Appeared in *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180–187.

An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces

Rajeev Thakur *William Gropp* *Ewing Lusk*

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
{thakur, gropp, lusk} @mcs.anl.gov

Abstract

In this paper, we propose a strategy for implementing parallel-I/O interfaces portably and efficiently. We have defined an abstract-device interface for parallel I/O, called ADIO. Any parallel-I/O API can be implemented on multiple file systems by implementing the API portably on top of ADIO, and implementing only ADIO on different file systems. This approach simplifies the task of implementing an API and yet exploits the specific high-performance features of individual file systems. We have used ADIO to implement the Intel PFS interface and subsets of MPI-IO and IBM PIOFS interfaces on PFS, PIOFS, Unix, and NFS file systems. Our performance studies indicate that the overhead of using ADIO as an implementation strategy is very low.

1 Introduction

Parallel computers are being used increasingly to solve large, I/O-intensive applications in a number of different disciplines. A limiting factor, however, is the lack of a standard, portable application-programming interface (API) for parallel I/O. Instead of a single standard API, a number of different APIs are supported by different vendors and research projects. Many commercial parallel file systems (e.g., IBM PIOFS [11] and Intel PFS [12]) and research parallel file systems (e.g., PPFS [10], Galley [18], HFS [15], Scotch [4], and PIOUS [17]) provide their own APIs. In addition, a number of I/O libraries with special APIs have been developed (e.g., PASSION [25], Panda [21], Chameleon I/O [7], SOLAR [28], Jovian [1], and ChemIO [6]). Different APIs are used by systems that support persistent objects (e.g., Ptool [9], ELFS [13], and SHORE [2]).

A group within the Scalable I/O Initiative [20] is developing a standard low-level interface for parallel I/O [5]. This low-level interface, however, is not intended to be used directly by application programmers, but instead at the operating-system level by developers of libraries for compilers, run-time systems, and applications. The only real effort to standardize an interface for parallel I/O at the application-programming level is the MPI-IO [27] proposal that is based on MPI [16]. The MPI Forum has recently started an effort to standardize an interface for parallel I/O as part of MPI-2. The Forum is using MPI-IO [27] as a starting point. The result of this effort may well become the standard API in the future.

In this paper, we propose a strategy for implementing parallel-I/O APIs portably and efficiently. For this purpose, we have defined an abstract-device interface for parallel I/O, called ADIO. Any parallel-I/O API can be implemented efficiently on multiple file systems by implementing the API portably on top of ADIO and implementing only ADIO separately on each different file system. We have used ADIO to implement the Intel PFS interface and subsets of MPI-IO and IBM PIOFS interfaces on PFS, PIOFS, Unix, and NFS file systems. Therefore, we are able to run applications (that use the above interfaces) portably on the IBM SP, Intel Paragon, and networks of workstations. Performance studies with two test programs and one real production application indicate that the overhead of using ADIO as an implementation strategy is very low.

We stress that ADIO is not intended to be a new API itself, i.e., it is not intended to be used directly by application programmers. Instead, it is a strategy for implementing other APIs.

The rest of the paper is organized as follows. In Section 2, we explain the ADIO concept in more detail. We describe the design of ADIO in Section 3 and discuss its use in implementing APIs such as the MPI-IO, PFS, PIOFS, PASSION, and Panda interfaces in Section 4. We present performance results in Section 5. We draw overall conclusions and discuss our plans for future work in Section 6.

2 The ADIO Concept

The main goal of ADIO is to facilitate a high-performance implementation of any existing or new parallel-I/O API on any existing or new file-system, as illustrated in Figure 1. ADIO consists of a small set of basic functions for performing parallel I/O. Any parallel-I/O API (including a file-system inter-

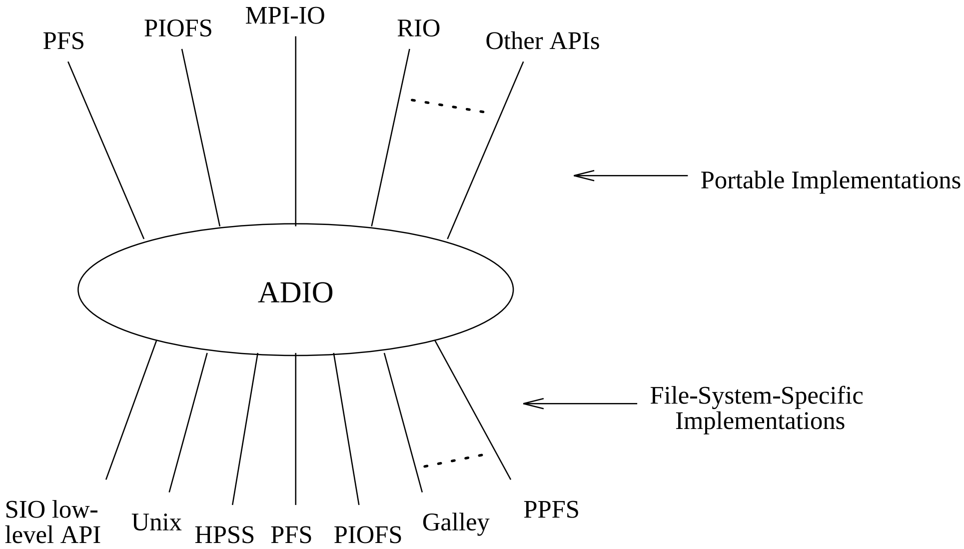

Figure 1: The ADIO concept

face) can be implemented in a portable fashion on top of ADIO. ADIO in turn must be implemented in an optimized manner on each different file system separately. In other words, ADIO separates the machine-dependent and machine-independent aspects involved in implementing an API. The machine-independent part can be implemented portably on top of ADIO. The machine-dependent part is ADIO itself, which must be implemented separately on each different system.

ADIO enables users to experiment with new APIs and new low-level file-system interfaces. Once a new API is implemented on top of ADIO, it becomes available on all file systems on which ADIO has been implemented. Similarly, once ADIO is implemented on a new file-system, all APIs implemented on top of ADIO become available on the new file system. This approach thus enables users to run applications on a wide range of platforms, regardless of the parallel-I/O API used in the applications.

The ADIO approach was motivated by the lack of consensus, within both the parallel-I/O community and the applications community, on any one standard API. Therefore, instead of mandating a particular API, we provide the framework for implementing any or all of them in a simple, efficient, and portable manner. When a standard API emerges, ADIO can be used to implement that API as well.

A similar abstract-device-interface approach for communication has been used very successfully in the MPICH implementation of MPI [8].

3 ADIO Design

ADIO is designed such that it can exploit the high-performance features of any file system, and any API can be expressed in terms of ADIO. We designed ADIO by first studying the interface and functionality provided by different parallel file systems and high-level libraries and then deciding how the functionality could be supported at the ADIO level portably and efficiently.

For portability and high performance, ADIO uses MPI [16] wherever possible. Therefore, ADIO routines have MPI datatypes and communicators as arguments. We describe the ADIO interface in the following subsections.

3.1 Open and Close

Open:

```
ADIO_File ADIO_Open(MPI_Comm comm, char
*filename, int file_system, int access_mode,
ADIO_Offset disp, MPI_Datatype etype,
MPI_Datatype filetype, int iomode, ADIO_Hints
*hints, int perm, int *error_code)
```

All opens are considered to be collective operations. The communicator `comm` specifies the participating processes. A process can open a file independently by using `MPI_COMM_SELF` as the communicator. The `file_system` parameter indicates the type of file system used. The `access_mode` parameter specifies the file access mode, which can be either `ADIO_CREATE`, `ADIO_RDONLY`, `ADIO_WRONLY`, `ADIO_RDWR`, `ADIO_DELETE_ON_CLOSE`, `ADIO_EXCLUSIVE`, or `ADIO_ATOMIC`. These modes may be combined by using the bitwise exclusive-or operator. The `ADIO_EXCLUSIVE` mode indicates that only the processes involved in this open call access the file; the ADIO implementation may use this information to perform client-side caching. The `ADIO_ATOMIC` mode indicates that the file system is required to guarantee atomicity of read/write operations. If this mode is not used, the file system need not provide atomicity and, therefore, may be able to improve performance.

The `disp`, `etype`, and `filetype` parameters are provided for supporting displacements, etypes, and filetypes as defined in MPI-IO [27]. The `iomode` parameter is provided for supporting the I/O modes of

Intel PFS [12]. The `ADIO_Hints` structure may be used to pass hints to the ADIO implementation for potential performance improvement. Examples of hints include file-layout specification, prefetching/caching information, file-access style, data-partitioning pattern, and information required for use on heterogeneous systems. Hints are purely optional; the calling program need not provide any hints, in which case ADIO uses default values. Similarly, the ADIO implementation is not obligated to use the specified hints. The `perm` parameter specifies the access permissions for the file. The success or failure of the open operation is returned in `error_code`. The `ADIO_Open` routine returns a file descriptor that must be used to perform all subsequent operations on the file.

Note that the displacement, etype, filetype, iomode, access mode, and hints associated with an open file can be changed by using the routine `ADIO_Fcntl`.

Close:

```
void ADIO_Close(ADIO_File fd, int
*error_code)
```

The close operation is also collective: All processes that opened the file must close it.

3.2 Contiguous Reads and Writes

```
void ADIO_ReadContig(ADIO_File fd, void *buf,
int len, int file_ptr_type, ADIO_Offset
offset, ADIO_Status *status, int *error_code)
```

Similarly `ADIO_WriteContig`.

ADIO provides separate routines for contiguous and noncontiguous accesses. The contiguous read/write routines are used when data to be read or written is contiguous in both memory and file. `ADIO_ReadContig` and `ADIO_WriteContig` are independent and blocking versions of the contiguous read and write calls (independent means that a process may call the routine independent of other processes; blocking means that the resources specified in the call, such as buffers, may be reused after the routine returns). Nonblocking and collective versions of the contiguous read/write calls are described in Sections 3.4 and 3.5, respectively.

In the case of `ADIO_ReadContig`, `buf` is the address of the buffer in memory into which `len` contiguous bytes of data must be read from the file. The location in the file from which to read can be specified either in terms of an explicit offset from the start of the file or from the current location of the file pointer. ADIO supports individual file pointers for each process; shared file pointers are not directly supported because of performance reasons. Shared file pointers can be emulated on top of ADIO if necessary. The `file_ptr_type` parameter indicates whether the routine should use explicit offset or individual file pointer. If `file_ptr_type` specifies the use of explicit offset, the offset itself is provided in the `offset` parameter. The `offset` parameter is ignored when `file_ptr_type` specifies the use of individual file pointer. The file pointer can be moved by using the `ADIO_SeekIndividual` function, described in Section 3.6. The `status` parameter returns information about the operation, such as the amount of data actually read or written.

3.3 Noncontiguous Reads and Writes

```
void ADIO_ReadStrided(ADIO_File fd, void
*buf, int count, MPI_Datatype datatype, int
file_ptr_type, ADIO_Offset offset,
ADIO_Status *status, int *error_code)
```

Similarly `ADIO_WriteStrided`.

Parallel applications often need to read or write data that is located in a noncontiguous fashion in files and even in memory. ADIO provides routines for specifying noncontiguous accesses with a single call. Noncontiguous access patterns can be represented in many ways, e.g., [19]; we chose to use MPI derived datatypes because they are very general and have been standardized as part of MPI. `ADIO_ReadStrided` and `ADIO_WriteStrided` are independent and blocking versions of the noncontiguous read and write calls; nonblocking and collective versions are described in Sections 3.4 and 3.5, respectively. Note that these routines support all types of noncontiguous accesses that can be expressed in terms of MPI derived datatypes, not just simple uniform strides.

In the case of `ADIO_ReadStrided`, `buf` is the address of the buffer in memory into which `count` items of type `datatype` (an MPI derived datatype) must be read from the file. The starting location in the file may be specified by using explicit offset or individual file pointer. The noncontiguous storage pattern in the file is indicated by the `filetype` (an MPI derived datatype) specified in `ADIO_Open` or `ADIO_Fcntl`.

Note that `ADIO_ReadContig` and `ADIO_WriteContig` are special cases of `ADIO_ReadStrided` and `ADIO_WriteStrided`. However, we consider contiguous accesses separately, because they are directly supported by all file systems and, therefore, may be implemented efficiently.

3.4 Nonblocking Reads and Writes

```
void ADIO_IreadContig(ADIO_File fd, void
*buf, int len, int file_ptr_type, ADIO_Offset
offset, ADIO_Request *request, int
*error_code)
```

```
void ADIO_IreadStrided(ADIO_File fd, void
*buf, int count, MPI_Datatype datatype, int
file_ptr_type, ADIO_Offset offset,
ADIO_Request *request, int *error_code)
```

Similarly `ADIO_IwriteContig`, `ADIO_IwriteStrided`.

ADIO provides nonblocking versions of all read and write calls. A nonblocking routine may return before the read/write operation completes. Therefore, the resources specified in the call, such as buffers, may not be reused before testing for completion of the operation. Nonblocking routines return a `request` object that must be used to test for completion of the operation. The ADIO routines for testing the completion of a nonblocking operation are described in Section 3.7.

3.5 Collective Reads and Writes

```
void ADIO_ReadContigColl(ADIO_File fd, void
*buf, int len, int file_ptr_type, ADIO_Offset
offset, ADIO_Status *status, int *error_code)
```

```
void ADIO_ReadStridedColl(ADIO_File fd, void
*buf, int count, MPI_Datatype datatype, int
file_ptr_type, ADIO_Offset offset,
ADIO_Status *status, int *error_code)
```

```
void ADIO_IreadContigColl(ADIO_File fd, void
*buf, int len, int file_ptr_type, ADIO_Offset
offset, ADIO_Request *request, int
*error_code)
```

```
void ADIO_IreadStridedColl(ADIO_File fd, void
*buf, int count, MPI_Datatype datatype, int
file_ptr_type, ADIO_Offset offset,
ADIO_Request *request, int *error_code)
```

Similarly `ADIO_WriteContigColl`, `ADIO_WriteStridedColl`, `ADIO_IwriteStridedColl`.

Several researchers have demonstrated that, for many common access patterns, collective I/O can greatly improve performance [3, 24, 14, 21]. To enable the use of collective I/O, ADIO provides collective versions of all read/write routines. A collective routine must be called by all processes in the group that opened the file. However, a collective routine does not necessarily imply a barrier synchronization.

3.6 Seek

```
ADIO_Offset ADIO_SeekIndividual(ADIO_File fd,
ADIO_Offset offset, int whence, int
*error_code)
```

This function can be used to change the position of the individual file pointer. The file pointer is set according to the value supplied for `whence`, which could be `ADIO_SEEK_SET`, `ADIO_SEEK_CUR`, or `ADIO_SEEK_END`. If `whence` is `ADIO_SEEK_SET`, the file pointer is set to `offset` bytes from the start of the file. If `whence` is `ADIO_SEEK_CUR`, the file pointer is set to `offset` bytes after its current location. If `whence` is `ADIO_SEEK_END`, the file pointer is set to `offset` bytes after the end of the file.

3.7 Test and Wait

It is necessary to test the completion of nonblocking operations before any of the resources specified in the nonblocking routine can be reused. ADIO provides three kinds of routines for this purpose: a quick test for completion that requires no further action (`ADIO_xxxxDone`), a test-and-complete (`ADIO_xxxxIcomplete`), and a wait-for-completion (`ADIO_xxxxComplete`). Separate routines exist for read and write operations.

```
int ADIO_ReadDone(ADIO_Request *request)
```

Similarly `ADIO_WriteDone`.

These routines check the `request` handle to determine whether the operation is complete and requires no further action. They return true if complete, and false otherwise.

```
int ADIO_ReadIcomplete(ADIO_Request *request,
ADIO_Status *status, int *error_code)
```

Similarly `ADIO_WriteIcomplete`.

If an operation is not complete, the above routines can be used. Note that these routines do not block waiting for the operation to complete. Instead, they perform some additional processing necessary to complete the operation. If the operation is completed, they return true and set the `status` variable; otherwise, they return false. If an error is detected, they return true and set the `error_code` appropriately.

```
void ADIO_ReadComplete(ADIO_Request *request,
ADIO_Status *status, int *error_code).
```

Similarly `ADIO_WriteComplete`.

These routines block until the specified operation is completed and set the `status` variable. If an error is detected, they set the `error_code` appropriately and return.

3.8 File Control

```
void ADIO_Fcntl(ADIO_File fd, int flag,
ADIO_Fcntl_t *fcntl, int *error_code)
```

This routine can be used to set or get information about an open file, such as displacement, etype, filetype, iomode, access mode, and hints.

3.9 Miscellaneous

ADIO also provides routines for purposes such as deleting files, resizing files, flushing cached data to disks, and initializing and terminating ADIO.

```
void ADIO_Delete(char *filename, int
*error_code)
```

```
void ADIO_Resize(ADIO_File fd, ADIO_Offset
size, int *error_code)
```

```
void ADIO_Flush(ADIO_File fd, int
*error_code)
```

```
void ADIO_Init(int *argc, char ***argv, int
*error_code)
```

```
void ADIO_End(int *error_code)
```

4 Implementation

Two aspects are involved in implementing ADIO: implementing an API on top of ADIO and implementing ADIO on a file system. The implementation may be done by using macros to eliminate the overhead of function calls (if it is not essential to check the correctness of function arguments).

4.1 Implementing an API on Top of ADIO

Here we explain how some of the different parallel-I/O APIs can be implemented by using ADIO routines. In particular, we explain how the main features of the API map to some feature of ADIO.

4.1.1 MPI-IO

MPI-IO [27] maps quite naturally to ADIO, because both MPI-IO and ADIO use MPI to a large extent. In addition, we included a number of features in ADIO specifically for being able to implement MPI-IO: displacement, etype, filetype, the ability to use explicit offsets as well as file pointers, and file delete-on-close.

4.1.2 Intel PFS

PFS [12] is the parallel file system on the Intel Paragon. In addition to a Unix-like read/write interface, PFS also supports several file-pointer modes that specify the semantics of concurrent file access. The Unix-like interface and the `M_UNIX` and `M_ASYNC` modes are straightforward to implement on top of ADIO. `M_LOG` mode can be implemented by emulating shared file pointers on top of ADIO. `M_SYNC`, `M_RECORD`, and `M_GLOBAL` modes can be implemented by using collective operations.

4.1.3 IBM PIOFS

PIOFS [11] is the parallel file system on the IBM SP-2. In addition to a Unix-like read/write interface, PIOFS also supports logical partitioning of files. A processor can independently specify a logical view of the data in a file, called a subfile, and then read/write that subfile with a single call. It is straightforward to implement the Unix-like interface of PIOFS on top of ADIO. The logical file views of PIOFS can be mapped to appropriate MPI derived datatypes and accessed by using the noncontiguous read/write calls of ADIO.

4.1.4 PASSION and Panda

PASSION [25] and Panda [21] are libraries that support input/output of distributed multidimensional arrays. I/O of this type involves collective access to (potentially) noncontiguous data. ADIO supports both collective I/O and noncontiguous accesses; therefore, PASSION and Panda can be implemented by using appropriate ADIO routines.

4.2 Implementing ADIO on a File System

Here we explain how ADIO can be implemented on PFS, PIOFS, Unix, and NFS file systems.

4.2.1 ADIO on PFS

Some ADIO functions, such as blocking and nonblocking versions of contiguous reads and writes, can be implemented by directly using their PFS counterparts. However, for functions not directly supported by PFS, the ADIO implementation must perform the task of expressing the ADIO functions in terms of available PFS calls. For example, noncontiguous requests can either be translated into several contiguous requests separated by seeks or can be implemented by using optimizations such as data sieving [23]. Collective operations can be implemented by using optimizations such as two-phase I/O [3, 24].

4.2.2 ADIO on PIOFS

As in the case of PFS, blocking and nonblocking versions of contiguous reads and writes can be implemented by directly using their PIOFS counterparts. Noncontiguous accesses can be implemented, in some cases, by using the logical views supported by PIOFS. In other cases, it may be necessary to implement noncontiguous accesses either in terms of several contiguous accesses or by using data sieving. Since PIOFS does not directly support collective I/O, the ADIO implementation can use two-phase I/O for improving performance.

4.2.3 ADIO on Unix and NFS

ADIO can be easily implemented on a Unix file system that supports all Unix semantics, such as atomicity and concurrent accesses from multiple processes to a file. However, the Network File System (NFS), which is widely used in a workstation environment, does not always guarantee consistency when multiple processes write to a file concurrently (even to distinct locations in the file), because it performs client-side caching [22]. To overcome this problem, we implemented ADIO on NFS by using *file locking* with the `fcntl` system call, which disables client-side caching. As a result, all requests from clients always go to the server, and consistency is maintained. Disabling client-side caching decreases the overall performance of NFS, but, nevertheless, it is necessary to ensure correctness of the result in the case of concurrent writes.

4.3 Current Status of Implementation

At present, we have implemented the PFS interface and subsets of MPI-IO and PIOFS interfaces on top of ADIO, and we have implemented ADIO on top of PFS, PIOFS, Unix, and NFS file systems, as illustrated in Figure 2. Therefore, we are able to run applications (that use these interfaces) portably on the SP, Paragon, and networks of workstations. We are actively working on implementing the entire MPI-IO interface on top of ADIO and implementing ADIO on additional file systems.

5 Performance

We studied the performance overhead of ADIO on PIOFS and PFS by using two test programs and one real production parallel application. Performance studies of ADIO on Unix and NFS are currently in progress.

We used the IBM SP at Argonne and the Intel Paragon at Caltech. The parallel I/O systems on these two machines were configured as follows during our experiments. On the SP, there were eight I/O server nodes for PIOFS, each with 3 Gbytes of local SCSI disks, and the operating system was AIX 3.2.5. On the Paragon, there were 16 I/O nodes for PFS, each connected to a 4.8-Gbyte RAID-3 disk array, and the operating system was Paragon/OSF R1.3.3. On both machines, users were not allowed to run compute jobs on the I/O nodes.

The performance results presented below are from an implementation of ADIO using functions, not

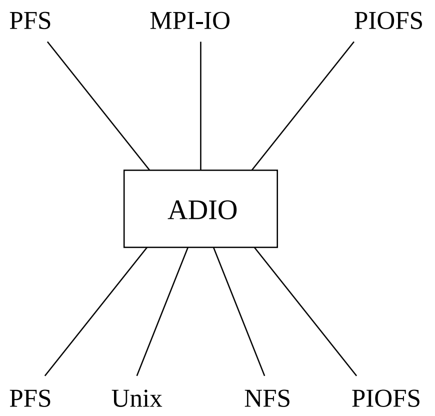

Figure 2: Current status of implementation

macros; the results may be slightly better if we use macros.

5.1 Test Programs

In the first program, called Program I, each process accesses its own independent file. Each process writes 1 Mbyte of data to its local file and reads it back, and this writing and reading procedure is performed ten times. We wrote three different versions of this program: for PFS, PIOFS, and MPI-IO.

The second program, called Program II, is similar to Program I except that all processes access a common file. The data from different processes is stored in the file in order of process rank. Each process writes 1 Mbyte of data to a common file and reads it back, and this writing and reading procedure is performed ten times. We also wrote three different versions of this program: for PFS, PIOFS, and MPI-IO.

To determine the overhead due to ADIO, we ran three cases of each program on the SP and Paragon. The three cases run on the SP were as follows:

1. The PIOFS version run directly on PIOFS.

2. The PIOFS version run through ADIO on PIOFS (PIOFS –> ADIO –> PIOFS). This case shows the overhead due to ADIO.

3. The MPI-IO version run through ADIO on PIOFS (MPI-IO –> ADIO –> PIOFS). This case shows the overhead of using the MPI-IO interface along with ADIO.

Table 1 shows the I/O time for all three cases of the two test programs, run on 16 processors on the SP. Clearly, the overhead of using ADIO was negligible.

The three cases run on the Paragon were as follows:

1. The PFS version run directly on PFS.

2. The PFS version run through ADIO on PFS (PFS –> ADIO –> PFS).

3. The MPI-IO version run through ADIO on PFS (MPI-IO –> ADIO –> PFS).

Table 1: I/O time for the test programs on 16 processors on the SP. The three cases are: PIOFS version run directly, PIOFS version run through ADIO on PIOFS, and MPI-IO version run through ADIO on PIOFS. Time in seconds.

Pro-	PIOFS	PIOFS–ADIO		MPI-IO–ADIO	
gram	time	time	ovhd.	time	ovhd.
I	7.42	7.44	0.27%	7.44	0.27%
II	8.44	8.69	2.96%	8.67	2.72%

Table 2: I/O time for the test programs on 16 processors on the Paragon. The three cases are: PFS version run directly, PFS version run through ADIO on PFS, and MPI-IO version run through ADIO on PFS. Time in seconds.

Pro-	PFS	PFS–ADIO		MPI-IO–ADIO	
gram	time	time	ovhd.	time	ovhd.
I	14.03	14.43	2.85%	14.41	2.78%
II	12.19	12.38	1.56%	12.31	0.98%

Table 2 shows the I/O time for all three cases of the two test programs, run on 16 processors on the Paragon. The overhead of using ADIO was negligible on the Paragon as well. For both test programs, the overhead of using MPI-IO through ADIO was slightly lower than that of PFS through ADIO, possibly because the MPI-IO versions had fewer I/O function calls than the PFS versions. The MPI-IO versions did not use any explicit seek functions. Instead, they used `MPIO_Read` and `MPIO_Write` functions that use an offset to indicate the location in the file for reading or writing. The PFS versions, however, used seek calls in addition to the read and write calls.

5.2 Production Application

The application we used is a parallel production code developed at the University of Chicago to to study the gravitational collapse of self-gravitating gaseous clouds. Details about the application and its I/O characteristics can be found in [26].

The application uses several three-dimensional arrays that are distributed in a (block,block,block) fashion. The algorithm is iterative and, every few iterations, several arrays are written to files for three purposes: data analysis, checkpointing, and visualization. The storage order of data in files is required to be the same as it would be if the program were run on a single processor. The application uses two-phase I/O for reading and writing distributed arrays, with I/O routines optimized separately for PFS and PIOFS [26]. I/O is performed by all processors in parallel.

We ran three cases of the application on the SP and Paragon. The three cases on the SP were as follows:

1. The PIOFS version run directly.

Table 3: I/O time for the production application on 16 processors on the SP. The three cases are: PIOFS version run directly, PIOFS version run through ADIO on PIOFS, and the Intel PFS version run through ADIO on PIOFS. Time in seconds.

PIOFS	PIOFS–ADIO		PFS–ADIO	
time	time	ovhd.	time	ovhd.
11.22	11.47	2.23%	11.68	4.10%

Table 4: I/O time for the production application on 16 processors on the Paragon. The three cases are: PFS version run directly, PFS version run through ADIO on PFS, and the IBM PIOFS version run through ADIO on PFS. Time in seconds.

PFS	PFS–ADIO		PIOFS–ADIO	
time	time	ovhd.	time	ovhd.
22.28	22.78	2.24%	22.92	2.87%

2. The PIOFS version run through ADIO on PIOFS (PIOFS –> ADIO –> PIOFS).

3. The Intel PFS version run through ADIO on PIOFS (PFS –> ADIO –> PIOFS).

The three cases on the Paragon were as follows:

1. The PFS version run directly.

2. The PFS version run through ADIO on PFS (PFS –> ADIO –> PFS).

3. The IBM PIOFS version run through ADIO on PFS (PIOFS –> ADIO –> PFS).

We could not run an MPI-IO version, because the application has not yet been ported to MPI-IO.

On both machines, we ran the application on 16 processors using a mesh of size $128 \times 128 \times 128$ grid points. The application started by reading a restart file and ran for ten iterations, dumping arrays every five iterations. A total of 50 Mbytes of data was read at the start, and around 100 Mbytes of data was written every five iterations. The sizes of individual read/write operations were as follows: there was one small read of 24 bytes and several large reads of 512 Kbytes; there were a few small writes of 24 bytes and several large writes of 128 Kbytes and 512 Kbytes.

Tables 3 and 4 show the I/O time taken by the application on the SP and Paragon, respectively. The overhead due to ADIO was very small on both systems. In addition, ADIO allowed us to run the SP version of the application on the Paragon and the Paragon version on the SP, both with very low overhead.

6 Conclusions and Future Work

We have described a strategy for implementing portable parallel-I/O APIs by using an abstract-device interface for parallel I/O, called ADIO. We have explained the design of ADIO and its use in implementing several APIs. Our performance studies indicate that the ADIO approach enables portable implementations with very low overhead.

We believe that ADIO has tremendous potential in solving many of the problems faced by application programmers regarding lack of portable standard API for parallel I/O. Therefore, we view the work described in this paper as only the beginning of a large project. We intend to develop a complete implementation of MPI-IO and track the interface definition as it evolves through the MPI Forum. We also intend to implement ADIO on other file systems for greater portability. We intend to distribute our code freely together with the MPICH implementation of MPI [8].

We note that the ADIO interface defined in this paper may change as our implementations and studies reveal the need for providing additional/different functionality at the ADIO level. The latest definition of the interface can always be obtained from `http://www.mcs.anl.gov/home/thakur/adio`.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

References

[1] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, October 1994.

[2] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, 1994.

[3] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[4] G. Gibson et al. The Scotch Parallel Storage Systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, Spring 1995.

[5] P. Corbett et al. Proposal for a Common Parallel File System Programming Interface, Version 0.60. On the World-Wide Web at `http://www.cs.princeton.edu/sio`, June 1996.

[6] I. Foster and J. Nieplocha. ChemIO: High-Performance I/O for Computational Chemistry Applications. World-Wide Web page at `http://www.mcs.anl.gov/chemio`, February 1996.

[7] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, November 1993.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] R. Grossman and X. Qin. Ptool: A Scalable Persistent Object Manager. In *Proceedings of ACM SIGMOD 94*, 1994.

[10] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, July 1995.

[11] IBM Corp. IBM AIX Parallel I/O File System: Installation, Administration, and Use. Document Number SH34-6065-01, August 1995.

[12] Intel Scalable Systems Division. Paragon System User's Guide. Order Number 312489-004, May 1995.

[13] J. Karpovich, A. Grimshaw, and J. French. Extensible File Systems ELFS: An Object-Oriented Approach to High Performance File I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, October 1994.

[14] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Technical Report PCS–TR94–226, Dept. of Computer Science, Dartmouth College.

[15] O. Krieger and M. Stumm. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. In *Proceedings of Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, May 1996.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Version 1.1, June 1995.

[17] S. Moyer and V. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[18] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[19] N. Nieuwejaar and D. Kotz. Low-level Interfaces for High-level Parallel I/O. In *Proceedings of the Third Annual Workshop on I/O in Parallel and Distributed Systems*, pages 47–62, April 1995.

[20] J. Pool. Scalable I/O Initiative. World-Wide Web page at `http://www.cacr.caltech.edu/SIO`, September 1995.

[21] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[22] H. Stern. *Managing NFS and NIS.* O'Reilly & Associates, Inc., 1991.

[23] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[24] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. Technical Report CACR–103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, Revised May 1996. (To appear in *Scientific Programming*).

[25] 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.

[26] R. Thakur, W. Gropp, and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with special emphasis on Parallel Databases and Parallel I/O*, September 1996.

[27] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. World-Wide Web `http://lovelace.nas.nasa.gov/MPI-IO`, April 1996.

[28] S. Toledo and F. Gustavson. The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations. In *Proceedings of Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, May 1996.