Implementing A Parallel NetCDF Interface for Seamless Remote I/O Using Multi-Dimensional Data

Yuichi Tsujita

Department of Electronic Engineering and Computer Science,
School of Engineering, Kinki University
1 Umenobe, Takaya, Higashi-Hiroshima, Hiroshima 739-2116, Japan
Phone +81-82-434-7000, FAX +81-82-434-7011 tsujita@hiro.kindai.ac.jp

Abstract. Parallel netCDF supports parallel I/O operations for a view of data as a collection of self-describing, portable, and array-oriented objects that can be accessed through a simple interface. Its parallel I/O operations are realized with the help of an MPI-I/O library. However, such the operations are not available in remote I/O operations. So, a remote I/O mechanism of a Stampi library was introduced in an MPI layer of the parallel netCDF to realize such the operations. This system was evaluated on two interconnected PC clusters, and sufficient performance was achieved with a huge amount of data.

Corresponding topics: Parallel and Distributed Computing, Cluster Computing

1 Introduction

Recent parallel scientific computations require not only a huge amount of computing power but also a huge amount of data storages. Scientific computations usually output intermediate data for check-point restart or analysis by using a visualization software after computation. In such the computation, common portable data format and I/O interfaces are very useful because users want to concentrate in their computations.

Several kinds of I/O interfaces such as netCDF [1] support such data format and simple I/O interface. NetCDF was developed to support a view of data as a collection of self-describing, portable, and array-oriented objects that can be accessed through a simple interface. It provides a portable I/O interface which supports not only fixed size arrays but also variable size arrays. It has been widely used in many kinds of scientific computations such as meteorology.

NetCDF is a useful interface library, however, it only supports serialized I/O operations. As a result, such I/O operations would be a bottleneck in parallel computation. A parallel I/O interface named parallel netCDF (hereafter PnetCDF) was developed in order to realize effective parallel I/O operations for netCDF data with the help of an MPI-I/O library [2] such as ROMIO [3]. It succeeded in scientific computation [4] and several visualization softwares support

its data format, however, the same operations among computers have not been available. Seamless remote I/O is useful for a user's client application such as a parallelized visualization software. A remote MPI-I/O mechanism of a Stampi library [5] has been introduced in a PnetCDF's MPI layer in order to realize this mechanism. The MPI-I/O mechanism supports automatic selection of local and remote I/O operations based on a target computer name which is specified in an `MPI_Info` object by an MPI program. MPI functions of a PnetCDF library have been replaced with the Stampi's MPI functions to support seamless remote I/O operations through a PnetCDF interface without paying attention to complexity and heterogeneity in underlying communication and I/O systems. In this paper, architecture and execution mechanism of this system are discussed in Section 2. Performance results are reported in Section 3. Related work is remarked in Section 4, followed by conclusions in Section 5.

2 Remote I/O Through A PnetCDF Interface

In this section, we describe decompositions of multi-dimensional data, a derived data type associated with the decompositions, and a remote I/O system with such data type.

2.1 Decompositions of multi-dimensional data and associated derived data types

In computer simulations, multi-dimensional data are frequently used to store calculated results, for example. In this section, decompositions of n-dimensional data sets and associated derived data types are discussed. We denote lengths of each axis with index of $1, 2, \ldots, n$ as $L_1, L_2, \ldots, L_n$, respectively. In the data sets, we suppose an array data in a C program. Let us assume that index of 1 is the most inner index and stands for the most significant dimension. On the other hand, index of n is the most outer index and stands for the least significant dimension.

Decompositions along the most inner and outer indexes make derived data types as shown in Figures 1 (a) and (b), respectively. In Fig. 1 (a), each user process accesses dotted non-contiguous data fields with L_1/np for a block length and $rank \times L_1/np$ for a stride length, where np and $rank$ stand for the number of user processes and an unique ID in an MPI program, respectively. Each offset is specified not to overwrite the data fields each other. On the other hand, Fig. 1 (b) shows a simple derived data type split evenly along the most outer index. It is clear that splitting evenly along the most significant axis provides the most complex data image and that along the least significant axis provides the most simplest one. This kind of data type is easily created by MPI functions for derived data types such as `MPI_Type_vector()`. In a PnetCDF interface, several kinds of MPI functions are used to create such derived data types. This issue is discussed in the next section.

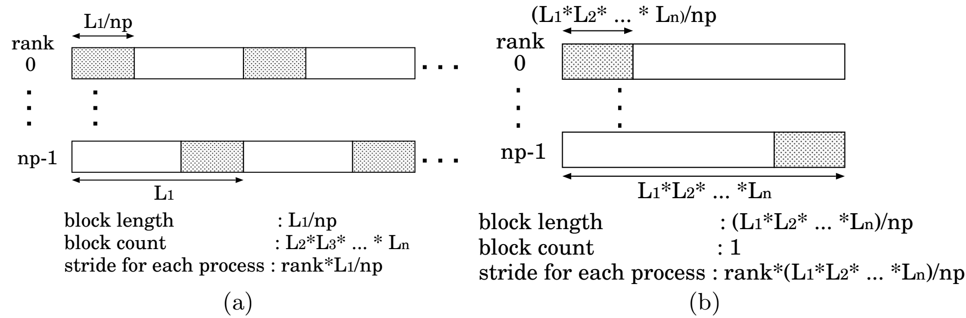


Fig. 1. Derived data types split evenly along (a) the most inner and (b) the most outer indexes for n user processes

2.2 Support of PnetCDF in remote I/O

PnetCDF supports many kinds of parallel I/O interface functions, and their parallel I/O mechanism is realized by using MPI functions inside them. As an example, several PnetCDF functions and used MPI functions inside them are listed in Table 1. In the PnetCDF, a native MPI library such as MPICH is

Table 1. Typical PnetCDF functions and MPI functions which are called inside them

PnetCDF functions	Used MPI functions
`ncmpi_create()`, `ncmpi_open()`	`MPI_File_open()`, `MPI_File_delete()`, etc.
`ncmpi_put_var_int()`	`MPI_Comm_rank()`, `MPI_File_set_view()`, `MPI_Type_hvector()`, `MPI_Type_commit()`, `MPI_Type_free()`, `MPI_File_write()`, etc.
`ncmpi_put_vars_int_all()`	`MPI_Comm_rank()`, `MPI_File_set_view()`, `MPI_Type_hvector()`, `MPI_Type_commit()`, `MPI_Type_free()`, `MPI_File_write_all()`, etc.
`ncmpi_close()`	`MPI_Allreduce()`, `MPI_File_close()`, etc.

used. Parallel I/O operations inside the same MPI implementations are available, however, remote I/O operations are not. As Stampi supports MPI functions listed in the table, we have replaced native MPI functions with Stampi's MPI functions which start with `_MPI_` in order to develop the remote I/O system. As the functions switch to local or remote I/O operations based on a destination computer automatically, seamless I/O operations are available in the PnetCDF layer. In the local I/O, portable interface functions of a native MPI library which start with `PMPI_` are called inside the Stampi layer. While in the remote I/O, an I/O request and associated parameters such as message data size and a data type are transfered to a corresponding remote MPI-I/O process. The MPI-I/O

process plays requested I/O operations. Details of this mechanism are discussed in 2.5.

2.3 Architecture of a remote I/O mechanism

Architecture of the I/O mechanism is depicted in Figure 2. MPI communica-

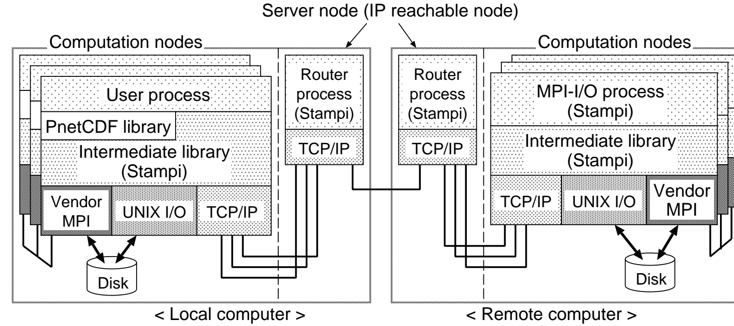


Fig. 2. Architecture of a remote I/O mechanism with PnetCDF interface support

tions inside a computer are carried out by using a native MPI library. When a PnetCDF interface is called for I/O operations inside a computer, associated Stampi's MPI interface functions are called, and high performance I/O operations are carried out by calling a native MPI library by the Stampi's functions. If the native one is not available, UNIX I/O functions are used instead of it.

While in MPI communications among computers, user's MPI processes invoke MPI processes on a remote computer by using rsh or ssh when a spawn function (`MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`) is called. MPI communications between the local and remote MPI processes are carried out via inter-connections established by TCP sockets. If computation nodes of a computer can not communicate outside directly, a router process is invoked on an IP reachable node to relay message data among computers. For remote I/O operations, an MPI-I/O processes are invoked on a remote computer to play I/O operations instead of the MPI processes when an MPI function to open such as `MPI_File_open()` is called. In I/O operations by the MPI-I/O processes, a native MPI library is used via a Stampi's MPI interface library as default. If the native one does not support MPI-I/O operations, UNIX I/O functions are used instead of it.

2.4 Execution mechanism

An execution mechanism of the remote I/O operations is illustrated in Figure 3. Firstly, a user issues a Stampi's start-up command to initiate a Stampi

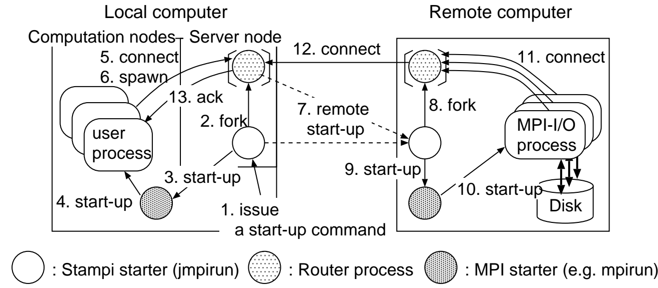


Fig. 3. Execution mechanism of remote I/O operations

starter (jmpirun) (**1**). The starter process invokes a native MPI start-up process (MPI starter) such as mpirun (**3**). Later the MPI starter process invokes user's MPI processes (**4**). For remote I/O operations, parameters such as a hostname of a target computer, a user ID, and a working directory are specified in an `MPI_Info` object. When `ncmpi_create()` (for creating a new file) or `ncmpi_open()` (for opening an existing file) is called by the user processes, a Stampi's `MPI_File_open()` is called by the PnetCDF function. After this operation, MPI-I/O processes are invoked on a remote computer according to the parameters in the `MPI_Info` object (**5**-**7**, **9**-**12**). If computation nodes where the MPI processes or MPI-I/O processes are running can not communicate outside directly, a router process is invoked on a server node by the Stampi starter (**2**, **8**). Once a communication path is established among them, remote I/O operations are available by sending I/O requests from the user processes to the MPI-I/O processes. After I/O operations, `ncmpi_close()` is called in the user processes to close a opened file, followed by calling `MPI_File_close()` to close the file and terminating the MPI-I/O processes inside it.

2.5 Execution steps of PnetCDF functions

I/O operations with a derived data type are essential mechanisms to support PnetCDF. Execution steps of remote I/O operations using `ncmpi_put_vars_int_all()` are explained as an example. Execution steps of I/O operations using the function are illustrated in Figure 4. Before I/O operations, we need to specify several parameters associated with the operations by using other PnetCDF and MPI functions. Inside each PnetCDF function, several MPI functions are used as listed in Table 1. Operations of this write function are grouped into two parts: one is for creation of a derived data type, and other for I/O operations by using the data type. The former part is carried out by using `MPI_Type_hvector()`, `MPI_Type_commit()`, `MPI_File_set_view()`, and so on. While the latter one is carried out by using `MPI_File_write_all()`. Execution steps for the former and the latter operations are depicted in Figures 5(a) and (b), respectively. Firstly, a derived data type is created by `MPI_Type_hvector()` and `MPI_Type_commit()` as shown in Fig. 5(a). After a file view is created by `MPI_File_set_view()`, several

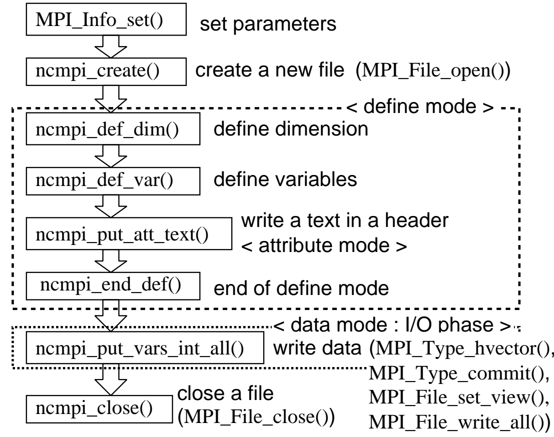
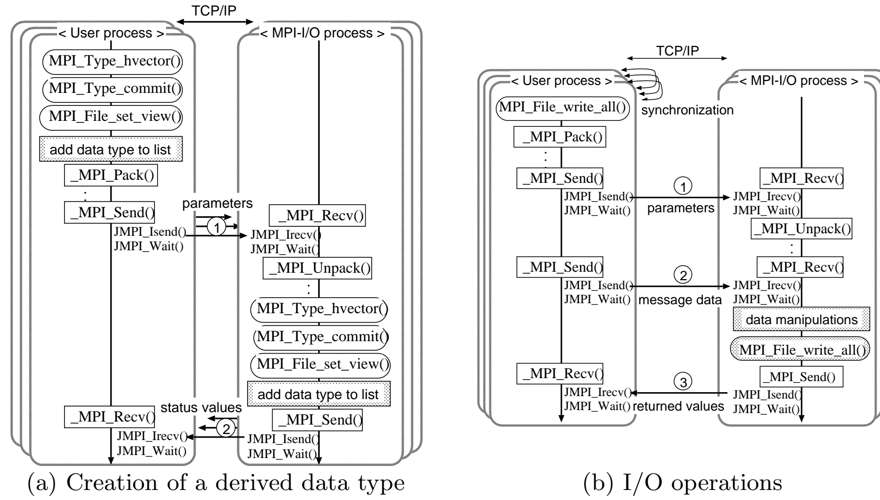


Fig. 4. Execution steps of typical collective write operations

parameters such as a unit data type, a block length, a stride length, and so on are stored in a list-based table provided by a Stampi library in each user process. A request of the function and the parameters are transfered to a corresponding MPI-I/O process by calling a Stampi's function which starts with `_MPI_`. As the data transfer is carried out by nonblocking TCP socket connections, overlap of computation by user processes and the data transfer would be expected. After each MPI-I/O process receives them, the same derived data type is created using the same functions, and they are stored in the similar table provided by a Stampi library in each MPI-I/O process. Each process returns a status value to a corresponding user process by using the Stampi's function at the final step.

Once we succeed to make a derived data type, we can start I/O operations using it as shown in Fig. 5(b). Associated MPI functions such as `MPI_File_write_all()` are called in an MPI layer of a PnetCDF interface. I/O requests of each MPI functions and associated parameters are transfered to a corresponding MPI-I/O process, and message data are also transfered later. Once each MPI-I/O process receives them, it plays I/O operations by using the derived data type. When the I/O operations finish, each MPI-I/O process returns a status value to the corresponding user process.

Concerning a derived data type, this system operates reordering striped data across I/O nodes by using intra-computer communications as shown in Figure 6. Each rectangle stands for an assigned memory buffer provided by this system. This mechanism has been adopted in order to reduce performance degradation by derived data type creation because times for reordering striped data with intra-computer communications are quite shorter than those with inter-computer communications.



(a) Creation of a derived data type (b) I/O operations

Fig. 5. Execution steps of (a) derived data type creation and (b) write operations. Functions in rectangles which start with `_MPI_` are Stampi's MPI interface functions.

3 Performance Evaluation

This system was evaluated among two PC clusters which were connected via 1 Gbps Ethernet. Specifications of the clusters are listed in Table 2. Each cluster had one server PC node and four computation PC nodes. Interconnection between the PC nodes was established via Gigabit Ethernet switches. In the both clusters, MPICH [6] (version 1.2.7p1) was available as a native MPI library. PnetCDF version 0.9.4 was used in this system. PVFS2 [7] (version 1.4.0) was available in the PC cluster II by collecting disk spaces of four computation nodes. Network connections between the clusters were established by connecting the both switches via a FreeBSD PC node which acted as a gateway.

In performance measurement, user processes were initiated on computation nodes of the cluster I, and remote I/O operations to the PVFS2 file system were carried out by invoking the same number of MPI-I/O processes on computation nodes of the cluster II. The notation, *np* also stands for the number of MPI-I/O processes in remote I/O operations. In this test, we used three-dimensional data sets. The following four different message data sets were prepared by using an integer data type (NC_INT):

- $16 \times 16 \times 16$ (16 Kbyte)
- $64 \times 64 \times 64$ (1 Mbyte)
- $128 \times 128 \times 128$ (8 Mbyte)
- $256 \times 256 \times 256$ (64 Mbyte)

Firstly, we measured times for non-collective (`ncmpi_put(get)_var_int()`) and collective operations (`ncmpi_put(get)_vars_int_all()`) as shown in Figure 7. It is remarked that the data were split along the z-axis evenly in the collec-

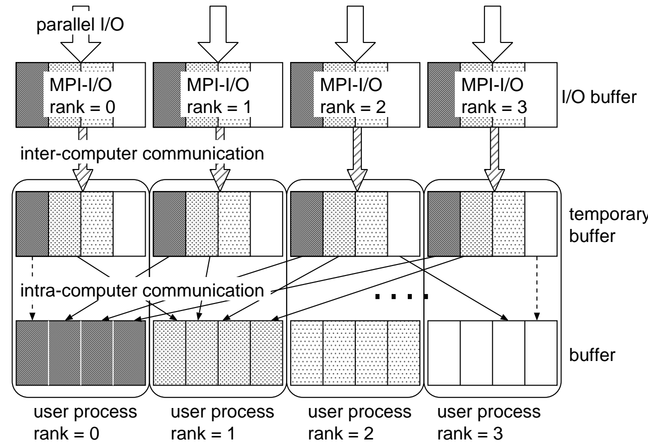


Fig. 6. Typical I/O and communications patterns in collective I/O with a derived data type

tive operations. In the both operations, the collective functions outperformed the non-collective ones at each message data size, however, the times were not minimized so much with an increase in the number of user processes. This was due to a bottleneck in network connection between the clusters. Inter-computer communication was bottleneck and communication times basically depended on total data size. Furthermore, there was a single network link between PC clusters and the total data size was constant even if we change the number of user processes. As a result, total I/O times were bounded by inter-computer network sustained bandwidth. Although we have such the restriction, this system would be usable because total I/O times are not degraded so much with an increase in the number of user processes. It is also noticed that it may be possible to increase its total performance if we have multiple physical network links by using link aggregation between PC clusters, for example.

Secondly, times for collective functions were measured with respect to axis to split a data image along. It is obviously expected that the more I/O pattern becomes complex, the more the I/O times increase. In this test, we measured the times for splitting along x, y, and z-axes. The results are shown in Figure 8. In both the read and the write operations, splitting the data image along the z-axis provided the most shortest times, and the times for the x-axis were the worst. Difference between the times for the x and z axes was around 0.5 s. It is also noticed that the I/O times were almost the same between the cases for two and four processes except the case of $256 \times 256 \times 256$ message data in the read operations.

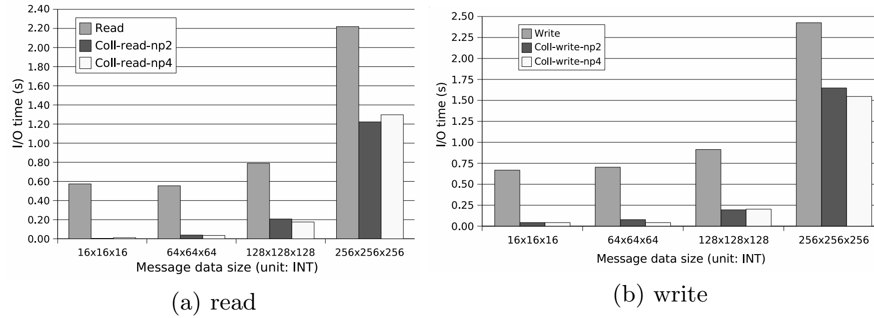
To find reasons for the increase in the I/O times with respect to axis to split along, we measured times for creation of a derived data type, synchronization of collective operations by `MPI_File_sync()`, and `MPI_File_write_all()` in remote

Table 2. Specifications of PC clusters used in performance measurement, where **serv** and **comp** in a bold font denote server and computation nodes, respectively

	PC cluster I	PC cluster II
serv	Dell PowerEdge800 × 1	Dell PowerEdge1600SC × 1
comp	Dell PowerEdge800 × 4	Dell PowerEdge1600SC × 4
CPU	Intel Pentium-4 3.6 GHz × 1	Intel Xeon 2.4 GHz × 2
Chipset	Intel E7221	ServerWorks GC-SL
Memory	1 Gbyte DDR2 533 SDRAM	2 Gbyte DDR 266 SDRAM
Disc system		
serv	80 Gbyte (Serial ATA) × 1	73 Gbyte (Ultra320 SCSI) × 1
comp	80 Gbyte (Serial ATA) × 1	73 Gbyte (Ultra320 SCSI) × 2
NIC	Broadcom BCM5721 (on-board PCI-Express)	Intel PRO/1000-XT (PCI-X board)
Switch	3Com SuperStack3 Switch 3812	3Com SuperStack3 Switch 4900
OS	Fedora Core 3	
kernel	2.6.12-1.1381_FC3smp(**serv**) 2.6.11-1SCOREsmp (**comp**)	2.6.12-1.1381_FC3smp (**serv**) 2.6.11 (**comp**)
Ethernet driver	Broadcom tg3 v3.71b (**serv**) Broadcom tg3 v3.58b (**comp**)	Intel e1000 version 6.0.54 (**serv**) Intel e1000 version 5.6.10.1 (**comp**)
MPI library	MPICH version 1.2.7p1	

I/O by using a pure MPI program. We supposed that those functions simulated I/O patterns which were carried out in the PnetCDF program.

Concerning creation of a derived data type, processing times were constant and negligible (around 0.05 ms and 0.08 ms for two and four user processes, respectively) in the total I/O times. Times for `MPI_File_sync()` are shown in Figure 9. They became long with an increase in the message data size. As the Stampi's `MPI_File_sync()` for remote I/O synchronized all the I/O synchronization by MPI-I/O processes, required time for it increased with an increase in the message data size. Figure 10 shows I/O times of `MPI_File_write_all()` and `MPI_File_read_all()` in remote I/O. For example, there is difference around 0.5 s between the times for splitting the data image along the z-axis and others. As it is hard to simulate operations of PnetCDF functions, this might be rough analysis, however, it is concluded that the differences in the I/O times with respect to axis to split along were mainly due to an increase in I/O times of `MPI_File_write_all()` and `MPI_File_read_all()`. To check whether this is coming from network data transfer or I/O operations on a remote computer, we also measured local I/O times on the PVFS2 file system as shown in Figure 11. It is obvious that the increase was coming from an increase in the times of the local I/O operations. Moreover, the I/O times were almost the same and did not scale with regard to the number of user processes. This might be due to bottleneck in intra-computer data transfer for collective operations or I/O operations on a PVFS2 file system.



(a) read (b) write

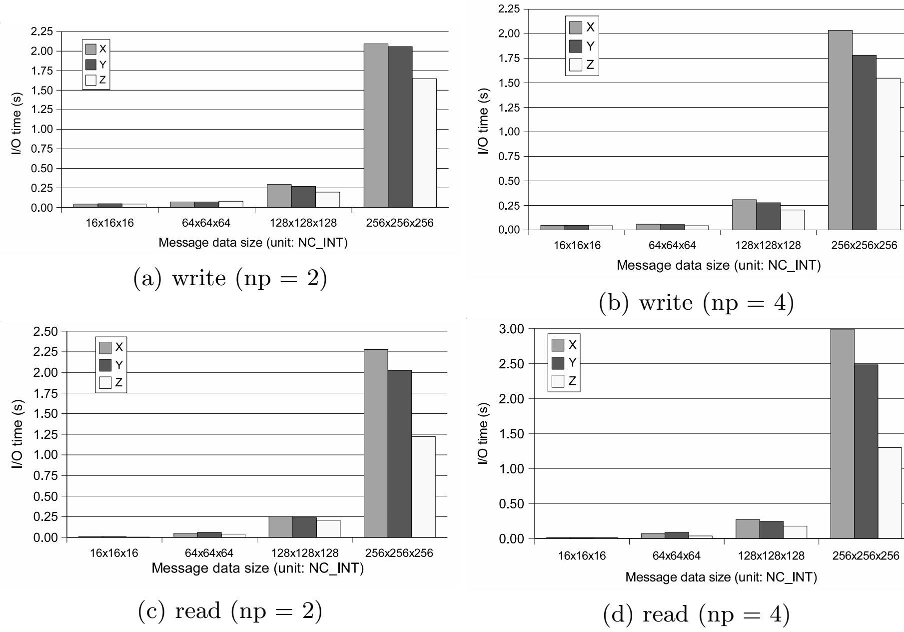
Fig. 7. Times of non-collective and collective remote I/O operations. *Read* and *Write* denote non-collective operations. While *Coll-read* and *Coll-write* denote collective operations, where numbers which follow *np* are the number of user processes and MPI-I/O processes.

4 Related Work

Providing a common data format makes data I/O operations portable for application programmers. This kind of implementations such as netCDF [8] and HDF5 [9] has been proposed. NetCDF provides a self-describing and common multi-dimensional data format and a simple interface. Its parallel I/O operations have been realized in PnetCDF, which is an extension of the interface, by introducing MPI-I/O functions as an underlying parallel I/O library [4]. On the other hand, HDF5 provides hierarchical data format in order to access huge amount of data effectively. An HDF5 interface has two objects, one is "Dataset" and another "Group". The Dataset manages multi-dimensional array data, while the Group provides relational mechanisms among objects. Parallel I/O operations are also available with this interface by introducing MPI-I/O functions as an underlying parallel I/O interface library [10].

An MPI-I/O interface in the MPI-2 standard [2] realizes parallel I/O operations in an MPI program. Several implementations of it are available such as ROMIO [3]. Its MPI-I/O operations to many kinds of file systems are realized through an ADIO interface [11]. It hides heterogeneity in architectures of each systems and provides a common interface to an upper MPI-I/O layer. Remote I/O operations using the ROMIO are available with the help of RFS [12]. An RFS request handler on a remote computer receives I/O requests from client processes and calls an appropriate ADIO library. On the other hand, Stampi itself is not an MPI implementation but a bridging library among different MPI implementations. It realizes seamless MPI operations among them by using TCP socket communications.

Inter-operability among different MPI implementations is also important issue, typically in a Grid computing environment. One of the representative works is PACX-MPI [13]. It realizes inter-operable MPI communications by deploying a common MPI interface library on top of each MPI implementation in order to realize seamless MPI operations. This system provides high performance MPI



(a) write (np = 2)

(b) write (np = 4)

(c) read (np = 2)

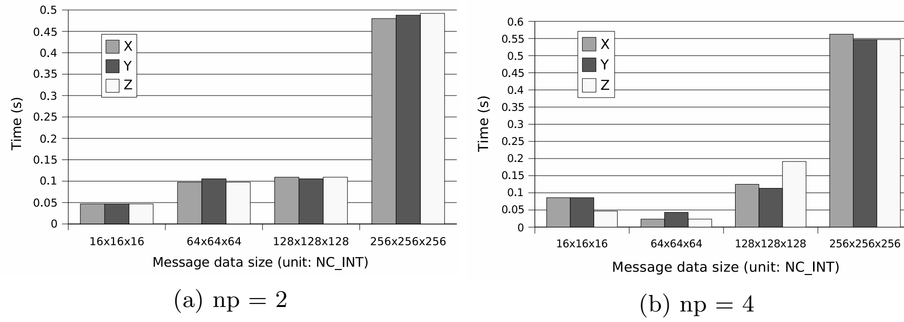
(d) read (np = 4)

Fig. 8. Times of collective PnetCDF functions in remote I/O with respect to axis to split a three-dimensional data along. X, Y, and Z denote axes to split along.

communications inside the same MPI implementations and seamless operations among different ones. GridMPI [14] also realizes such mechanism in a Grid computing environment. Inter-operable MPI communications are realized through an IMPI interface by using P2P data communications. It is supported on many kinds of MPI implementations. On the other hand, Stampi realizes the similar communication mechanism with PACX-MPI with regard to a method to implement a common communication layer on top of each MPI implementation. However, it realizes a flexible communication mechanism where invocation of proxy process is dynamically selected according to communication topology of each parallel computer.

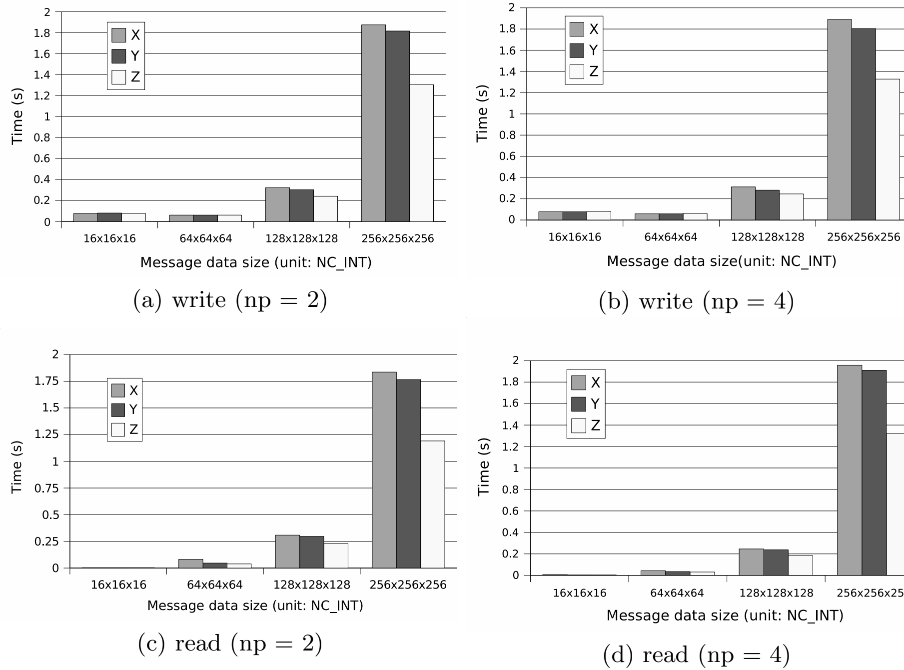
5 Conclusion

We have developed a seamless remote I/O system using a PnetCDF interface among different MPI implementations by using a Stampi library. Its collective I/O interface outperformed its non-collective one in remote I/O. We also measured I/O times of collective one with respect to axis to split evenly along using three-dimensional data sets. Derived data types were created based on the associated splitting pattern. It was expected that the more complex the data type became, the more its I/O time increased due to an increase in the number of intra-computer data transfers for collective I/O. We have evaluated such the



(a) np = 2 (b) np = 4

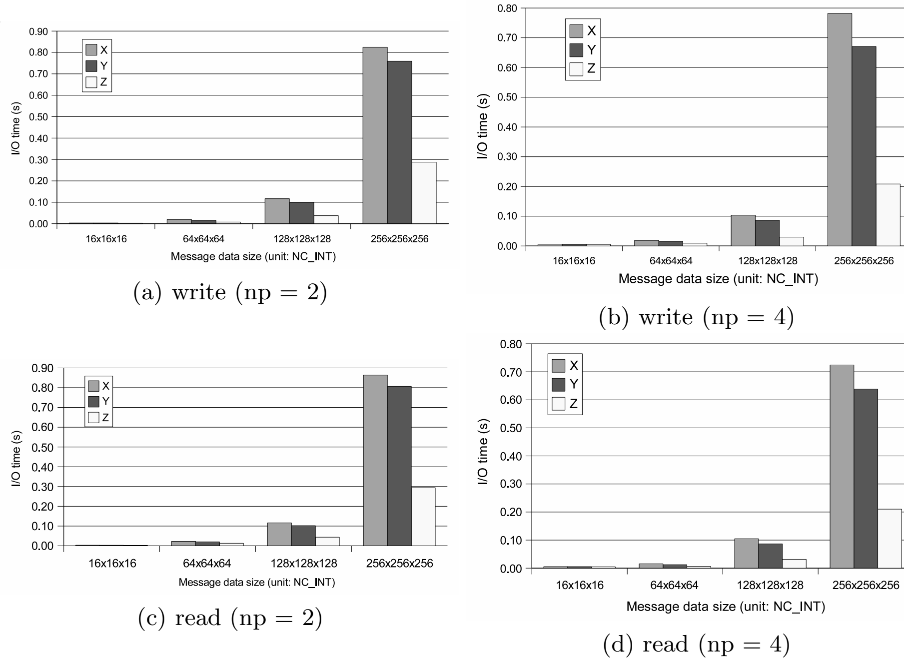
Fig. 9. Times for `MPI_File_sync()` in remote I/O with (a) two and (b) four user processes and MPI-I/O processes. X, Y, and Z denote axes to split a three-dimensional data along.



(a) write (np = 2) (b) write (np = 4)

(c) read (np = 2) (d) read (np = 4)

Fig. 10. Times for `MPI_File_write_all()`/ `MPI_File_read_all()` in remote I/O. X, Y, and Z denote axes to split a three-dimensional data along.



(a) write (np = 2)

(b) write (np = 4)

(c) read (np = 2)

(d) read (np = 4)

Fig. 11. Times of local collective I/O on a PVFS2 file system, where X, Y, and Z denote axes to split a three-dimensional data along

I/O patterns in both remote and local I/O operations. In remote I/O by using a PnetCDF interface, splitting along the most inner index provided the most complex derived data type. As a result, its I/O times were the most longest. On the other hand, times for splitting along the most outer index were the most shortest. The increase in the times with regard to axis to split along was coming from local I/O operations by using a native MPI library. The system did not scale due to a bottleneck in inter-computer data transfer or I/O operations on a PVFS2 file system, however, its performance was almost the same with respect to the number of user processes. Optimization in message data transfer between computers is considered as a future work. Implementation of non-blocking I/O functions is also considered to minimize visible I/O times.

Acknowledgment

The author would like to thank staff at Center for Computational Science and e-Science (CCSE), Japan Atomic Energy Agency (JAEA), for providing a Stampi library and giving useful information.

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 18700074 and by the CASIO Science Promotion Foundation.

References

1. Rew, R.K., Davis, G.P.: The unidata netCDF: Software for scientific data access. In: Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology, American Meteorology Society (February 1990) 33–40
2. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. (July 1997)
3. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems. (1999) 23–32
4. Li, J., Liao, W.K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A high-performance scientific I/O interface. In: SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, IEEE Computer Society (November 2003) 39
5. Tsujita, Y., Imamura, T., Takemiya, H., Yamagishi, N.: Stampi-I/O: A flexible parallel-I/O library for heterogeneous computing environment. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, Volume 2474 of Lecture Notes in Computer Science., Springer (2002) 288–295
6. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI Message-Passing Interface standard. Parallel Computing **22**(6) (1996) 789–828
7. PVFS2: `http://www.pvfs.org/pvfs2/`.
8. Rew, R., Davis, G., Emmerson, S., Davies, H., Hartnett, E.: NetCDF User's Guide. Unidata Program Center. (June 2006) `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/`.
9. The National Center for Supercomputing Applications: `http://hdf.ncsa.uiuc.edu/HDF5/`.
10. Ross, R., Nurmi, D., Cheng, A., Zingale, M.: A case study in application I/O on Linux clusters. In: SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), ACM Press (November 2001) 11
11. Thakur, R., Gropp, W., Lusk, E.: An abstract-device interface for implementing portable parallel-I/O interfaces. In: Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation. (1996) 180–187
12. Lee, J., Ma, X., Ross, R., Thakur, R., Winslett, M.: RFS: Efficient and flexible remote file access for MPI-IO. In: Proceedings of the 6th IEEE International Conference on Cluster Computing (CLUSTER 2004), IEEE Computer Society (September 2004) 71–81
13. Gabriel, E., Resch, M., Beisel, T., Keller, R.: Distributed computing in a heterogeneous computing environment. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, Volume 1497 of Lecture Notes in Computer Science., Springer (1998) 180–187
14. GridMPI: `http://www.gridmpi.org/`.