Compressed Cannon's Algorithm

1st Louis Jencka
Department of Computer Science)
University of New Mexico
Albuquerque, USA
ljencka@unm.edu

2nd Amanda Bienz
Department of Computer Science)
University of New Mexico
Albuquerque, USA
bienz@unm.edu

***Abstract*—Parallel matrix-matrix multiplication (GEMM) is a linear-algebra operation underpinning many applications in high-performance and scientific computing. With strong scaling, GEMM's communication latency among processes, nodes, and accelerators can be a leading bottleneck in performance.**

Data-compression is a common solution for reducing transport costs, but is often not applied for this purpose in HPC, where low-latency & high-throughput networks are difficult for compressors to match. However, some parallel GEMM algorithms – such as Cannon's method – exhibit a communication pattern where data is rarely altered as it travels between processes, providing an opportunity for a simple compression strategy: compress submatrices once at the outset, and then circulate the compressed version. This minimizes compression, which is typically significantly slower than decompression, at a price of additional memory utilization.

In this work, we have measured and modeled scaling studies of this strategy, using six existing and widely-used lossless compressors (NDZip, ZStandard, Zip, BloscLZ, ZFP, & LZ4). We find most of these compressors can improve communication latency, with speedups of up to 1.2 times observed. Additionally, we evaluate and discuss the accuracy of our model against our measured results.

***Index Terms*—compression, matrix multiplication, cannon, communication, HPC, MPI**

I. Introduction

Parallel matrix-matrix multiplication (GEMM) is a linear-algebra operation underpinning many applications in high-performance and scientific computing.

With strong scaling of problem-size and process counts in parallel GEMM, the communication costs between processes, nodes, and accelerators can become a leading bottleneck in overall latency. Data-compression is a common solution for mitigating the costs of network communication, but is often not used in contexts like HPC, where low-latency & high-throughput networks are available.

Some parallel GEMM algorithms like Cannon's method[1] exhibit a pattern of communication wherein submatrices are repeatedly sent between processes without being repartitioned or otherwise altered. This presents an opportunity for an efficient compression strategy: let all processes compress their local submatrices once at the outset, and then reuse those compressed versions for every subsequent exchange of data.

This strategy plainly reduces the number of compressor operations, but importantly it minimizes the amount of time spent *compressing* - which is a far more expensive operation than decompression in most compressors. Additionally, a dense GEMM algorithm such as Cannon's is a better candidate for compression than the sparse varieties, as sparse matrices are inherently compressed by their encoding schemes.

We've undertaken a scaling study of this compressed variant of Cannon's algorithm using four general-purpose lossless compressors (LZ4[2], BloscZ[3], Zip[4], & ZStandard[5]), and two lossless compressors specialized for multidimensional numerical arrays (ZFP[6] & NDZip[7]). Further, we have developed a model for the communication costs of compressed Cannon's algorithm, and compare its predictions here with the measured results from our scaling study. The main contributions of this paper are as follows:

1) A simple strategy for efficient compression within Cannon's algorithm, in which compression is minimized,
2) A comparison of compressed Cannon's across six existing and widely-used compressors (NDZip, ZStandard, Zip, BloscLZ, ZFP, & LZ4),
3) Measured and modeled scaling studies, showing the impact of matrix size and process count on compressed Cannon's algorithm, and
4) An in-depth performance analysis of the proposed technique.

The remainder of the paper is outlined as follows. Section II provides detailed background information on GEMM algorithms, compression techniques, and related works. Implementation details for the compressed Cannon's algorithm are presented in Section III and a corresponding model is detailed in Section III-B. A performance study is presented in Section IV, and concluding remarks are provided in Section V.

II. Background

A. Cannon's Algorithm

Matrix-matrix multiplication is a linear-algebra operation that's ubiquitous in computer science, and key to many high-performance and scientific-computing applications. In these latter contexts, parallel GEMM algorithms are frequently used on distributed systems to handle large-scale problems.

There are many different algorithms for parallel GEMM, but a common variety of approach is divide-and-conquer, in which the problem is partitioned and scattered among a grid of processes. In a 2D-grid strategy, matrices of $C = A \times B$ are block-partitioned and apportioned among a Cartesian grid of

$$\begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n,1} & C_{n,2} & \cdots & C_{n,n} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n,1} & B_{n,2} & \cdots & B_{n,n} \end{pmatrix}$$

Fig. 1: Two square matrices A and B are block-partitioned into $n \times n$ submatrices. Their product, C, would correspondingly be computed blockwise. For example, the highlighted block $C_{2,2}$ is calculated as $C_{2,2} = \sum_{k=1} nA2, k \times Bk, 2$.

processes, such that the the submatrices of C may be locally computed blockwise as $C_{i,j} = A_{i,k}B_{k,j}$.

Figure 1 highlights how this approach works. Note that calculating a given block of C requires knowledge of an entire row of blocks from A and column of blocks from B. When C is being computed in parallel, distributing entire rows and columns to every process can be inefficient, and in the case of very large matrices is often infeasible due to memory constraints. There exists a trade-off between memory efficiency and communication efficiency which parallel GEMM algorithms tackle in different ways.

Cannon's algorithm is a parallel matrix-matrix multiplication method for dense matrices. It is a memory-efficient and communication-avoiding approach; only one copy of the data collectively stored among the processes, and that data is communicated among the processes with an optimally minimal number of messages.[1]

In Cannon's algorithm, the matrices A and B are partitioned into equally-sized blocks and are scattered to a grid of N processes. Each process calculates the local portion $C_{i,j}$ of $C = A \times B$ in a piece-wise fashion, shifting blocks of A and B round-robin to neighboring processes each step (Figure 1).

Algorithm 1: Cannon's Algorithm

Input: A {local portion of A}
B {local portion of B}
row {local rank's row in process grid}
col {local rank's column in process grid}
Output: C {local portion of C}

// Initial Shift of A & B
shift_left(A,i)
shift_up(B,j)

// Iterative Multiply and Rotation
for j *in* $0..\sqrt{n} - 1$ **do**
    C += A· B
    shift_right(A,1)
    shift_down(B,1)

The *shift* method in Algorithm 1 communicates sub-matrices across the process grid. For example, `shift_right(A[i;:], i)` will cause each process p to send its local block of A to the right by i processes, and receive a replacement block of A from a process i positions to the left (wrapping around as needed).

In evaluating and modelling compressed Cannon's we will frequently be discussing *speedup*. Speedup, in terms of latency, is defined as $S_{\text{latency}} = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}}$.

B. Data Compression

Data compression algorithms are classified into two major groups – *lossless*, and *lossy*. Lossless compressors are able to perfectly reproduce data from its compressed form, whereas lossy compressors will only produce an approximation. Because of their relaxed fidelity, lossy compressors are fundamentally able to achieve much greater reductions in size than lossless compressors.

Several general-purpose lossless compressors were used in this project. These include LZ4, ZStandard, Zip, and BLZ; all of which are byte-oriented descendants of the LZ77 Lempel-Ziv compressor. This family of compression algorithms operate by replacing repeated sequences of data with references to earlier occurrences. We leveraged C-Blosc, a "meta-compression" library, for its implementation of these compression algorithms [3]. C-Blosc performs compression in cache-sized chunks in order to minimize blocking, and additionally preprocesses blocks of data via a shuffling technique to improve their compressability.

Our scaling study also included two specialized compression algorithms. ZFP[6] is a compression library and format designed for multidimensional numerical arrays. ZFP supports lossy and lossless compression modes; we used its lossless mode in our study. NDZip[7] is a lossless compressor for floating-point data, designed for high-throughput use in distributed HPC applications.

A common measure used in data compression is the *compression ratio* of a compressor, which is calculated as $\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$.

C. Related Work

Rasouli et al. have written on compressing network communications in GEMM[8]. They've detailed and benchmarked a novel divide-and-conquer GEMM for sparse matrices, along with a compression scheme for the indices of their CSC-stored matrices. Our work differs in that our focus is on dense matrices, and in that we investigate the *complete* compression of data for matrices A and B.

There has been significant research into compressing communications for MPI-based (Message Passing Interface) applications. This includes schemes for transparent adaptive compression[9], GPU-accelerated compression[10][11], and compression-enhanced variants of MPI collective

operations[12][13]. The approach of offloading data to a GPU for compression prior to sending over the network, as Zhou et al. have done, is very promising and a potential future avenue for our work.

In this paper we discuss Cannon's algorithm as used for GEMM on dense datasets; Solomonik and Demmel[14] published on an extension of Cannon's to a 2.5D topology, capable of efficiently handling sparse data. The viability of lossless compression for Cannon's algorithm, as shown here, suggests that a similar approach could be adapted to their work.

III. Methods

A. Compressed Cannon's

Pseudo-code for compressed Cannon's algorithm can be seen in Figure 2. For the most part, it's identical to Cannon's algorithm as detailed in Figure 1. The main difference is that the local blocks A and B are compressed at the outset, with all further use of A and B requiring a decompression call. There is an added cost of two calls to `compress()` and $2\sqrt{n}$ calls to `decompress()`. Additionally there are increased memory requirements – A' and B' need to be stored for each step, and in the worst case these compressed copies could be of the same size as uncompressed A and B themselves.

Algorithm 2: Compressed Cannon's Algorithm

```
Input: A                                   {local portion of A}
B                                          {local portion of B}
row                                        {local rank's row in process grid}
col                                        {local rank's column in process grid}
Output: C                                  {local portion of C}

// Compress local (i,j) blocks of matrices A and B
A' ← compress(A)
B' ← compress(B)

// Initial shift of blocks
shift_left(A', col)
shift_up(B', row)

// Multiply and Rotate
for i in 1..√n do
    // Decompress received blocks
    A ← decompress(A')
    B ← decompress(B')

    // Multiply currently-held blocks
    C += A· B

    // Shift blocks left and up
    shift_right(A', 1)
    shift_down(B', 1)
```

B. Model

We have devised a simple performance model for predicting the communication latency of Cannon's algorithm with compression. We began with a standard *postal model*[15], which predicts the latency T using a per-message latency α and a per-byte transport cost β, coupled with the number of messages n and the quantity of bytes transmitted s:

$$T_{standard} = \alpha \cdot n + \beta \cdot s \tag{1}$$

$T_{standard}$ can capture the communication costs of Cannon's algorithm, but with the addition of a compressor we need to also account for the time spent on compressing and decompressing data, and the corresponding reduction in bytes transmitted:

$$T_{compressed} = \alpha \cdot n + \frac{\beta \cdot s}{r} + \frac{s}{c \cdot m_c} + \frac{s}{d \cdot m_d} \tag{2}$$

where r is the compression ratio; m_c is some positive real number which captures the fraction of bytes transmitted that are compressed; m_c captures the fraction of bytes transmitted which are decompressed; and c & d are the compression and decompression rates in units of bytes per second.

In the case of Cannon's algorithm, n, m_c, m_d, and s can be rewritten in terms of the number of processes p and the problem size. Given a grid of p processes and two matrices of size A bytes each, a process will exchange $n = 4\sqrt{p}$ messages totalling $s = \frac{4A}{\sqrt{p}}$ bytes. Only $\frac{2A}{p}$ bytes will be compressed however, making $m_c = 2\sqrt{p}$. As only the received messages are decompressed, $m_d = 2$.

$$T_{cannon} = 4\sqrt{p} \cdot \alpha + \frac{2A}{\sqrt{p}} \left(\frac{2\beta}{r} + \frac{1}{c \cdot \sqrt{p}} + \frac{1}{d} \right) \tag{3}$$

Let's consider only the cases where compression would be more efficient, or those where $T_{cannon} < T_{standard}$. This inequality is reducible to the following relationship:

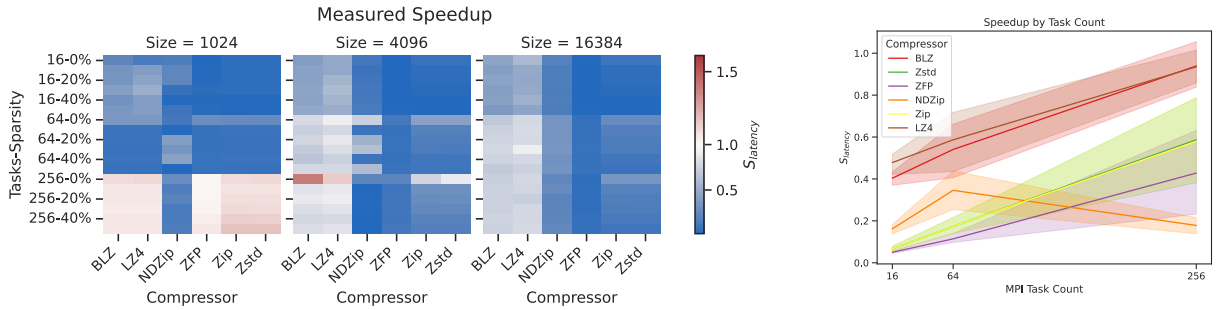
$$\left(\tau = \frac{r}{r-1} \cdot \frac{d + c\sqrt{p}}{2dc\sqrt{p}} \right) < \beta \tag{4}$$

The left-hand-side of this relation, which we'll refer to as τ, captures the throughput a compressor is capable of achieving.

With increasing $c\sqrt{p}$, τ approaches $\frac{1}{2d}$. Therefore with a strong reuse of compressed messages (corresponding to a large number of processes), or very fast compression (c), the decompression-speed will become the limiting factor. The decompression rate d is typically much greater than c, so ameliorating the cost of compression by increasing $\sqrt{p}$ will correspondingly improve the margin of improvement in throughput.

The compression ratio is also quite important in this relation, as it determines how much faster than β the compressor must be. For example, if a compressor is able to reduce messages by 10% in size, than the aggregate compression/decompression rate must be about $10\times$ smaller than β. If the compressor can achieve a 20% reduction in size, than this requirement drops to $5\times$.

IV. Experimental Results



(a) Observed speedup in latency across all tested configuration of our benchmark. (b) Trends in S_{latency} by process count.

Fig. 2

A. Implementation

In this study we measured the communication costs for compressed Cannon's algorithm; this includes time spent on compression, on decompression, and on MPI calls, as well as the size of compressed and uncompressed messages. We used the mean of these measurements, taken across MPI rank. Five trials were performed for each configuration, with the best-performing trial kept for analysis.

All testing was performed on a computer cluster at the Center for Advanced Research Computing at the University of New Mexico. Nodes in this cluster were equipped with Intel Xeon Gold 6226R processors, with 32 cores per node, and support for several SIMD instruction extensions (SSE1-4, AVX, AVX2, AVX512). The network interconnect was a 1x IniniBand HDR.

We profiled OpenMPI[16] on this cluster to determine values of α and β for use in our modelling. We found a maximum throughput of approximately 53 Gbit/s for larger MPI messages sent via a rendezvous strategy, and approximately 18 Gbit/s for smaller messaged sent eagerly. The coefficients for α and β as described by the postal model are listed in Figure 3.

MPI Strategy	α	β
Eager	0.000019	4.532×10^{-9}
Rendezvous	0.000624	1.494×10^{-10}

Fig. 3: Measured values for α and β, as used in equations detailed in Sec. III-B. Two sets are given, corresponding to the eager and rendezvous strategies used by MPI. For the instance of MPI evaluated here, the eager strategy was used for all messages smaller than 65536 bytes.

For data, we generated two-dimensional matrices of single-precision floating-point numbers. These numbers were drawn from a uniform distribution in the range $[-1, 1]$. A certain percentage of the elements were zeroed, corresponding to the "sparsity" parameter. The sizes of our matrices were chosen to sample a power-of-two scaling, while sparsity was linearly sampled between values of 0% & 50% (see Figure 5). We used NumPy[17] and Pandas[18][19] in creating this dataset.

We evaluated compressed Cannon's algorithm across 378 different configurations of parameters, covering a variety of compressors, problem sizes, data sparsities, and process counts. The sets of values tested are listed in Figure 5.

Compressor	None, ZFP, LZ4, NDZip, BLZ, ZStandard, Zip
Matrix Size	1024, 4096, 16384
MPI Task Count	16, 64, 256 (16 per node)
Data Sparsity	0%, 10%, 20%, 30%, 40%, 50%

Fig. 5: The cartesian-product of these parameters was tested in our study, corresponding to 378 distinct configurations.

Measured Performance

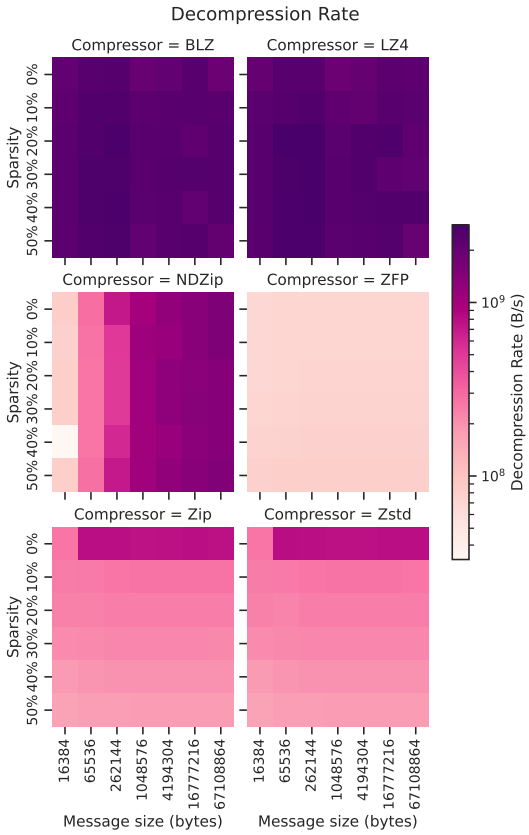
Of the 378 configurations we tested, there were 31 which resulted in a speedup of communication latency in comparison with the no-compression case. Figure 2a shows the observed speedups by configuration. Best performance was generally correlated with increasing data sparsity and task counts.

Compressor	Tasks	Size	Sparsity	S_{latency}
BLZ	256	4096	0%	1.610785
Zip	256	1024	50%	1.277837
Zstd	256	1024	50%	1.275302
LZ4	256	4096	0%	1.222417
Zip	256	1024	40%	1.217530

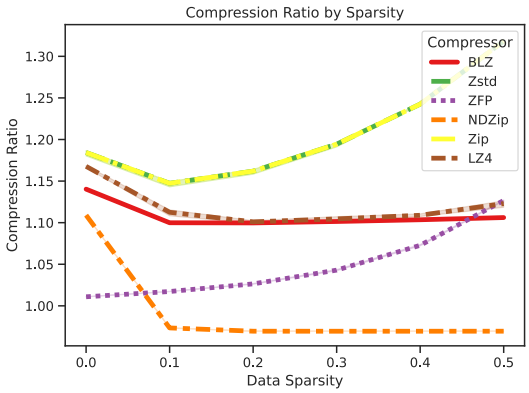
Fig. 6: Top performing trials, by Compressor and data parameters.

Our study tested four general-purpose lossless compressors – BLZ, LZ4, Zip, & Zstd – as well as two floating-point compressors, NDZip and ZFP. Figure 4 contains several visualizations which help characterize the performance of the compressors used in our study.

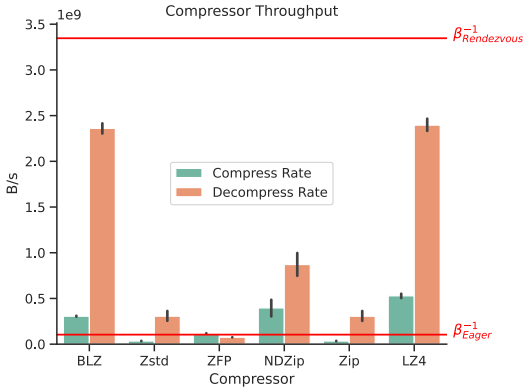
Reduction in message size, as captured by the compression ratio, was strongly correlated with sparsity (Figure 4b). At the point of 0% sparsity, we can see how these compressors perform with very-dense floating-point data. Although these data were randomly selected, their constrained range ($U_{-1,1}$)



(a) The decompression-rates achieved by compressors evaluated in our scaling study, in bytes per second.



(b) The measured reduction in size of submatrices due to compression, relative to uncompressed Cannon's.



(c) Measured compression & decompression rates for each compressor, compared against the measured MPI bitrates as are detailed in Figure 3.

Fig. 4

results in similar exponents in their underlying floating-point encoding.

BLZ & LZ4 led by a significant margin in compression and decompression rates (Figure 4c). Most of the tested compressors were able to process data faster than the MPI-eager bitrate, but none were able to surpass the MPI-rendezvous rate.

Model Evaluation

We can also evaluate the performance of these compressors by computing their τ values, as described in Eq. 4. In Figure 7, the distributions of τ is shown per compressor, as derived from measured compressor attributes in our study. Interestingly, NDZip straddles the β_{Eager} line, despite failing to achieve a speedup in testing over uncompressed Cannon's algorithm. This is likely due to NDZip's poor performance for the small message sizes where β_{Eager} is applicable, as can be seen in Figure 4a.

In addition to the scaling study performed and detailed above, we modeled how compressed Cannon's algorithm might behave with stronger scaling. Figure 8 shows the speedup in communication latency with up to 4096 processes, for submatrices ranging from 1024 kB to 32768 kB in size. Modest improvements in overall time spent are predicted, with larger matrices benefiting more strongly.

V. Conclusions and Future Directions

We have presented a variant of Cannon's algorithm in which communication is compressed, using a "compress once, reuse often" strategy. We undertook a scaling study of the communication costs, finding that BLZ and LZ4 are strong candidates for use in compressing floating-point data that falls within MPI's eager communication limit. Additionally, we modeled further scaling by task-count and problem size, predicting that both are positively correlated with a speedup in latency.

In our future work we'll consider lossy compression algorithms, GPU-offloaded compression, and threaded compressors. We will also be investigating the applicability of similar compression strategies in sparse distributed GEMM and algebraic multigrid applications.

Acknowledgment

This work was performed with partial support from the National Science Foundation under Grant No. CCF-2151022

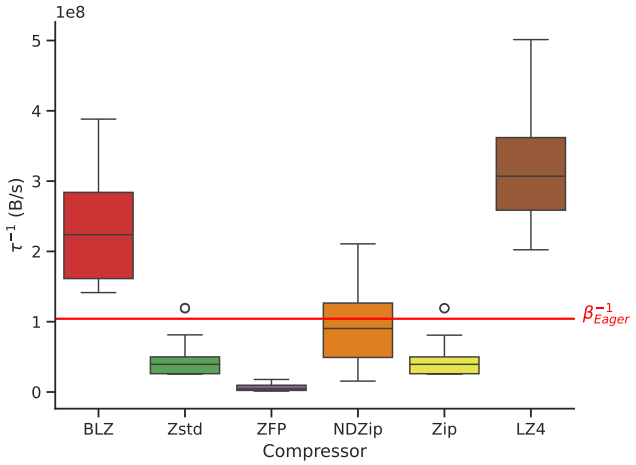


Fig. 7: The distribution of values for τ (described in Eq. 4) as derived from scaling-study measurements.

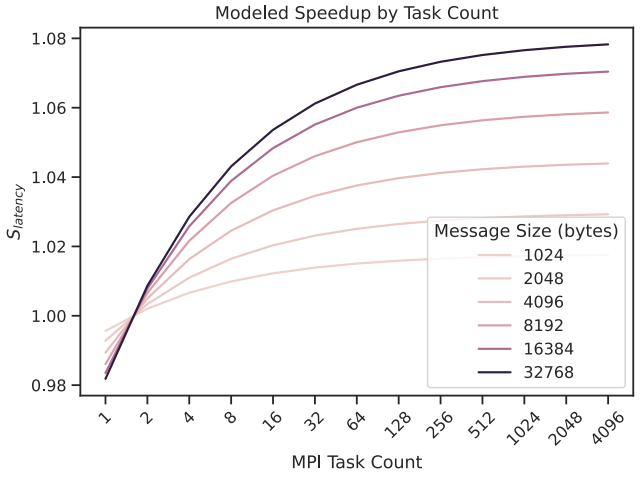


Fig. 8: Modeled improvement in S_{latency} predicted by our model as matrix size and process count are scaled.

and the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the U.S. Department of Energy's National Nuclear Security Administration.

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the high performance computing resources used in this work.

Figures in this paper were generated with the assistance of Seaborn[20] and Matplotlib[21].

References

[1] H.-J. Lee, J. P. Robertson, and J. A. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 44–51.

[2] Y. Collet. Lz4 - extremely fast compression. [Online]. Available: https://github.com/lz4/lz4

[3] B. D. Team. Blosc: A blocking, shuffling and lossless compression library. [Online]. Available: https://www.blosc.org

[4] I. PKWARE. Appnote.txt - .zip file format specification. [Online]. Available: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[5] Y. Collet and M. Kucherawy, "Zstandard compression and the application/zstd media type," Tech. Rep., 2018.

[6] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[7] F. Knorr, P. Thoman, and T. Fahringer, "ndzip: A high-throughput parallel lossless compressor for scientific data," in *2021 Data Compression Conference (DCC)*. IEEE, 2021, pp. 103–112.

[8] M. Rasouli, R. M. Kirby, and H. Sundar, "A compressed, divide and conquer algorithm for scalable distributed matrix-matrix multiplication," in *The International Conference on High Performance Computing in Asia-Pacific Region*, 2021, pp. 110–119.

[9] R. Filgueira, D. E. Singh, J. Carretero, A. Calderón, and F. García, "Adaptive-compi: Enhancing mpi-based applications' performance and scalability by using adaptive compression," *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 93–114, 2011.

[10] Q. Zhou, C. Chu, N. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, "Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 444–453.

[11] Q. Zhou, P. Kousha, Q. Anthony, K. Shafie Khorassani, A. Shafi, H. Subramoni, and D. K. Panda, "Accelerating mpi all-to-all communication with online compression on modern gpu clusters," in *International Conference on High Performance Computing*. Springer, 2022, pp. 3–25.

[12] H. Shan, S. Williams, and C. W. Johnson, "Improving mpi reduction performance for manycore architectures with openmp and data compression," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2018, pp. 1–11.

[13] J. Huang, S. Di, X. Yu, Y. Zhai, Z. Zhang, J. Liu, X. Lu, K. Raffenetti, H. Zhou, K. Zhao *et al.*, "An optimized error-controlled mpi collective framework integrated with lossy compression," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 752–764.

[14] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.

[15] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, 1992, pp. 13–22.

[16] E. G. et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[17] C. R. H. et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[18] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[19] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[20] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. [Online]. Available: https://doi.org/10.21105/joss.03021

[21] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.