To apper in Cluster Computing 2 (1999)

Implementing Noncollective Parallel I/O In Cluster Environments Using Active Message Communication

Jarek Nieplocha
<j_nieplocha@pnl.gov>
Pacific Northwest National Laboratory
Richland, WA 99352

Holger Dachsel
<dachsel@scm.com>
Scientific Computing & Modelling
1081 HV Amsterdam, The Netherlands

Ian Foster
<foster@mcs.anl.gov>
Argonne National Laboratory
Argonne, IL 60439

1 Introduction

Recent advances in low-latency, high-speed network technology coupled with inexpensive commodity processors have greatly improved cost-effectiveness of parallel computing. Disk technology has also been advancing rapidly especially with respect to the storage density, capacity and bandwidth. I/O, on the other hand, remains a major bottleneck in many parallel applications, such as climate modeling, computational chemistry, and computational fluid dynamics. There are several contributing factors to this problem, for example: non-optimal I/O behavior of the applications, insufficient I/O resources, or filesystem configuration not matching the application needs. Whereas application I/O behavior have been studied extensively in the literature, see for example [1,2], the resource planning and configuration issues have received less attention. A challenging question that many computing centers face, first in the procurement and then in the operational phase, is how much secondary storage resources are needed and how to configure them to meet increasing needs of the variety of applications running on their systems.

Parallel I/O systems such as IBM PIOFS, Intel PFS, and PIOUS [3] use the traditional client-server model in which some processors with attached disks act as I/O servers, while other processors are used as compute nodes and I/O clients. This model can work well in dedicated (single-application) environments if the I/O subsystem characteristics match the needs of the particular application. In the multi-application environment, the parallel I/O system is a shared resource which usually impacts the I/O performance delivered to the simultaneously running applications. In many cases, it is not economically viable to configure a parallel filesystem to support more than one I/O-intensive parallel application running simultaneously. The most expensive cost component is not the disk storage. It is the cost of server nodes (including their infrastructure) which in the client-server systems are separated from the compute pool. However, in scientific applications, the client-server model is frequently not the only possible implementation of parallel I/O. In particular, the client-server dichotomy is not required to implement collective I/O [4]. For example, systems such as Panda [5] and DRA [6] can effectively exploit all local disks on the compute nodes during a collective I/O operation thus not requiring dedicated I/O servers. Nevertheless, although collective I/O is effective for some applications, it is not appropriate for others that require more dynamic and independent (noncollective) access to secondary storage. Certain computational chemistry applications have such access characteristics, for example [7]. This paper demonstrates that more flexible communication protocols in the style of Active Messages [8] or VIA [9] when employed in context of I/O on clustered systems with inexpensive local storage help to achieve very high performance rivaling that of dedicated parallel filesystems for noncollective I/O, and allow very efficient exploitation of the available system resources by effectively providing only the necessary subset of functionality of a parallel filesystem but with performance characteristics matching the application needs.

To support noncollective parallel I/O in a clustered environment without dedicated server nodes, we introduced a new parallel I/O model called Distant I/O (DIO). We implemented a parallel shared file I/O model using DIO and demonstrate its advantages in a very I/O-intensive computational chemistry application on the IBM SP. At present time, the SP represents the largest commercial cluster architecture. DIO targets clustered computer systems in which disks are attached to compute nodes. This hardware configuration is currently supported by networks of workstations (NOWs) and by multicomputers (e.g., the IBM SP) and is likely to remain popular in the future because of the economic and physical attributes of such architectures. DIO supports one-sided access to storage on remote nodes that execute the same user application; hence, all processors can participate in a computation and operate as both I/O clients and (if they have attached disks) servers. A DIO implementation relies for performance on the existence of a single-sided communication protocol able to initiate an I/O operation on a processor that may be performing other computation. Lightweight communication protocols such as Active Messages and Fast Messages [10] are well suited for this purpose. Such facilities allow a DIO implementation to avoid the complexities and overheads associated with the server processes and daemons required to support a client-server–based approach. The Remote DMA provided by VIA combined with memory mapped files could be also used to implement the DIO model. However, as VIA requires locking of memory pages in physical memory, this implementation would not be appropriate for large scale out-of- core applications.

A primitive Distant I/O mechanism can be used as a building block for higher-level parallel I/O libraries and systems. We demonstrate how this can be accomplished by describing a DIO-based implementation of the Shared Files (SF) library on a collection of local disks attached to compute nodes of the IBM SP cluster. This parallel noncollective I/O library was originally developed by the ChemIO project [7,11]. It allows a user to define “shared files” that can be accessed and updated independently by any processor in a parallel computation. On the IBM SP, our DIO-based SF implementation outperforms a comparable implementation based on the IBM PIOFS parallel file system by a wide margin, when used by a large computational chemistry application, namely COLUMBUS, a multireference configuration interaction code [12]. An out-of-core version of this code [13] using the Shared Files library was able to solve a problem [14] six time larger than previously reported which involved moving 1.5 terabyte of data between disk and main memory per iteration. Furthermore, with the DIO implementation described in this paper, the execution time for this calculation on 128 processors of the IBM SP was reduced from 305.5 to 79.6 wall clock hours when compared to the execution on top of the well equipped PIOFS system.

The rest of this paper is organized as follows. In Section 2, the DIO model and implementation on the IBM SP are described. In Section 3, a parallel shared file implementation using DIO is presented. Section 4 contains microbenchmark performance results for DIO. Section 5 compares the performance of a large application running on top of DIO with the performance of PIOFS on the IBM SP. Finally, Section 6 presents our conclusions.

1 Distant I/O Model and Architecture

One-sided communication allows a process in a distributed-memory system to access data residing in the address space of another process, without the explicit cooperation of the second process. Distant I/O combines one-sided communication with I/O to secondary storage at remote processors. Distant I/O has several useful properties, including:

- *Distributed view of secondary storage*: Secondary storage is used as an extension of main memory in distributed- memory systems and accessed with the convenient one-sided communication paradigm.
- *Flexibility*: DIO can be used to implement parallel I/O models and libraries even on systems that lack parallel/shared filesystems. Furthermore, such systems can be customized (for example, by setting striping factors and caching policies) to match the needs of a parallel application, rather than relying on system-wide settings. This follows findings presented in [2] that recommend filesystem customization to match the individual application I/O patterns.
- *Capacity and bandwidth scalability*: As the number of application processors with attached disks grows, the aggregate I/O bandwidth and aggregate capacity proportionally increases.

In order to explore the practical utility of Distant I/O, we have defined a DIO application programmer interface (API) and constructed an implementation of this API on the IBM SP. The API is based on the C run-time library, but extends it in four areas:

1. Instead of a separate seek operation, offset is used as an additional argument to read/write operations.
2. The file descriptor is replaced by a handle that references a file on a local or remote processor.
3. A “home” process/processor id is added to identify the location of the file.
4. A request handle is added to support nonblocking versions of calls.

For example, the following are the DIO counterparts to the standard *read* and *write* operations:

```
dio_read(handle, offset, buffer, bytes, proc, request)

dio_write(handle, offset, buffer, bytes, proc, request)
```

Nonblocking DIO functions are designed to allow overlapping of remote I/O operations with other activities. Such operations are completed by calling the *dio_wait* function, which takes as an argument the request handle returned by the corresponding *dio_write/read* operation. The remote file handle is a local representation of the remote file descriptor. It can be obtained by registering a file for DIO access either through a collective DIO operation similar to the MPI-2 *MPI_Win_create* [15] or through a directory service.

1.1 Implementation on the IBM SP

We discuss the architecture of DIO by describing its implementation on the IBM Scalable POWERparallel (SP) system. This massively parallel computer has a cluster architecture and employs as building blocks processor nodes similar to the IBM RS/6000 workstation. Every node contains at least one local disk based on SSA or SCSI technology. The nodes are connected through a high-speed network that supports 38 μsec one-directional latency and 100 MB/s bandwidth in the point-to-point user-space communication [16]. In addition, some larger SP configurations support an optional parallel filesystem, PIOFS. PIOFS is striped on multiple disks connected to dedicated PIOFS server nodes. IBM recently introduced another parallel filesystem, GPFS. However, it is not yet available on large SP systems.The IBM SP supports LAPI [16], a commercial implementation of Active Messages [1,17] that provides active message functionality, put, and get one-sided remote memory copy operations. LAPI is a multithreaded system compatible with Pthreads, which are supported by the IBM AIX 4.2 operating system on the IBM SP.

Our DIO implementation on the IBM SP uses LAPI Active Messages (AM) to send specifications of read/write operation to remote nodes. Upon arrival, the AM completion handler is invoked and executed by a separate thread. Within the handler code, LAPI remote memory copy and Unix I/O are used. For *dio_write*, *LAPI_Get* transfers data to an internal DIO buffer. This step is followed by a blocking write. For *dio_read*, blocking read is followed by *LAPI_Put*, which transfers data read from the disk to the internal DIO buffer and then to the user buffer at the requesting processor. This process is illustrated in Figure 1 for *dio_read.*

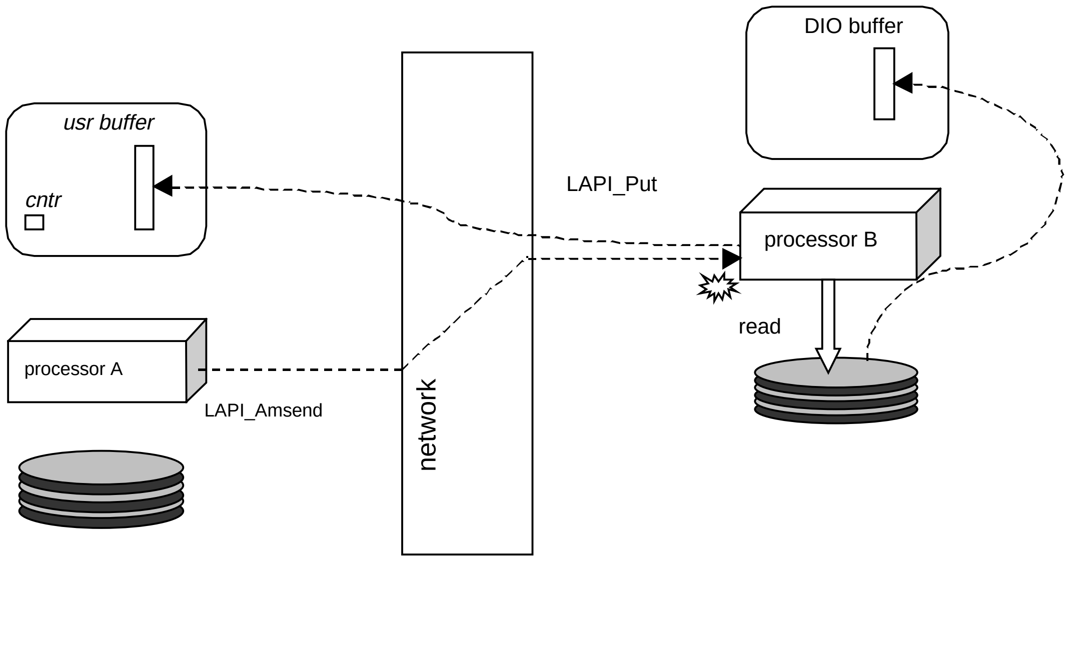

Figure 1: Implementation of *dio_read*

The IBM SP implementation of DIO nonblocking operations exploits Posix asynchronous I/O (AIO) when a DIO operation references a file on the local disk. If the data is on a remote disk, the operation returns when an active message is sent to the remote processor. The AM completion handler is executed by a separate thread, which is activated by the LAPI dispatcher when a message arrives [16]. This thread executes concurrently with the main (application) thread and makes the blocking I/O read/write call. When the I/O is completed and the completion handler finishes, LAPI implicitly sends a low-level control message to the requesting processor, which then increments the AM completion counter *cntr*, see Figures 1. This optional feature is enabled when a non-NULL completion counter address is specified in the *LAPI_Amsend* interface. If the data is on a remote file, the *dio_wait* operation waits until this counter is incremented, a process that occurs when the completion handler and I/O operation it executes are completed. If the data is on a local file, the *dio_write* simply calls its Posix AIO counterpart. We note that although the counter mechanism in LAPI AM interface is convenient, it is not mandatory to implement DIO. Other active-message–style facilities, including the Berkley/Cornell AMs [8]and Illinois FM [10], could be used in a similar fashion to implement DIO on other platforms, including networks of workstations. In particular, the notification of I/O completion can be accomplished with an explicit message sent back to the requesting node. In addition, the implementation described in this section is applicable both to clusters with uniprocessor and multiprocessor nodes.

The LAPI-based implementation of DIO can be used by MPI programs. However, it cannot be used by applications that rely on MPL, the original IBM proprietary message-passing library still available on the IBM SP. The reason for this incompatibility is that the message delivery subsystem of MPL is based on signals. The most recent implementation of MPI and LAPI both use threads. Unlike MPL, MPI is available in two implementations: signal and thread-based. In our experience, the thread-based implementation of MPI is competitive with the signal-based implementation of that library.

1.1.1 Memory management in Active Message processing

With the LAPI Active Message library, the application provides memory that the library uses to copy data arriving from the network. This requires from DIO to do memory management and flow control to avoid overflowing its I/O buffers or depleting local memory resources, which for example could happen when many *dio_write* request targeted one particular node. A common solution used in implementation of message passing libraries is to allocate one buffer per task on every node. However, with this solution memory consumption on every node is proportional to the number of tasks which would limit scalability of DIO. We designed a protocol that allows DIO to minimize the memory consumption and use only one buffer (~100KB) on each node. In *dio_write*, instead of sending request specification with the data to be written (LAPI_Amsend supports very large messages) we separated transfers of the request information and the data. The transfer of request is initiated by sender, and transfer of data is initiated by receiver. The increased latency of this protocol (~80µS extra) is negligible comparing to the disk access time. Since *dio_write* is nonblocking, it does not have to appear on the critical path of computations performed by the sender.

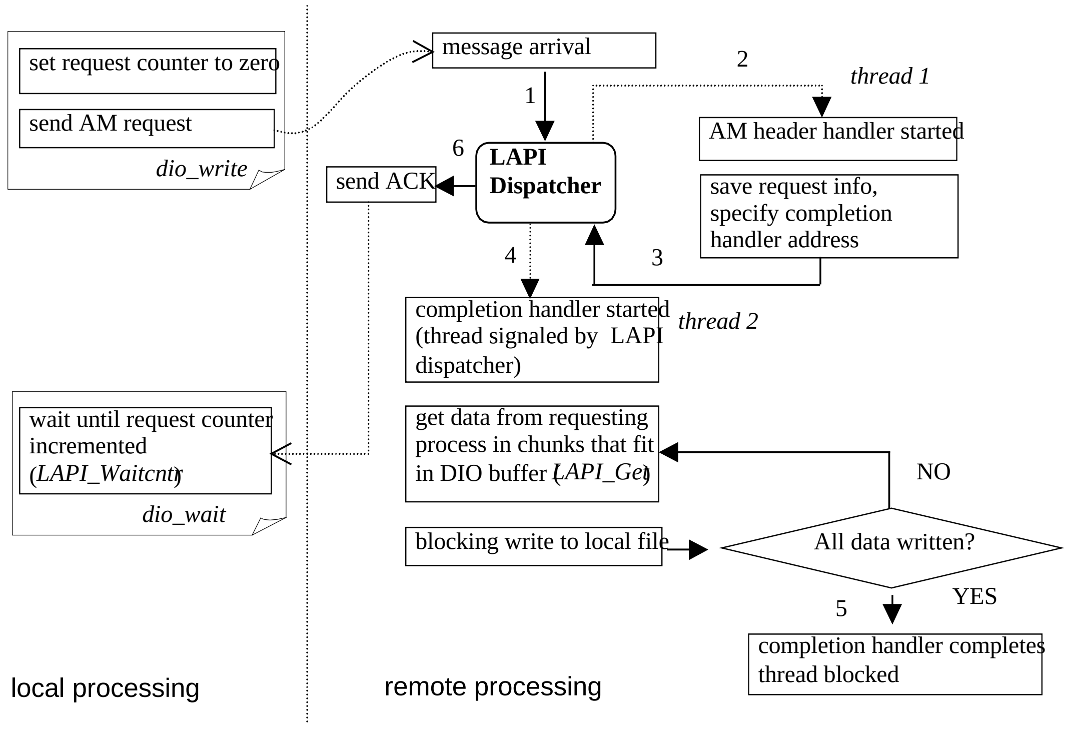
set request counter to zero
send AM request
dio_write
message arrival
1
2
thread 1
AM header handler started
6
LAPI Dispatcher
send ACK
save request info, specify completion handler address
4
3
completion handler started (thread signaled by LAPI dispatcher)
thread 2
wait until request counter incremented (LAPI_Waitcntr)
dio_wait
get data from requesting process in chunks that fit in DIO buffer (LAPI_Get)
NO
blocking write to local file
All data written?
5
YES
completion handler completes thread blocked
local processing
remote processing

Figure **2**: Processing in dio_write and dio_wait operation by requesting process and at remote location. The numbers specify sequence of operations.

Figure 2 demonstrates the flow of data and processing used in the *dio_write* (nonblocking) operation and *dio_wait* operation that completes processing started by *dio_write*. On the requesting side, a *dio_write* operation returns very fast to the caller, immediately after the specification of the I/O operation is sent to remote node. The operation involves sending an LAPI active message describing parameters of the request: offset in remote file, request size, file handle, pointer to the user buffer with the data and requesting process(or) id. The header is very short (20 bytes), and LAPI is able to send asynchronously 32 of such messages in time less than 13 μS each. Upon the message arrival, the LAPI dispatcher runs the user-specified callback function, header handler. The role of header handler is to copy the request parameters to a temporary space and return address of another callback function, the completion handler. The header handler must not block or perform any communication operations because when it is running a portion of the LAPI dispatcher code is locked which prevents progress in the communication on this node. The completion handler is executed by another thread which up to this time was blocked waiting for a condition variable to be signalled by the dispatcher. Once completion handler starts, it is able to access the DIO request specification. In *dio_write*, data is transferred from the requesting node to the internal DIO buffer in chunks. Each chunk is written to the local file and then next chunk is transferred. Since there can be only one instance of completion handler running per task, one buffer is sufficient. After I/O operation in completion handler thread are finished, the handler returns. LAPI dispatcher blocks the thread and then sends an acknowledgment to the requesting node.

2 Parallel Shared Files

We used the Distant I/O model to construct an implementation of Shared Files (SF), a parallel I/O library we had developed before [7]. This library supports the concept of a parallel file with every process in a parallel computation being able to read and write independently at arbitrary locations. This disk access model is similar to the parallel files created in the M_UNIX mode on the Intel PFS and the default parallel file mode supported by the IBM PIOFS. The differences between SF and these other systems include the following:

- Shared files are not guaranteed to be persistent. Persistency is a property of the filesystem on which SF is implemented, rather than the model itself.
- Shared Files support files larger than 2 GB in the Fortran API [7].
- The user can specify hints for “typical request” size, and “maximum expected” file size in *sf_create* interface when created a new shared file.
- Shared Files read and write operations are nonblocking (a feature not yet available in standard Fortran-77/90) and contain an offset argument rather than a separate seek operation:

```
rc = sf_write(handle, offset, bytes, request_id)

rc = sf_read(handle, offset, bytes, request_id)
```

Similarly to parallel filesystems, the SF library does not perform explicit control of consistency in concurrent accesses to overlapping sections of the shared files. For example, SF semantics allows a write operation to return before the data transfer is complete, which requires special care in programs that perform write operations in critical sections, since unlocking access to a critical section before write completes is unsafe. An *sf_wait* function is provided to enforce completion of the data transfer so that the data can be safely accessed by another process after access to the critical section is released by the writing process.

The actual size of a shared file might grow as processes perform write operations beyond the current end-of-file boundary. Data in shared files are implicitly initialized to zero, meaning that read operations at locations that have not been written return zero values. However, reading beyond the current end-of-file boundary is erroneous and the result undefined. We take advantage of this property in the optimized implementation of shared files on collection of local disks.

In essence, the Shared Files library at run-time presents to the application a view of a file residing in a parallel filesystem. The file is not guaranteed to persist beyond the program execution and cannot be accessed with the usual set of operating system commands. However, because of the restricted access model and availability of hints directing the implementation, a high performance implementation matching the application needs becomes possible.

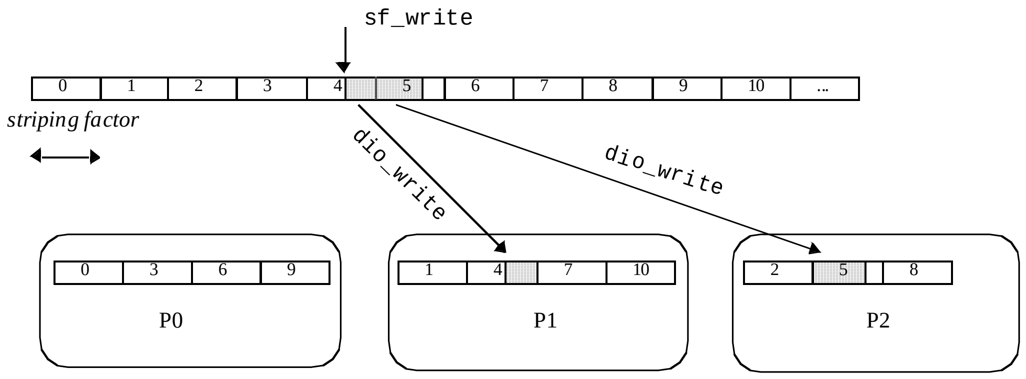

Figure3. Example sf_wait request decomposed into DIO operations on component files for a shared file implementation on three processors with local disks.

1.1 High performance implementation of parallel shared files

The DIO implementation of shared files uses a collection of component files, each located on a disk on a different compute node, to represent a parallel file. On IBM SP, the shared file is implemented by striping a parallel logical file across multiple physical files located on all disks available on the computing nodes on which the parallel application is running, see for Figure 3.

It uses Distant I/O to perform remote read/write operations. Based on an input from the user for the “typical request size,” SF determines a value of the striping factor when a file is created. If this (optional) information is not available, an empirically determined value (e.g., 32 kB) is used.

The DIO implementation of Shared Files exploits the properties of this I/O model to optimize performance. In particular, there is no central directory that stores and updates dynamically changing parameters of the component files such as their size. When the application writes beyond the logical end-of-file boundary in the shared file, a gap can be created. In Unix and Windows NT (NTFS, but not FAT), file gaps are interpreted implicitly as filled with zeros. A gap in a shared file might create a situation when the size of one or more component files becomes not consistent (too small) with the size of the shared file, see Figure 4.

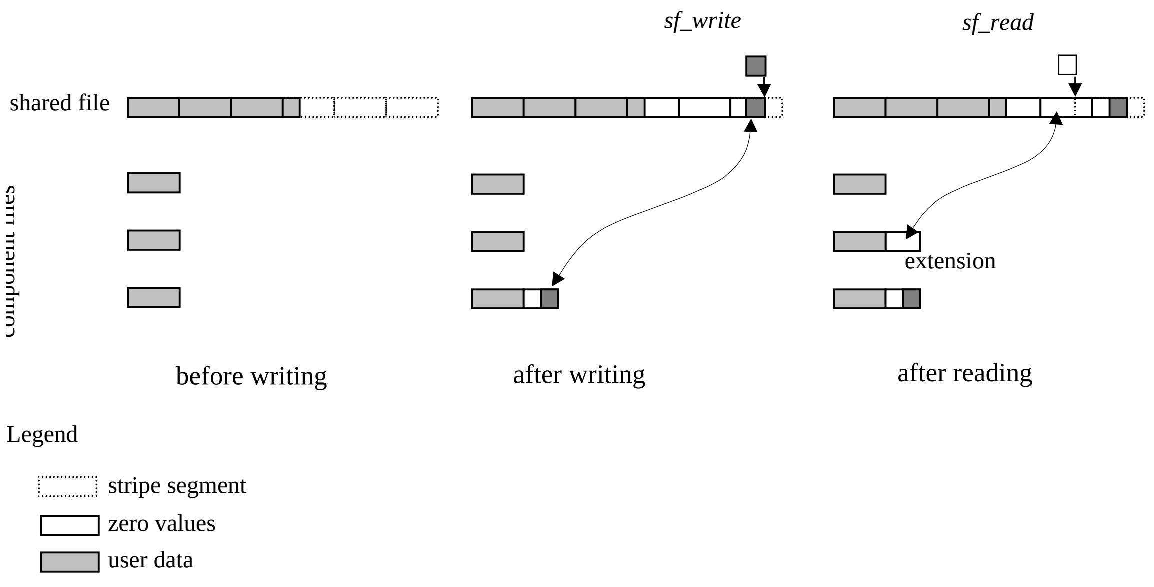

Figure **4**: Example component file representation of a shared file with a gap created by *sf_write* operation beyond end of file followed by a *sf_read* operation within the gap area.

In some circumstances, with this discrepancy a valid read request to shared file could generate a DIO read request beyond the end of component file. To address this problem and avoid the costs of maintaining and updating a centralized directory holding component file state, we modified the standard behavior of Distant I/O read operation which under normal circumstances disallowed reading beyond end of file. With this modified behavior, a DIO read operation beyond the end of file returns values read found in the component file and extends its size. This adjustment is done only when read or write operation on that file is performed. In such a case, we extend the component file by writing a zero value at the appropriate location beyond the end of the file.

The original Shared Files implementation used the ELIO (Elementary IO) device library [7,11] as its portability layer. We added DIO as another device library compatible at run time with ELIO. This allows multiple filesystem implementations (e.g., PIOFS and local JFS files in AIX) in the same application and enables one to dynamically select, at run time, which implementation to use, see Figure 5. The library uses the path specified in the metafile name to determine whether it points to a filesystem shared by all the processors (like PIOFS); if it does not, SF selects a DIO-based implementation for this particular shared file.

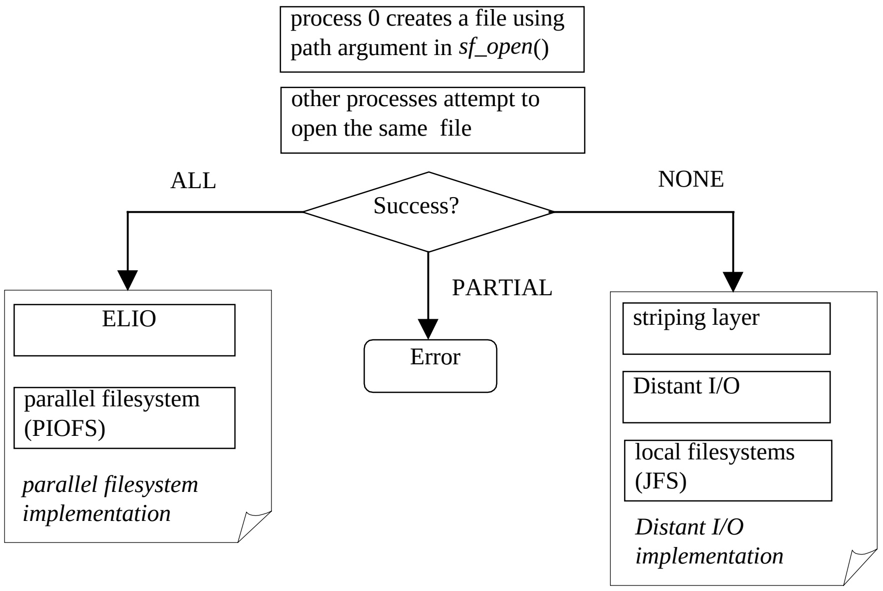
process 0 creates a file using path argument in *sf_open*()
other processes attempt to open the same file
ALL
Success?
NONE
PARTIAL
ELIO
parallel filesystem (PIOFS)
parallel filesystem implementation
Error
striping layer
Distant I/O
local filesystems (JFS)
Distant I/O implementation

Figure 5: Support for parallel filesystem and DIO implementation of Shared Files at run-time

With the DIO implementation, the I/O activity performed by the application is localized to the set of resources on the compute nodes the application is running. With exception of sharing the switch bandwidth, it does not have other effects on other applications in the system that, for example, might be performing I/O to the parallel filesystem.

2 Performance

We measured the performance of DIO by writing and reading 1 GB (eight times the amount of main memory on a processor, to avoid caching effects) of data from both a local and a remote file. The request size varied from 4096 to 32,768 bytes. We tested two disk configurations on the 512-node IBM SP with 120MHz Power2 Super processors at Pacific Northwest National Laboratory (PNNL). In one configuration, the local JFS filesystem was mounted on a single SCSI disk; in the second configuration, it was striped on two identical SCSI disks (a bandwidth optimization).

Figures 6 and 7 show that the I/O rates for remote requests approach local rates as the request size increases. The relatively small differences between local and remote I/O can be attributed to the efficiency of LAPI, especially when comparing it to the disk access time. (The bandwidth in *LAPI_Put* (used in *dio_read*) grows much faster with increased message size than in the MPI point-to-point message-passing [16] and, for example, achieves more than 70

MB/s bandwidth for messages as short as 4096-bytes.) However, the overhead associated with Active Messages processing (including the handler execution by a dedicated thread scheduled by AIX) contributes to the larger difference in performance for smaller requests.

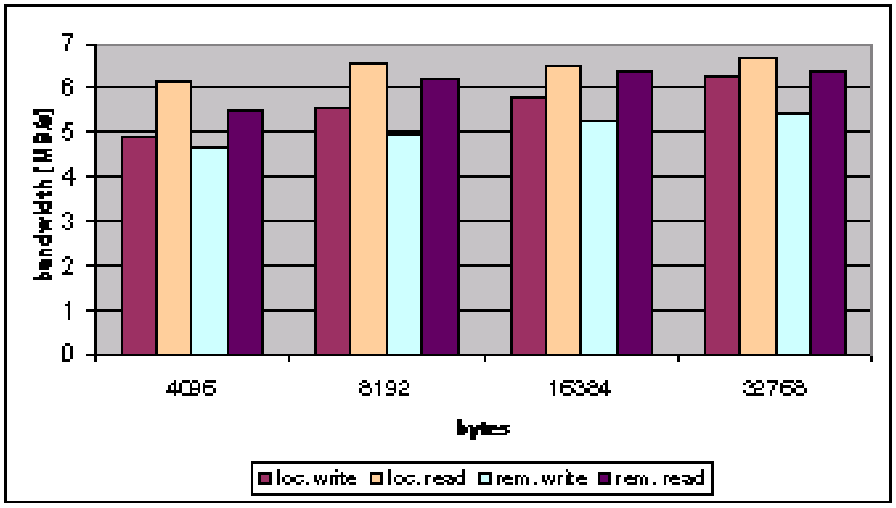

Figure 6: Performance of DIO *read* and *write* in access to local and remote files on the IBM SP (two disks striped)

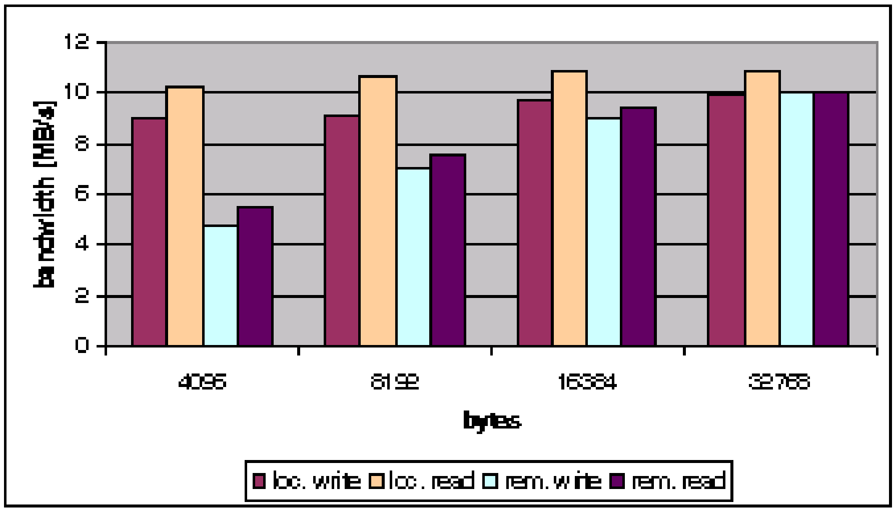

Figure 7: Performance of DIO *read* and *write* in access to local and remote files on the IBM SP (two disks striped)

2 Application Experience

The multireference configuration interaction (MRCI) method is widely used in computational chemistry to obtain accurate predictions of properties of chemical systems. The MRCI method is arguably the most generally-applicable, high-accuracy method for determination of the electronic structure of small molecules. It can be used to predict the energetics and other properties of molecular systems from which may be derived chemical information including thermochemistry, spectroscopy and detailed models of chemical reactions. Since the computational expense scales as greater than the sixth power of the molecular system size, it is only applicable to small molecules. Details of the algorithms and methodology used in the COLUMBUS MRCI code are given elsewhere [12,13]. From the mathematical perspective, the program solves the eigenvalue problem for a very large sparse symmetric Hamilton matrix by using the standard Davidson method. A set of expansion vectors is used to project the original eigenvalue problem to a subspace eigenvalue problem that has a much smaller dimension. Because of the very high accuracy of the MRCI method, even for a very small molecule, the matrix becomes very large. To date, the state-of-the-art calculations in this area have been for 100 to 200 million of the configuration state functions (Hamilton matrix dimension). Despite using sparsity techniques and highly effective compression scheme to reduce the storage requirements such large calculations cannot be performed in-core.

To address memory limitations, COLUMBUS adopted a disk-based approach using the SF library [7]. This application is using shared memory programming model with data located in distributed system memory and in secondary storage. The calculations are performed in an MIMD task-parallel fashion driven by data-dependent dynamic load balancing. The program allocates all possible distributed main memory for a one-dimensional global array [18] and then creates an SF shared file to store data that does not fit in the distributed main memory. Secondary storage is accessed in a noncollective fashion, with individual processors reading and writing records containing compressed data. Data is read from disk, uncompressed, updated, compressed, and written back to the disk. Since the update can affect the compression rate, the size of the data written to the disk might be different from that of the original data. A distributed lock mechanism supported by the Global Arrays library [18] is used by COLUMBUS to assure atomicity when updating critical sections. The average I/O request size in this program is approximately 30 KB. Due to the algorithm properties and the amount of available in-core memory, the request size could not be increased.

We used both PIOFS and DIO implementations of Shared Files to solve the largest MRCI problem ever attempted, represented by a matrix of dimension 1.3 billion (1,295,937,374 configuration state functions). The calculations were performed on 128 processors of the IBM SP at PNNL. We monitored the PIOFS activity during the program execution to assure that there were no other I/O-intensive applications using PIOFS at the same time. The science results obtained from this challenging calculation are presented in [14]. We report timing results for two execution environments:

- SF-PIOFS (44 servers with 4 SSA disks each, which appears to be the largest PIOFS configuration available at that time anywhere) and MPL communication and
- SF-DIO on top of local disks (not striped) and the LAPI Active Messages library.

We were not able to use GPFS, a new IBM parallel filesystem, since as the time of this writing it does not support systems as large as 512 nodes.

Table 1: I/O read performance in COLUMBUS using Shared Files library on top of PIOFS and DIO

	Data Volume	I/O time	Bandwidth per CPU
PIOFS	900.963GB	906297.07s	0.994MB/s
DIO	900.957GB	235708.32s	3.823MB/s

The I/O-related statistics are given (per iteration) in Table 1. The first column contains the amount of data read from secondary storage, corresponding to “reading expansion vectors” phase in Figure 8.

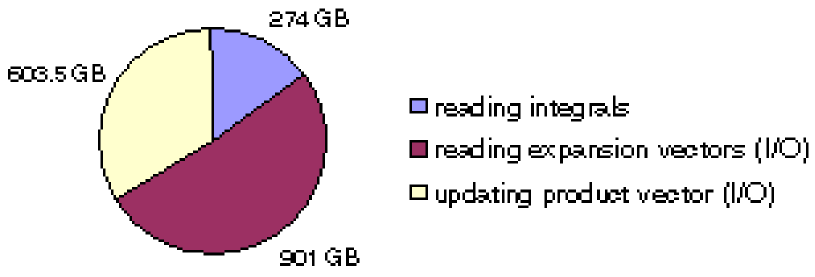

Figure **8**: Data volume moved per iteration in main computational phases of the calculation: 1.5TB between disk and memory, the rest in interprocess communication

It is slightly different for the two versions because of the dynamic nature of the load balancing and the algorithm properties. The second column indicates the total time wall-clock time spent reading data, summed over all processors. The 900GB constitutes the larger portion of 1.5 TB aggregate I/O volume per iteration in this calculation. We do not report write rates because the I/O buffering layer in AIX makes them difficult to measure reliably and non-intrusively in a context of a complex application such as COLUMBUS rather than in a synthetic benchmark. The last column gives the average bandwidth rate per I/O request and per processor. These results show that the DIO-based implementation of SF outperforms PIOFS by a factor of four in this particular application. The SF-DIO read rates measured in COLUMBUS are only 25% lower than those measured by our microbenchmark for DIO alone (see Figure 6). This small difference is due to the overhead of the SF layer (striping), some degree of contention when accessing data, and the fact that the every node used in calculation played a part-time role of an I/O client and I/O server.

The overall execution time of COLUMBUS is over 70% shorter (79.6 vs. 305.5 hours) for the DIO-based implementation than for PIOFS. The improvement can be contributed to increased efficiency of I/O and faster interprocessor communication. The original implementation of this application used the IBM MPL communication library and PIOFS. Although LAPI is faster (and incompatible) with MPL, this I/O-bound application is spending only 10% time on the interprocessor communication in the computational phase labelled “reading integrals” in Figure 8. Therefore, the increase of the overall performance than can be contributed to the improvement in I/O is estimated to be at least 65%.

2 Conclusions

We have proposed a new approach to parallel I/O, called Distant I/O, that exploits Active Message communication on clustered systems with attached disks. This model avoids scalability problems and software overhead associated with client-server implementations of parallel filesystems, and supports the construction of scalable secondary storage systems based on attached disks. DIO provides one-sided and asynchronous access to disks attached to remote processors: in effect, it extends the one-sided communication model to secondary storage. We implemented this model using Active Messages and Posix asynchronous I/O on the IBM SP. Our implementation approach is directly applicable to clusters of workstations on which various flavors of active-message–style communication facilities are available. We have used DIO to implement a higher-level parallel noncollective I/O library. In particular, we have constructed a DIO-based implementation of the Shared Files library by striping parallel-shared files on local disks attached the IBM SP nodes. The obtained I/O performance is excellent when measured by a microbenchmark and by a large I/O- intensive scientific application, COLUMBUS. It shows that a very high utilization of I/O resources by a complex application is achievable with our approach. This I/O technology enabled COLUMBUS to solve a problem six times larger than previously reported in its particular area of science.

Acknowledgments

This work was supported through the U.S. Department of Energy by the Mathematical, Information, and Computational Science Division phase II Grand Challenges of the Office of Computational and Technology Research. This work was performed under contract DE-AC06-76RLO 1830 (Pacific Northwest National Laboratory) and under contract W-31-109-Eng-38 (Argonne National Laboratory). This research was performed in part using the Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory (PNNL). Pacific Northwest National Laboratory is operated for the Department of Energy by Battelle..

References

[1] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel and Distributed Systems*, 7(10):1075-1089, October 1996.

[2] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. InProceedings of Supercomputing '95, San Diego, CA, December 1995. IEEE Computer Society Press.

[3] S. A.Moyer and V. S. Sunderam, PIOUS: A scalable I/O system for distributed computing environments. In *Proc. Scalable High-Performance Computing Conf.*, 1994.

[4] Y. Chen, I. Foster, J. Nieplocha, and M.Winslett, Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study, *Proc. 11th ACM Intl. Conf. on Supercomputing*, ACM Press, 1997.

[5] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. Supercomputing '95*, December 1995.

[6] J. Nieplocha and I. Foster. Disk Resident Arrays: An array-oriented library for out-of-core computations. In *Proc. Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[7] J. Nieplocha I. Foster, R.A. Kendall, ChemIO: High-performance parallel I/O for computational chemistry applications, *Int. J. Supercomp. Apps. High Perf. Comp.* vol. 12, no. 3, 1998.

[8] D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. Generic active message interface specification. Technical report, University of California at Berkeley, November 1994.

[9] Compaq Computer Corp., Intel Corp., Microsoft Corp., Virtual Interface Architecture Specification, Dec. 16, 1997.

[10] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. *In Proc. Supercomputing'95,* 1995.

[11] J. Nieplocha, I. Foster, R. Kendall, ChemIO, http://www.emsl.pnl.gov:2080/parsoft/chemio.

[12] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, and R. J. Harrison. A massively parallel multireference configuration interaction program - the parallel COLUMBUS program. *J. Chemical Physics*, 18:430, 1997

[13] H. Dachsel, J. Nieplocha, R.J. Harrison, An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interactionprogram, *Proceedings of High Performance Networking and Computing Conference SuperComputing'98*, 1998. (SC'98 Best Overall Paper Award)

[14] H. Dachsel, R.J. Harrison, D. Dixon, Multireference Configuration Calculations on Cr2: Passing the one billion limit in MRCI/MRACPF calculations, to appear in *Journal of Physical Chemistry*, *1999*.

[15] MPI Forum, MPI-2: Extensions to Message Passing Interface, University of Tennessee, July 18, 1997.

[16] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison , R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, Performance and experience with LAPI – a new high-performance communication library for the IBM RS/6000 SP. *Proc. 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing IPPS'98,* pages 260-266, 1998.

[17] C. C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, November 1996.

[18] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, 10:197–220, 1996.