Supporting Efficient Noncontiguous Access in PVFS over InfiniBand

Jiesheng Wu, Pete Wyckoff[†], and Dhabaleswar K. Panda

Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

[†]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Technical Report
OSU-CISRC-5/03-TR26

Supporting Efficient Noncontiguous Access in PVFS over InfiniBand *

Jiesheng Wu† Pete Wyckoff‡ Dhabaleswar Panda†

†Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

‡Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

Noncontiguous I/O access is the main access pattern in many scientific applications. Noncontiguity exists both in access to files and in access to target memory regions on the client. This characteristic imposes a requirement of native noncontiguous I/O access support in cluster file systems for high performance. In this paper, we address two main issues on supporting efficient noncontiguous I/O access in cluster file systems over a high performance network. One is noncontiguous data transmission between the client and the I/O server. The second is noncontiguous disk access on the I/O server itself.

We propose a novel approach, RDMA Gather/Scatter*, to transfer noncontiguous data for such I/O accesses, and design a new scheme,* Optimistic Group Registration*, to reduce memory registration costs associated with this approach. For the second issue, we deploy data sieving on the I/O server to process a large number of small noncontiguous disk accesses. Unlike other data sieving implementations, our I/O server uses a cost model actively and intelligently to decide whether it is beneficial to perform data sieving or not. We have designed and incorporated these approaches in a version of PVFS over InfiniBand. Through a range of PVFS and MPI-IO micro-benchmarks, the MPI-IO tiled access test and the NAS BTIO benchmark, we demonstrate that our approaches attain significant performance gains compared to other existing approaches.*

1 Introduction

I/O is quickly emerging as the main bottleneck limiting performance in modern day clusters. The need for scalable parallel I/O and file systems is becoming more and more urgent. There has been a significant amount of work on parallel and cluster file systems, which has repeatedly demonstrated that a viable infrastructure consists of *commodity storage units connected with commodity networking technologies*, to provide high performance and scalable I/O support in cluster systems [2, 4, 25, 22, 32, 33, 8]. PVFS (Parallel Virtual File System) [4] is a good example of such an architecture and a leading cluster file system for parallel computing.

On the other hand, although file systems are designed for high performance, previous research shows that only about a tenth or less of the peak I/O performance can be realized by many applications [28, 15]. One of the main reasons is that the I/O interfaces available to applications and the I/O methods supported by file systems do not match well to applications' access characteristics. Most file systems are optimized for large contiguous file accesses, while in many applications, each process tends to access a large number of relatively small regions that are not located sequentially in the file [3, 19, 24]. Noncontiguity can exist in both the file itself and in the memory of the client.

Traditionally, noncontiguous access is achieved with a set of contiguous calls, each of which accesses only a single contiguous piece. Several techniques [26, 7, 23, 14, 13] were proposed to optimize noncontiguous accesses in situations where only contiguous I/O access support is available. Thakur *et al.* [27] noted that native noncontiguous access support in file systems themselves is important. They proposed an interface that describes noncontiguity in both memory and the file in a simple manner. This interface not only can be used to implement noncontiguous I/O access functions in the upper programming interfaces such as MPI-IO [18] efficiently, but also allows the file systems themselves to make further optimization on the noncontiguous accesses. Ching *et al.* [1] implemented this interface in PVFS. Their implementation is called *list I/O*.

There are two important issues in providing efficient noncontiguous accesses in cluster file systems wherein the compute nodes and the I/O nodes are connected by high performance networks. First, in a noncontiguous access, data may be written from or read into a large number of noncontiguous buffers. So high-performance noncontiguous data transmission between the compute node and the I/O node is critical in this case. Second, a noncontiguous access may result in a large number of small requests that access relatively small pieces of data in a noncontiguous manner. Efficient processing these small requests on the I/O nodes is crucial to application performance. These two issues result in more serious performance problems when the network is not the bottleneck in a cluster file system.

The issue of noncontiguous data transmission is often ignored in conventional networks. The performance differ-

*This research is supported in part by Sandia National Laboratory's contract #30505, Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052 and #CCR-0204429.

ences between different ways to handle noncontiguous data transmission might not have much impact on the performance of noncontiguous I/O accesses because of the high overhead and low bandwidth in these networks. We observe that noncontiguous data transmission becomes an important factor affecting the performance of noncontiguous I/O accesses in high performance networks such as InfiniBand [12]. Similarly, the inefficiency of processing noncontiguous small requests on the I/O node might not be realized by applications on conventional networks since network performance is the main bottleneck. However, as high performance networks become a popular means to connect the compute nodes and the I/O nodes in cluster file systems, this inefficiency has direct and significant impact on the performance of applications.

In this paper, we address these two issues by designing PVFS list I/O over the InfiniBand network. We describe how efficient noncontiguous data transmission can be achieved for PVFS noncontiguous I/O accesses and how PVFS I/O nodes can efficiently process many small noncontiguous file operations to serve a noncontiguous request. In this paper, we make the following observations and contributions:

1. Noncontiguous data transmission plays an important role in the performance of noncontiguous I/O accesses in cluster systems over high performance networks.
2. Gather/Scatter functionality in Remote Direct Memory Access (RDMA) operations offered by modern high performance networks can be used to transfer noncontiguous data efficiently.
3. Memory registration and deregistration for networks with remote DMA capabilities adds a new dimension to data transport issues. Our new memory registration scheme, Optimistic Group Registration, permits the efficient use of RDMA Gather/Scatter for noncontiguous data transmission.
4. A new technique, called *Active Data Sieving*, is shown to reduce excessive disk access costs caused by noncontiguous I/O accesses. The I/O nodes use a cost model to decide whether to perform data sieving to service a noncontiguous I/O request or to access contiguous file regions separately.

Our results show that the RDMA Gather/Scatter approach with Optimistic Group Registration can achieve a factor of 1.5 improvement on PVFS list I/O performance compared to other noncontiguous data transmission approaches. Active Data Sieving on the I/O node achieves a factor of 1.3–1.9 improvement for small noncontiguous I/O accesses. The performance results of the MPI-IO tiled access test show that our approaches attain the best performance compared to other existing methods in many cases. We have also evaluated the performance of the NAS BTIO benchmark with our implementation. The results obtained show that our approaches can offer a 20% improvement over the previous best result in a complex combination of noncontiguous data transmission and noncontiguous I/O accesses.

The schemes we explore for noncontiguous data transmission can be used for other purposes such as MPI noncontiguous data communication. Also, Active Data Sieving is not network specific: it can be considered to be a generic technique to process noncontiguous disk accesses on the I/O nodes in cluster file systems.

The rest of the paper is organized as follows. We first give a brief overview on PVFS, InfiniBand and ROMIO in Section 2. Section 3 states two issues on noncontiguous I/O accesses over high performance networks. In Sections 4 and 5, we address noncontiguous data transmission and noncontiguous disk accesses, respectively. The performance results are presented in Section 6. We examine some related work in Section 7 and draw our conclusions and discuss possible future work in Section 8.

2 Background

We recently designed and implemented a version of PVFS over the InfiniBand network. In work [31], we examined the feasibility of leveraging the InfiniBand technology to improve I/O performance and scalability of PVFS in clusters connected by the InfiniBand network. We focused on a software architecture which can take full advantage of InfiniBand features, efficient transport layer to support PVFS protocols, and buffer management. Our work shows that the InfiniBand network with its user-level communication and RDMA features can improve all aspects of PVFS, including throughput, access time, and CPU utilization. In the following subsections, we give brief overviews of PVFS, InfiniBand, and ROMIO.

2.1 Overview of PVFS

PVFS is a leading parallel file system for Linux cluster systems. It was designed to meet increasing I/O demands of parallel applications in cluster systems. A number of nodes in a cluster system can be configured as I/O servers and one of them is also configured to be the metadata manager.

PVFS achieves high performance by striping files across a set of I/O server nodes to achieve parallel accesses and aggregate performance. PVFS uses the native file system on the I/O servers to store individual file stripes. An I/O daemon runs on each I/O node and services requests from compute nodes, particularly read and write requests. Thus, data is transferred directly between I/O servers and compute nodes. A metadata manager provides a clusterwide consistent name space to applications. In PVFS, the metadata manager does not participate in read/write operations. PVFS supports a set of feature-rich interfaces, including support for both contiguous and noncontiguous accesses in both memory and files [5].

2.2 Overview of InfiniBand

The InfiniBand Architecture [12] defines a System Area Network for interconnecting both processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O.

Both channel and memory semantics are available for transferring data. In channel semantics, send/receive operations are used for communication. In memory semantics, Remote Direct Memory Access (RDMA) write and

read operations are used. Gather/Scatter are also supported in RDMA operations. RDMA write operation can gather multiple data segments together and write all data into a contiguous buffer on the peer side in one single operation. RDMA read operation can read data from a contiguous buffer on the peer side and place all data into several local buffers in one single operation.

2.3 Overview of ROMIO

MPI-IO, the I/O part of the MPI-2 standard [18], is an interface specifically designed for portable, high-performance parallel I/O. It acts as a higher-layer client which uses features of a parallel file system such as PVFS. MPI-IO uses MPI Datatype structures to describe the data layouts in the user's buffer and also to define the data layout in the file.

ROMIO [27] is a well-known implementation of MPI-IO with high-performance and portability on different file systems and platforms, including PVFS. It has four different methods to handle noncontiguous accesses on PVFS [1]: Multiple I/O, Data Sieving, Collective I/O and list I/O. MPI-IO applications can use hints or perform different I/O calls to choose one of methods.

3 Efficient Noncontiguous Access in PVFS

In this section, we first describe the current design and implementation of PVFS list I/O. We then show two different ways in which noncontiguous accesses arise, both of which pose challenges on efficient noncontiguous I/O access in PVFS. As illustrated in the example in Figure 1, the top set of communications shows *noncontiguous data transmission between the compute nodes and the I/O nodes*. The second source of noncontiguity is *noncontiguous disk accesses*, as shown at the bottom, when I/O nodes access their local files. We try to solve these two issues.

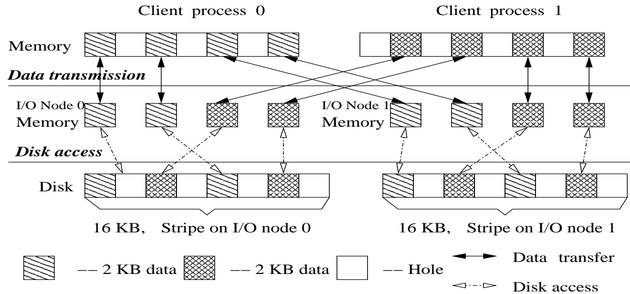

Figure 1. A PVFS list I/O example.

3.1 PVFS List I/O

PVFS provides a list I/O interface to applications which can be used to perform the transfers in Figure 1 in a single operation. This interface confirms with the interface proposed by Thakur *et al.* in [27]. The following is the PVFS list I/O read interface (the write interface is similar):

```
pvfs_read_list(int fd,
    int       mem_list_count,
    void *    mem_offsets[],
    int       mem_lengths[],
    int       file_list_count,
    int64_t   file_offsets[],
    int32_t   file_lengths[])
```

This interface allows a set of buffers to be used as read or write destinations in memory on the client and a set of offsets in the file on the I/O node. Noncontiguity in both the file and the memory is thus possible.

A naive implementation of list I/O would translate a list I/O request into a set of individual requests, each of which accesses one contiguous piece separately. Obviously, this would provide no advantages for list I/O.

PVFS has designed and implemented its list I/O in an efficient manner as described in [5]. The `pvfs_read_list` and `pvfs_write_list` functions take list I/O parameters and perform the noncontiguous access in a single PVFS operation. The current implementation is based on TCP/IP, a stream-based transport layer, therefore the buffer offset-length pairs are not required by the I/O nodes. When an I/O node receives a list I/O request with a number of file offset-length pairs, it services them individually; merge happens only when the actual file accesses from the same compute node are contiguous with each other.

This implementation effectively reduces the number of request and reply message pairs and increases data amount transfered in each pair of request and reply messages, significantly improving the data transfer efficiency. However, as it is based on TCP/IP, noncontiguous data transmission is not considered as an issue. Also, the I/O accesses to the local files in each I/O node are processed separately. Given a PVFS list read operation as shown in Figure 1, the I/O nodes read each contiguous piece of data from the local files. After each read, they initiate a socket `write` operation to send data to client processes. Each client process reads data into 4 different buffers using 4 operations. In total, 8 `read` and `lseek` operations are used for file accesses. 8 socket `write` and `read` operations are used to transfer data between the client processes and the server I/O nodes.

3.2 Network Support for List I/O

Many conventional communication interfaces, including TCP/IP, only support data transmission in contiguous blocks, defined by a memory address and a length. Based on these interfaces, to move data from and into a list of buffers specified in the PVFS list I/O, two schemes can be used. The first scheme is to send and receive one message for each contiguous block of data. The second scheme is to pack noncontiguous data into a temporary buffer before transmitting it, and unpacking it when it has arrived.

Both schemes have been widely used for noncontiguous data transmission. For example, the current PVFS list I/O design follows the first scheme using the socket interface over TCP/IP, and the second scheme is a generic method deployed in MPICH [9] to transmit noncontiguous data. Communication performance suffers in the first scheme since the message startup costs accrue for each message. Furthermore, the data size in each message is small. In the second scheme, one additional copy is required on both the send and the receive sides; however, it does use one large message to transfer all data.

Performance issues in noncontiguous data transmission are often ignored in conventional networks because of their high overhead and low bandwidth. The message startup

costs or the extra memory copy overheads do not have much impact on the communication performance when the network is comparatively slow. However, in low overhead and high bandwidth networks such as InfiniBand, these overheads have a significant impact on performance. For example, in our InfiniBand testbed, the network bandwidth is 820 MB/s and memory copy bandwidth is 1300 MB/s, therefore a scheme to pack, send, and unpack data can offer an aggregate bandwidth of only 362 MB/s.

Due to the emergence of high-performance networks, traditional methods used for noncontiguous data transmission become very inefficient. In section 4, we address how we can achieve efficient noncontiguous data transmission for list I/O over high-speed networks.

3.3 Disk Operations for List I/O

As shown in Figure 1, file regions specified by the offset-length pairs in a list I/O request may not be contiguous, as is found in many parallel applications. Thus even though an I/O node can receive a large number of file accesses in one list I/O request, if these accesses were serviced individually, the performance would be quite low. There are three major factors which affect performance of file accesses on the I/O node:

- Most file systems favor large accesses; small requests can not obtain peak performance.
- The cost of making many read/write system calls, each for small amounts of data, is extremely high, even when caching is performed by the file system [28].
- Minimizing file seeks is important to maximize performance when doing multiple file accesses.

To reduce the effects of these factors, it is critical to make as few requests to the file system as possible, generating as many large accesses to the file system as possible. Data sieving is such a technique that enables an implementation to make a few, contiguous requests to the file system even if the user's request consists of several small, noncontiguous accesses [26]. However, data sieving reduces the number of file access calls and increases file access sizes at the cost of extra data read/written. For write data sieving, costs are also paid to perform read-modify-write and synchronization to prevent concurrent updates to the same file region. The extra data may actually be good for later requests, though [27].

In section 5, we address the feasibility of performing data sieving on the I/O node to service list I/O requests and describe a cost model used by the I/O nodes to decide whether to perform data sieving or to access each contiguous piece of data separately in the presence of the advantages and disadvantages given above.

4 Noncontiguous Data Transmission

PVFS list I/O allows a set of discrete memory buffers to be used as read or write destinations in memory on the client. A typical example of such buffers is rows in a subarray of a multidimensional array, separated by gaps (*noncontiguous buffers*). As previously noticed [31], buffers on the I/O nodes are usually contiguous. An important issue is to transfer data between PVFS list I/O buffers on the compute nodes and buffers on the server nodes.

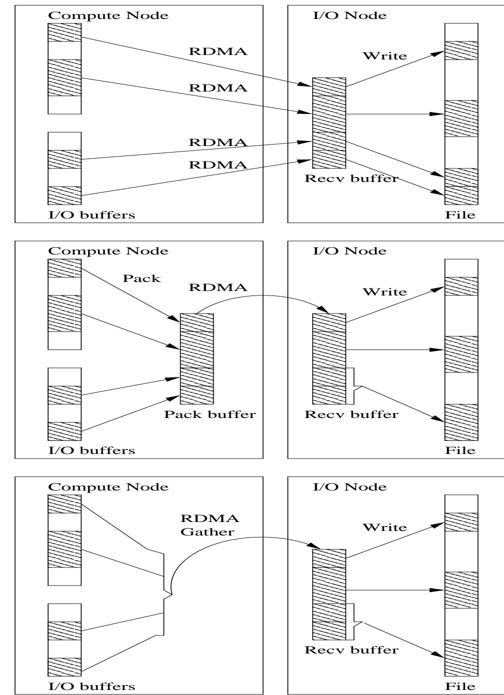

Figure 2. Noncontiguous data transfer. *Top: Multiple Message. Middle: Pack/Unpack. Bottom: RDMA Gather/Scatter.*

4.1 Mechanism Tradeoffs

As discussed in section 3.2, two schemes have been widely used to transfer noncontiguous data: 1) send and receive one message for each contiguous block of data, 2) pack noncontiguous data into a temporary buffer before transmitting it, and unpack it after its arrival. We call them *Multiple Message* and *Pack/Unpack*, respectively. The top two panels in Figure 2 illustrate these schemes.

A third way exists to transfer noncontiguous data in modern communication networks such as InfiniBand that support RDMA Gather/Scatter operations. RDMA Write operations can gather multiple data segments together within one operation and place them in a single buffer on the receiver side. RDMA Read operations can read data from a single buffer on the peer side into multiple buffers on the local initiator. This gather/scatter functionality is a perfect match with the requirement of PVFS list I/O noncontiguous data transfer. The bottom panel in Figure 2 shows an example of RDMA gather write. In this *RDMA Gather/Scatter* scheme, the message startup costs which occur in the Multiple Message scheme can be reduced dramatically, since a large number of data segments can be specified in one operation. It also avoids data copies which are required in the Pack/Unpack scheme.

There are many tradeoffs among the three schemes, however, which complicates the design decision about when to use a particular scheme. These are listed in the following paragraphs.

Copy or memory registration. Buffers must be registered before any data transmission occurs in InfiniBand. This requires that all list I/O buffers be registered in both the Multiple Message and the RDMA Gather/Scatter schemes,

and that the temporary buffer in the Pack/Unpack scheme be registered. Sometimes it is desirable to unregister these buffers after the completion of noncontiguous I/O access as well. A tradeoff exists between choosing to accept the overhead of an extra copy versus the overhead of memory registration and possible deregistration.

Communication startup overhead. The number of communication operations is different in these three schemes. In the Multiple Message scheme, it is equal to the number of list I/O buffers. In the Pack/Unpack scheme, only one transfer is required. In the RDMA Gather/Scatter scheme, some number of segments, 64 currently in InfiniBand, can be gathered into a single communication. Choosing fewer, larger messages results in better performance.

Buffer alignment. Networks which use RDMA are sensitive to buffer alignment and can generate large delays to compensate for misaligned buffers. Since the Pack/Unpack scheme itself allocates a temporary buffer for RDMA operations, this buffer can be aligned. However, it is possible that list I/O buffers given by users may not be aligned and cause the performance of the Multiple Message and RDMA Gather/Scatter schemes to suffer.

Application buffer access patterns. The costs of memory registration and deregistration can be amortized across multiple operations by registration caching mechanisms such as pin-down cache [10]. But if the application chooses buffers in such a way that caching is not very frequent, performance of the Multiple Message and RDMA Gather/Scatter schemes might be hurt. It is likely that a Pack/Unpack implementation will reuse the same buffer and not be affected.

Since it is clear that the Multiple Message scheme will likely perform poorly compared to the other two, it is ignored now for clarity. From the tradeoffs listed above, though, it is not clear which of the remaining two schemes will be better. The answer depends on the total effects of the above factors in each scheme. We use the following test to show the performance of noncontiguous data transmission with these two schemes.

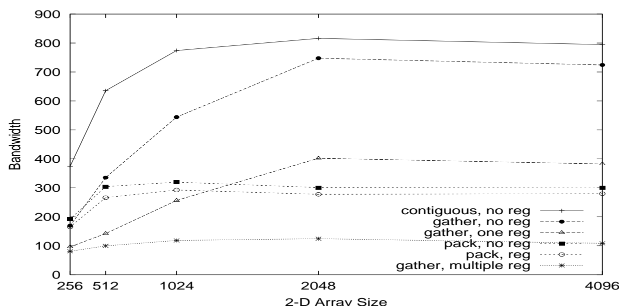

Figure 3. Bandwidth achieved in various transfer schemes.

In Figure 3, we show the bandwidth achieved in transferring a 2-D subarray from a compute node to an I/O node in our testbed. We consider the following scenario which is a common case of I/O access patterns in scientific applications. A 2-D array of varying size is distributed across 4 processes using a block distribution in both dimensions. One of the subarrays is then sent using different schemes.

In the Pack/Unpack scheme, the temporary buffer can be allocated from a pre-registered buffer pool or from the system. In the former case, registration and deregistration are not needed. These two cases are termed as *pack, no reg* and *pack, reg*, respectively. In the RDMA Gather/Scatter scheme, two ways to register list I/O buffers are considered. One is to register each list I/O buffer separately, termed as *gather, multiple reg* in the graph. Another is to register the memory region which covers all list I/O buffers from a subarray, termed as *gather, one reg*. We also show its best case, where memory registrations are always found in the cache, called *multiple, no reg* in the graph. Finally, the maximum achievable bandwidth obtained by a single write is labeled *contiguous, no reg* in the graph.

Several observations can be made from Figure 3. First, the packing and memory registration costs have a dramatic impact on performance. Second, the Pack/Unpack scheme performs comparatively better when the array size is small, Third, the RDMA Gather/Scatter scheme has the potential for high performance if registrations are handled well.

The above test results show that the RDMA Gather/Scatter scheme is very promising when the costs of memory registration and deregistration can be controlled in a certain range. The issue of how we can reduce the costs of memory registration and deregistration is addressed in the following subsection.

4.2 Minimizing Memory Registration Overhead

As seen in Figure 3, if the costs of memory registration and deregistration on the list I/O buffers could be reduced, noncontiguous data transmission can be done in a very efficient way by using the RDMA Gather/Scatter scheme. However, it is not trivial to reduce these costs. The complication comes from the number of registration and deregistration operations on list I/O buffers and the total size of memory space to be registered and deregistered.

A large number of buffer registration and deregistration events occur for each PVFS list I/O operation without careful design. Take the above test for example, if the subarray is from a 4096×4096 array, there are 2048 buffers, one for each row, that must be registered and deregistered. This may result in serious performance problems, for many reasons. First, it results in high registration overhead, since it is likely that some of the individual rows are not already registered. On our InfiniBand testbed, 1 020 μs is required to register and deregister 100 buffers of size 4 kB each. Second, the total number of buffers registered is limited. When the system hits this limitation, some registered buffers must be deregistered. This may lead to registration thrashing. Thus, reducing the number of buffers needed to be registered as much as possible is critical to alleviate these problems.

On the other hand, the total size of memory space to be registered and deregistered should also be considered. We could register or deregister the whole memory region which covers all list I/O buffers with a single operation. The total amount of memory registered increases because even unused areas are included. But the benefit of reducing the number of buffer registration calls in this way may not arise

because more time is required to perform the registration as the memory size grows.

Based on these observations, the reigning design principle which dictates how to perform memory registration and deregistration for PVFS list I/O buffers is to reduce the number of buffers as much as possible, while also minimizing the total size of memory regions. There are several design alternatives, which can be separated into two classes, depending on whether the application must be changed or not.

4.2.1 Application-aware memory registration

The first class of design alternatives requires changes to the application to allow it to take a more active role in memory registration.

First, the PVFS application can be given explicit control of this task and must call routines in the PVFS library to register regions which it plans to use with PVFS. A similar approach is taken in [6]. This suffers from the obvious drawback of putting more work into the application layer, and disallows some optimizations by the library, such as using inline data transfer where registration would not be required.

Second, one could consider not requiring full control by applications, but just asking them to specify to the PVFS library the *actual allocation* which was used to generate buffers in a list I/O call. For example, in the above subarray example, the actual allocation buffer address is the initial address of the whole two-dimensional array, with length of the entire array. This permits PVFS to optimize for the common case of an application using *malloc* to create an array, then sending pieces of that array using list I/O. This suffers the same drawback of requiring application modification, but is not quite as invasive as the previous scheme.

4.2.2 Library-controlled memory registration

If we reject the above schemes on the grounds that they change current PVFS semantics and require application changes, there are still other mechanisms by which the library itself can try to optimize memory registration. These require no interface changes, but now the PVFS library is not aware of how the application memory is arranged: a valid list I/O operation may use memory regions from widely disparate areas of the application virtual memory space.

The first, naive scheme is to register the entire memory region which covers all the list I/O buffers. Although the number of registration operations is thus reduced to one, there are two practical problems. The "holes" between two buffers may not really have been allocated by the application, causing the registration call to fail. Or, even if the whole memory region has been allocated, the total size of "holes" may be so large that no benefit is gained over simply registering each of the buffers separately.

The second scheme is to group list I/O buffers into several memory regions. This scheme overcomes the problems in the naive scheme by the following two steps. In the first step, it controls the sizes of memory regions which are going to be registered by sorting and grouping buffer regions to avoid attempting to allocate truly large "holes" of memory between buffers. This avoids the failure of the naive scheme to gain any benefit over individual registration. In the second step, it queries the operating system to find out if a "hole" which was not rejected by the first step is actually in the process allocated memory space and thus safe to register. Where they are not, buffers again must be registered independently.

The third scheme is a combination of the previous two. First, it sorts and groups list I/O buffers into candidate regions for registration. Then, it optimistically attempts to register each memory region as the first scheme does, but if the operating system denies one of these registrations, it must query the operating system to find out actual boundaries of application memory allocation and register exactly those. We call this scheme *Optimistic Group Registration*. It is quite efficient in the common case where all list I/O buffers come from one or more bigger buffers, but is also safe by virtue of relying on queries to the operating system if it must.

4.3 Our Design and Implementation

We use the Optimistic Group Registration scheme for two reasons. First, in common cases, list I/O buffers are from one encompassing allocation, such as rows in a subarray of a multidimensional array. Second, it is transparent to PVFS applications. There are three steps in this scheme: 1) group list I/O buffers into candidate regions, 2) optimistically register each region, and 3) to filter out "holes" which resulted in registration failures, if any.

The following equation is used to sort and group list I/O buffers. The cost of registering a buffer is modeled as $T = a \times p + b$, where a is the registration cost per page, b is the overhead per operation, and p is the size of the buffer in pages. The same cost equation can be applied to deregister a buffer with different values of a and b. In our testbed, we found the costs per page in buffer registration and deregistration to be 0.77μs and 0.23μs, respectively. The overheads per registration and deregistration operations are 7.42μs and 1.1μs, respectively. According to this cost model, a tradeoff can be made between the number of operations and the buffer size. In our implementation, we compare the cost to register a large combined region which includes extra unneeded "holes" against the cost to perform multiple small regions to determine candidate groupings.

These candidate memory regions are optimistically registered, one at a time, in the second step. If all registration operations are successful, the procedure is finished. This is the common case in most applications.

When an optimistic registration fails, if there are not too many buffers inside the failed region, we simply allocate them as given. But if there are many buffers which would make that too expensive, we query the operating system to find the "true" holes in virtual memory space. There are a few ways to find out this information. In Linux, one can read from the file /proc/$pid/maps, but that is quite slow. Instead we added a system call which walks the virtual memory structures in the kernel to find the same information. This system call requires about 70 μs when querying about 1000 holes compared to 1100 μs when reading from /proc. A third mechanism is system independent and involves using signal handling to catch segmentation violations while

reading from one word on each page in order to find if the pages are resident or not. Alternatively, some systems support the *mincore* system call which perhaps will provide the same information. We have investigated these last two and plan to use them to deploy PVFS on other systems.

With the Optimistic Group Registration scheme, RDMA Gather/Scatter works well in most cases, especially for large data transfers. When the total size of noncontiguous data regions is not large, decreasing the copy overhead is not important, but increasing the request size is. We decide to use the Pack/Unpack to transfer noncontiguous data when the total size of data is not large than the default PVFS stripe size (64 kBytes). There are several reasons for this choice. First, as seen in Figure 3, when the pack size is less than 256 kBytes, *i.e.* in a 1024×1024 array, Pack/Unpack is still beneficial. Second, in our PVFS implementation over InfiniBand [31], when the transfer size is less than 64 kBytes, Fast RDMA is used. Noncontiguous data is packed into the Fast RDMA buffer on the compute node and then transferred to the I/O node for writes. For reads, the I/O node RDMA writes data into the Fast RDMA buffer on the compute node, then the compute node unpacks data into the list I/O buffers.

5 Efficient Noncontiguous File Access on the I/O Node

In this section, we set out to answer this question: *how a single I/O node can efficiently perform file accesses to service a list I/O request*. This is the second issue in noncontiguous I/O access. Combined with efficient noncontiguous data transmission as discussed in Section 4, a complete solution is achieved that can offer high performance noncontiguous I/O access in PVFS.

5.1 Active Data Sieving on the I/O node

As previously mentioned, in many parallel applications, each process tends to access a number of relatively small, noncontiguous portions of a file. Information of a large number of file accesses can be sent in one single list I/O request; however, performance would suffer if these accesses were serviced separately, as we discussed in Section 3.

We propose to apply data sieving on the I/O node to process PVFS list I/O requests. We refer to this as *Active Data Sieving (ADS)*. In data sieving, for read, instead of reading each contiguous portion separately, the I/O node reads a large contiguous chunk into a temporary buffer. It then transfers the desired data to the list I/O buffers on the compute node. For write, a read-modify-write is performed locally. First, the I/O node reads a large contiguous chunk of data into a temporary buffer. Second, it copies data received from the compute nodes into appropriate locations in the temporary buffer. Then, it writes that temporary buffer back to the file. In general, the portion of the file being accessed must be locked to prevent concurrent updates, but that is not a concern here since the I/O node has exclusive access to its own local data. Active Data Sieving fits nicely with scatter/gather capable networks such as InfiniBand where just parts of the buffer can be sent naturally to the requesting client.

When a list I/O request arrives, the I/O node analyzes all file accesses in the request and decides whether it is beneficial to apply data sieving to process these accesses or not. A cost model could be very comprehensive and complicated to take access patterns, system parameters, disk characteristics and cache effects into account. Here, we deploy a simple but effective model in the design of ADS on the I/O node. The following parameters in table 1 are considered.

Table 1. Model Parameters

System Parameters	
B_{mem}	Memory bandwidth
$B_r(s)$	File read bandwidth without cache for size *s*
$B_w(s)$	File write bandwidth without cache for size *s*
O_r	File read overhead
O_w	File write overhead
O_{seek}	File seek overhead
O_{lock}	File lock overhead
O_{unlock}	File unlock overhead
Request Parameters	
N	Number of noncontiguous accesses
S_i	Size of the i*th* file access
S_{req}	Total size of wanted data
S_{ds}	Total size of data sieving data

Given these parameters, the costs to serve a PVFS list read and write without data sieving and with data sieving are represented by T_r, T_w, T_{dsr}, and T_{dsw}, respectively. In T_{dsw}, the costs for read, modify and write with synchronization are included.

$$T_{read} = N \times (O_r + O_{seek}) + \sum_{i=1}^{N} \frac{S_i}{B_r(S_i)}$$

$$T_{write} = N \times (O_w + O_{seek}) + \sum_{i=1}^{N} \frac{S_i}{B_w(S_i)}$$

$$T_{dsr} = O_r + O_{seek} + \frac{S_{ds}}{B_r(S_{ds})}$$

$$T_{dsw} = T_{dsr} + \frac{S_{req}}{B_{mem}} + O_{lock} + O_w + \frac{S_{ds}}{B_w(S_{ds})} + O_{unlock}$$

This model gives a conservative estimate of costs for data sieving, since cache effects are not taken into account. This indicates that with this cost model, if data sieving is chosen by the I/O node, it is highly probable that data sieving is beneficial due to the added effect of caching discussed earlier.

5.2 Why not Just Use ROMIO Data Sieving?

Data sieving is used in ROMIO to handle independent noncontiguous accesses [28] on each process. In the latest ROMIO implementation over PVFS, it performs data sieving for noncontiguous reads if the corresponding hints are enabled. However, it does not perform data sieving for noncontiguous writes since the current PVFS does not support file locking. Although both ADS and ROMIO data sieving over PVFS tend to reduce the number of I/O calls to the local file system in the I/O node and to increase I/O access sizes, ROMIO data sieving is a *client* side implementation

of data sieving, while ADS is a *server* side implementation of the same task. Particularly, there are a few key differences between the two.

First, list I/O with ADS enables list I/O write operations to take advantage of data sieving benefit. Second, the undesired data is not transfered through the network in list I/O with ADS. Third, data copies can be avoided on the compute node with ADS. In addition, ADS is more capable of making better decisions on whether to perform data sieving or access contiguous data segments separately since in ROMIO data sieving, the compute node may be not aware of underlying file system data layouts and parameters.

MPI-IO applications can choose to use either list I/O with ADS or ROMIO Data Sieving by setting file system hints to override the defaults.

6 Performance Results

This section presents performance results from a range of benchmarks on our implementation of PVFS over InfiniBand. Based on our previous work [31], we added noncontiguous data transmission and active data sieving. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [17], which is a user-level programming interface developed by Mellanox and compatible with the InfiniBand Verbs specification. We use both PVFS and MPI-IO micro-benchmarks as well as applications to quantify our design choices in noncontiguous data transmission and noncontiguous file accesses. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

6.1 Experimental setup

Our experimental testbed consists of a cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.1.2-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. Each node has a Seagate ST340016A, ATA 100 40 GB disk. We used the Linux RedHat 7.2 operating system.

6.2 Network and File System Performance

Table 2 shows the raw 4-byte one-way latency and bandwidth of VAPI and our MVAPICH [16], a MPICH implementation over InfiniBand. Table 3 shows the read and write bandwidth of an *ext3fs* file system on the local 40 GB disk with and without cache effect. The *bonnie* [11] file-system benchmark is used.

Table 2. Network performance

	Latency (μs)	Bandwidth (MB/s)
VAPI RDMA Write	6.0	827
VAPI RDMA Read	12.4	816
MVAPICH	6.8	822

Table 3. File system performance

	Write (MB/s)	Read (MB/s)
without cache	25	20
with cache	303	1391

It can be seen that there is a large difference in bandwidth realizable over the network compared to that which can be obtained to a disk-based file system without cache effects. However, applications can still benefit from fast networks for many reasons in spite of this disparity. Data is frequently in server memory due to file caching and read-ahead when a request arrives. Also, in large disk array systems, the aggregate performance of many disks can approach network speeds. Caches on disk arrays and on individual disks also serve to speed up transfers. Therefore, we design two sets of experiments. One is to take cache effects into account and to stress the network data transfer independent of any disk activities. Another set of experiments is to eliminate file cache effects. Performance evaluation of noncontiguous data transmissions is based on the first set of experiments. Performance evaluation of noncontiguous file accesses is based on both sets of experiments.

6.3 Effects of Data Transfer Mechanism

We design a PVFS-level micro-benchmark to show the effects of the design choice whether to use Pack/Unpack or RDMA Gather/Scatter to transfer noncontiguous data between the compute nodes and I/O nodes. In this test, there are four I/O nodes and four compute nodes. Each process wants to write or read variable sizes of data using PVFS list I/O operations. The number of noncontiguous data segments is set to 128. The size of each segment is equal, and varies from 128 bytes to 8 kB.

Three design choices are compared: Pack/Unpack, RDMA Gather/Scatter, and the hybrid scheme which we use in our final design. Figure 4 shows that Pack/Unpack works better when the total request size is not large, while RDMA Gather/Scatter performs better when the request size is large. The hybrid scheme we choose combines these two schemes and works well in both cases.

6.4 Optimistic Group Registration Performance

This test is designed to study the impact of Optimistic Group Registration on the PVFS list I/O performance. The test writes a 2-D integer array of size 2048×2048 into one file in row-major order. The array is distributed across 4 processes using a block distribution in both dimensions. Each process writes its subarray into the file contiguously at different non-overlapping file locations.

Four cases are considered. The first case is the ideal one where no registration is needed. This happens when all buffer registrations have been previously cached. The second case is individual registration and deregistration on each buffer. The third case is to use the Optimistic Group Registration scheme to register list I/O buffers that come from the subarray. The fourth case is similar to the third case, except that the list I/O buffers are not all part of the same large array. We take 1024 buffers from several arrays, and intentionally create 10 holes which are not allocated yet between these buffers. By this, we can see the costs for registration failures and querying the operating system in the

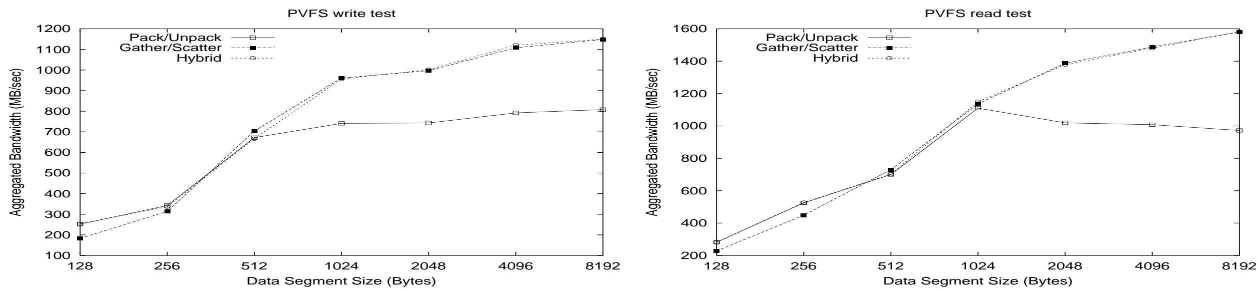

Figure 4. Performance of noncontiguous data transfer schemes.

Optimistic Group Registration scheme. We call these four test cases “Ideal”, “Indiv.” “OGR” and “OGR+Q”, respectively.

Table 4 lists the write bandwidth, the number of registrations, and the overhead for registration in each test case. Compared to the ideal case, the other three cases have 57%, 6% and 13% dregradation, respectively in write without sync. In write with sync, when disk access time is dominant, however, the overhead of memory registration and deregistration in the individual case still results in 11% degradation.

Table 4. Optimistic Group Registration Impact

case	no sync (MB/s)	sync (MB/s)	# reg	overhead (μs)
Ideal	1010	82	0	0
Indiv.	424	73	1024	5254
OGR	950	≈82	1	227
OGR+Q	879	≈82	11	496

The number of registration operations and their costs are also shown in the table. It can be observed that Optimistic Group Registration reduces costs of registration on list I/O buffers dramatically. In addition, a faster file system leads to a larger impact from memory registration and deregistration.

6.5 MPI-IO Noncontiguous Access Benchmarks

To evaluate the impact of Active Data Sieving on the performance of PVFS list I/O operations, we designed a benchmark using MPI-IO. The file view is one dimensional block column distribution. As shown in Figure 5, the file accesses are noncontiguous: each process accesses only one unit out of every four in the file. In this test, we vary the size of the array from 512 to 8192, thus the numbers of columns touched by each process changes from 128 to 2048.

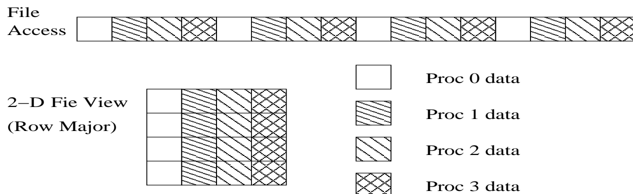

Figure 5. Accesses in the file view with one-dimensional block column distribution.

In the test, we set different hints to enable the potential techniques: Multiple I/O, ROMIO Data Sieving, and PVFS list I/O. In the PVFS list I/O method, we also test using Active Data Sieving or not. We compared the performance of these four methods for read, where the data is in cache or uncached, and write, where the data is potentially flushed to disk.

Figure 6 shows write results for each method. As mentioned earlier, ROMIO Data Sieving over PVFS for noncontiguous write is actually implemented with the Multiple I/O method, thus their performance is nearly identical. The next feature to notice is that using list I/O always outperforms ROMIO Data Sieving by a factor of anywhere from 3.5–12.1 depending on the array size. This is true both when considering just the network transfer aspects (no sync), and when considering the full time to commit the data to disk. Regarding the two list I/O curves, it can be seen that using ADS shows a significant benefit in the small array size range. Starting at 2048, the model used by the I/O node decides that there is no benefit to be gained from using ADS, hence the curves merge.

Figure 7 shows read results for each method. In these graphs it can be seen that list I/O is comparable to, or outperforms ROMIO Data Sieving. As the array size increases, ROMIO Data Sieving must transfer the whole array from the I/O nodes to the compute nodes, and its performance suffers while list I/O selectively transfers only the data required by each compute node. Using ADS again improves performance in the small array cases. In the read without cache case, ROMIO Data Sieving is comparable to list I/O for a wide range of array sizes, up to 2048, because disk access time dominates in this range. Eventually the overheads of reading three times as much data and sending it across the network catch up with ROMIO Data Sieving and its performance falls off. However, list I/O with ADS can decide that there is no benefit to perform data sieving and then accesses each file region separately in the large array cases.

6.6 MPI-IO Tiled Access Test

The test application *mpi-tile-io* [21] implements tiled access to a two dimensional dense dataset. This type of workload is seen in visualization applications and in some numerical applications. For our tests, we used four compute nodes and four I/O server nodes. Each compute node renders to one of a 2×2 array of displays, each with 1024×768 pixels. The size of each element is 24 bits, leading to a file

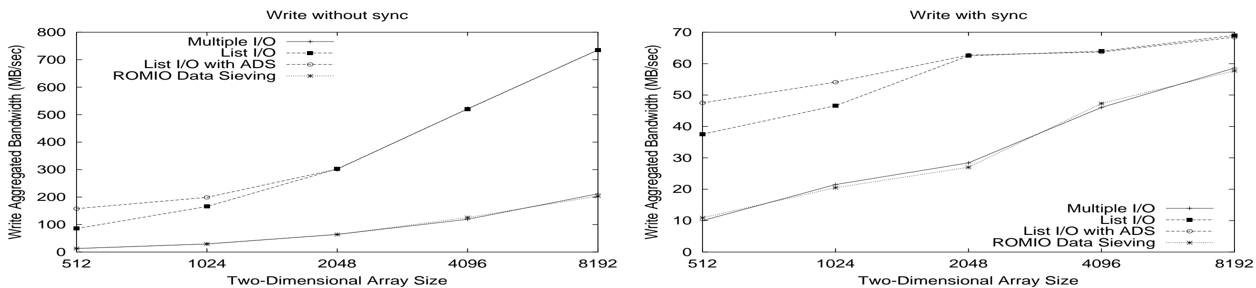

Figure 6. Write results with different methods in the block-column file view.

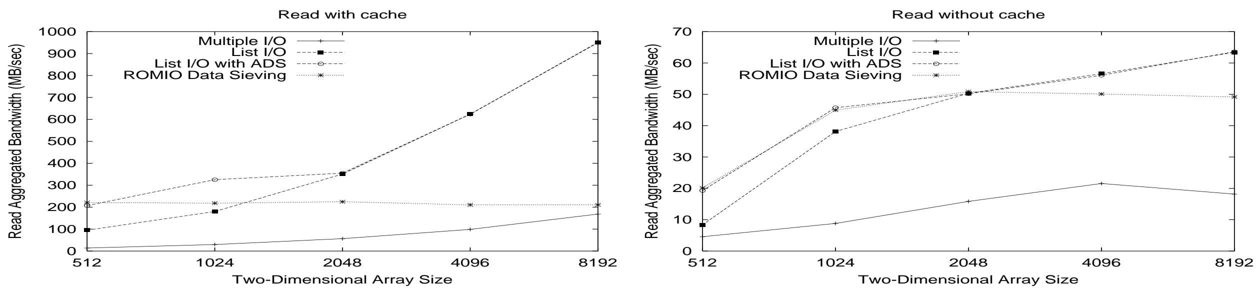

Figure 7. Read results with different methods in the block-column file view.

size of 9 MB.

The access pattern in this test is noncontiguous in file space but contiguous in memory. We consider the same four cases as in the previous test: Multiple I/O, List I/O, List I/O with ADS, and ROMIO Data Sieving for both read and write, again either considering or ignoring disk effects.

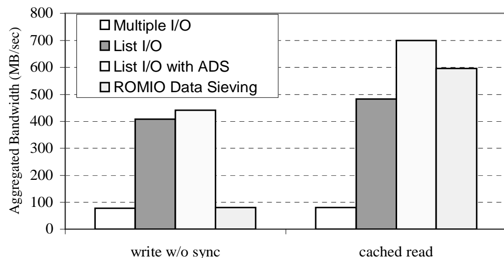

Figure 8. Tiled I/O, without disk effects.

Figure 8 shows the results for the four test cases when data is written without sync and read from the file cache. Compared to the Multiple I/O case, List I/O with ADS has a factor 5.7 improvement for write, and 8.8 for read. Compared to the List I/O case, it has 8.4% improvement for write, and 45% improvement for read. Write is more costly than read with ADS due to the need to perform a read-modify-write cycle. Compared against ROMIO Data Sieving, list I/O with ADS still has a factor of 5.7 improvement for write, but just 18% improvement for read.

Figure 9 shows the results for the four test cases when the disk is the bottleneck in data transfers. For write, list I/O with ADS still outperforms the other methods. For read, ROMIO Data Sieving now outperforms list I/O with ADS. There are two reasons. First, the increased network transfer time in ROMIO Data Sieving does not matter when the

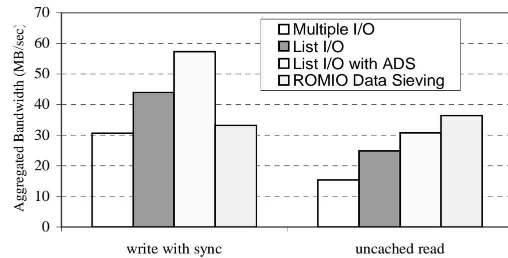

Figure 9. Tiled I/O, with disk effects.

disk dominates. Second, list I/O with ADS generates 6 pairs of request and reply messages, compared to just one with ROMIO Data Sieving, adding more overhead to the entire operation.

It is expected that the list I/O with ADS will improve with increasing number of file accesses in one list I/O request. Currently, we use the default value in PVFS which is 128, but a larger number can be used to decrease the number of request and reply pairs needed to complete the operation.

6.7 NAS BTIO Benchmark

The BTIO benchmark was recently added into the 2.4 version of NAS Parallel Benchmarks (NPB) and is used to test the output capabilities of high-performance computing systems, especially parallel systems. It is based on the Block-Tridiagonal problem of the NPB Suite. The details of the numerical algorithm, data partition, and data distribution can be referred to [20].

There is a very high degree of fragmentation in data sets of the BT problem. The main access pattern in BTIO is noncontiguous in memory and in the file. Thus, this test can be used for us to quantify our design choices in both noncontiguous data transmission and noncontiguous file accesses.

Results for a class A problem size are shown in Table 5, where we show the total problem execution time and the I/O overhead, which is the amount of time the benchmark spends performing I/O operations. It can be seen that list I/O with ADS performs best even for this complex application.

Table 5. BTIO Performance

case	Time (s)	I/O overhead (s)
no I/O	165.6	0
Multiple I/O	180.0	14.4
Collective I/O	169.6	4.0
List I/O	168.2	2.6
List I/O with ADS	167.7	2.1
Data Sieving	177.3	11.7

We profiled the I/O characteristics of this test for the above five I/O methods. Details are presented in Table 6. In both list I/O cases, the number of request messages is reduced to 1360, a significant reduction compared to the Multiple I/O method (163840) and the Data Sieving method (82040). A similar reduction is seen in the number of memory registration operations due to Optimistic Group Registration. In List I/O with ADS, the number of the file access call pairs (lseek, write) and (lseek, read) on each I/O node is also reduced to 7680, compared to Multiple I/O (163840), List I/O (163840), and ROMIO Data Sieving (85060). This reduction is attributed to Active Data Sieving on the I/O node.

This demonstrates that our proposed RDMA Gather/Scatter transmission mechanism with the Optimistic Group Registration can reduce the number of registration and deregistration operations and increase the registration cache hit effectively. Active Data Sieving reduces I/O calls to the I/O node file system significantly and increases I/O access size dramatically.

Table 6. Characteristics in each test case

	Mult.	Coll.	List	ADS	DS
client request characteristics					
req #	163840	160	1360	1360	82040
reg #	163840	160	1360	1360	82040
reg cache hit	52000	158	1324	1324	26039
disk access characteristics					
read #	81920	1600	81920	5120	3140
write #	81920	1600	81920	2560	81920
communication between the compute and I/O nodes					
size (MB)	200	200	200	200	490
communication between the compute nodes for I/O					
size (MB)	0	150	0	0	0

7 Related Work

Noncontiguous data transmission is traditionally implemented with a list of contiguous data transmissions or the Pack/Unpack scheme [9]. Worringen *et al.* [30] used remote memory operations provided in the SCI network to send noncontiguous datatypes in MPI. Their work is based on memory copy semantics. We explore an RDMA approach in this paper.

Memory registration and deregistration are important issues in modern networks which provide RDMA capabilities. Work in [29] and [33] focus on schemes to reduce overheads of system memory registration and deregistration operations. Tezuka *et al.* [10] propose a pin-down cache to reduce memory registration and deregistration overhead. In our work, we propose a novel, general scheme, Optimistic Group Registration, to reduce costs of registration and deregistration on a list of buffers for noncontiguous data transmission.

A few papers have addressed optimizing noncontiguous disk access in file and storage systems, including data sieving [26], two-phase I/O [7], server-directed I/O [23], disk-directed I/O [14], and data-shipping and controlled prefetching [13]. Our work in optimizing noncontiguous disk accesses is close to server-directed I/O in the sense that the I/O node performs optimizations. However, server-directed I/O focuses on cooperation between the I/O servers to perform collective I/O operations. Our work focuses on active and intelligent data sieving for independent noncontiguous I/O accesses.

We implemented a version of PVFS over InfiniBand in work [31]. Ching *et al.* [5, 1] implemented PVFS list I/O and evaluated their implementation over TCP/IP. Latham *et al.* [15] examined the performance problems in PVFS and ROMIO for noncontiguous I/O access. However, both the noncontiguous data transmission issue and the noncontiguous disk access issue were not addressed.

8 Conclusions and Future Work

Parallel scientific applications, data mining applications, and visualization engines all need high performance parallel file systems. The access patterns generated by many of these sometimes tend to be many small accesses scattered widely across a striped file, a model which has to date not been well supported. The advent of recent interconnects such as InfiniBand which are capable of scatter/gather remote direct memory access permit the use of new techniques to make such noncontiguous accesses significantly better. Furthermore, network is not the bottleneck in cluster file systems any more. Disk access performance should be further improved to stress the network, especially for noncontiguous disk accesses.

In this paper, we address two issues involved in noncontiguous I/O accesses in cluster file systems over high performance networks: noncontiguous data transmission and noncontiguous disk accesses. For noncontiguous data transmission, we propose a novel approach, *RDMA Gather/Scatter*, to transfer noncontiguous data between the clients and the I/O servers. Associated with this approach, we propose a new registration scheme, *Optimistic Group Registration*, to reduce memory registration costs. For noncontiguous disk accesses in the I/O server nodes, we have implemented a new scheme termed as *Active Data Sieving* to reduce disk access costs for a large number of small and noncontiguous accesses. Unlike other data sieving implementations, a cost model is used by the I/O nodes to actively and intelligently decide whether it is beneficial to perform data sieving or

not.

We have designed and incorporated these approaches in a version of PVFS over InfiniBand. Our results show a performance improvement of up to 1.5 times for the RDMA Gather/Scatter approach with Optimistic Group Registration on PVFS list I/O performance compared to the other approaches. Intelligent and active data sieving on the I/O node achieves a factor of 1.3–1.9 improvement on small noncontiguous I/O accesses. The NAS BTIO benchmark performance results show that our approach attains a 20% improvement compared to the best result across all other approaches in an environment which is a complex combination of noncontiguous data transmission and noncontiguous I/O accesses.

The approaches proposed in this paper for noncontiguous data transmission in noncontiguous I/O access can be used elsewhere such as for MPI noncontiguous data transfer and database multiple data segment transfer. The design of Active Data Sieving is not network specific. We plan to incorporate our current implementation into PVFS implementations over other networks.

Acknowledgments

We would like to thank the PVFS team at Argonne National Laboratory and Clemson University for giving us access to the latest versions of PVFS and for providing us with crucial insights into the implementation. We are also thankful to Jiuxing Liu and Pavan Balaji for discussion with us.

References

[1] A. Ching, A. Choudhary, K. Coloma, W.-K. Liao, R. Ross and W. Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *Proceedings of CCGrid2003*, Tokyo, Japan, May 2003.

[2] M. Bancroft, N. Bear, J. Finlayson, R. Hill, , R. Isicoff, and H. Thompson. Functionality and Performance Evaluation of File Systems forStorage Area Networks (SAN). In *the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies*, 2000.

[3] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[5] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[6] DAFS Collaborative. Direct Access File System Protocol, V1.0, August 2001.

[7] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[8] G. A. Gibson and R. V. Meter. Network attached storage architecture. *COMMUNICATIONS OF THE ACM*, 43(11), Nov. 2000.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[10] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th Int. Parallel Processing Symposium*, March 1998.

[11] http://www.textuality.com/bonnie/. Bonnie: A File System Benchmark.

[12] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24, 2000.

[13] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation on top of GPFS. In *Supercomputing 2001*, Nov. 2001.

[14] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[15] R. Latham and R. Ross. PVFS, ROMIO, and the noncontig Benchmark. http://www.mcs.anl.gov/romio/noncontig-perf.pdf, April 2003.

[16] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing*, June 2003.

[17] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 0.95, March 2003.

[18] Message Passing Interface Forum. MPI-2: A Message Passing Interface Standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[19] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[20] Rob F. Van Der Wijngaart and Parkson Wong. NAS Parallel Benchmarks I/O Version 2.4. http://www.nas.nasa.gov/Research/Reports/Techreports/2003/nas-03-002-abstract.html.

[21] R. B. Ross. Parallel I/O Benchmarking Consortium. http://www-unix.mcs.anl.gov/ rross/pio-benchmark/html/.

[22] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *First USENIX Conference on File and Storage Technologies*, pages 231–244. USENIX, Jan. 2002.

[23] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[24] E. Smirni and D. Reed. Workload characterization of input/output intensive parallel applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, June 1997.

[25] Storage Networking Industry Association. Shared Storage Model. www.snia.org/tech_activities/shared_storage_model.

[26] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[27] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[28] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *to appear in Parallel Computing*, 2002.

[29] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proc. Hot Interconnects V*, August 1997.

[30] J. Worringen, A. Gaer, F. Reker, and T. Bemmerl. Exploiting Transparent Remote Memory Access for Non-Contiguous- and One-Sided-Communication. In *Workshop on Communication Architecture for Clusters 2002 (in conjunction with IPDPS)*, April 2002.

[31] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. Technical Report, OSU-CISRC-04/03-TR, April 2003.

[32] R. Zahir. Lustre Storage Networking Transport Layer. http://www.lustre.org/docs.html.

[33] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI communication for database storage. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 257–268. IEEE Computer Society, 2002.