MPI-IO on DAFS over VIA: Implementation and Performance Evaluation*

Jiesheng Wu and Dhabaleswar K. Panda
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

Abstract

In this paper, we describe an implementation of MPI-IO on top of the Direct Access File System (DAFS) standard. The implementation is realized by porting ROMIO on top of DAFS. We identify one of the main mismatches between MPI-IO and DAFS is memory management. Three different design alternatives for memory management are proposed, implemented, and evaluated. We find that memory management in the ADIO layer performs better in situations where the DAFS Provider uses Direct data transfer to handle I/O requests. For the other case of Inline data transfer, it may hurt performance. We propose that the DAFS Provider can expose such implementation information for applications to take full advantage of Inline and Direct data transfers and memory management. Comparative analysis of MPI-IO performance over DAFS, network file system (NFS) and local file system (LFS) shows that MPI-IO on DAFS over VIA on cLAN performs 1.6-5.6 times better than on NFS over UDP/IP on cLAN. The performance of MPI-IO on DAFS is found to be comparable to the performance on local file system. Additional experiments show that MPI-IO nonblocking I/O primitives implemented by DAFS nonblocking operations can completely overlap I/O and computation. These results show that MPI-IO on DAFS can take full advantage of DAFS features to achieve high performance I/O over VI Architectures.

1. Introduction

In recent years, user-level communication systems and memory-to-memory networks are increasingly used in Network Based Computing platforms and data center environment. The Virtual Interface (VI) Architecture [5] and InfiniBand (IB) Architecture [10] standardize these communication systems for inter-processor-communication and I/O. The Direct Access File System (DAFS) [6, 7], a new local file-sharing standard, is designed to provide applications with high-throughput, low-latency access to shared file servers over memory-to-memory networks with VI-compliant capabilities. The DAFS Protocol is based on NFS version 4 [13], with new features for direct data transfer, asynchronous operations, scatter/gather list I/O, and locking and recovery features for a data-center or cluster environment. It takes full advantage of memory-to-memory networks such as cLAN [8] with remote DMA support to provide high performance file I/O, improved CPU utilization and reduced system overhead due to data copies, user/kernel context switches, thread context switches, and network protocol processing. DAFS is a *user level* file system: a DAFS client may access network storage directly with protection and without kernel involvement. This offers the application full control over data movement and caching.

MPI-IO, the I/O part of the MPI-2 standard [12], is to provide an interface specifically intended for portable and high performance parallel I/O. MPI-IO defines a comprehensive interface with many features for both portability and performance. ROMIO [16] is a well-known implementation of MPI-IO with high-performance and portability. The current version runs on different types of machines, including parallel machines, SMP and networks of workstations (NOW). Supported file systems are IBM PIOFS, Intel PFS, HP HFS, SGI XFS, NEC SFS, Clemson PVFS, NFS and any Unix file system (UFS) [16].

Even though MPI-IO implementations are available for different file systems, so far, there is no implementation to take advantage of the emerging DAFS standard. In this paper, we take on such a challenge. We design, develop, and evaluate an implementation of MPI-IO (ROMIO) on DAFS over cLAN VIA. We demonstrate that our implementation can take full advantage of all DAFS features for high performance. The main contributions of this paper are as follows:

*This research is supported in part by Department of Energy's Grant # DE-FC02-01ER25506 and an NSF Grant #EIA-9986052.

- We analyze MPI-IO desired features and compare them with the DAFS capabilities. We show that DAFS meets almost all desired features for implementing MPI-IO correctly with high performance.
- Memory management is a major issue in implementing MPI-IO on DAFS. We propose three design alternatives for memory management.
- We implement MPI-IO on DAFS by porting the ROMIO ADIO layer on DAFS APIs. Three design alternatives of memory management are implemented and evaluated. We find that there could be a dilemma for DAFS applications to take advantage of Inline and Direct data transfers when the effects of memory management are considered. We propose that the DAFS Provider can export the threshold at which data transfers switch from Inline to Direct so that the DAFS-enabled applications can avoid the cost of memory management by using Inline data transfer.
- We compare and analyze the performance of basic MPI-IO operations in our implementation with implementations on two other file systems: network file system (NFS) and local file system (LFS). We show that MPI-IO can take full advantage of VI-compliant transport and user-level file access employed by the DAFS Protocol and interfaces. Compared to the two other implementations, MPI-IO on DAFS delivers high-throughput, low-latency performance and better capability for overlapping I/O and computation.

The rest of the paper is organized as follows. An overview of DAFS is presented in Section 2. In Section 3, we give a brief overview of ROMIO. In Section 4, we analyze and compare the MPI-IO demands and the DAFS capabilities. Different approaches for memory management are also discussed. Section 5 describes our implementation of MPI-IO on DAFS. Various performance evaluation results are presented in Section 6, followed by conclusions and future work in Section 7.

2. Overview of DAFS

2.1. Communication model

The Direct Access File System (DAFS) [7] is a new file sharing standard. It's designed to provide application servers with high-throughput and low-latency access to shared file servers over memory-to-memory networks, such as Virtual Interface (VI) Architecture [5] and InfiniBand (IB) Architecture [10]. Memory-to-memory networks are characterized by remote direct memory access (RDMA), user-level networking and offloading transport protocol processing to the network adapters. The DAFS protocol takes full advantage of these features to provide low-latency, high-throughput, and low overhead data movement. These capabilities are referred as the Direct Access Transport (DAT) by the DAFS protocol.

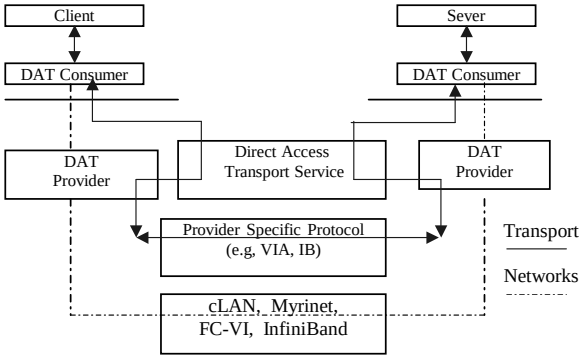

Figure 1. DAFS communication framework

The DAT semantic is the minimal set of transport capabilities that DAFS requires to provide high-performance DAFS implementations. DAT can be mapped easily onto networks that support features mentioned earlier, such as VI Architecture and IB Architecture. Figure 1 shows the communication framework of the DAFS protocol, in which interactions between the local and remote DAT Consumers are enabled through direct access transport (DAT) services.

DAFS provides two types of data transfer operations. *Inline* data transfer uses a send-receive model to make data movement. Usually, there is a data copy from user buffers to pre-registered buffer pools. Another data transfer option is *Direct*, in which I/O operations are translated to RDMA I/O operations. The DAFS user is responsible to manage the memory for Direct data transfer.

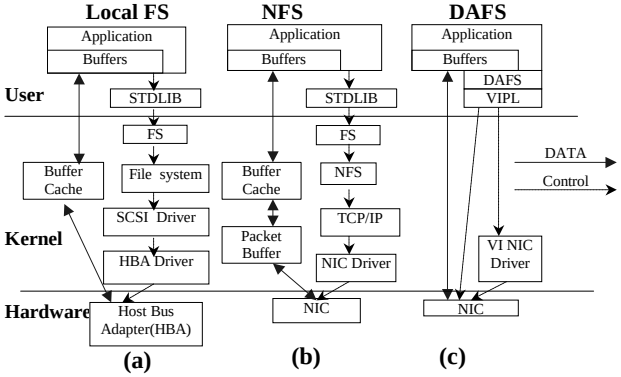

Figure 2. DAFS file access methods

2.2. DAFS file access methods

The DAFS protocol suggests a fundamental change in file access method in order to take advantage of DAT capabilities of underlying networks. Figures 2(a), (b) and (c) describe the general file access methods for local file systems,

network file systems and DAFS respectively. In DAFS, by using the remote memory addressing capability of DAT, a client's read or write request causes the DAFS server to issue RDMA operations back to the client, thus data can be transfered directly to and from a client application's buffer without copy or any CPU overhead at all on the client's side.

Applications can take advantage of DAFS capabilities in several ways depending on DAFS implementation. Both DAFS server and client can be implemented in two main ways: *user level* and *kernel level*, several implementation demonstrations are listed in [7].

3. Overview of ROMIO

MPI-IO, the I/O part of the MPI-2 [12] standard, is to design a new interface specifically for portable, high-performance parallel I/O. MPI-IO defines a comprehensive interface with many features which specifically support portable, high-performance parallel I/O. Details about these features can be found in [9].

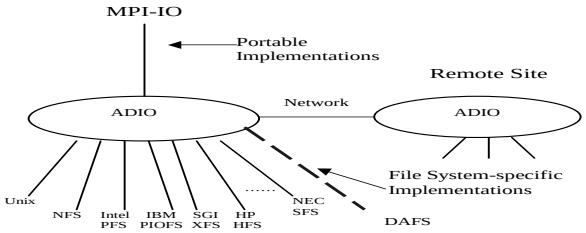

Figure 3. ROMIO architecture (from [15]) and our approach

Multiple implementations of MPI-IO, both portable and machine specific, are available. ROMIO [16] is a well-known implementation of MPI-IO with high-performance and portability on different platforms [16, 15]. ROMIO has a key internal layer called ADIO: an abstract-device interface for parallel I/O, which consists of a small set of basic functions for parallel I/O. ROMIO achieves portability and high performance for MPI-IO by implementing MPI-IO on top of ADIO, and implementing ADIO separately on each different file system. Implementation of ADIO on a particular file system can be completely optimized for performance on the targeted file system. An overview of ROMIO architecture is shown in Figure 3.

4. Challenges in implementing MPI-IO on DAFS

In this section, we compare and analyze the desired features of MPI-IO with DAFS capabilities. We show that the major mismatch between MPI-IO and DAFS is that read or write buffers must be registered for the DAFS Provider. This is called memory management. We present and evaluate different design alternatives for memory management.

4.1. Desired features and DAFS capabilities

Rajeev, et al. [15] proposed a list of features desired from a file system that would help in implementing MPI-IO correctly and with high performance. We describe how these features can be supported by DAFS capabilities in Table 1.

It is very interesting to note that DAFS provides all these desired features, except control over file striping and file preallocation. In addition, DAFS provides nonblocking I/O which is optional in the MPI-IO demands list. Thus, it can be expected that MPI-IO can exploit DAFS features to the maximum extent possible. One of the main objectives of our work is that the power and complexity of DAFS can be encapsulated in the ADIO layer which is appropriate for MPI-IO and subsequently MPI-IO applications take full advantage of DAFS features and high performance networks.

4.2. Memory management

A mismatch between MPI-IO and DAFS, not shown in Table 1, is as follows: as a basic requirement for memory-to-memory networks, all read or write buffers are required to be in registered memory regions in DAFS. To enable applications to flexibly manage their buffers by their buffer access patterns, memory registration is exported to the DAFS applications by DAFS memory management APIs. This important difference from POSIX and WIN32 enables the performance advantages of DAFS by removing memory registration operations and their user-to-kernel transitions from the critical I/O paths. Thus, the DAFS Consumer's explicit registration is preferable, though the DAFS API accepts a NULL memory handle,

Note that at the cost of pre-registering a system buffer and an additional memory-to-memory copy, the constraint of explicitly registering read and write buffers can be removed, however this is beneficial only for small data transfers. The DAFS *Inline* data transfer also takes advantage of it. But the threshold at which data transfers switch from *Inline* to *Direct* is not visible to the DAFS users. Based on this and the above mentioned performance concern, it is inevitable to register user buffers on top of the DAFS.

Memory management should address the following three problems:

1. Allow applications to register memory once and then subsequently use it for many times to avoid the per-operation registration overhead.

2. Balance performance and critical resources (pinned memory and resources on the network adapters), may require memory management to be machine-dependent.

3. Avoid memory registration in situations where registration is not required. For example, when a read or

Table 1. MPI-IO demands for correctness and high-performance and DAFS capabilities

MPI-IO Demands	DAFS Capabilities	Remark
High Performance Parallel File Access: concurrent requests (particularly writes) not serialized	• high-throughput, low-latency file access • high-speed byte level locking • session-based communication model • write-batch	
Data-Consistency Semantics: byte-level consistency, the data written by a process visible to other processes immediately after the write from this process returns without any explicit cache flush	• byte-level locking • various types of locking • the delegation of file access authority • no cache, read-ahead, or write-behind on the client side	*asynchronous operations can be used by applications to do read-ahead and write-behind if need.*
Atomicity Semantics	• atomic write append • atomicity semantics enforced by locking semantics	*DAFS makes common case fast.*
File Attribute Consistency	• mechanism to enable applications to maintain file attribute consistency • variable size of buffers used to fetch attribute data.	*In DAFS, application control achieves higher performance*
Interface Supporting Noncontiguous Access	• gather/scatter list I/O • noncontiguity in both memory and a file	*DAFS request chaining optimization*
Support file larger than 2 Gbytes	• support	
Byte-Range locking	• support	
Control over File Striping	• no Support	
Variable Caching/Pre-fetching Policies	• server caching hints • variable caching/pre-fetching policies used in the server side.	*DAFS Client also can provide application-controlled read-ahead and write-behind.*
File Preallocation	• no support	
Leave Collective I/O to the MPI-IO implementation	• no collective I/O • collective I/O may benefit from DAFS nonblocking operations • collective I/O may benefit from DAFS cache hints	
No shared file pointers	• not support shared file pointers	*DAFS uses explicit offset in operations*
Nonblocking I/O optional	• native nonblocking operations	*extra threads or processes are not needed*

write request is small enough to fit for Inline data transfer and registration is not required. This requires memory management to be dependent on the implementation of the DAFS API Provider library.

Since MPI-IO interface does not contain any function related to memory registration, memory management only can be done in the ADIO layer and/or in the DAFS layer. We propose three design alternatives for memory management in implementing MPI-IO on DAFS.

- *ADIO-on-the-fly*: In this approach, registration and deregistration of buffers is done in the ADIO layer per I/O call. The DAFS Provider is not required to provide on-the-fly memory registration. Both the ADIO layer and the DAFS Provider can be easily implemented. With this approach, it can get benefits from the DAFS Provider registration caching, if available. This approach incurs per-operation registration overhead. It solves the second problem. To solve the third problem, implementation information about threshold of data transfer from Inline to Direct must be exported from the DAFS Provider.

- *ADIO-CACHE-on-the-fly*: In this approach, registration caching is deployed in the ADIO layer. Clearly, this approach is not dependent on the DAFS Provider registration caching for high performance, which is suggested to be used sparingly [7]. Furthermore, application specific caching scheme can be provided by the ADIO layer according to the hints of application buffer usage pattern dynamically. It solves the first problem. Using a moderate number of caching entries, the second problem for performance can be also solved. Similarly, threshold of data transfer from Inline to Direct is needed to solve the third problem.

- *DAFS-on-the-fly*: In this approach, the ADIO layer calls DAFS APIs directly without pre-registering buffers. Memory registration and deregistration are offloaded to the DAFS Provider. Memory management completely depends on the DAFS Provider. The Provider *may* cache memory registrations. This approach solves the third problem, but it depends on implementation of the DAFS API Provider library to solve the first and second problems.

All these three approaches are used to hide memory management for MPI-IO applications. Alternatively, MPI-IO can be extended to have explicit memory management functions. However, compared to the ADIO-CACHE-on-the-fly method, we believe the benefit of this extension may not offset the cost for changing existing MPI-IO applications with these extended functions. Thus, we believe future implementations of MPI-IO will use one of the above mentioned memory management techniques. We implemented these three approaches and evaluated their impact on MPI-IO performance.

5 Implementation of MPI-IO on DAFS

In this section, we present details of our implementation of MPI-IO on DAFS. As analyzed in Section 4.1, most of MPI-IO features can be supported directly by DAFS capabilities. From the interface point of view, most of ADIO functions can be implemented by DAFS API functions in a straightforward manner, including basic file operation, noncontiguous access and nonblocking I/O.

Memory management was implemented using three different approaches as discussed in Section 4.2. To implement *ADIO-CACHE-on-the-fly*, we added a simple LRU component in the ADIO layer with 200 cache entries (this number is based on our analysis of NAS benchmark [1] in *communication* aspect, registration hit can be up to 80% with a LRU cache using *200* entries). Instead of DAFS memory registration function, specific memory registration function provided by the above LRU component is called by the ADIO layer. Minor modifications were done to the **ADIO_Init**() and **ADIO_End**() functions [14]. The third approach of *DAFS-on-the-fly* was implemented directly by passing a NULL memory handle into the DAFS API functions in all cases when the ADIO layer calls the DAFS API functions. In our current implementation, an application can choose one of these three approaches by setting a corresponding environment variable before it executes. We are planing to enable applications to choose one of them by adding an option in **MPI_Info_set** function dynamically. By this method, an application can choose different memory management approaches during different phases of its execution by calling **MPI_Info_set** function according to its buffer usage pattern.

6. Performance evaluation

In this section, we evaluate three alternatives of memory management in our implementation in order to consider the impact of memory management on latency. We compare and analyze the performance of basic MPI-IO operations in our implementation with implementations on two other file systems often used in cluster systems: local file system (LFS) and network file system (NFS). We also compare the capability of MPI-IO nonblocking operations for overlapping I/O and computation in these three implementations.

6.1. Experimental setup

We used a cluster system consisting of 8 dual 1 GHz Intel PIII systems, built around the ServerWorks LE chipset, which has a 64-bit 66MHz PCI bus for all experiments. These nodes are equipped with 512MB of SDRAM and 256K L2-level cache. All systems run unmodified Linux 2.2.18 with kernel built-in NFS version 3 stack.

We installed user level DAFS package from Duke [11] over an Emulex/Giganet cLAN switched VI network [8] with 1.25Gb/s link speed. The DAFS server has a memory-based file system. Local file system is a Linux ext2 file system mounted over a RAM disk. We used NFS version 3 which is required by MPI-IO ROMIO [16]. NFS was configured on UDP/IP over the same cLAN network. The *rsize* and *wsize* [4], i.e., the buffer size network file system uses to exchange data in a block are 8K bytes. The file system exported by the NFS server is a Linux ext2 file system over a RAM disk.

All MPI-IO experiments run over the above three file systems. With such configuration, we can emphasize effects of underlying communication protocols (VIA vs UDP/IP), user-level file sharing protocols and user-level file access on MPI-IO performance. In the rest of this paper, we refer these three systems as ROMIO_DAFS, ROMIO_NFS and ROMIO_LFS, respectively.

6.2. Impact of memory management

Effects of memory management depend on three factors: 1) approach of memory management being used, 2) cost of memory registration and deregistration, and 3) application buffer access pattern. To consider these factors together, we evaluated three memory management approaches discussed in Section 4.2 with different buffer reuse ratio and buffer sizes.

Figure 4 shows the impact of memory management on read latency. The buffer size that an application uses on each I/O request is called *block size* in the figures. Our benchmark program sequentialy reads data from a file with a certain block size for many times. Latency is the average time of each read operation. In Figure 4 (a), the block size of each request is 1024 bytes, in (b) and (c), it is 2048 and 16384 bytes, respectively. Legend *DAFS* means *DAFS-on-the-fly* approach is used. Legend *ADIO* means *ADIO-on-the-fly* approach is used. Legend *ADIO_C* means *ADIO-CACHE-on-the-fly* approach is used. We make the following four observations.

First, for small requests (less than 2048 bytes, Figure 4 (a)), registration provided by the DAFS Provider performs better than other twos, since the DAFS Provider uses Inline

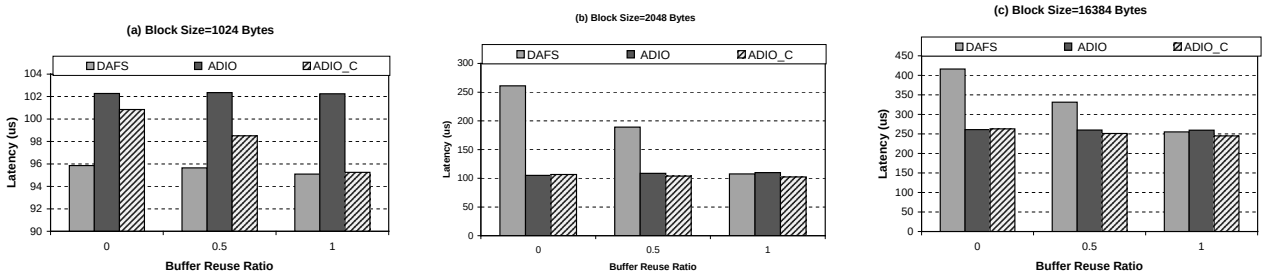

Figure 4. Impact of memory management on read latency

data transfer to handle such requests. Thus, registration in any layers other than the DAFS Provider is not efficient.

Second, memory management (with/out cache) in the ROMIO layer performs significantly better than memory management in the DAFS Provider when Direct data transfer is used to handle requests (Figures 4 (b) and (c)). In Duke's implementation, there is a simple cache for the DAFS Provider on-the-fly memory management. Our result shows that there is some room for optimization of their cache implementation.

Third, *ADIO-CACHE-on-the-fly* reduces latency compared to *ADIO-on-the-fly*. The difference in their performance grows with the buffer reuse ratio and registration and deregistration cost (block size). Figure 5 shows this difference for read operations. When the buffer reuse ratio is 0, *ADIO-on-the-fly* performs better, since cache in *ADIO-CACHE-on-the-fly* adds 1.5 microseconds overhead. Note that cLAN VIA has best performance of memory registration [2], in other platforms with more costly registration, *ADIO-CACHE-on-the-fly* is expected to have more benefits. It is also noted that this difference can be removed or decreased when the DAFS Provider provides efficient caching mechanism, though this feature should be used sparingly [7].

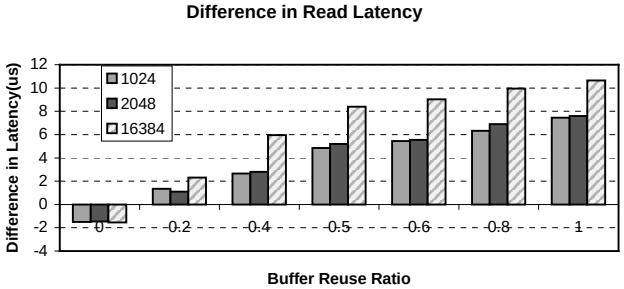

Figure 5. Difference in read latency between ADIO-CACHE-on-the-fly and ADIO-on-the-fly

Fourth, to take advantage of Inline data transfer, the ADIO layer has to know the threshold to switch from Inline to Direct in the DAFS Provider. We propose that the DAFS API interface can be extended to export such information about implementation of the DAFS API Provider to the DAFS API Consumer.

Simliar results are achieved for write operations and can be referred in [17]. Since the *ADIO-CACHE-on-the-fly* approach performs better in general, the implementation is simple and the overhead is low, we use this approach for evaluation in the remaining part of this paper.

6.3. Performance comparison

Figures 6 and 7 present latency and bandwidth of MPI-IO blocking read and write on the three file systems. In this experiment, we measured the latency and bandwidth for **MPI_File_read** and **MPI_File_write** over ROMIO_DAFS, ROMIO_NFS and ROMIO_LFS, respectively. Figure 6 shows latency for both sequential and random read, and latency for sequential write (append) only. Figure 7 shows sequential read and write bandwidth. In the test for sequential read operations, the file location distance between two consecutive read requests is block size. While in the test for random read operations, the file location distance between two consecutive read requests is 0.5Mbytes. This random test is intended to eliminate read-ahead and caching effect because we believe the system will not prefetch more than 0.5Mbytes data after each read request. Thus the latency for random read operations can be considered as overhead for each read operation.

As shown in Figure 6, for sequential read operations (legend "S"), local file system read-ahead can significantly help ROMIO_LFS achieve a much lower latency because one read from RAM disk can make the following hundreds of small read requests to read from I/O cache. It takes only 1.5 microsecond in average for reading 4 bytes. ROMIO_NFS also benefits from read-ahead. A value of *rsize* equals to 8K bytes helps ROMIO_NFS to do sequential small reads. It takes 102.3 microseconds to read 4 bytes. ROMIO_DAFS takes 75 microseconds to read 4 bytes.

In random read test (legend "R" in Figure 6), for ROMIO_LFS, every read request issues a new request to RAM disk in the unit of *Pagesize*. For block sizes up to *Pagesize*, it takes almost same time. ROMIO_LFS takes 26 microseconds for reading 4 bytes, ROMIO_NFS takes 310 microseconds and ROMIO_DAFS takes 75 microseconds. There is no difference between sequential and nonsequential versions of ROMIO_DAFS read, since there is no cache

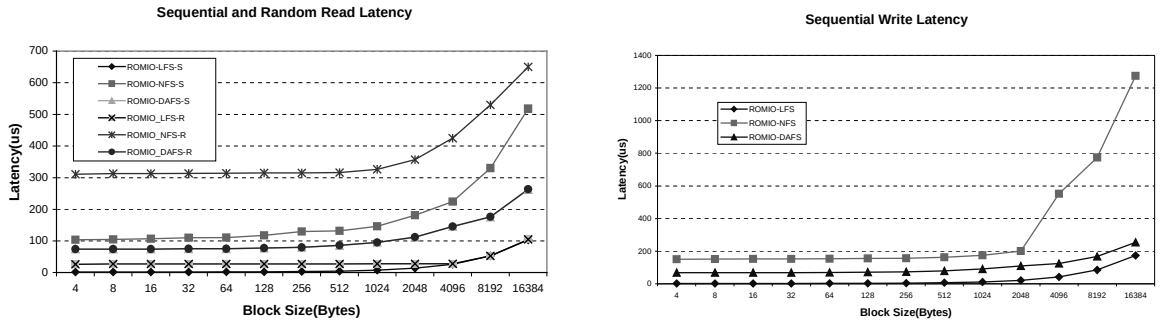

Figure 6. MPI-IO blocking read and write latency on Local File System, NFS and DAFS. In read latency, "S" stands for sequential reads and "R" means random reads.

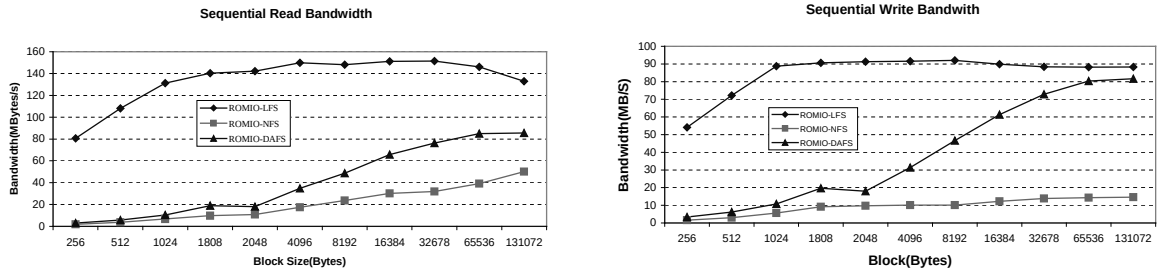

Figure 7. MPI-IO blocking read and write bandwidth on Local File System, NFS and DAFS

on the client side and no disk activities on the server side.

As shown in Figure 7, the sequential read bandwidth of ROMIO_DAFS is 1.6-2.2 times better than ROMIO_NFS. Write bandwidth of ROMIO_DAFS is 1.8-2.4 times higher than ROMIO_NFS with block sizes less than 2048 bytes and 4.4-5.6 times higher for block sizes larger than 2048 bytes. There are three factors which contribute to this significant improvement. First, overhead reducing features in DAFS have been fully exploited by MPI-IO. Second, cLAN can support 65519 bytes MTU frames. The MTU of UDP/IP over cLAN is still 1500 bytes. Third, MPI-IO ROMIO implementation over NFS requires clients to use *"noac"* [4] mount option to disable all forms of attribute caching entirely for correctness. This extracts NFS server performance penalty and increases interactions between the server and clients, specifically for write operation.

At 2048 bytes, the write and read bandwidth of ROMIO-DAFS drops slightly. This is because in Duke's implementation, the DAFS Provider handles all requests with length less than 2048 bytes (actually 2004 bytes payload) using Inline data transfer. In addition, VIPL of cLAN does not support RDMA read. Thus, by using Direct data transfer to write data to the server, more message exchanges than read operation are needed since a hand-shake is needed for the client to get target buffers on the server before the client can initiate RDMA write. This also reduces the write bandwidth.

The threshold of Inline and Direct depends on system memory performance (copy), cost of memory management and performance of underlying communication layer (PIO vs DMA [3]). Our experimental results shows a number larger than 2048 bytes as the switch point.

It can be observed that ROMIO_DAFS performs significantly better than ROMIO_NFS. This indicates that MPI-IO applications can take full advantage of the features inherent in DAFS file system to reduce end-to-end overhead.

6.4. Impact of nonblocking (asynchronous) I/O

In our implementation, MPI-IO nonblocking operations use the counterparts of DAFS API directly. Nonblocking operations are used to overlap I/O and computation. To study the impact of nonblocking I/O on DAFS, we conducted an additional experiment in which a variable amount of computation was added after each non-blocking I/O operation. Figure 8 presents the results when the I/O block size is 64K bytes.

We note that nonblocking I/O has a large impact on the performance of applications. The X axis in Figure 8 is the CPU fraction given by *T(len)/[T(len)+ T_IO(len)]*, where *T_IO(len)* is a function of message length, indicating the latency to read or write a block data with length of *len* using blocking operations, and *T(len)* is computation time. When the computation takes less time than the blocking MPI-IO operations (points before 0.5 in the X axis), i.e., $T(len) < T\_IO(len)$, the bandwidth achieved by ROMIO_DAFS re-

mains almost same as bandwidth achieved by blocking operations (CPU fraction is 0). This indicates that MPI-IO on DAFS can perfectly overlap I/O and computation. There is no overlap between I/O and computation in ROMIO_NFS and ROMIO_LFS.

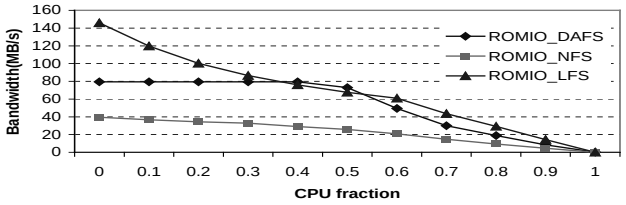

Figure 8. Overlap between I/O and computation

7 Conclusions and future work

In this paper, we have presented a MPI-IO implementation on DAFS over VIA and compared its performance with two other file systems in cluster environments. We designed and implemented three approaches for memory management. Performance evaluation shows that memory management in the ROMIO ADIO layer with cache performs better when Direct data transfer is used to handle I/O requests in the DAFS Provider, while for requests handled by Inline data transfer, memory management provided by the DAFS Provider performs better. Based on this finding, we propose that the DAFS Provider provides mechanism to export the threshold of Inline and Direct data transfers to the DAFS Consumer.

We configured all file systems on memory-based file system or RAM disk and focused on the effects of underlying communication protocols (VIA in DAFS vs UDP/IP in NFS), user-level file sharing protocols, and user-level file access on MPI-IO performance. The performance of MPI-IO on DAFS over VIA on cLAN was found to be 1.6-5.6 times better than the performance of MPI-IO on NFS over UDP/IP on cLAN. Write bandwidth on DAFS was close to LFS with large request size. We also demonstrate that our implementation exploits all features of DAFS for high-performance I/O and is capable of delivering better overlap between I/O and computation.

We are working on improving the performance of collective I/O operations by taking advantage of DAFS features. We are also engaged in application-level evaluation. Another direction for future work is studying how coherent client-side caching intended for portable, high-performance parallel I/O can be incorporated into the ADIO layer on DAFS.

Acknowledgments

We would like to thank Duke DAFS team for giving us the latest version of DAFS implementation and Richard Kisley for providing us with insights into their implementation. We are also thankful to Dr. Pete Wyckoff and Jiuxing Liu for many discussions with us.

References

[1] M. Banikazemi, B. Abali, and D. K. Panda. Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA). In *Fourth Int'l Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00)*, Jan 2000.

[2] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishna, P. Sadayappan, H. Sah, , and D. K. Panda. VIBe: A Microbenchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations. In *IPDPS*, April 2001.

[3] M. Banikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for the IBM SP Switch-Connected NT Clusters. In *IPDPS*, pages 33–42, May 2000.

[4] B. Callaghan. *NFS Illustrated*. Addison Wesley, 1999.

[5] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[6] DAFS Collaborative. Direct Access File System Application Programming Interface(DAFS API), V1.0, November 2001.

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

[8] Emulex Corp. cLAN: High Performance Host Bus Adapter, September 2000.

[9] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA, 1998.

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

[11] R. Kisley. Structure, Performance, and Implementation of the Direct Access File System (Master Thesis). Department of Computer Science, Duke University, August 2001.

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

[13] S. Shepler, B. Callaghan, D. Robinson, and et al. NFS Version 4 Protocol. RFC 3010, December 2000.

[14] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct. 27–31, 1996.

[15] 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.

[16] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation(Revised Version)*. ANL/MCS-TM-234, Sept. 2000.

[17] J. Wu and D. K. Panda. MPI-IO on DAFS over VIA: Implementation and Performance Evaluation. Technical Report OSU-CISRC-11/01-TR23, CIS Dept. the Ohio State University, Dec 2001.