An Efficient MPI-IO for Noncontiguous Data Access over InfiniBand

Ding-Yong Hong, Ching-Wen You and Yeh-Ching Chung[1]
Department of Computer Science, National Tsing Hua University, Taiwan
{dyhong, chingwuen}@sslab.cs.nthu.edu.tw, ychung@cs.nthu.edu.tw

Abstract

Noncontiguous data access is a very common access pattern in many scientific applications. Using POSIX I/O to access many pieces of noncontiguous data segments will generate a lot of I/O requests that cause the I/O system perform poorly. Tow-phase I/O, also called collective I/O, applied by MPI-IO provides a good method to optimize noncontiguous I/O operations. However, this method requires many data segments being redistributed. This redistribution needs some information been calculated in advance and lots of data segments are transmitted via network more than once. With I/O-intensive applications, the aggregate size of these data segments being redistributed becomes significant large. This additional overhead will degrade the system performance. In this thesis, we extend the collective I/O method and propose a new I/O scheme to avoid retransmission of data segments by applying RDMA gather/scatter operations supported by InfiniBand hardware. We also extend the "view" and "datatype" concepts of MPI into the file system to help complete our design. The experiments show that the method we design improves the performance and is more efficient than the collective I/O approach.

1. Introduction

In a cluster system, many I/O-intensive scientific applications frequently request I/O operations between memory and files which reside in different machines. With the computing ability and the network transmission rate of processors become more and more powerful, I/O is emerging as the main bottleneck degrading the system performance. Therefore, efficient and scalable parallel I/O schemes and high performance parallel file systems are needed. Many parallel I/O designs and parallel file systems have been proposed in the literature [6,7,8,9]. Sometimes the data layout is noncontiguous in the memory or in the file. In the traditional way, like POSIX I/O, it will access these noncontiguous data segments using individual request for each contiguous data piece. Inevitably, as the number of noncontiguous data segments is numerous, I/O clients will generate a large amount of I/O operations. Such large amount of I/O requests will cause I/O servers busy and the performance of I/O servers will be degraded.

Collective I/O [8] is an optimizing approach for these kinds of noncontiguous data layouts. Through the analysis of the collection of independent I/O operations, what data regions must be transferred is determined. These data regions are then split up among I/O clients and the noncontiguous data segments are redistributed to appropriate positions in appropriate clients. Each data region formed by combining many I/O operations can be read/write by only one request call. Thus, total numbers of I/O requests are reduced. Redistribution, however, needs extra overheads such as redistribution information calculation and data transmissions. These additional calculations will add more computing loads to I/O clients which may run some CPU-intensive application programs, and additional data transmissions will also increase network traffic. How to reduce these overheads is important to the performance of collective I/O.

In this thesis, we address these issues of noncontiguous data accesses and extra data transmissions of collective I/O on InfiniBand architecture (IBA) [3]. We extend the collective I/O method and propose a new I/O scheme to avoid re-transmission of data segments based on InfiniBand RDMA read/write operations. We also extend the "view" and "datatype" concepts of MPI into the file system to help our design and shift a greater part of computing loads from I/O clients to I/O servers without increasing the loads of I/O servers. The experiments show that the proposed I/O scheme can improve the performance and is more efficient than the collective I/O approach.

[1] The corresponding author

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

The rest of the thesis is organized as follows. In Chapter 2, a brief survey of related work on collective I/O is given. Chapter 3 gives an overview of collective I/O. In Chapter 4, we describe the method for the noncontiguous data accesses based on InfiniBand RDMA read/write operations. Chapter 5 shows the performance results of our method and collective I/O. In Chapter 6, we give some conclusion remarks regarding to the proposed method and point out the directions of future work.

2. Related Work

Many methods for improving collective I/O have been proposed in the literature. Dickens *et al*. [2] presented that collective I/O can be further improved with threads. They demonstrated that using I/O threads to perform the collective I/O in the background while the main thread continues with computations is frequently a worst implementation option. The reason was that some parts of collective I/O cannot overlap with computations. They developed an alternate approach where some parts of the collective I/O are performed in the background, and some parts are performed in the foreground. Their technique provides up to 80% improvement over sequential collective I/O.

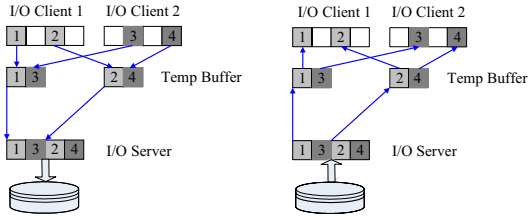
Kandemir [4] proposed a compiler-directed collective I/O approach which detects opportunities for collective I/O and inserts the necessary I/O calls in the code automatically. The characteristic of his approach is that instead of indiscriminately applying collective I/O by programmers, it uses collective I/O selectively only in cases where independent parallel I/O would not be possible or would lead to an excessive number of I/O calls. He showed that the approach he proposed is only 5.23 percent worse than an optimal scheme.

Ching *et al*. [1] addressed the issue of noncontiguous data accesses. Their implementation adds a method of noncontiguous data accesses, list I/O, supporting in the ROMIO MPI-IO implementation. Similar to collective I/O, their scheme can also reduce the numbers of I/O operations. They also proposed a hybrid scheme involving data sieving and list I/O. This scheme can cover a large range of access patterns.

3. Preliminaries

3.1. Overview of Collective I/O

Collective I/O consists of read, write, and read-modify-write operations. In its two-phase approach, every I/O client is an aggregator process. The role of the aggregator process is to collect the data segments to match the file layouts. In this thesis, we will focus on read and write operations.



(a) write **(b) read**

Figure 1: An example of the collective I/O operations.

3.1.1. The write operation. Figure 1(a) shows the behavior of the write operation of collective I/O. In Figure 1(a), I/O clients 1 and 2 are responsible to write the file layouts {1, 3} and {2, 4} to the I/O server, respectively. At first, the noncontiguous data layout in each I/O client does not match the file layout specified. In the first phase, the two I/O clients collectively match file layouts {1, 3} and {2, 4} by analyzing and redistributing some data segments between them. After the redistribution, in the second phase, the contiguous data region of each I/O client can be transferred to the I/O server with only one write operation. Once the I/O server receives the contiguous data regions from each I/O client, it can write the received data regions to disk.

Assume that there are m I/O clients collectively write n data segments to one I/O server and each data segment is k-byte. Each I/O client is responsible to write n/m data segments. The two-phase write operation consists of four steps, the analysis step, the redistribution step, the write step, and the disk I/O step. In the analysis step, an I/O client needs to compute the information for data segments redistribution. In the redistribution step, an I/O client is responsible for redistributing the data segments among I/O clients. In the write step, an I/O client writes the contiguous data regions to an I/O server. In the disk I/O step, an I/O server writes data to disk. The total time of the write operation, T_{CIO_write}, is

$$T_{CIO\_write} = T_{CIO\_analysis} + T_{CIO\_redistribution} + T_{CIO\_write} + T_{disk\_io},$$

where $T_{CIO\_analysis}$, $T_{CIO\_redistribution}$, T_{CIO_write}, and T_{disk_io} is the time to perform the analysis step, the redistribution step, the write step, and the disk I/O step, respectively.

For $T_{CIO\_analysis}$, it can be divided into two different computing parts. The first part is to transform the datatype to positions and lengths of data segments. The time needed for transformation is proportional to the number of data segments, n. This part will cost time αn, where α is a constant. The second part is to pick up all data segments held by an I/O client and calculate final positions where these segments will be redistributed. It will cost time βn, where β is a constant. We have

$$T_{CIO\_analysis} = \alpha n + \beta n.$$

For $T_{CIO\_redistribution}$, each client needs to distribute $\frac{m-1}{m} \times \frac{n}{m}$ data segments to other I/O clients. We have

$$T_{CIO\_redistribution} = \frac{m-1}{m} \times \frac{n}{m} \times \frac{k}{S_n},$$

where S_n is the bandwidth of the network.

In the write step, each I/O client writes the data region to the I/O server. The write time is

$$T_{CIO\_write} = \frac{n}{m} \times \frac{k}{S_n}.$$

After the I/O server receives these data regions, they will be written into disk. The disk access time is

$$T_{disk\_io} = n \times \frac{k}{S_d},$$

where S_d is the bandwidth of disk I/O.

The total time to finish the write operation is

$$T_{CIO\_write} = T_{CIO\_analysis} + T_{CIO\_redistribution} + T_{CIO\_write} + T_{disk\_io}$$
$$= (\alpha n + \beta n) + (\frac{m-1}{m} \times \frac{n}{m} \times \frac{k}{S_n}) + (\frac{n}{m} \times \frac{k}{S_n}) + (n \times \frac{k}{S_d}).$$

3.1.2. The read operation. Figure 1(b) shows the behavior of the read operation of collective I/O. In Figure 1(b), I/O clients 1 and 2 need to get data segments {1, 2} and {3, 4} from the I/O server, respectively. However, the order of data segments in the I/O server is (1, 3, 2, 4). The data layout in the I/O server does not match the layout of each I/O client. In the first phase, these two I/O clients analyze the best layout information for I/O server and ask I/O server to read the data segments from disk. In the second phase, the I/O server sends contiguous data regions to each client according to the request of each client. In our example, I/O sever will send contiguous data regions {1, 3} and {2, 4} to I/O clients 1 and 2, respectively. After each client received the contiguous data region, it redistributes some data segments to corresponding clients. Therefore, client 1 and client 2 redistribute data segments 3 and 2 to client 2 and client 1, respectively.

The two-phase read operation also consists of four steps, the analysis step, the disk I/O step, the read step, and the redistribution step. With the same assumption of the write operation, the total time to finish the noncontiguous I/O read operation is

$$T_{CIO\_read} = T_{CIO\_analysis} + T_{disk\_io} + T_{CIO\_read} + T_{CIO\_reditribution}$$
$$= (\alpha n + \beta n) + (n \times \frac{k}{S_d}) + (\frac{n}{m} \times \frac{k}{S_n}) + (\frac{m-1}{m} \times \frac{n}{m} \times \frac{k}{S_n}),$$

where $T_{CIO\_analysis}$, T_{disk_io}, T_{CIO_read}, and $T_{CIO\_redistribution}$ is the time to perform the analysis step, the disk I/O step, the read step, and the redistribution step, respectively.

4. Extended Collective I/O Based on InfiniBand RDMA

The InfiniBand architecture is a new industry-standard architecture for server I/O and inter-server communication [3]. It defines two kinds of semantics, the channel semantic and the memory semantic. These two semantics define send/receive and Remote Direct Memory Access (RDMA) read/write operations. The RDMA operation also supports the gather/scatter operations. The RDMA write can collect multiple data segments and then write to a contiguous region of memory with one request. The RDMA read can read a contiguous region of memory and dispatches the contiguous subsets of these data into separate memory locations. With the RDMA read/write operations, the time for the data segments redistribution in collective I/O can be reduced a lot. In the following, we will describe how to use InfiniBand RDMA read and write operations to efficiently perform read and write operations of collective I/O in details. We call our design as ***Extended Collective I/O*** (**ECIO**).

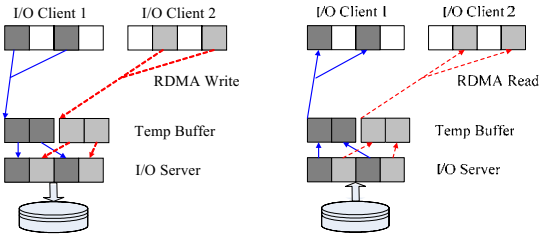
4.1. The write Operation of ECIO

To use RDMA gather/scatter write operation to efficiently perform the write operation of collective I/O, the goal is to avoid retransmission of data segments among I/O clients. Therefore, we extend the design of the write operation of collective I/O. The write operation of ECIO consists of the following five steps:

1. Analysis step – An I/O client needs to compute the corresponding lengths and locations of data segments specified by the I/O client.
2. Write step – An I/O client writes noncontiguous data segments to the I/O server by using RDMA gather/scatter write operation.
3. Transformation step – The I/O server transforms the datatype to offsets and lengths of the data segments.
4. Local redistribution step – The I/O server is responsible to move the data segments received from I/O clients to their corresponding locations in its local memory.
5. Disk I/O step – The I/O server writes data to disk.

Since the redistribution step of collective I/O will cause the data segment transmissions among I/O clients, in our design, we divide this step to the transformation step and the local redistribution step. In this way, we can reduce the overhead of data segment transmissions from interconnection network to local memory. This will reduce the execution time a lot.

In the analysis step, our design is similar to the analysis step of the write operation of collective I/O. Each client still calculates information about its own data segments.



(a) write **(b) read**

Figure 2: An example of noncontiguous data accesses using RDMA gather/scatter operations.

In the write step, if I/O clients detect that the noncontiguous data segments are all subsets of any contiguous data regions, these data segments will be written directly from I/O clients to I/O servers. In order not to act as the POSIX I/O operation, we apply the RDMA gather/scatter write operation. With this write operation, many pieces of data segments are combined and fit into one RDMA gather/scatter list.

In the transformation step, after the I/O server receives data segments from I/O clients, it has no idea about the corresponding lengths and locations of all received data segments in a file. To solve this problem, we take the advantage of the "view" and "datatype" concepts of MPI. That is, the I/O server must have the same information of the datatype which has been defined in the user program. After the datatype is set, we must make a copy of the datatype in the I/O server. The datatype can be passed to the I/O server as a hint either when the user program calls the function *MPI_File_set_view* or be attached to the I/O request message sent to the I/O server. With this hint, the I/O server can transform the datatype to offsets and lengths.

Adding extra datatype calculations will bring some overheads to the I/O server. Since the I/O server is usually in I/O or network communication phases, the idle CPU time can be used to perform this calculation. Therefore, it is easy to overlap these computations with I/O and network communication without degrading the performance of the I/O server.

In the local redistribution step, the I/O server moves data segments received to their corresponding locations. Since this redistribution is performed in the local memory and can be overlapped with communication, the time to execute this operation is insignificant. In the disk I/O step, the design is the same as that of the write operation of collective I/O.

Figure 2(a) shows an example of our design. In Figure 2(a), I/O clients want to write the shadowed noncontiguous data segments to the I/O server. The I/O clients perform the RDMA gather/scatter write operation to transfer data segments to the I/O server in parallel. After all data segments are received, the I/O server performs local copies to redistribute them to correct positions. Then, the data segments are written to the disk. Assume that there are m I/O clients collectively write n data segments to one I/O server and each data segment is k-byte. The total time to complete the write operation, T_{ECIO_write}, of ECIO is

$$T_{ECIO\_write} = T_{ECIO\_analysis} + T_{RDMA\_write} + T_{ECIO\_transform} + T_{local\_redistribution} + T_{disk\_io},$$

where $T_{ECIO\_analysis}$, T_{RDMA_write}, $T_{ECIO\_transform}$, $T_{local\_redistribution}$ and T_{disk_io} is the time to perform the analysis step, the write step, the transformation step, the local redistribution step, and the disk I/O step, respectively.

In the analysis step, since an I/O client only needs to compute the information of data segments specified by I/O client, we have $T_{ECIO\_analysis} = \alpha n$.

In the write step, an I/O client writes the noncontiguous data segments to an I/O server using RDMA gather/scatter write operation. We have

$$T_{RDMA\_write} = \frac{n}{m} \times \frac{k}{S_{sg\_write}},$$

where S_{sg_write} is the bandwidth of the network for the RDMA gather/scatter write operation.

In the transformation step, after the data segments from I/O clients are received, the file system has to transform the datatype to offsets and lengths for later local redistribution. This time of transformation is the same as $T_{ECIO\_analysis}$. We have, $T_{ECIO\_transform} = \alpha n$. Since this calculation can be overlapped with other I/O operations, $T_{ECIO\_transform}$ can be hidden and saved, that is, we can assume that $T_{ECIO\_transform} = 0$.

The local redistribution time, $T_{local\_redistribution}$, is relative to the computing power of the processor. However, the local copies among memory are fast. In our design, this redistribution can be overlapped with other I/O operations. Therefore, we can also assume that $T_{local\_redistribution} = 0$. In the disk I/O step, an I/O server writes data to disk. T_{disk_io} is the same as the disk access time of collective I/O. The total time to finish the extended collective write operation is

$$\begin{aligned} T_{Extend\_CIO\_write} &= T_{ECIO\_analysis} + T_{RDMA\_write} + \\ &\quad T_{ECIO\_fransform} + T_{local\_redistribution} + T_{disk\_io} \\ &= \alpha n + \frac{n}{m} \times \frac{k}{S_{sg\_write}} + n \times \frac{k}{S_d} \end{aligned}$$

4.2. The read Operation of ECIO

The design of the read operation of ECIO is similar to that of the write operation of ECIO. The read operation of ECIO consists of the following five steps:

1. Analysis step – An I/O client needs to compute the corresponding lengths and locations of data segments specified by the I/O client.
2. Disk I/O step – The server reads data from disk.
3. Transformation step – The I/O server transforms

datatype to offsets and lengths of data segments.

4. Local redistribution step – The I/O server is responsible to move data segments retrieved from disk to the correct locations in its local memory.
5. Read step – An I/O client reads noncontiguous data segments from the I/O server by using RDMA gather/scatter read operation.

Figure 2(b) shows an example of our design. The assumption is the same as that of the write operation of ECIO. The analysis step is the same as that in extended collective write design. We have $T_{ECIO\_analysis} = \alpha n$. In the disk I/O step, an I/O server reads data segments from disk. T_{disk_io} is the same as the disk access time of collective I/O. The time of transformation step is the same as that of extended collective write operation. We have $T_{ECIO\_transform} = \alpha n$. Since this transformation can be overlapped with other I/O operations, the time required can be hidden and saved. We can assume that $T_{ECIO\_transform} = 0$. The local redistribution among memory is fast and in our design this redistribution can be overlapped with I/O or network communication, we can also assume that $T_{local\_redistribution} = 0$. Finally, the time for the RDMA gather/scatter read operation is

$$T_{RDMA\_read} = \frac{n}{m} \times \frac{k}{S_{sg\_read}},$$

where S_{sg_read} is the bandwidth of RDMA gather/scatter read operation. The total time to perform noncontiguous data read in our implementation is

$$\begin{aligned} T_{Extend\_CIO\_read} &= T_{ECIO\_analysis} + T_{disk\_io} + T_{ECIO\_transform} + \\ &\quad T_{local\_redistribution} + T_{RDMA\_read} \\ &= \alpha n + n \times \frac{k}{S_d} + \frac{n}{m} \times \frac{k}{S_{sg\_read}} \end{aligned}$$

4.3. Theoretical Analysis

For the write operation of ECIO, our scheme can save time $T_{CIO\_write} - T_{extend\_CIO\_write}$, that is,

$$\begin{aligned} &T_{CIO\_write} - T_{Extend\_CIO\_write} \\ &= \beta n + \left(\frac{m-1}{m} \times \frac{n}{m} \times \frac{k}{s_n}\right) + \left(\frac{n}{m} \times \frac{k}{S_n}\right) - \left(\frac{n}{m} \times \frac{k}{S_{sg\_write}}\right). \quad (1) \end{aligned}$$

From Equation (1), we can predict when the number of I/O clients, m, becomes larger, the difference between collective I/O and our scheme will become smaller. However, as the number of data segments, n, increases, the time difference will also increase.

For the read operation of ECIO, our scheme can save time $T_{CIO\_read} - T_{extend\_CIO\_read}$, that is,

$$\begin{aligned} &T_{CIO\_read} - T_{Extend\_CIO\_read} \\ &= \beta n + \left(\frac{m-1}{m} \times \frac{n}{m} \times \frac{k}{s_n}\right) + \left(\frac{n}{m} \times \frac{k}{S_n}\right) - \left(\frac{n}{m} \times \frac{k}{S_{sg\_read}}\right). \quad (2) \end{aligned}$$

From Equation (2), we have similar predictions as those in Equation (1).

5. Performance Evaluation

This section demonstrates the performance results of our implementations. The I/O servers and I/O clients run Linux with kernel version 2.6.8 on 5 IBM e-Servers. Each IBM e-Server has dual Xeon 2.4 GHz CPUs and 1 GB DRAM. All machines are connected with InfiniHost 7000 dual port 4x HCA adapters. The I/O server contains one 36GB SCSI hard disk. The InfiniBand interface is VAPI [5] which is a user-level API. We use MVPICH [6] and PVFS2 [7] as our MPI-IO software and file system, respectively. Figure 3 represents the "datatype" structure of our test data. This datatype can be defined by calling the function *MPI_Type_struct*(*count, blocklens, indices, old_type, &new_type*) supported by MPI, where *count* is the total numbers of data blocks, *blocklens* means the number of elements in each block, *indices* means displacement between two joint blocks and *old_type* is the basic datatype of the data element in the data blocks.

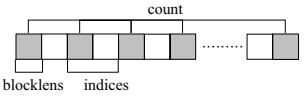


Figure 3: A structure of an MPI datatype.

5.1. Noncontiguous Data Write

In the first experiment of write case, we set m=2, *blocklens*=1, *indices* = 2, *old_type* = MPI_INT, and *count* is set to {4000, 8000,…,512000, 1024000}. The data segments are written to one I/O server and this noncontiguous data write case is executed 100 times.

Table 1 shows the performance results of our design for the first experiment. In this test case, one I/O client is responsible to write shadow blocks shown in Figure 3 and the other client is responsible to write the white blocks. In Table 1, column 1 is the number of data elements in a block. Columns 2 and 3 show the file size and the total size of data redistribution after the write operation is executed 100 times, respectively. Columns 4 and 5 are the execution time of collective I/O and our scheme, respectively. Column 6 shows the improvement of our scheme over collective I/O.

Table 1: Noncontiguous data write: 2 clients + 1 server.

count (10^3)	File size (MB)	Resent size (MB)	MPI CIO (sec)	ECIO (sec)	Improvement (%)
4	1.6	0.8	1.65	1.48	10.3
8	3.2	1.6	1.94	1.77	8.8
16	6.4	3.2	2.48	2.20	11.3
32	12.8	6.4	3.62	3.22	11.0
64	25.6	12.8	6.03	5.12	15.0
128	51.2	25.6	10.72	9.29	13.3
256	102.4	51.2	20.20	17.26	14.6
512	204.8	102.4	39.62	33.37	15.8
1024	409.6	204.8	77.92	65.66	15.7

From Table 1, as the number of noncontiguous data segments increases, our scheme can save more time than collective I/O. This result matches the theoretical analysis described in Chapter 4.3.

In the second experiment of write case, the settings are the same as those of the first experiment except that the total number of I/O clients, *m*, is set to 4. Table 2 shows the performance results of our design. From Table 2, as the number of noncontiguous data segments increases, our scheme can save more time than the collective I/O method. From Tables 1 and 2, we also observe that as the number of client increase, the time difference between collective I/O and our I/O scheme decreases. These results match the theoretical analysis described in Chapter 4.3.

Table 2: Noncontiguous data write: 4 clients + 1 server.

count (10^3)	File size (MB)	Resent size (MB)	MPI CIO (sec)	ECIO (sec)	Improvement (%)
4	1.6	1.2	1.76	1.56	11.3
8	3.2	2.4	1.92	1.71	10.9
16	6.4	4.8	2.34	1.96	16.2
32	12.8	9.6	2.85	2.49	12.6
64	25.6	19.2	4.04	3.73	7.6
128	51.2	38.4	6.36	6.04	5.3
256	102.4	76.8	11.45	10.86	5.2
512	204.8	153.6	21.55	20.21	6.2
1024	409.6	307.2	41.51	39.05	5.9

5.2. Noncontiguous Data Read

In the third and forth experiments, the settings are the same as those for the first and second experiments, respectively, except that data segments are read from disk to I/O clients. Table 3 and Table 4 show the performance results of these two test cases respectively. From these results, we have similar observations as those of the first and second experiments.

6. Conclusions and Future Work

In this thesis, we have proposed a new scheme for parallel I/O over InfiniBand. By taking advantage of the InfiniBand hardware, fewer I/O requests and reduction of extra data transmission can be achieved. The methods we proposed extend the collective I/O design and are more efficient for noncontiguous data access. Some experimental tests have been conducted in our implementation. The experimental results show that the performance of our schemes is better than that of collective I/O for all test cases. In this thesis, we only consider the read/write operations of collective I/O over InfiniBand architecture. In the future, we will consider how to implement the read-modify-write operation of collective I/O. Also, we will do more experimental tests, such as multiple clients and multiple servers, to verify the proposed schemes.

Table 3: Noncontiguous data read: 2 clients and 1 server.

count (10^3)	File size (MB)	Resent size (MB)	MPI CIO (sec)	ECIO (sec)	Improvement (%)
4	1.6	0.8	1.56	1.42	9.0
8	3.2	1.6	1.80	1.63	9.4
16	6.4	3.2	2.35	2.08	11.5
32	12.8	6.4	3.44	2.89	16.0
64	25.6	12.8	5.62	4.53	19.4
128	51.2	25.6	10.31	8.15	20.9
256	102.4	51.2	19.25	14.87	22.8
512	204.8	102.4	38.31	29.20	23.8
1024	409.6	204.8	76.29	58.02	23.9

Table 4: Noncontiguous data read: 4 clients and 1 server.

count (10^3)	File size (MB)	Resent size (MB)	MPI CIO (sec)	ECIO (sec)	Improvement (%)
4	1.6	1.2	1.52	1.42	6.6
8	3.2	2.4	1.66	1.54	7.2
16	6.4	4.8	1.95	1.74	10.8
32	12.8	9.6	2.59	2.27	12.4
64	25.6	19.2	3.60	3.17	11.9
128	51.2	38.4	5.85	5.31	9.2
256	102.4	76.8	10.59	9.67	8.7
512	204.8	153.6	20.02	18.51	7.5
1024	409.6	307.2	39.75	36.52	8.2

Acknowledgements

The work in this paper was partially supported by National Science Council and Ministry of Economic Affairs of the Republic of China under contract NSC-94-2213-E-007-080, NSC-94-2752-E-007-004-PAE, 94-EC-17-A-04-S1-044 and 94-EC-17-A-01-S1-038.

References

[1] A. Ching, A. Choudhary, K. Coloma, W.K. Liao, R. Ross, W. Gropp. "Noncontiguous I/O Accesses Through MPI-IO." *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003.

[2] P.M. Dickens, R. Thakur, "Improving Collective I/O Performance Using Threads", *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.

[3] InfiniBand™ Trade Association, *InfiniBand™ Architecture Specification Volume 1, Release 1.1*, 2002.

[4] M. Kandemir, "Compiler-Directed Collective I/O", *IEEE Transaction on Parallel and Distributed System*, 2001

[5] VAPI, Mellanox IB-Verbs API.

[6] MVPICH, MPICH 1.2.5 implementation on InfiniBand.

[7] PVFS2, Parallel Virtual File System 2.

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary. "Improved parallel I/O via a two-phase run-time access strategy." *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, 1993

[9] X. Shen, A. Choudhary, "A Distributed Multi-Storage Resource Architecture and I/O Performance Prediction for Scientific Computing." *Ninth IEEE International Symposium on High Performance Distributed Computing*, August 2000.