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 noncontiguous data transmission between the client and the I/O server in cluster file systems over a high performance network.

We propose a novel approach, RDMA Gather/Scatter, *to transfer noncontiguous data for such I/O accesses. We also propose a new scheme,* Optimistic Group Registration, *to reduce memory registration costs associated with this approach. We have designed and incorporated this approach in a version of PVFS over InfiniBand. Through a range of PVFS and MPI-IO micro-benchmarks, and the NAS BTIO benchmark, we demonstrate that our approach attains 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 consisting of *commodity storage units connected with commodity networking technologies* can provide high performance and scalable I/O support in cluster systems [3, 23, 20, 32, 33]. PVFS (Parallel Virtual File System) [3] 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 [26, 14]. 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 [2, 17, 22]. 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 [24, 7, 21, 13, 12] were proposed to optimize noncontiguous accesses in situations where only contiguous I/O access support is available. Thakur *et al.* [25] 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 [16] 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 file requests that access relatively small pieces of data in a noncontiguous manner. Efficiently 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. In this paper, we focus on the issue of noncontiguous data transmission. Due to space limitation, we leave the discussion of the second issue in [31].

The issue of noncontiguous data transmission is often ignored in conventional networks. The performance differences 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 Infini-Band [11].

In this paper, we address the issue of noncontiguous data transmission by designing PVFS list I/O over the Infini-Band network. We describe how efficient noncontiguous data transmission can be achieved for PVFS noncontiguous I/O accesses. We make the following contributions:

1. We observed that the existing methods to support non-contiguous data transmission have serious problems on the performance of noncontiguous I/O accesses in cluster file systems over high performance networks for many cases.

2. Gather/Scatter functionality in Remote Direct Memory Access (RDMA) operations offered by modern high performance networks can be used to transfer noncon-tiguous 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 ef-ficient use of RDMA Gather/Scatter for noncontiguous data transmission.

Our results show that the RDMA Gather/Scatter ap-proach with Optimistic Group Registration can achieve a factor of 1.5 improvement on PVFS list I/O performance compared to other noncontiguous data transmission ap-proaches. We have also evaluated the performance of the NAS BTIO benchmark with our implementation. The re-sults obtained show that our approaches can offer a 20% improvement over the previous best result.

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 the issue of noncontiguous data transmission on noncontiguous I/O accesses over high per-formance networks. In Section 4, we address noncontigu-ous data transmission. In Section 5, we describe our im-plementation of the PVFS list I/O. 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 [30], we exam-ined the feasibility of leveraging the InfiniBand technology to improve I/O performance and scalability of PVFS in clus-ters connected by the InfiniBand network. We focused on a software architecture which can take full advantage of In-finiBand 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, includ-ing 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 com-pute nodes. A metadata manager provides a clusterwide consistent name space to applications. In PVFS, the meta-data 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 [4].

2.2 Overview of InfiniBand

The InfiniBand Architecture [11] defines a System Area Network for interconnecting both processing nodes and I/O nodes. It provides a communication and management in-frastructure for inter-processor communication and I/O.

Both channel and memory semantics are available for transferring data. In channel semantics, send/receive op-erations are used for communication. In memory seman-tics, 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 [16], is an in-terface specifically designed for portable, high-performance parallel I/O. It acts as a higher-layer client which uses fea-tures 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 [25] 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 dif-ferent ways in which noncontiguous accesses arise, both of which pose challenges on efficient noncontiguous I/O ac-cess in PVFS. As illustrated in the example in Figure 1, the top set of communications shows *noncontiguous data trans-mission between the compute nodes and the I/O nodes*. The second source of noncontiguity is *noncontiguous disk ac-cesses*, as shown at the bottom, when I/O nodes access their local files. In this paper, we focus on the first issue.
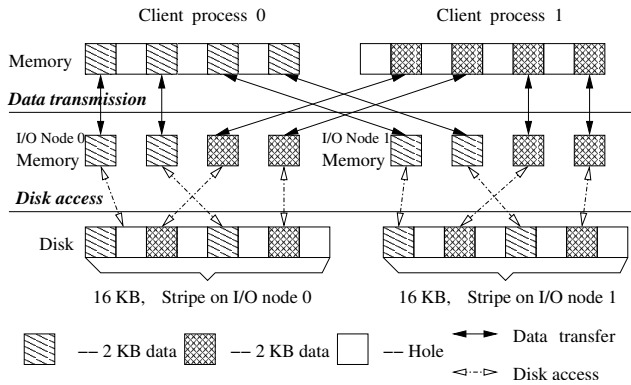
Figure 1. A PVFS list I/O example.

3.1 PVFS List I/O

PVFS provides a list I/O interface to applications which can be used to perform noncontiguous accesses in Figure 1 in a single operation. This interface confirms with the interface proposed by Thakur *et al.* in [25]. 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 [4]. 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, noncontiguous data transmission is not considered as an issue due to the stream semantics of TCP/IP.

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 are usually 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.

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 830 MB/s and memory copy bandwidth is 1300 MB/s (with cache effect) or 640 MB/s (without cache effect), therefore a scheme to pack, send, and unpack data can offer an aggregate bandwidth of only 364 MB/s or 230 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.

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 [30], 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.

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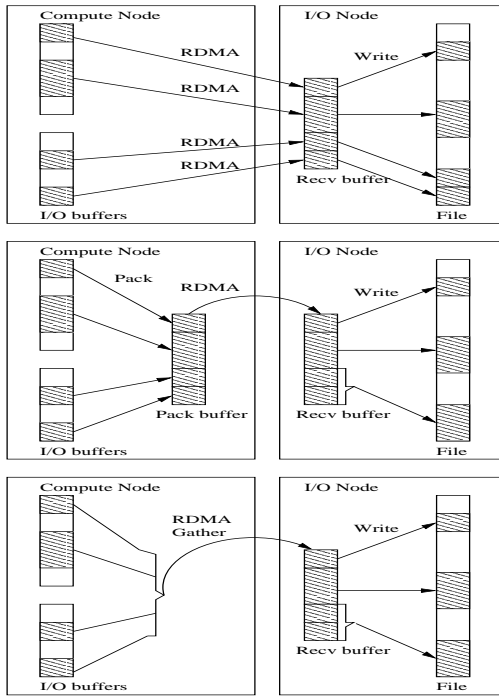
Figure 2. Noncontiguous data transfer. *Top: Multiple Message. Middle: Pack/Unpack. Bottom: RDMA Gather/Scatter.*

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 [9] and the *batched deregistration mechanism* [30]. 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.
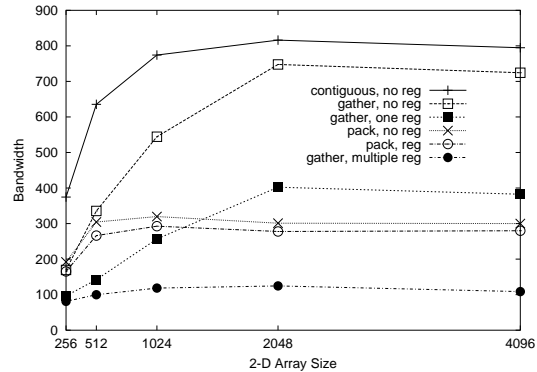

Figure 3. Bandwidth achieved in various transfer schemes.

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. 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. However, those unused areas between list I/O buffers may offset the benefit.

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 [29].

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 PVFS library can deploy several schemes to balance the number of registration and deregistration calls and the total size of registered and deregistered regions, as discussed in [31]. Here we focus on one of them, namely *Optimistic Group Registration (OGR)*, due to space limitation.

The OGR scheme first sorts and groups list I/O buffers into candidate regions for registration. It controls the sizes of memory regions which are going to be registered and deregistered. First, it groups nearby buffers, which avoids attempting to register truly large "holes" of memory between buffers. Then, it optimistically attempts to register each memory region. If the operating system denies one of these registrations due to some holes are not allocated, it must query the operating system to find out actual boundaries of application memory allocation and register exactly those.

This scheme is expected to be quite efficient in the common case where all list I/O buffers come from one or more bigger buffers and unallocated holes are rare, but is also safe by virtue of relying on queries to the operating system if it must. This scheme is also transparent to PVFS applications.

5 Implementation

In this section, we briefly describe how we implement the Optimistic Group Registration scheme and how we choose different transfer mechanisms for different cases.

5.1 Implementation of OGR

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 $0.77\mu s$ and $0.23\mu s$, respectively. The overheads per registration and deregistration operations are $7.42\mu s$ and $1.1\mu s$, 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 register 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.

5.2 Choices of Transfer Mechanisms

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 [30], 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.

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 [30], we added noncontiguous data transmission. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [15], 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. 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 1 shows the raw 4-byte one-way latency and bandwidth of VAPI. Table 2 shows the read and write bandwidth of an *ext3fs* file system on the local 40 GB disk with and without cache effect. The *bonnie* [10] file-system benchmark is used.

Table 1. Network performance

	Latency (μs)	Bandwidth (MB/s)
VAPI RDMA Write	5.8	832
VAPI RDMA Read	11.2	821

Table 2. 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.

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

Table 3 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 3. 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	$\approx$82	1	227
OGR+Q	879	$\approx$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   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
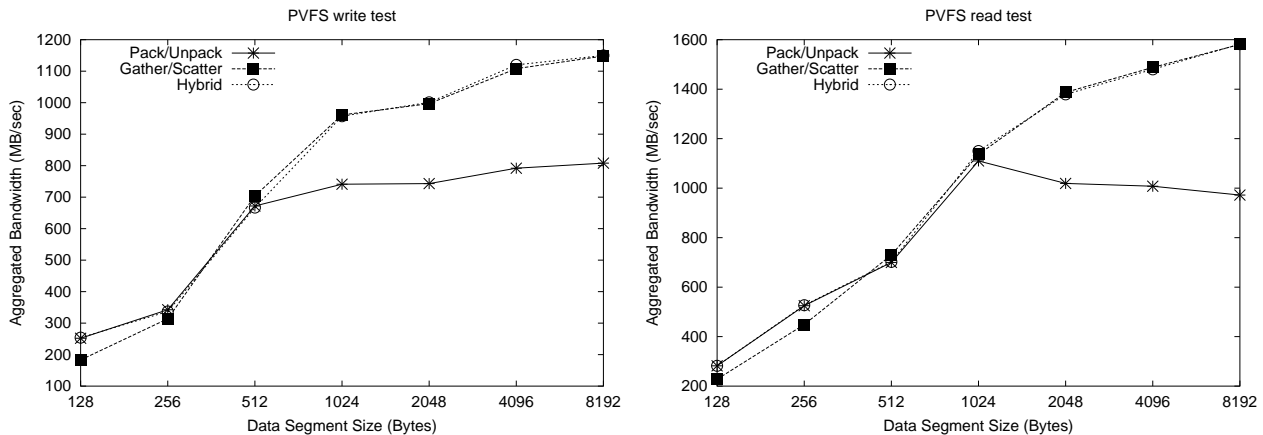
Figure 4. Performance of noncontiguous data transfer schemes.

Block-Tridiagonal problem of the NPB Suite. The details of the numerical algorithm, data partition, and data distribution can be referred to [18].

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 noncontiguous data transmission. Results for a class A problem size are shown in Table 4, 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 performs best even for this complex application.

Table 4. 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
Data Sieving	177.3	11.7

We profiled the I/O characteristics of this test for the above four I/O methods. In the list I/O case, 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.

7 Related Work

Noncontiguous data transmission is traditionally implemented with a list of contiguous data transmissions or the Pack/Unpack scheme [8]. Worringen *et al.* [28] 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 [27] and [33] focus on schemes to reduce overheads of system memory registration and deregistration operations. Tezuka *et al.* [9] 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.

We implemented a version of PVFS over InfiniBand in work [30]. Ching *et al.* [4, 1] implemented PVFS list I/O and evaluated their implementation over TCP/IP. Latham *et al.* [14] examined the performance problems in PVFS and ROMIO for noncontiguous I/O access.

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.

In this paper, we address one of issues involved in noncontiguous I/O accesses in cluster file systems over high performance networks: noncontiguous data transmission. 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.

We have designed and incorporated this approach 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. Optimistic Group Registration effectively reduces memory registration costs. The NAS BTIO benchmark performance results show that our approach attains a 35% improvement compared to the best result across all other approaches.

As a future work, we plan to combine MPI datatype structure information and buffer registration information to

reduce the size of request messages for noncontiguous I/O accesses. Work in [19] and [5] motivates us to follow this direction. The approach 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.

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] 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.

[3] 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.

[4] 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.

[5] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In *Euro PVM/MPI*, 2003.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing, (28)1:83–105*, 2002.

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

[28] 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.

[29] J. Wu and D. K. Panda. MPI-IO on DAFS over VIA: Implementation and Performance Evaluation. In *Workshop on Communication Architecture for Clusters 2002 (in conjunction with IPDPS)*, April 2002.

[30] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[31] J. Wu, P. Wyckoff, and D. K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. Technical Report, OSU-CISRC-05/03-TR, May 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.