Vol.2022-HPC-187 No.16
2022/12/2

Development of the data buffer holding time-series data across multiple applications

Jingde Zhou[1] Keiichiro Fukazawa[2] Takeshi Nanri[3]

Abstract: Cross-reference simulation is a calculation model coupling multiple parallel simulation codes when at least one simulation code needs to read the data in other simulation codes to do its own calculation. To provide a solution that can carry out cross-reference simulation efficiently and conveniently, a cross-reference simulation framework CoToCoA is being developed. It allows CoToCoA users to implement cross-reference simulation with minimal modification on simulation codes while the overhead is restricted as much as possible. In this paper, a function with a specific data buffer is developed to implement one-sided data communication among multiple simulation codes with minimal communication loss. Generally, a simulation code calculates and updates the data in each time step. Then the former data will be overwritten. If there is no synchronization or barrier, other simulation codes may miss data at some time step due to the overwriting. To solve this issue, a specific data buffer is utilized to save the data at each time step. No data will be skipped as long as the data buffer is not full. Meanwhile, the number of data communication calls can be reduced significantly.

1. Introduction

Cross-reference simulation is a calculation model combining multiple parallel simulation codes when some simulation codes need to read the data calculated by other simulation codes to do their own calculation. The data communication between different simulation codes is the most important part of a cross-reference simulation. In a large number of research fields, cross-reference simulation is used to study complicated phenomena involving multiple simulations which may have substantial differences in the spatial and temporal scales.

In some research fields like Solar-Terrestrial Physics (STP), many simulation codes are very sophisticated. In most cases, they are developed independently and individually. Also, different simulation codes may have substantial differences in the spatial and temporal scales. Therefore, the implementation and construction of different simulation codes are quite unlike and hard to understand for each simulation code developer. This makes it difficult to do the coupling unless the developers are in the same research team. In addition, the coupling overhead may be high when the coupling is implemented by traditional file-based data communication. Due to the difficulty of coupling between different simulations and the overhead, much research based on cross-reference simulation progresses tardily.

To execute cross-reference simulation efficiently, a cross-reference simulation framework called Code-To-Code-Adapter (CoToCoA) [1] is being developed based on Message Passing Interface (MPI) [2]. CoToCoA is a framework to connect a requester application to multiple worker applications through a coupler application. CoToCoA can be used to execute cross-reference simulation in an efficient and smooth way called strong cross-reference simulation. Different from other cross-reference simulation frameworks like preCICE[3], CoToCoA mainly focuses on the data communication between different simulation codes. The user only needs to add minimal modifications to the simulation codes to implement the data communication when other couplers may require several sophisticated settings. Therefore, CoToCoA users can easily couple the simulation codes which are developed by other developers.

In some kinds of cross-reference simulation, There is one main simulation code. In order to keep the efficiency and stability of the whole cross-reference simulation, the main simulation code's overhead brought by data communication should be as little as possible. One-sided communication is a technology that allows one process to read or write the memory of another process without its response. In CoToCoA, MPI Remote Memory Access (RMA) is used to implement one-sided communication. With MPI RMA, other simulation codes (usually executed as the workers) can directly read the data calculated by the main simulation code (usually executed as the requester). Then the main simulation code's overhead brought by data communication comes to zero theoretically. Also, one-sided communication is considered as a communication method having better performance than traditional two-sided communication since it does not need the response of the target process and the data movement caused by extra intermediate buffering.

When one-sided communication is utilized as the communication method in CoToCoA, a large amount of data may be lost due to the different execution speeds. An example is shown in **Fig .1**.

[1] Graduate School of Informatics, Kyoto University
[2] Academic Center of Computing and Media Studies, Kyoto University
[3] Research Institute for Information Technology, Kyushu University

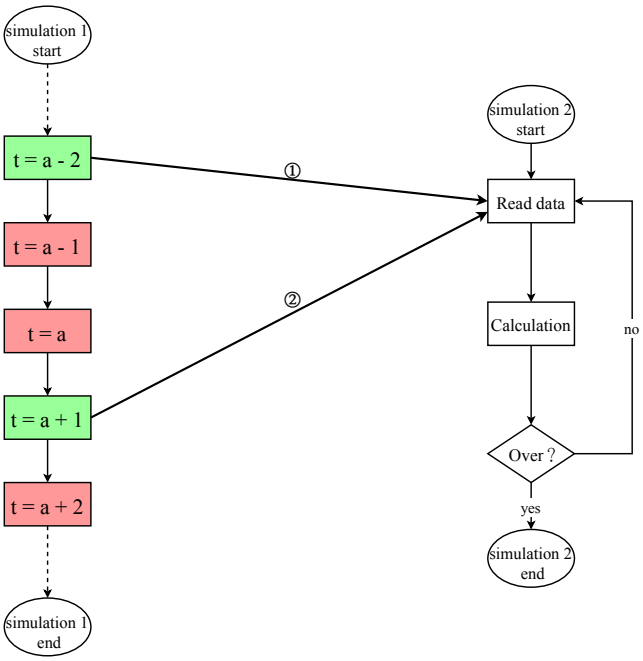

Fig. 1 Traditional one-sided communication in cross-reference simulation

At some point, simulation 2 reads the data at time step $a-2$ from Simulation 1 to do its calculation. Next time, simulation 2 reads the data at time step $a+1$. The data at time step $a-1$ and time step a are lost unavoidably.

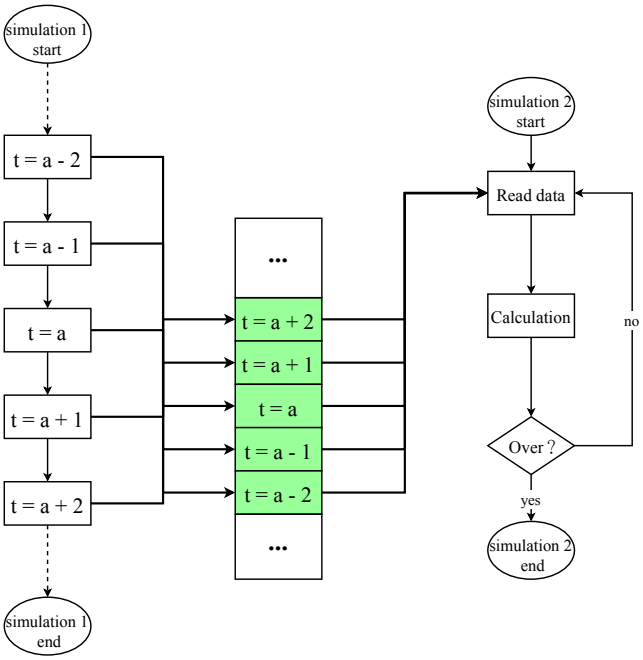

Fig. 2 One-sided communication with a specific data buffer

This paper develops a new CoToCoA function utilizing a specific data buffer to avoid frequent communication loss. Meanwhile, Each process of each worker can read a particular part of data (a line in one-dimension data, a rectangular in two-dimension data, a cuboid in three-dimension data) when CoToCoA user specifies the start position and end position of the data. In addition, each process of each worker reads multiple time steps data instead of only one time step data in one communication call.

The working principle of this function is shown in **Fig. 2**. A specific data buffer temporarily saves the calculated data, then the data will not be overwritten in the next time step. The workers read data from the data buffer. Then no data will be lost as long as the data buffer is not full.

The performance of this new function is evaluated on SQUID and ITO which belong to the Large-scale Computer System of Cybermedia Center, Osaka University[4], and the Research Institute for Information Technology, Kyushu University[5], respectively. The result of SQUID shows performance of the new function is better than ordinary one-sided communication without the specific data buffer. The result of ITO demonstrates the robustness of the new function.

2. Research objectives

2.1 Code-to-code-adapter (CoToCoA)

CoToCoA is a newly developed library for cross-reference simulation. CoToCoA treats every simulation code in a cross-reference simulation as an application and executes them simultaneously. Users can use it to couple multiple simulation codes with minimal modification to the codes while the overhead brought by the coupling is restricted as much as possible. At present, CoToCoA supports Fortran and C. MPI is the only fundamental communication layer at this point. A CoToCoA program can only be executed based on an MPI environment.

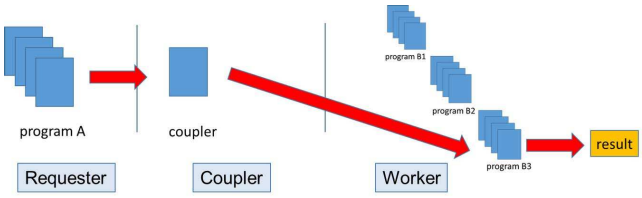

Fig. 3 Workflow of the CoToCoA program [6]

As shown in **Fig. 3**, there are three roles in a CoToCoA program. They are requester, coupler, and workers, respectively. The requester executes the main application, calculates data, and sends requests to the coupler. The coupler receives requests from the requester. It manages these requests and sends each request to the appropriate worker. The workers get requests from the coupler, read data from the requester or the coupler, and do their calculations. In CoToCoA, data communication is implemented by communication between memory, not file I/O.

CoToCoA can implement one-sided data communication. Compared to traditional two-sided communication, it needs neither the response of the target process nor the data movement caused by extra intermediate buffering. Moreover, some particular hardware can take charge of the one-sided data communication in some advanced supercomputer systems. In this case, both two sides can focus on their own calculation with minimal data communication overhead.

2.2 Research proposal

In previous research, CoToCoA has been used to carry out many cross-reference simulations. With the deepening of research, it is found that the data communication overhead of the

main simulation (executed as the requester) greatly limited the performance. Also, data communication may cause an imbalance in workload. In order to reduce the data communication overhead of the requester and the communication loss mentioned above, a new CoToCoA function is developed. In the new CoToCoA function, the workers utilize one-sided communication based on MPI RMA to directly read data from a specific data buffer of the requester. Theoretically, the data communication overhead of the requester will be zero and the communication loss will be largely reduced. Also, a convenient feature is added to this function. **Fig. 4** shows an example of this feature. In this example, there is a requester with nine processes and a worker with four processes. CoToCoA users can easily implement this many-processes non-contiguous data communication by specifying the start position and the end position of the data in each process of each worker. Generally, if a worker wants to read a particular part of data, the data communication codes need to be written carefully on both two sides. And it may need to modify the codes extensively when the computing resource allocation is changed. In conclusion, a new CoToCoA function is developed to efficiently implement one-sided data communication with minimal communication loss in a more user-friendly way.

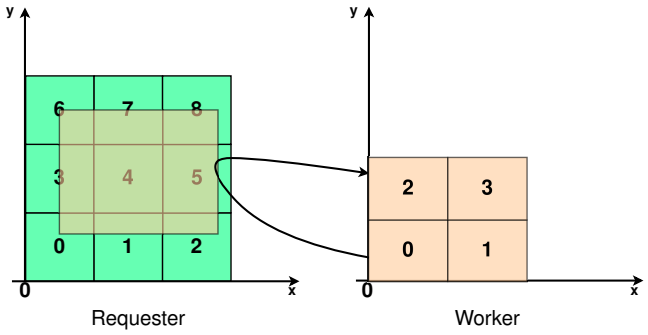

Fig. 4 A worker reads user-specified data from the requester

Four main properties of this new function are shown as follows.

- **CoToCoA users can transfer a user-specified part of data from the requester to the workers without complicated programming.** As shown in Fig. **4**, each worker's process may need to read different layouts of non-contiguous data from each requester's process. With the new function, each process of the workers will automatically figure out which part of data it should read from each process of the requester. The CoToCoA users do not need to pay much attention to the codes about the data communication.
- **Workers read the data in the data buffer with minimal communication calls.** To reduce the data communication overhead of the workers, communication calls should be minimized as much as possible. In this function, each process of the workers reads all useful data across many time steps from the data buffer with only one communication call.
- **CoToCoA Users can determine whether the requester waits for the workers or not when the data buffer is full.** In the new function, users can select the Execution Mode. In Execution Mode 1, the requester will not suspend when the data buffer is full. The requester only needs to copy the calculated data to the data buffer at the price of communication loss (Communication loss still happens when the data buffer is full). In Execution Mode 2, the requester will wait for the workers when the data buffer is full. No communication loss will happen but the requester may sometimes suspend. CoToCoA users can select the Execution Mode based on their requirements.
- **The data communication overhead brought to the requester is minimal.** Except for the data copy, the requester does not need to do anything about data communication unless Execution Mode 2 is selected. Usually, the data copy time is about a tenth of the data communication time.

3. Data buffer holding time-series data across multiple applications

3.1 Create time-series data buffer

In the new function, a specific data buffer is created in the memory of each process of the requester and read by the workers. This data buffer saves time-series data. The data at each time step is saved in one buffer unit.

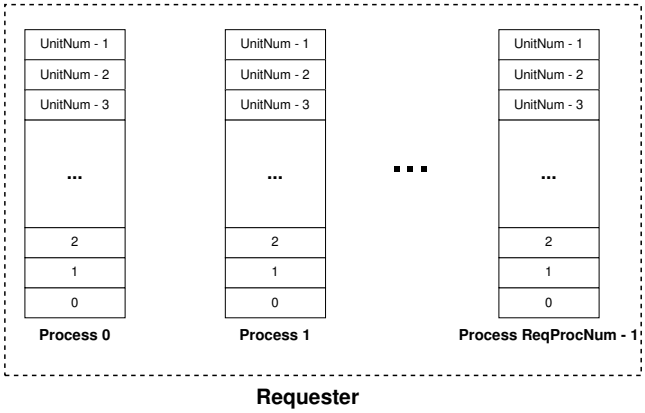

Fig. 5 Construction of the data buffer

Fig. 5 shows the construction of the data buffer, *UnitNum* is the number of buffer units, *ReqProcNum* is the number of processes of the requester. The value of *UnitNum* is determined by the user-specified buffer size and the size of data at each time step. The requester sequentially saves calculated data in the data buffer after it does its calculation in each time step. If the data buffer is full, the new data will overwrite the oldest data.

In this function, two routines are used to initialize the data buffer.

- **CTCAR_buffer_init_int(int** *BufferSize*, **int** *DimensionSize*, **int*** *StartPosition*, **int*** *EndPosition*)[*1]
  Where *BufferSize* is the size of the data buffer, *DimensionSize* is the dimension of data, *StartPosition* and *EndPosition* are the start position and the end position of the data among all processes of the requester. This routine is called by all processes of the requester.

[*1] These routines have two variants to handle different data type. For float data, users should call CTCAR_buffer_init_real4. For double data, users should call CTCAR_buffer_init_real8. Other routines in the new function also have these two variants.

- **CTCAW_buffer_init_int(int*** *StartPosition*, **int*** *EndPosition*, **int** *ExecutionMode*)
  Where *StartPosition* and *EndPosition* are the start position and the end position of the user-specified data. *ExecutionMode* determines the Execution Mode of this function. This routine is called by all processes of the workers.

3.2 Create derived datatype

In this new function, the non-contiguous data communication is implemented by MPI's derived datatype. Two derived datatype array is created in each process of the workers. $Datatype\_req_{i,j}$ denotes the layout of data in the data buffer for the data communication between the requester's process i and this worker's process j. $Datatype\_wrk_{i,j}$ denotes the layout of data in the receive buffer for requester's process i and worker's process j.

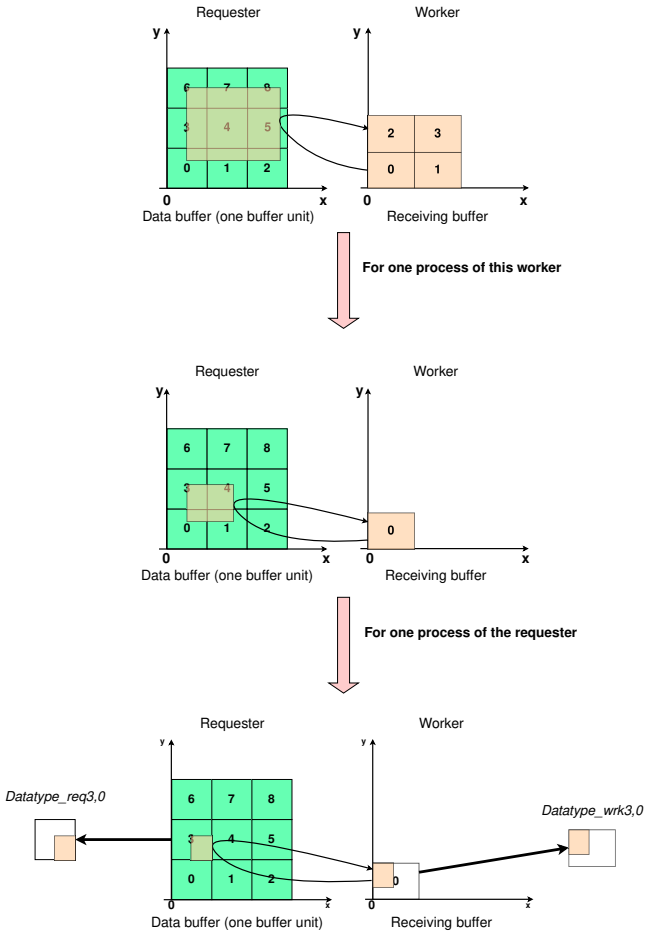

Fig. 6 The use of MPI's derived datatype

Fig. 6 is an example illustrating the layout of the derived datatype. In this example, there is only one worker. There are nine processes of the requester and four processes of the worker. Each process of the requester is in charge of one part of the calculation. For each process of the worker, it needs to read different layouts of data from different process of the requester. For example, the process 0 of the worker reads $Datatype\_req_{3,0}$ layout of data from the data buffer in process 3 of the requester, and saves it with $Datatype\_wrk_{3,0}$ layout of data in its receiving buffer.

3.3 Mechanism of counter

Counters are used in the first process of the requester and each worker. they are used to control the execution of both the requester and the workers. There are two Execution Mode existing in this function.

In **Fig .7** and **Fig .8**, $TotalTimestep$ denotes the number of total time step of the requester's simulation code, $ThisWrk$ is the serial number of this worker, *ReqProcNum* presents the number of processes of the requester, *WrkNum* denotes the number of the workers. *Counter_Req_r* is an integer in the requester. It indicates the current time step of the requester. $Counter\_Req\_w_{0,1...WrkNum-1}$ is an integer array in the requester. It indicates the current time step of each worker. *Counter_Wrk_r* is an integer in each worker. It indicates the current time step of the requester. *Counter_Wrk_w* is an integer in each worker. It indicates the current time step of this worker. use_i shows if this process of worker needs to read data from process i of the requester.

Fig. 7 shows the program flow of Execution Mode 1. In Execution Mode 1, the requester can copy data to the data buffer only after the data buffer is not full. For each worker, it reads data from the data buffer as long as there is still useful data in the data buffer. This worker reads *Counter_Wrk_r* - *Counter_Wrk_w* timesteps data every time. However, the *Counter_Wrk_r* may be updated by the requester during the data communication. To avoid mistaken data communication, $tmp1$ is used to temporarily save the value of *Counter_Wrk_r*. The worker reads $tmp1$ - *Counter_Wrk_w* time steps data instead of *Counter_Wrk_r* - *Counter_Wrk_w* time steps data.

Fig. 8 shows the program flow of Execution Mode 2. In Execution Mode 2, the requester copy calculated data to the data buffer after each time step without any suspension. Different from Execution Mode 1, *Counter_Wrk_w* may be discontinuous. a new variant $tmp2$ is used to record the starting time step. the worker reads $tmp1 - tmp2$ time steps data instead of $tmp1 - Counter\_Wrk\_w$ timesteps data.

3.4 Read data from requester

Two main routines are provided to implement the data communication.

- **CTCAR_buffer_loaddata_int(int*** *address*)
  Where *address* is the address of calculated data. This routine should be called by each process of the requester after the calculation is done in each time step.
- **CTCAW_buffer_readdata_int(int*** *address*)
  Where *address* is the address of the receiving buffer. This routine should be called by each process of the workers after the calculation is done in each time step.[*2]

Also, **CTCAR_buffer_free** (called by the requester) and **CTCAW_buffer_free** (called by the workers) should be called after the use of data buffer is finished.

To minimize the data communication overhead of the workers, the number of communication calls should be reduced as much as possible. In the new function, the workers can read data across multiple time steps by only one (sometimes two) communication

[*2] The call of this routine does not directly mean the data communication is carried out.

```
the requester:
    initialize:
        Counter_Req_r = -1
        Counter_Req_w_{0,1...WrkNum-1} = -1

    for j in 0,1...TotalTiemstep - 1
        do calculation
        Counter_Req_r++

        while (Counter_Req_r > UnitNum +
        min(Counter_Req_w_0, Counter_Req_w_1,
        ..., Counter_Req_w_{WrkNum-1})):
            the requester waits for the workers

        copy the calculated data to the data buffer
        for i in 0,1...WrkNum - 1:
            (Counter_Wrk_r in worker i) = Counter_Req_r

requester end

Each worker:
    initialize:
        Counter_Wrk_r = -1
        Counter_Wrk_w = -1
        for i in 0,1...ReqProcNum - 1:
            if (this process needs to read data
            from the requester's process i):
                use_i = 1
            else:
                use_i = 0

    while there is still useful data in the data buffer:
        while Counter_Wrk_r <= Counter_Wrk_w:
            this worker waits for the requester

        tmp1 = Counter_Wrk_r
        for i in 0,1...ReqProcNum - 1:
            if use_i = 1:
                Read tmp1 - Counter_Wrk_w time steps data
                from data buffer by Datatype_req_{i,ThisWrk}
                and Datatype_wrk_{i,ThisWrk}

        Counter_Wrk_w = tmp1
        (Counter_Req_w_{ThisWrk} in the requester) = Counter_Wrk_w
        do calculation

this worker end
```

Fig. 7 Algorithm of the counter in Execution Mode 1

```
the requester:
    initialize:
        Counter_Req_r = -1

    for j in 0,1...totalTiemstep - 1:
        do calculation
        Counter_Req_r++

        copy the result data to the data buffer

        for i in 0,1...WrkNum - 1:
            (Counter_Wrk_r in worker i) = Counter_Req_r

the requester end

Each worker:
    initialize:
        Counter_Wrk_r = -1
        Counter_Wrk_w = -1

        for i in 0,1...ReqProcNum - 1:
            if (this process needs to read data
            from the requester's process i):
                use_i = 1
            else:
                use_i = 0

    While there is still useful data in the data buffer:
        while Counter_Wrk_r <= Counter_Wrk_w:
            this worker waits for the requester

        tmp1 = Counter_Wrk_r
        if tmp1 > UnitNum + Counter_Wrk_w:
            tmp2 = tmp1 - UnitNum
        else:
            tmp2 = Counter_Wrk_w

        for i in 0,1...ReqProcNum - 1:
            if use_i = 1:
                Read tmp1 - tmp2 time steps data
                from data buffer by Datatype_req_{i,ThisWrk}
                and Datatype_wrk_{i,ThisWrk}

        Counter_Wrk_w = tmp1
        do calculation

this worker end
```

Fig. 8 Algorithm of the counter in Execution Mode 2

call.

- **Both in Execution Mode 1 and 2**
  In general, workers read data from the data buffer by only one time as shown in **Fig .9**.
  When a worker's process reads data across the boundary of the data buffer, it needs to read two times. An example is shown in **Fig .10**.
- **Extra issue only in Execution Mode 2**
  In Execution Mode 2, some data may not be read by the workers because of the imbalanced execution speed. A Co-ToCoA routine **CTCAW_get_timestep** is provided to get the exact time step of current data.

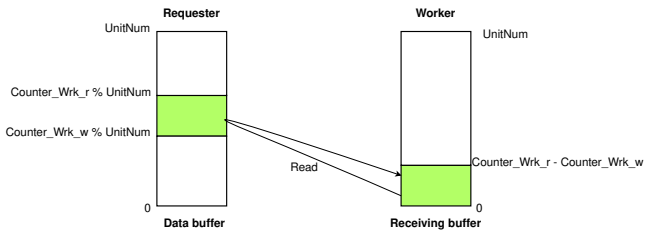

Fig. 9 Worker reads data from data buffer (situation 1)

The workers may read the data mixed with different time steps due to the overwriting. An example is as shown in **Fig .11**. During the data communication,

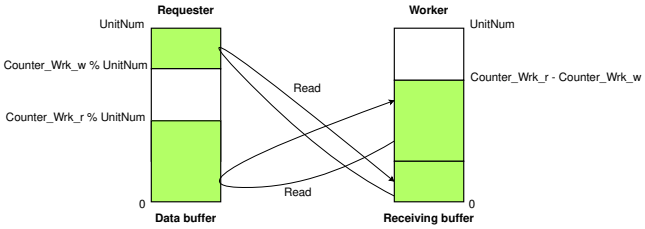

Fig. 10 Worker reads data from data buffer (situation 2)

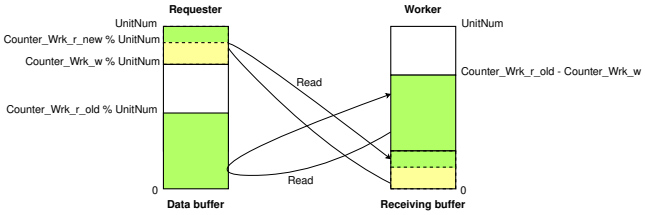

Fig. 11 Worker reads data from data buffer (situation 3)

Counter_Wrk_r is increased from *Counter_Wrk_r_old* to *Counter_Wrk_r_new*. The received data from *Counter_Wrk_w* to *Counter_Wrk_r_new*, which is the yellow part in the figure, may be mixed with different time steps data unavoidably. Though overwriting is inevitable, CoToCoA users can use **CTCAW_get_overwrite_flag** to distinguish if the current time step data is mixed or not.

Fig. 12 shows an example of the new function. For the worker, **CTCAW_buffer_readdata_int** will return a value to indicate if useful data still exists. The routine **CTCAR_get_address** returns the address of current time step data.

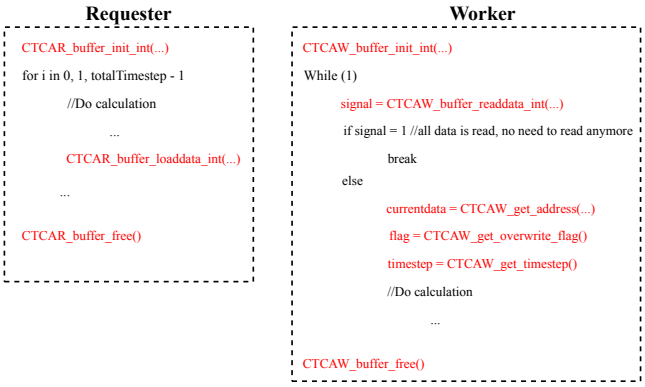

Fig. 12 An example of the new function

4. Evaluation results

4.1 Evaluation on VE

An evaluation is implemented to measure the performance of three different data communication methods on the Vector Engine (VE) of SQUID. They are traditional two-sided communication, ordinary one-sided communication without data buffer[*3] and the new function. In this evaluation, data communication is between two processes distributed in different VEs. The $BufferSize$ is 4GB. The total timestep is $100N$. Process 1 calculates 900×900 2-dimension integer data array in each timestep. Meanwhile, Process 2 reads 600×600 2-dimension integer data array from Coordinates $(150, 150)$ to Coordinates $(750, 750)$ in the data buffer until there is no useful data in the data buffer. This program is executed in Execution Mode 1. VE's specification is shown in **table 1**.

Table 1 Specification of VE in SQUID

Machine	NEC SX-Aurora TSUBASA B401-8 x36
Vector engine (VE)	
	System
Number of Nodes	288
Total Number of Cores	2,880
Total Amount of Memory	13,824 GB
Total Theoretical Peak Performance	0.922 PFLOPS
Interconnect	Mellanox InfiniBand HDR (200 Gbps)
	Node
Processor	NEC SX-Aurora TSUBASA Type20A
Theoretical Peak Performance	307 GFlops / core × 10 cores
Memory	48 GB

The evaluation result on VE of SQUID is as shown in **Table 2** and **Fig .13**. The percentage value attached to the data communication time is the ratio to the one-sided communication time without data buffer. According to the evaluation result, when $N < 7$, the new function is 12.6% faster than ordinary one-sided communication without data buffer. However, when $N >= 7$, one-sided communication with data buffer is 4.5% slower than one-sided communication without data buffer. The performance of the new function decreased rapidly. The reason may be that there is a limitation in one one-sided communication call. It is broken when $N >= 7$. Then the data communication is carried out in an implicit and inefficient way.

Different from one-sided communication, the requester in two-sided communication needs to handle the data communication during the data communication occurs. The requester in one-sided communication can do its own calculation in the saved time.

4.2 Evaluation on ITO-A

An evaluation is implemented on ITO-A to evaluate the performance of the new function when a large number of nodes are used. ITO-A's specification is shown in **table 3**. All 2,000 nodes of ITO-A are available. In this evaluation, a CoToCoA program is used to test the robustness of the new function. There is only one worker in this evaluation. The number of the requester's core is $4N$ when the number of the worker's core is N. Each requester's core generates 100×100 2-dimension int data array in one time step. Then, One worker'core reads data from 4 cores of the requester and saves the data in a 200×200 2-dimension integer data array. The $BufferSize$ is 4GB. The total timestep is 1000. This program is executed in Execution Mode 1. The detail of this evaluation is shown in **Table 4**.[*4]

The evaluation result is shown in **Table 5** and **Fig .14**. The data communication time shown in the table is the average data communication time of all worker's cores. When N value is fixed, the data communication time increases slowly following the increase of the number of nodes. When all 2000 nodes are used,

[*3] The main purpose of this evaluation is to compare the data communication time of three different communication methods. A barrier is used to prevent communication loss. (the extra waiting time is not included in the result)

[*4] In this evaluation, each node uses the same number of cores to do the execution.

Table 2 Data communication time (ms) on VE (Table)

N value	1	2	3	4	5
two-sided	12.2(94.3%)	24.4(94.3%)	36.7(94.4%)	48.9(94.4%)	61.1(94.2%)
one-sided without data buffer	13.0(100%)	25.9(100%)	38.8(100%)	51.7(100%)	64.8(100%)
one-sided with data buffer	11.3(87.4%)	22.6(87.3%)	33.9(87.4%)	45.3(87.5%)	56.6(87.3%)
N value	6	7	8	9	10
two-sided	73.3(94.4%)	85.5(94.3%)	97.7(94.4%)	109.9(94.3%)	122.2(94.4%)
one-sided without data buffer	77.6(100%)	90.6(100%)	103.5(100%)	116.5(100%)	129.5(100%)
one-sided with data buffer	68.0(87.6%)	94.8(104.5%)	108.2(104.6%)	121.8(104.5%)	135.3(104.5%)

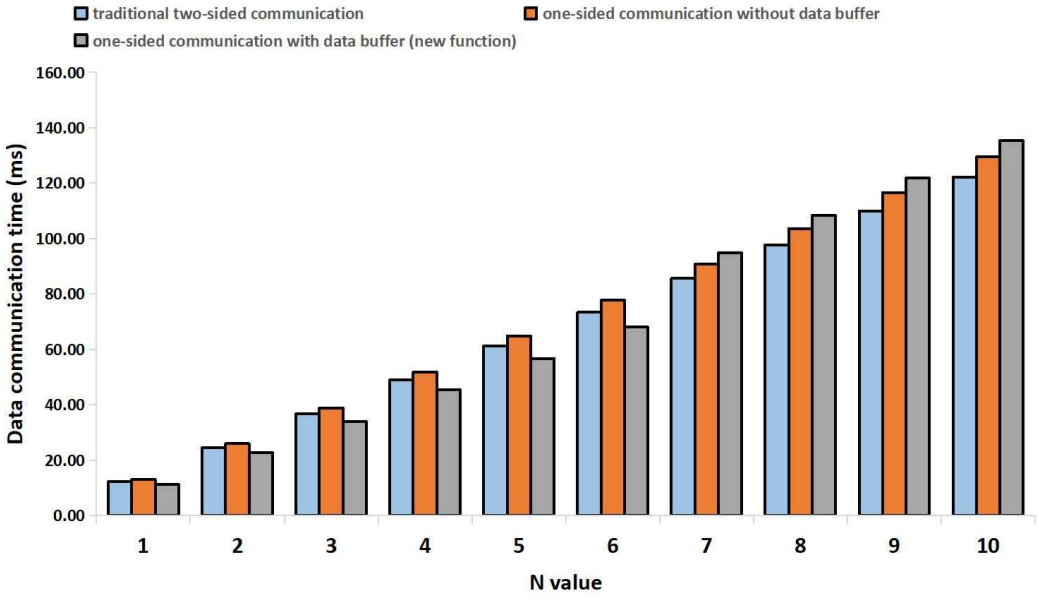

Fig. 13 Data communication time (ms) on VE (Histogram)

Table 3 Specification of ITO-A

Machine	Fujitsu PRIMERGY CX2550/CX2560 M4
	System
Number of Nodes	2,000
Total Number of Cores	72,000
Total Amount of Memory	384 TB
Total Theoretical Peak Performance	6.91 Pflops
Interconnect	InfiniBand EDR 4x (100 Gbps)
	Node
Processor	Intel Xeon Gold 6154 (Skylake-SP) 3.0 GHz (Turbo 3.7 GHz) 18 core x 2 / node
Theoretical Peak Performance	3,456 Gflops
Memory	DDR4 192 GB
Memory Bandwidth	255.9 GB/sec

Table 4 Experiment setting on ITO-A

N value	Nodes number	Array size	Data size	Timestep
100	16 32 64 128	$4 \times 100 \times 100$	160 KB	1000
1600	256 512 1024 2000	$4 \times 100 \times 100$	160 KB	1000

Table 5 Data communication time (ms) on ITO-A (Table)

N value	Nodes number	Data communication time (ms)
100	16	437
	32	471
	64	478
	128	488
1600	256	452
	512	471
	1024	495
	2000	663

the data communication time rises sharply. An assumption is that the enormous data communication may reach the limitation of the hardware.

5. Related Work

Other than CoToCoA, several frameworks are also used to deal with cross-reference simulation. preCICE[3] is an open-source coupling library for partitioned multi-physics simulations. It is mainly used to deal with fluid-structure interaction and conjugate heat transfer simulations. preCICE needs to be configured at runtime via an xml file that specifies the pattern of mapping, communication, coupling scheme, acceleration, and mesh exchange. The xml file may be very complicated when the simulation codes are difficult. This creates difficulties for the developers. The Multiphysics Object Oriented Simulation Environment (MOOSE) framework[7] is a high-performance, open source, C++ finite element toolkit developed at Idaho National Laboratory. While the core MOOSE framework itself does not contain code for simulating any particular physical application, it is distributed with a number of physics "modules" which are tailored to solving e.g. heat conduction, phase field, and solid/fluid

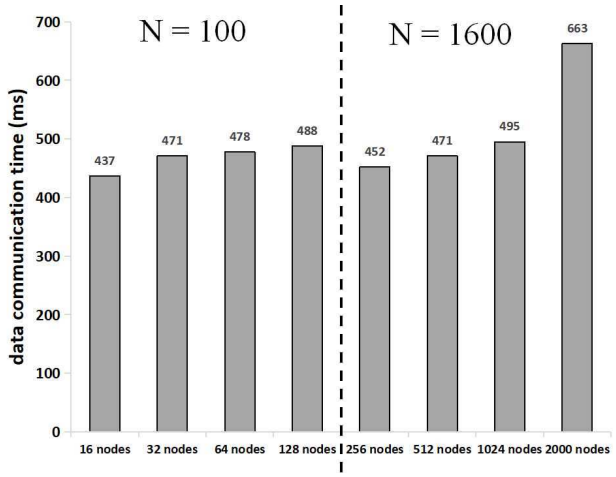

Fig. 14 Data communication time (ms) on ITO-A (Histogram)

mechanics problems. It is mainly developed to deal with many specific cross-reference simulations. Many developers have used MOOSE to contribute in many fields.

Several studies have been carried out to evaluate the performance of one-sided communication. Ichitaro Yamazaki et al.[8] tested the asynchronous optimized Schwarz domain-decomposition iterative method using various one-sided (remote direct memory access) communication schemes with passive target completion. Their experimental results suggested that the performance of the asynchronous solver depends heavily on the software and hardware support for remote memory access communication. In some situation, asynchronous solver may outperform the synchronous solver. Weihang Jiang et al.[9] proposed a design of MPI-2 one-sided communication over InfiniBand. Through performance evaluation, they found that their design can achieve lower overhead and higher communication performance. Moreover, experimental results have shown that the RMA based approach allows for better overlap between computation and communication. It also achieves better scalability with multiple number of origin processes.

In addition, William Gropp et al.[10] evaluated the performance of MPI's derived datatype. According to the experimental result, using datatype to do the data communication directly is generally faster than combining packing and sending manually.

6. Conclusion

A CoToCoA function is newly developed to implement one-sided communication between the requester and the workers. This function utilizes a data buffer in each requester's process. In each time step, the requester saves the result data in the data buffer after calculation. The size of the data buffer can be specified by CoToCoA user. When the size of the data buffer is large enough, the data buffer can save multiple time steps data. On the other hand, the worker reads target data from the data buffer to do its calculation. In this way, little data will be lost due to the overwriting of data. In addition, the worker is able to read multiple time steps data in the data buffer at one time. The reduction of one-sided communication calls results in the reduction of data communication time. Moreover, the worker reads data from the requester in a one-sided way. Therefore, the requester does not need to cope with the data communication when it is doing its calculation. According to the evaluation result, the advantage of the new function is significant.

References

[1] Keiichiro Fukazawa, Yuto Katoh, Takeshi Nanri, and Yohei Miyake. Application of cross-reference framework cotocoa to macro- and micro-scale simulations of planetary magnetospheres. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pages 121–124, 2019.

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2022.

[3] Hans-Joachim Bungartz, Florian Lindner, Bernhard Gatzhammer, Miriam Mehl, Klaudius Scheufele, Alexander Shukaev, and Benjamin Uekermann. precice–a fully parallel library for multi-physics surface coupling. *Computers & Fluids*, 141:250–258, 2016.

[4] Cybermedia Center, Osaka University. *SQUID: Large-Scale Computer System in Osaka University*, June 2022.

[5] Research Institute for Information Technology, Kyushu University. *ITO: Supercomputer system in Osaka University*, June 2022.

[6] Keiichiro Fukazawa, Yuto Katoh, Takeshi Nanri, and Yohei Miyake. Application of cross-reference framework cotocoa to macro-and micro-scale simulations of planetary magnetospheres. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pages 121–124. IEEE, 2019.

[7] Derek R. Gaston, Cody J. Permann, John W. Peterson, Andrew E. Slaughter, David Andrš, Yaqi Wang, Michael P. Short, Danielle M. Perez, Michael R. Tonks, Javier Ortensi, Ling Zou, and Richard C. Martineau. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45–54, 2015. Multi-Physics Modelling of LWR Static and Transient Behaviour.

[8] Ichitaro Yamazaki, Edmond Chow, Aurelien Bouteiller, and Jack Dongarra. Performance of asynchronous optimized schwarz with one-sided communication. *Parallel Computing*, 86:66–81, 2019.

[9] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K Panda, William Gropp, and Rajeev Thakur. High performance mpi-2 one-sided communication over infiniband. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pages 531–538. IEEE, 2004.

[10] William Gropp and Rajeev Thakur. An evaluation of implementation options for mpi one-sided communication. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 415–424. Springer, 2005.