construct based on the COO format, which allows creating sparse arrays indexed by multi-dimensional coordinates. An array based on a Chapel sparse domain can be used with the shared memory mapped `BlockDist`, hence allowing random access to sparse tensor values. But in this case, the array's values are the sparse tensor's values (and not their coordinates), meaning that the shared mapped memory `BlockDist` distribution would only provide access to the values and not their coordinates. Thus, in the interoperability use case presented above, GentenMPI could access the sparse tensor values but not know where they reside in the sparse tensor.

Our proposed solution is to extend Chapel's sparse domain and array facilities while aligning with GentenMPI's specific layout requirements. Chapel's sparse domain stores coordinates in a COO-style format, meaning that the coordinates are stored in an array internal to the sparse domain data structure. We plan to refactor this internal coordinates array to use shared mapped memory as in the `BlockDist` distribution described above, while maintaining all Chapel functionality of the sparse domain and of the arrays which use the domain.

Our plan for extending the Chapel/C++ interoperability for sparse tensors presented above to Chapel sparse domains includes the following steps:

(1) Allow the Chapel sparse domain's coordinates array to be stored in shared mapped memory. Currently, sparse domain coordinates are stored internally, initially in a size zero array, to which sparse coordinates may be added.
(2) Store the Chapel sparse domain's coordinates array elements using a memory layout compatible with GentenMPI. That is, our modified GentenMPI implementation wraps shared mapped memory arrays in `Kokkos::Views`, which is possible only for arrays whose elements have a fixed separation (stride) between them in memory.
(3) Partition the Chapel sparse domain (using `BlockDist` or similar) so that the partitions satisfy bounding box requirements for GentenMPI. This would be the expected behavior for `BlockDist` on a Chapel sparse domain.
(4) Store the Chapel sparse domain coordinates on the same locale as their corresponding values. *This is yet to be determined.*
(5) Store the Chapel sparse domain coordinates without duplicating them across locales. *This is yet to be determined.*

In this talk, we will present the details of our initial Chapel and C++/MPI interoperability for sparse tensor decompositions

described above along with the progress to date on completing the sparse domain steps listed above.

ACKNOWLEDGMENTS

We thank Karen Devine and Andrew Younge from Sandia National Laboratories for valuable help in extending their previous work on Chapel/C++ interoperability. We also thank Jee Choi from the University of Oregon for suggestions regarding computational experiments and performance analyses. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2023-01930C.

REFERENCES

[1] Ronald F. Boisvert, Roldan Pozo, and Karin Remington. 1996. *The Matrix Market Exchange Formats: Initial Design.* Technical Report Technical ReportNISTIR 5935. National Institute of Standards and Technology, Gaithersburg, MD, USA.

[2] Bradford Chamberlain, Steven Deitz, David Iten, and Sung-Eun Choi. 2011. Authoring User-Defined Domain Maps in Chapel. "https://chapel-lang.org/publications/cug11-final.pdf". In *Chapel User's Group 2011*. Cray Inc, 1–6.

[3] Karen Devine and Grey Ballard. 2020. *GentenMPI: Distributed Memory Sparse Tensor Decomposition.* Technical Report SAND2020-8515. https://www.osti.gov/biblio/1656940

[4] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Proc. Extreme Scaling Workshop*. 18–24. https://doi.org/10.1109/XSW.2013.7

[5] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[6] Tamara G. Kolda and David Hong. 2020. Stochastic Gradients for Large-Scale Tensor Decomposition. *SIAM Journal on Mathematics of Data Science* 2, 4 (jan 2020), 1066–1095.

[7] Trevor McCrary, Karen Devine, and Andrew Younge. [n. d.]. Integrating Chapel programs and MPI-Based Libraries for High-performance Graph Analysis. ([n. d.]). https://chapel-lang.org/CHIUW/2022/McCrary.pdf

[8] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. University of Tennessee, USA.

[9] Teresa M Ranadive and Muthu M Baskaran. 2021. An all–at–once CP decomposition method for count tensors. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[10] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools.* http://frostt.io/

[11] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70. https://doi.org/10.1109/IPDPS.2015.27

[12] Keita Teranishi, Daniel M. Dunlavy, Jeremy M. Myers, and Richard F. Barrett. 2020. SparTen: Leveraging Kokkos for On-node Parallelism in a Second-Order Method for Fitting Canonical Polyadic Tensor Models to Poisson Data. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC43674.2020.9286251