International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 13, Number 1, pages 48–61, January 2023

Data Transfer API and its Performance Model for Rank-Level Approximate Computing on HPC Systems

Yoshiyuki Morie
Faculty of Fukuoka Medical Technology, Teikyo University
Omuta, Fukuoka, 836-8505, JAPAN

Yasutaka Wada
School of Information Science, Meisei University
Hino, Tokyo, 191-8506, JAPAN

Ryohei Kobayashi
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, 305-8577, JAPAN

and

Ryuichi Sakamoto
Global Scientific Information and Computing Center, Tokyo Institute of Technology
Meguro, Tokyo, 152-8550, JAPAN

Received: July 26, 2022
Revised: October 31, 2022
Accepted: November 27, 2022
Communicated by Masahiro Shibata

Abstract

The application of approximate computing (AC) in optimizing tradeoffs among performance, power consumption, and accuracy of computation results can be improved by adjusting data precision in applications. The importance of AC has increased over the years as it is used to maximize performance even with limited power budget and hardware resources in high performance computing (HPC) systems that require more precise computations. To apply AC for HPC applications effectively, we must consider the character of each message passing interface (MPI) rank in an application and optimize it by adjusting its data precision. This rank-level AC ensures that ranks and threads in an application run with data precision and perform data transfer while converting the precision of target data. In this paper, we have proposed and evaluated data pack/unpack application programming interfaces (APIs), which are applicable for standard MPI programs run on HPC systems, for converting the precision of target data. The proposed APIs enable us to express data transfer among ranks with different precisions. In addition, we have also developed a reasonable performance model to select an appropriate data transfer API for maximizing performance with rank-level AC based on performance evaluation with various HPC systems.

Keywords: Approximate Computing, Data Transfer API, High Performance Computing Systems, Performance Modeling

1 Introduction

In recent years, we have encountered various limitations regarding the hardware resources in a computer system, such as a limited number of processor cores, limited amount of memory, limited network performance, and limited power budget available. In addition, we have also faced challenges in extracting performance from various applications even with their limited parallelism and complicated structures. However, the expectations from advanced scientific computation are high, and the software technology must be improved to address the aforementioned limitations. Approximate computing (AC), which changes/adjusts the precision of data used in an application, is a promising approach to improve computer systems' and applications' performance/effectiveness under such limitations.

High performance computing (HPC) systems are particularly affected by such limitations because they must execute parallel applications that consist of multiple ranks/processes with varied performances. To improve the effectiveness of the execution of such an application, we need to optimize load balance among ranks by applying AC for each rank. However, ranks in HPC application are executed while communicating with each other, and it is important to address the difference in data precision among ranks. To solve such problems, we have developed a rank-level AC method for HPC applications to optimize load balance and data transfer among ranks in an application simultaneously. Applications of HPC such as stencil calculation require data to be packed/unpacked for data transfer. Therefore, we have proposed a prototype to implement data pack/unpack APIs with precise data conversion. In addition, we evaluated its performance and quantified its overhead [10]. However, the performance degraded in its preliminary evaluation in contrast to our assumption.

The contributions of this paper are summarized as follows:

- We have clarified that fully simdization improves the performance of the proposed API.
- We have evaluated the performance of various strides during pack/unpack.
- We have created a performance model to clarify the criteria for converting data precision.

The rest of this paper is organized as follows: Section 2 presents a comprehensive review of related research works. Section 3 describes the data transfer process for an AC-applied MPI application. Section 4 explains the proposed model of data transfer APIs for converting the data precision of target data. Section 5 evaluates the performance of the proposed APIs and their implementation. Section 6 explains the performance model for selecting an effective data transfer method. Finally, Section 7 presents the conclusion.

2 Related Works

AC can be applied on various levels in a system; that is, from hardware to application [4, 9].

Some applications, such as image processing and deep learning, are robust against degradation of calculation precision; therefore, we can apply AC more efficiently for these applications. Chen et al. proposed a network parameter compression method for deep learning to reduce the communication overhead of distributed learning [5]. However, in terms of HPC system usage, AC must be applied to various MPI applications.

Karakoy et al. proposed a slicing-based approach to reduce memory access while ensuring that accuracy lies within the specified error bound [8]. Shafique et al. implemented open-source libraries of AC-enabled arithmetic components to develop AC-enabled systems considering multiple system layers (from logic to architecture) simultaneously [13]. Fujiki et al. proposed an approximate interconnection network that ignores minor communication errors among compute nodes for HPC systems [6].

Some research works utilize both hardware and software characteristics based on the target applications. Hara and Hanawa utilized field programming gate array (FPGA) flexibility to offload

* A preliminary version of this paper was published in the 24th Workshop on Advances in Parallel and Distributed Computational Models (APDCM), May 2022. [10]

some tasks in an HPC application [7]. In addition, certain research works memorize the input/output of repeated computations and reuse them while allowing some accuracy degradation [11, 12]. As a result, AC can be applied to various applications; however, we still need to consider the characteristics of each thread/rank in a parallel application.

Though the above-mentioned challenges are for HPC systems, we have encountered similar challenges during the design of the proposed model as well. [6] focuses on FPGA utilization for AC but does not consider differences among ranks. [8] utilizes optical interconnect for lower latency by allowing soft errors; however, it assumes error-tolerant applications only. To expand the scope of AC techniques for HPC systems, it is important to consider characteristics of parallel threads/ranks and communication among them. This paper proposes rank-level AC and data transfer APIs to optimize the performance of each rank in various applications and develop a data transfer performance model with the API.

3 Data Transfer in an MPI application with Rank-Level AC

This section summarizes the proposed rank-level AC and its challenges from the viewpoint of data transfer.

3.1 Rank-Level AC

HPC application with MPI consists of multiple ranks and each rank is executed while transferring data with other ranks. Depending on the application characteristics, load unbalance exists among ranks, which degrades the execution performance. In this paper, we have proposed rank-level AC to improve the application execution performance by reducing the load unbalance among ranks.

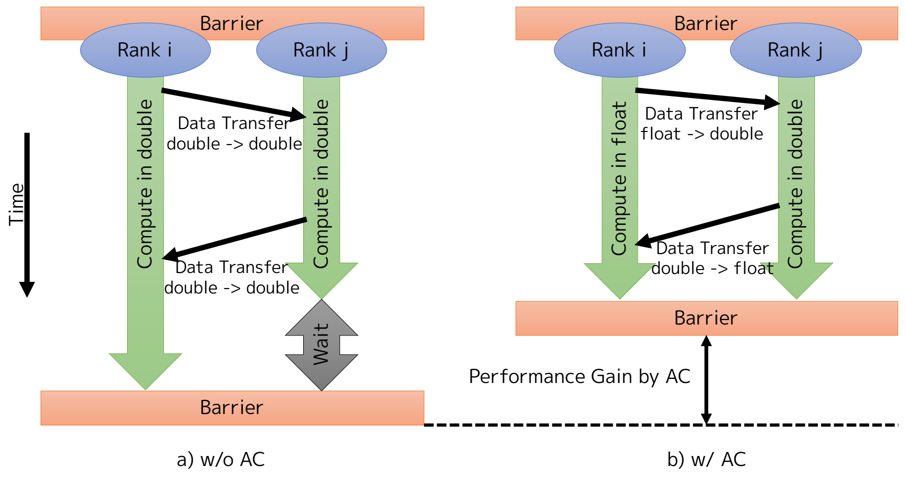

Figure 1: Example of rank level AC with MPI application [10]

Fig. 1 shows an example of rank-level AC. Fig. 1 (a) shows the load unbalance between the ranks i and j in an application. To detect such load unbalance among ranks, we can utilize compiler analysis information, profiling results from test runs, and time measurement functions implanted in the runtime libraries used for parallel computation. The execution performance of this application can be improved by reducing the load unbalance. In addition, we can apply AC (change the data precision from double to float) to the slower rank and shorten its execution time. Then, we can reduce the wait time in the faster rank and obtain the performance gain (Fig. 1 (b)). However, to apply this rank-level AC, we need to change the precision of data transferred between ranks. This paper proposes and implements data transfer APIs that resolve this problem.

3.2 Data Transfer Problem for Rank-Level AC

Fig. 2 shows the change in communication performance of double and float for the same count. This figure shows that the communication improvement is around 200% when double is converted to float in large messages with AC.

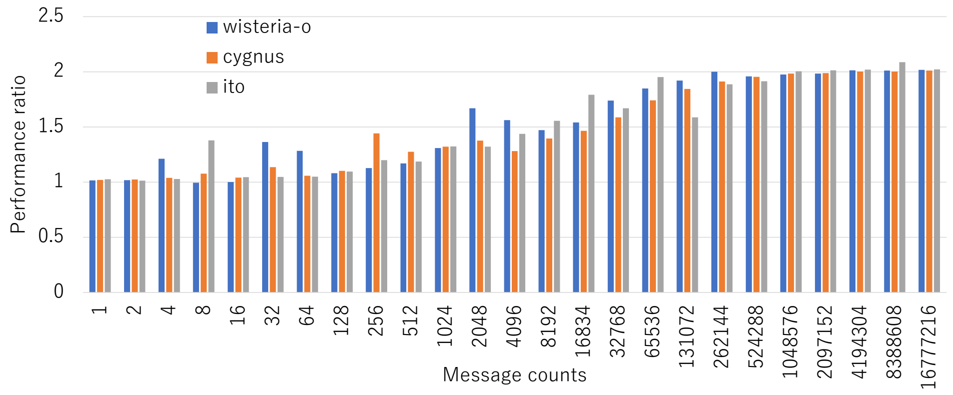

Figure 2: Data transfer overhead comparison with float and double on Wisteria-O [3]/Cygnus [2]/ITO [1]

The communication performance is relatively simple. For applications like stencil calculation, the cost of estimating cast and pack/unpack data is essential for backward and forward communication. However, this is complicated due to various factors.

4 Data Transfer APIs and Their Implementation for Rank-Level AC

When applying AC to data transfer, shorter data transfer time implies reduced precision data. However, we must cast the target data to lower precision while transferring it. In MPI applications such as stencil calculation, it is necessary to pack/unpack the target data to the buffer before and after transferring. Since the pack/unpack process copies the data independently, we can perform the data cast and copy processes simultaneously.

4.1 Data Transfer API Definitions

Table 1: Pack/unpack interfaces

Interface	abstract
inline static int aac_pack_d2f(double *indata, float *outdata, int ncount, int stride)	pack with converting double to float
inline static int aac_unpack_d2f(float *indata, double *outdata, int ncount, int stride)	unpack with converting float to double
inline static int aac_pack(double *indata, double *outdata, int ncount, int stride)	pack without converting the data precision
inline static int aac_unpack(double *indata, double *outdata, int ncount, int stride)	unpack without converting the data precision

The pack/unpack APIs are summarized in Table 1. The aac_pack/unpack_d2f APIs are used to pack/unpack APIs with converting the data precision, and acc_pack/unpack APIs are without converting the data precision. We have specified the inline expansion to prevent performance degradation caused by dereferencing array during data cast in these APIs.

4.2 Data Transfer API Implementation

The proposed API is implemented using a simple array access. The implementations of pack and unpack functions are described below.

Pack reads memory at stride interval specified in a single loop and writes to consecutive memory addresses. In the case of converting the data precision, the read data is cast (double to float) into write memory address.

However, unpack also reads data consecutively and writes each data into stride interval specified in a single loop. In the case of converting the data precision, data is cast (float to double) when writing it into each address.

Algorithm 1 Implementation of aac_pack_d2f() and aac_unpack_d2f()

```
1: function ACC_PACK_D2F(input, output, NCOUNT, stride)
2:     for $i = 0 \dots \text{NCOUNT} - 1$ do
3:         Output[i] = float(input[i*stride])
4:     end for
5: end function
1: function ACC_UNPACK_D2F(input, output, NCOUNT, stride)
2:     for $i = 0 \dots \text{NCOUNT} - 1$ do
3:         Output[i*stride] = double(input[i])
4:     end for
5: end function
```

The pseudo code is shown in Algorithm 1. It is a simple loop; however, SIMD may not work for some compilers. Therefore, it is necessary to manually optimize the loop by expanding it. However, in this evaluation, we did not perform manual optimization, but we increased the optimization level of the compiler.

5 Performance Evaluation of the Proposed API

In this evaluation, we have measured and compared two types of communication times, namely with and without converting the data precision (double to float).

We have evaluated the performance of the proposed APIs with a ping-pong communication program. This program executes data pack/unpack processes before and after each ping-pong communication and repeats them according to the number of message counts to transfer. In this time, we have measured the average communication time of each message count (changing from 1 to 16777216) and four types of stride intervals (1, 8, 256, 8192).

5.1 Evaluation Environments

We have employed Wisteria-O at the University of Tokyo, Cygnus at University of Tsukuba, and ITO at Kyushu University for our performance evaluation.

Table 2 shows the specification of Wisteria-O, which is family of super computer "Fugaku."

Table 3 shows the specification of Cygnus, which is Infiniband HDR100 system. Though each node in Cygnus has four InfiniBand HDR100 ports, we have utilized only one port in this experiment to eliminate the performance instability caused by link aggregation. We have restricted the number of

Table 2: Specification of Wisteria-O

	Computing node on Wisteria-O
Processor	A64FX 48C 2.2GHz
Memory	32GB
Interconnect	Tofu interconnect D
Topology	6D Mesh/torus
OS	Red Hat Enterprise Linux
Compiler and MPI Library	FUJITSU Software Technical Computing Suite V4.0
Compile option	`-O3 -Kfast -Ksimd_packed_promotion -fopenmp`

available interconnection ports to one by setting the option "`-x UCX_MAX_RNDV_RAILS=1`" for `mpirun` with OpenMPI.

Table 3: Specification of Cygnus

	Computing node on Cygnus
Processor	Intel Xeon Gold 6125 x2
Memory	192GiB (DDR4)
Interconnect	InfiniBand HDR100 x 4 port
Link speed	100Gpbs x4
Topology	Fat-tree
Compiler	gcc version 8.3.1
MPI Library	Open MPI 4.0.3
Compile option	`-O3 -mavx2`

Table 4 shows the specification of ITO, which is Infiniband EDR100 system.

Table 4: Specification of ITO

	Computing node on ITO
Processor	Intel Xeon Gold 6154 (3.0GHz 18 cores) x2
Memory	192 GB (DDR4)
Interconnect	InfiniBand EDR100
Link speed	100Gpbs
Topology	Fat-tree
Compiler	Intel oneAPI 2021.3
MPI LIbrary	Intel(R) MPI Library for Linux* OS, Version 2021.3
Compile option	`-O3 -mavx2`

5.2 Evaluation Results

Fig. 3 shows the data transfer performance improvement with converting the data precision (from double to float) on Wisteria-O/Cygnus/ITO. In this figure, y-axis shows the performance improvement of data transfer with converting the data precision versus without converting the data precision (from double to double). Then, 1.0 or higher means that converting the data precision improved the performance. The average performance improvement of with data conversion versus without conversion was approximately 41% on Wisteria-O, 31% on Cygnus, 35% on ITO, and the message count of 32M on Wisteria-O gave us the maximum performance improvement, which was approximately 68%. The performance improvement of larger message sizes was approximately 50 – 60 %. This is because the memory copy and communication bandwidth have a significant effect as the message size increases. Upon converting data from double to float, the communication size and memory

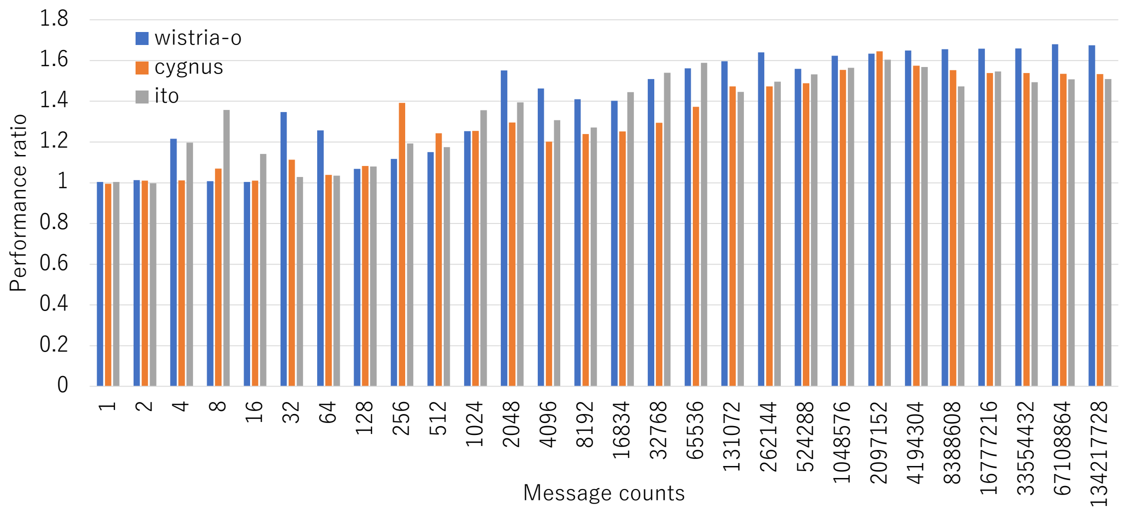

Figure 3: Data transfer and pack/unpack overhead comparison between with and without converting the data precision on Wisteria-O/Cygnus/ITO

access size are halved. By reducing the amount of data, the overall performance was improved as in this evaluation.

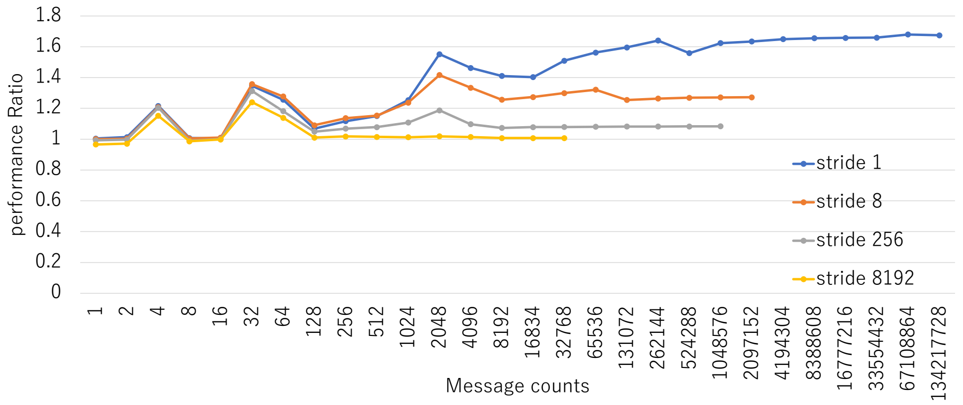

Figure 4: Data transfer and pack/unpack overhead comparison between with and without converting the data precision for each stride interval (1, 8, 256, 8192) on Wisteria-O

Fig. 4 shows the data transfer performance improvement with converting the data precision for each stride interval (1, 8, 256, 8192) on Wisteria-O. As the stride increases, the rate of improvement of performance decreases. This is because, when the stride increases, a cache miss occurs while accessing the data owing to the fact that the penalty for cache miss is greater than cost of data transfer. In all stride intervals of data transfer, 32 counts was local peak performance because the effect of throughput is larger than latency cost. If the number of counts is the same, double data types become twice the data size for approximately twice the data transfer time. The peak performance of 2048 counts is assumed to be due to switching of Eager/Rendezvous. Stride 8192 does not improve performance above 128 counts. In these strides, the cost of pack/unpack was much larger than the performance gain of data transfer.

Fig. 5 shows the pack/unpack overhead comparison of with and without converting the data precision for each stride (1, 8, 256, 8192) on Wisteria-O. In sequential access, data copied between doubles was 1.5 to 2 times higher performance for most message sizes. This may be because, SIMD is not used efficiently. For this reason, the bandwidth is not fully used for float data. Stride 8 and 256

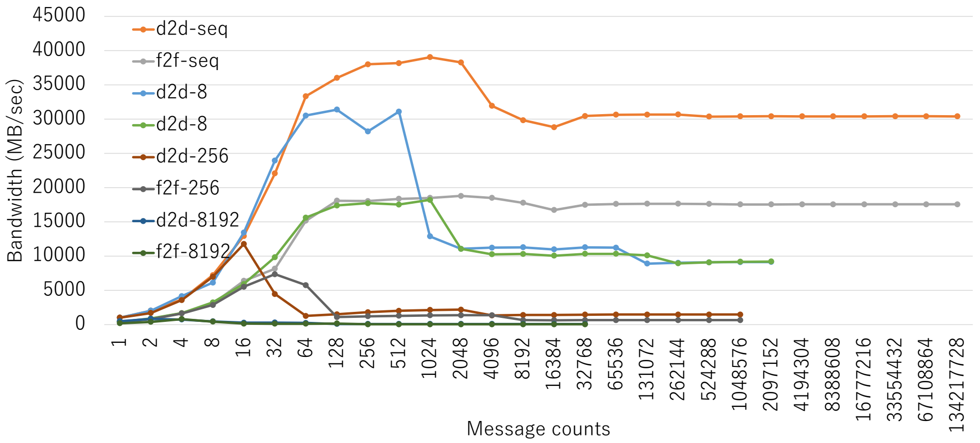

Figure 5: Pack/unpack overhead comparison between with and without converting the data precision for each stride interval (1, 8, 256, 8192) on Wisteria-O

were equivalent to the bandwidths of float data and double data at 1024 counts (stride 8) and 128 counts (stride 256), respectively. As a result, when transferring the same counts, the data transfer can be performed in half the time. However, if the number of counts is higher, the performance may improve even if the cache miss penalty for each data becomes larger than the communication cost like stride 8. In contrast, stride 8192, for example, has no bandwidth at all. There is no performance improvement in pack/unpack and memory copy dominates due to cache miss. Therefore, there is no overall performance improvement.

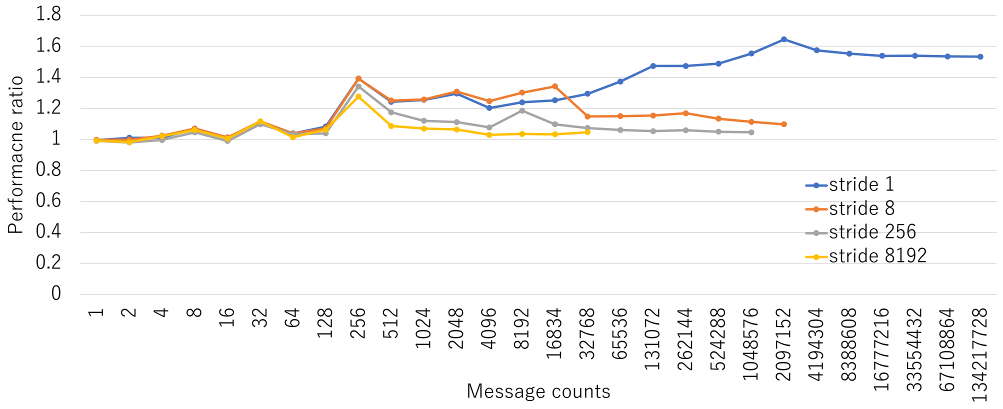

Figure 6: Data transfer and pack/unpack overhead comparison of with and without converting the data precision for each stride interval (1, 8, 256, 8192) on Cygnus

Fig. 6 shows the data transfer performance improvement with converting the data precision for each stride interval (1, 8, 256, 8192) on Cygnus. For Wisteria-O, the performance improvement rate decreases as the stride increases. Thus, for stride interval of data transfer, 256 counts were used to evaluate local peak performance due to effect of throughput that is larger than latency cost. If the number of counts is same, double data types become twice the data size and data transfer time is approximately twice. If the number of counts is same, double data types become twice the data size and data transfer time is approximately twice. The performance of 2048 counts on stride 1 and 8 was peak. We believe this may be originated from switching of eager/rendezvous as like Wisteria-O.

Fig. 7 shows the pack/unpack overhead comparison of with and without converting the data precision for each stride (1, 8, 256, 8192) on Cygnus. In sequential access, data copied between doubles exhibited 1.6 times higher performance until 64K counts. All stride intervals are equal to

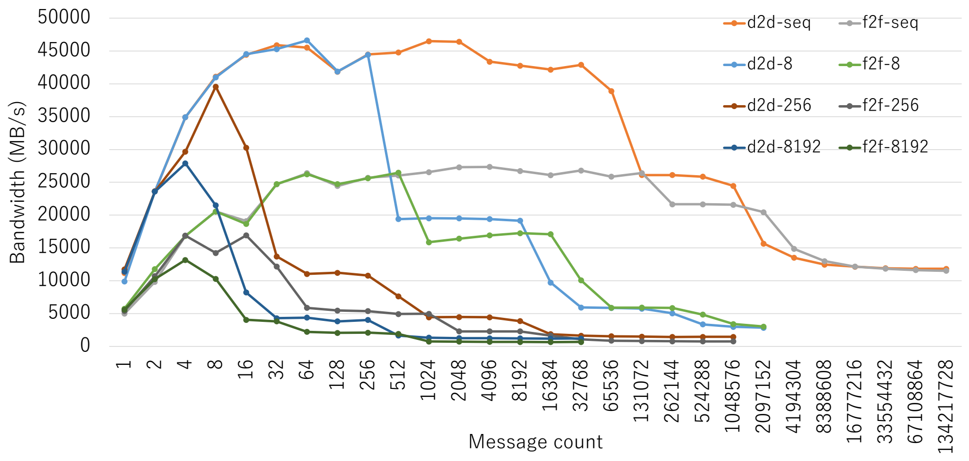

Figure 7: Pack/unpack overhead comparison of with and without the converting the data precision for each stride interval (1, 8, 256, 8192) on Cygnus

the bandwidths of float data and double data at 262144 counts (stride 1), 512 counts (stride 512), and 16 counts (stride 256 and 8192). From this, performance was improved for medium message sizes with stride 8 and 256, and for medium and large message sizes with stride 8192. However, except for sequential copy, when the transfer counts were large, bandwidth was almost significantly reduced. At this point, there was no performance improvement in pack/unpack and memory copy dominates due to cache miss. Therefore, there was no overall performance improvement.

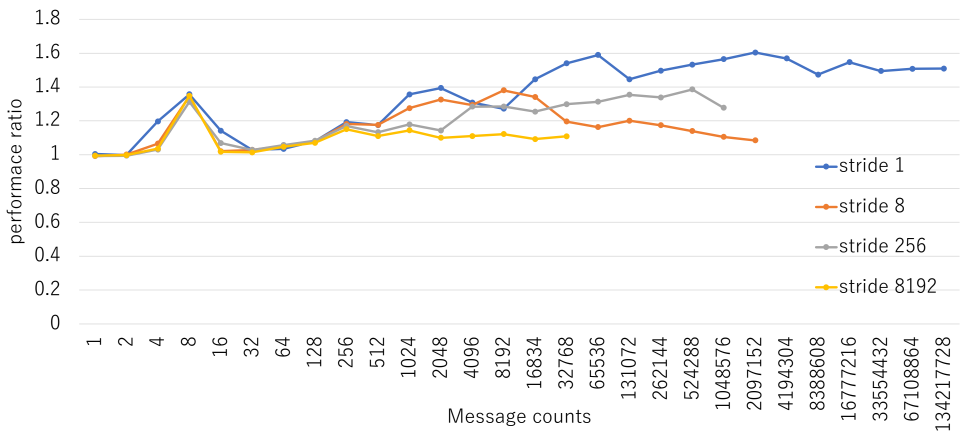

Figure 8: Data transfer and pack/unpack overhead comparison of with and without converting the data precision for each stride interval (1, 8, 256, 8192) on ITO

Fig. 8 shows the data transfer performance improvement with converting the data precision for each stride interval (1, 8, 256, 8192) on ITO. For Wisteria-O/Cygnus, the rate of improvement of performance decreases as the stride increases. In all stride counts of data transfer, 8 counts depicted local peak performance due to the effect of throughput that was larger than latency cost. If the number of counts is same, double data types become twice the data size and the data transfer time approximately doubled. The performance of 2048 counts on stride 1 and 8 was at its peak because preeager/rendezvous acts like two computer systems.

Fig. 9 shows the pack/unpack overhead comparison with and without converting the data precision for each stride interval (1, 8, 256, 8192) on ITO. All the stride intervals almost all the equivalent the bandwidths of float data and double data to 262144 counts (stride 1), 512 counts (stride 512),

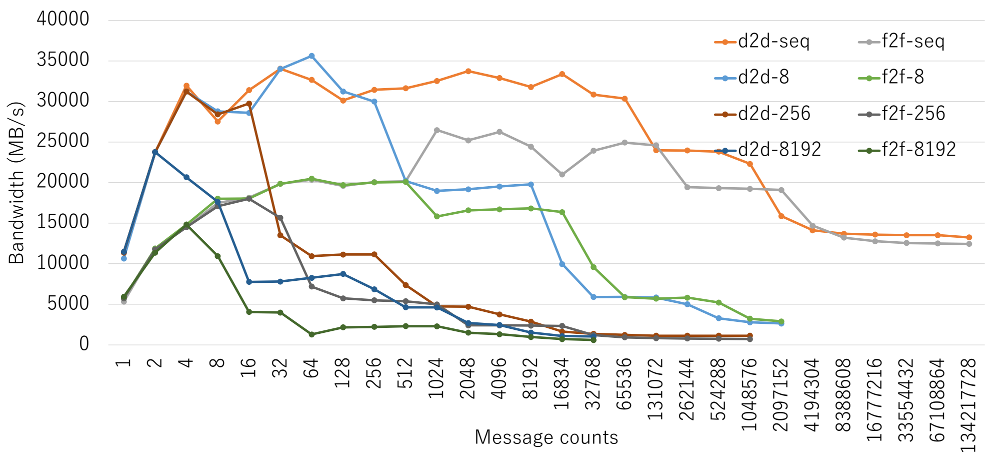

Figure 9: Pack/unpack overhead comparison of with and without converting the data precision for each stride interval (1, 8, 256, 8192) on ITO

and 32 counts (stride 256 and 8192). Thus, performance was improved for medium message sizes with stride 8 and 256. However, except for sequential copy, when the transfer counts were large, the bandwidth was almost significantly reduced. At this point, there is no performance improvement in pack/unpack and memory copy dominates due to cache miss, which is the same as Cygnus.

Similar results were obtained in various experimental environments, indicating that the proposed API can be widely used.

6 Performance Modeling to Select Effective Data Transfer Method with Rank-Level AC

We have created a modeling of data transfer parallel computing to determine with/without converting the data precision.

First, the cast process that converts the data precision is CPU processing, which is faster than memory transfer; therefore, we have not considered it.

Next, the model of data transfer between ranks is divided into a communication process and pack/unpack process. This part of communication can be estimated by latency and communication bandwidth, and the pack/unpack part can be estimated by memory bandwidth.

Eq. 1 shows the model equation of time $Time_{f2d}$ usec that transfers data with converting the data precision. L usec is network latency, $Messagesize_{double}$ MB/s is the size of one double data. $Messagesize_{float}$ MB/s is the size of one float data, $CommBW_{f2f}$ MB/s is the communication bandwidth of float data, $MemBW_{float}$ MB/s is the memory bandwidth of float data, and $MemBW_{double}$ MB/s is the memory bandwidth of double data.

$$Time_{f2d} = L + \frac{Messagesize_{double}}{MemBW_{double}} + \frac{Messagesize_{float}}{MemBW_{float}} + \frac{Messagesize_{float}}{CommBW_{f2f}} \tag{1}$$

To estimate the performance gain with converting the data precision, we have also created a model equation of $Time_{f2d}$ usec that converts without data precision. Eq. 2 shows the model equation of time $Time_{d2d}$ usec that transfers data with converting the data precision. $CommBW_{d2d}$ MB/s is the communication bandwidth of float data.

$$Time_{d2d} = L + 2 \times \frac{Messagesize_{double}}{MemBW_{double}} + \frac{Messagesize_{double}}{CommBW_{d2d}} \tag{2}$$

The accuracy of the model is determined by comparing the performance ratio between the estimated value and measured value, which is shown in the figure below. In this evaluation, the parameters were obtained from actual measurements of communication and pack/unpack message counts. The parameters except for L use the maximum bandwidth of message count. The communication time of the minimum message size was applied for L.

To indicate the accuracy of the model, Fig. 10 shows the comparison between the measured and estimated performance ratios on Wisteria-O. The parameters of the model are as follows.

L is 1.20 usec, and $CommBW_{f2f}$ is 5413 MB/s. Then, $CommBW_{d2d}$ is 5369 MB/s, $MemBW_{float}$ is 17559 MB/s, and $MemBW_{double}$ is 30390 MB/s.

A large message count indicates that the performance ratio is accurately estimated by the proposed model. There is a great deal of variability in the intermediate sizes; however, the average accuracy was 11.8% for f2d data transfer and 12.7% for d2d data transfer.

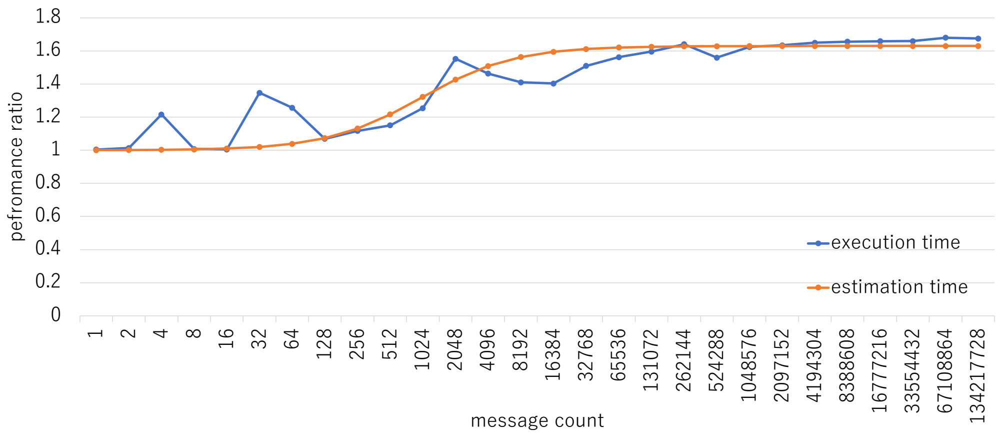

Figure 10: Comparison between estimation time ratio and execution time ratio on Wisteria-O

To indicate the accuracy of the model, Fig. 11 shows the comparison between the measured and estimated performance ratios on Cygnus. The model parameters of Cygnus are as follows.

L is 1.59 usec, $CommBW_{f2f}$ is 11162 MB/s, $CommBW_{d2d}$ is 9511 MB/s, $MemBW_{float}$ is 11520 MB/s, and $MemBW\_double$ is 11817 MB/s.

A large message count indicates that the performance ratio is accurately estimated by the proposed model similar to Wisteria-O. The average accuracy was 7.6% for f2d data transfer and 7.8% for d2d data transfer.

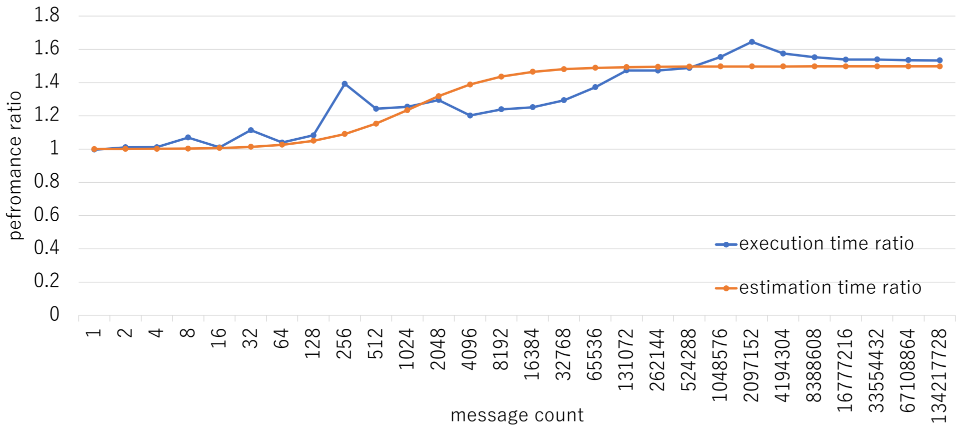

Figure 11: Comparison between estimation time ratio and execution time ratio on Cygnus

To indicate the accuracy of the model, Fig. 12 shows the comparison between the measured and estimated performance ratios on ITO. The model parameters of ITO are as follows.

L is 1.20 usec, $CommBW_{f2f}$ is 11203 MB/s, $CommBW_{d2d}$ is 10987 MB/s, $MemBW_{float}$ is 12436 MB/s, and $MemBW_{double}$ is 13246 MB/s.

A large message count indicates that the performance ratio is accurately estimated by the proposed model similar to the other two computer systems. The average accuracy was 1.9% for f2d data transfer and 0.1% for d2d data transfer.

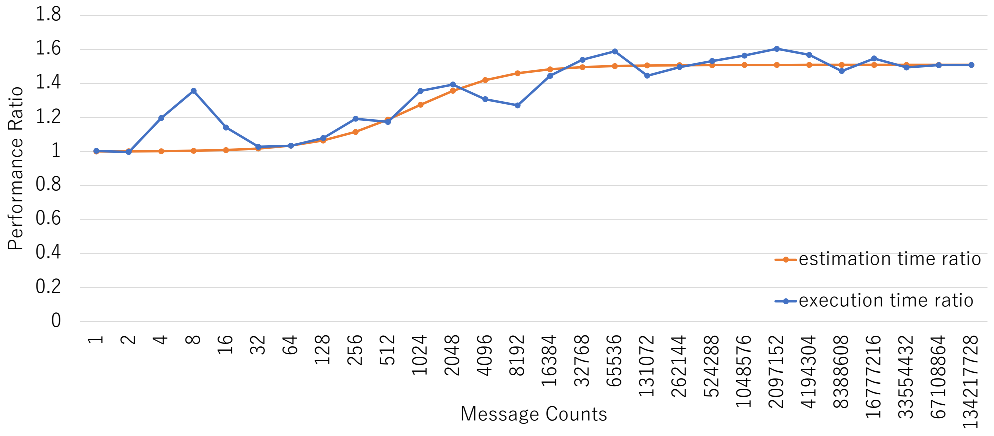

Figure 12: Comparison between estimation time ratio and execution time ratio on ITO

The three model evaluations indicate that the accuracy of the proposed model was high when the message size was large. The communication performance ratio deviated up and down for medium level messages and below. The reason for this was probably the hierarchical memory. Since the estimation time uses the main memory bandwidth for performance for all message sizes, the graph showed a smooth performance transition. However, since there is a hierarchical memory in the measurement with the real machine, the performance varies depending on the type of memory used. Double data has twice the message size of float data; therefore, float data accesses faster memory and double data accesses slower memory for some message counts. For example, ITO system had up to L3 cache and its memory performance changed four times: line size, L1 cache, L2 cache, and L3 cache; therefore, four major performance peaks could be seen in the measured performance ratio. L2 cache of ITO system is 1024KB. In this measurement, when the count is 64K, 1024KB is used for pack/unpack process due to which cache misses occurred. In contrast, the float data used half of double data, that is, 512KB, due to which cache misses did not occur and the performance of the proposed method was further improved and appeared as a performance peak. Our future work is to investigate whether it is necessary to reflect the performance of hierarchical memory in the model when applying AC.

7 Conclusions

AC is one of the promising techniques to improve the effectiveness of HPC systems under limited resources. To apply rank-level AC for HPC applications, we need to convert the precision of the target data while transferring data among ranks. In this paper, we proposed the implementation of data transfer APIs . We successfully quantified data transfer overhead with the data pack/unpack APIs through performance evaluation. We also created a performance model of the proposed API for rank-level data transfer and showed that the model is accurate with large message sizes. In the future, we plan to utilize the proposed APIs in MPI applications to realize rank-level AC on HPC systems and improve our model's accuracy of data transfer for rank-level AC considering real applications.

Acknowledgment

This research was supported in part by Multidisciplinary Cooperative Research Program in CCS, University of Tsukuba and by the Priority Research Funding from Meisei University. The computation was performed using the computer resource offered under the category of General Projects by Research Institute for Information Technology, Kyushu University.

References

[1] Introduction of ITO. `https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/01_intro.html`.

[2] Overview of cygnus: a new supercomputer at CCS. `https://www.ccs.tsukuba.ac.jp/wp-content/uploads/sites/14/2018/12/About-Cygnus.pdf`.

[3] Wisteria/BDEC-01 supercomputer system. `https://www.cc.u-tokyo.ac.jp/en/supercomputer/wisteria/service/`.

[4] Ankur Agrawal, Jungwook Choi, Kailash Gopalakrishnan, Suyog Gupta, Ravi Nair, Jinwook Oh, Daniel A. Prener, Sunil Shukla, Vijayalakshmi Srinivasan, and Zehra Sura. Approximate computing: Challenges and opportunities. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016.

[5] Chia-Yu Chen, Jungwook Choi, Kailash Gopalakrishnan, Viji Srinivasan, and Swagath Venkataramani. Exploiting approximate computing for deep learning acceleration. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 821–826, 2018.

[6] Daichi Fujiki, Kiyo Ishii, Ikki Fujiwara, Hiroki Matsutani, Hideharu Amano, Henri Casanova, and Michihiro Koibuchi. High-Bandwidth Low-Latency Approximate Interconnection Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2017.

[7] Tadayoshi Hara and Toshihiro Hanawa. Offloading transprecision calculation using FPGA. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops*, HPCAsia 2022 Workshop, page 19–28, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Mustafa Karakoy, Orhan Kislal, Xulong Tang, Mahmut Taylan Kandemir, and Meenakshi Arunachalam. Architecture-aware approximate computing. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2), 2019.

[9] Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 48(4):62:1–62:33, 2016.

[10] Yoshiyuki Morie, Yasutaka Wada, Ryohei Kobayashi, and Ryuichi Sakamoto. Performance evaluation of data transfer API for rank level approximate computing on HPC systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 445–448, 2022.

[11] Hisashi Osawa and Yuko Hara-Azumi. Approximate data reuse-based accelerator design for embedded processor. *ACM Trans. Des. Autom. Electron. Syst.*, 24(5), August 2019.

[12] Yuuki Sato, Takanori Tsumura, Tomoaki Tsumura, and Yasuhiko Nakashima. An approximate computing stack based on computation reuse. In *2015 Third International Symposium on Computing and Networking (CANDAR)*, pages 378–384, 2015.

[13] Muhammad Shafique, Rehan Hafiz, Semeen Rehman, Walaa El-Harouni, and Jörg Henkel. Invited - Cross-layer approximate computing: From logic to architectures. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 1–6. Association for Computing Machinery, 2016.