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

{thakur, gropp, lusk} @mcs.anl.gov

Abstract

Portable parallel programming has been hampered by the lack of a single, standard, portable application-programmer's interface (API) for parallel I/O. Instead, the programmer must choose from several different APIs, many of which are not portable. To alleviate this problem, we have developed an abstract-device interface for parallel I/O, called ADIO. ADIO is not intended as a new API; rather, it is a strategy for implementing other APIs in a simple, portable, and efficient manner. ADIO facilitates the implementation of any existing or new API on any existing or new file system. ADIO thus enables users to experiment with different APIs, a feature that, we think, would help in the definition of a standard API. It also makes existing applications portable across a wide range of platforms.

In this paper, we introduce the concept of ADIO. We describe the design of ADIO and its use in implementing APIs. We have currently implemented subsets of the Intel PFS, IBM PIOFS, and MPI-IO APIs on both the PFS and PIOFS file systems. As a result, we are able to run IBM PIOFS applications on the Intel Paragon, Intel PFS applications on the IBM SP, and MPI-IO applications on both systems. We report performance results obtained from two test programs and one real production application on the SP and Paragon. These results indicate that the performance overhead of using ADIO as an implementation strategy is negligible.

Key words: parallel I/O, I/O interfaces, portable parallel programming

*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.

DISCLAIMER

1 Introduction

Parallel computers are being used increasingly to solve large I/O-intensive applications in a number of different disciplines. But the lack of a standard, portable application-programmer's interface (API) for parallel I/O is a limiting factor in this regard. The Message-Passing Interface (MPI) [16] has been very successful as a standard, portable interface for interprocess communication in parallel programs. However, no standard API exists for parallel I/O. (In the rest of this paper, the term API means API for parallel I/O.)

Although there is no single standard API, a number of different interfaces are supported by different vendors and research projects. Many commercial parallel file systems (e.g., IBM PIOFS [11] and Intel PFS [12]) provide their own API. Also, several research parallel file systems have their own API (e.g., PPFS [10], Galley [19], RAMA [17], Scotch [7], HFS [15], Vesta [3], and PIOUS [18]). In addition, a number of I/O libraries with special APIs have been developed (e.g., PASSION [21], Panda [20], Chameleon I/O [6], SOLAR [25], Jovian [1], and ChemIO[1]). Different APIs are used by systems that provide support for persistent objects (e.g., Ptool [9], ELFS [13], and SHORE [2]).

A group within the Scalable I/O Initiative[2] is developing a low-level interface for parallel I/O [4]. This low-level interface 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-programmer level is the MPI-IO [24] proposal that is based on MPI. However, both the parallel-I/O community and the applications community are far from a consensus about the suitability of MPI-IO as a standard API.

The problem of defining a standard API for parallel I/O is not an easy one—particularly because parallel I/O is itself a relatively new field, with limited user experience with existing APIs. To alleviate this problem, we have developed an abstract-device interface for parallel I/O, called ADIO. Our main objectives in defining this interface are:

1. to facilitate efficient and portable implementations of parallel-I/O APIs,
2. to enable users to experiment with existing and new APIs, and
3. to make applications portable across a wide range of platforms.

We stress that ADIO is not intended to be used directly by application programmers. It is also not a standardization effort: we do not propose ADIO as a standard API. Instead, ADIO is a

[1] http://www.mcs.anl.gov/chemio
[2] http://www.cacr.caltech.edu/SIO/

strategy for implementing other APIs portably and efficiently.

In this paper, we introduce the concept of ADIO. We describe in detail the design of ADIO (Section 3) and discuss its use in implementing APIs such as MPI-IO, PASSION, Panda, PFS, and PIOFS (Section 4). We have currently implemented subsets of the MPI-IO, PFS, and PIOFS APIs on two file systems—PFS and PIOFS. As a result, we are able to run IBM PIOFS applications on the Intel Paragon, Intel PFS applications on the IBM SP, and MPI-IO applications on both systems. Performance results obtained from two test programs and one production application indicate that the overhead of using ADIO as an implementation strategy is very low (Section 5).

2 The ADIO Concept

The main goal of ADIO is to facilitate a high-performance implementation of any existing or new API on any existing or new file-system interface, as illustrated in Figure 1. Any API (including a file-system interface) can be implemented in a portable fashion on top of ADIO. ADIO is, in turn, implemented in an optimized manner on each different file system separately. This approach enables the porting of applications (that perform I/O) to a wide range of platforms without committing to a particular API. For example, we have implemented subsets of the Intel PFS, IBM PIOFS, and MPI-IO interfaces on top of ADIO and implemented ADIO on top of PFS and PIOFS. Therefore, we are able to run Intel Paragon applications on the IBM SP, SP applications on the Paragon, and MPI-IO applications on both systems.

ADIO also allows us 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 top of a new file-system interface, all APIs implemented on top of ADIO become available on the new file system.

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. This approach promotes the widespread use of existing and proposed APIs. The resulting knowledge and experience gained by API developers and users should help in determining the features desirable in a standard API. We believe that such implementation and experimentation are necessary before users and I/O researchers can converge toward a standard API.

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

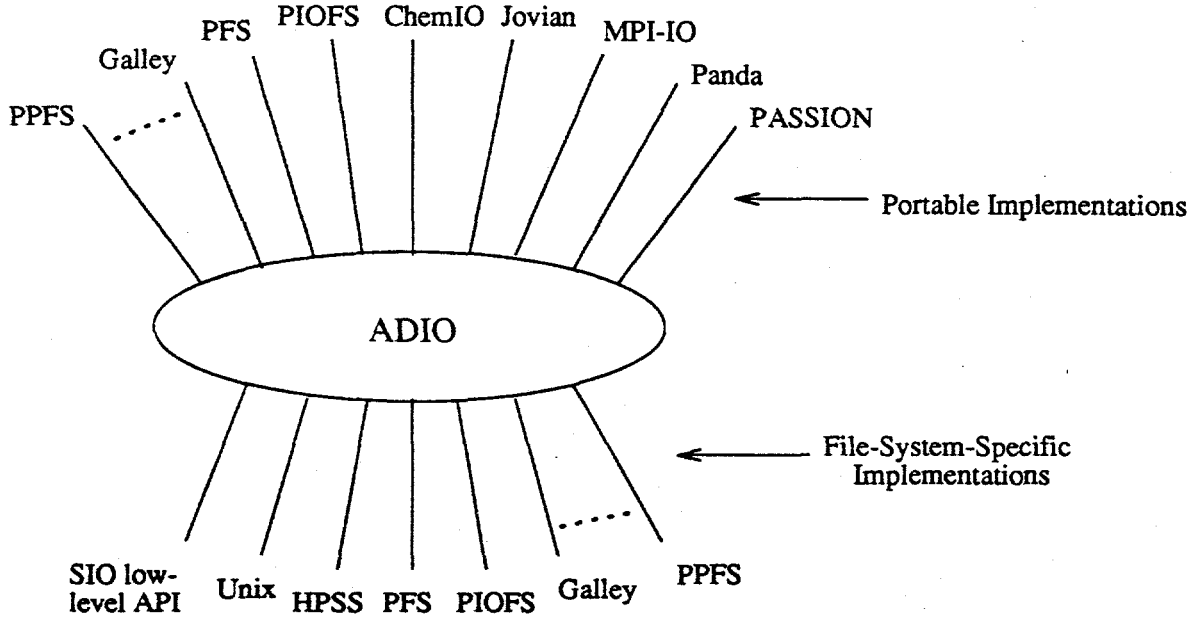

Figure 1: The ADIO concept

3 ADIO Design

ADIO is designed so 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 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, void *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, which in turn indicates the actual functions to use for I/O. The access_mode parameter specifies the file access mode, which can be 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 [24]. 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 opened file.

Close:

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

The close operation is also collective. The processes that opened the file, indicated by the communicator in the file descriptor, 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)

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

ADIO provides separate routines for contiguous and strided 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. Offsets are 64 bits long. 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 Strided 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)

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

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 strided accesses with a single call. Strided access patterns can be represented in many ways; 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 strided read and write calls. Nonblocking and collective versions are described in Sections 3.4 and 3.5, respectively.

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 stride in the file is indicated by the `filetype` (an MPI derived datatype) specified when the file was opened.

Note that `ADIO_ReadContig` and `ADIO_WriteContig` are special cases of `ADIO_ReadStrided` and `ADIO_WriteStrided`. We consider contiguous operations 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_IwriteContig(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)

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

ADIO provides nonblocking versions of all read/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 is 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_WriteContigColl(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_WriteStridedColl(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_IwriteContigColl(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)

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

Several studies have shown that, in many cases, the I/O performance of parallel programs can be improved greatly by using collective I/O [5, 22, 14]. 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 two kinds of routines for this purpose: a quick test for completion that requires no further action (ADIO_xxxxDone) and a test-and-complete (ADIO_xxxxIcomplete). Separate routines exist for read and write operations.

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

```
int ADIO_WriteDone(ADIO_Request request)
```

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)
```

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

If a request is not complete, the above routines can be used. These routines call the I/O device and perform some additional processing.

3.8 Miscellaneous

ADIO also provides routines for purposes such as deleting files, resizing files, flushing the cache, 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 top of a file-system interface. 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 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 [24] maps quite naturally to ADIO because both MPI-IO and ADIO use MPI to a large extent. In addition, a number of features were included in ADIO specifically for being able to implement MPI-IO; these include displacement, etype, filetype, the ability to use explicit offsets as well as file pointers, and file delete-on-close.

4.1.2 PASSION and Panda

PASSION [21] and Panda [20] are libraries that support input/output of distributed multidimensional arrays. I/O of this type involves collective access to (potentially) strided data. ADIO supports both collective I/O and strided accesses; therefore, PASSION and Panda can be implemented by using the ADIO routines for collective and strided accesses.

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 strided read/write calls of ADIO.

4.1.4 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.

4.2 Implementing ADIO on Top of a File-System Interface

Here we explain how ADIO can be implemented on top of the PFS and PIOFS interfaces.

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, strided requests can either be translated into several contiguous requests separated by seeks or can be implemented by using optimizations such as data sieving [21]. Collective operations can be implemented by using optimizations such as two-phase I/O [5, 22].

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. Strided accesses can be implemented, in some cases, by using the logical views supported by PIOFS. In other cases, it may be necessary to implement strided 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.

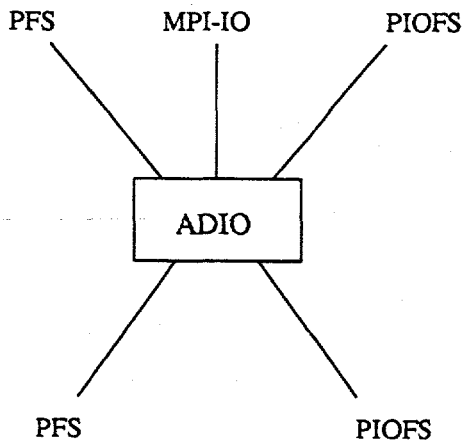

Figure 2: Current status of implementation

4.3 Current Status of Implementation

At present, we have implemented subsets of the MPI-IO, PFS, and PIOFS interfaces on top of ADIO, and we have implemented ADIO on top of PFS and PIOFS, as illustrated in Figure 2. This implementation has enabled us to run PFS applications on PIOFS, PIOFS applications on PFS, and MPI-IO applications on both PFS and PIOFS. We discuss the performance of these implementations in Section 5.

The ADIO implementation is an ongoing effort, and we intend to add other APIs and file systems to the above list.

5 Performance

To study the performance overhead of ADIO, we used two test programs and one real production parallel application. We ran these codes on the SP and Paragon, with and without ADIO. As shown below, we found the overhead due to ADIO to be negligible.

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

Table 1: I/O time on the SP for the test programs. The timings shown are for three cases of the programs: PIOFS version run directly, PIOFS version run through ADIO (PIOFS -> ADIO -> PIOFS), and MPI-IO version run through ADIO (MPI-IO -> ADIO -> PIOFS).

Program	Direct PIOFS time (sec.)	PIOFS through ADIO time (sec.)	PIOFS through ADIO ovhd.	MPI-IO through ADIO time (sec.)	MPI-IO through ADIO ovhd.
I	7.42	7.44	0.27%	7.44	0.27%
II	8.44	8.69	2.96%	8.67	2.72%

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 ADIO overhead, 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 on the SP for all three cases of the two test programs. Clearly, the overhead of using ADIO was negligible.

The three cases run on the Paragon were:

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 2 shows the I/O time on the Paragon for all three cases of the two test programs. 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 seek functions. Instead, they used `MPIO_Read` and `MPIO_Write` functions that use an

Table 2: I/O time on the Paragon for the test programs. The timings shown are for three cases of the programs: PFS version run directly, PFS version run through ADIO (PFS -> ADIO -> PFS), and MPI-IO version run through ADIO (MPI-IO -> ADIO -> PFS).

	Direct PFS	PFS through ADIO		MPI-IO through ADIO	
Program	time (sec.)	time (sec.)	ovhd.	time (sec.)	ovhd.
I	14.03	14.43	2.85%	14.41	2.78%
II	12.19	12.38	1.56%	12.31	0.98%

offset to indicate the location in the file for reading/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 study the nonlinear evolution of Jeans instability in self-gravitating gaseous clouds, a process considered to be the basic mechanism for the formation of stars and galaxies. Details about the application and its I/O characteristics can be found in [23].

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 (restart), 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 [23]. 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.
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 3 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).

Table 3: I/O time on the SP for the production application. The timings shown are for three cases of the application: PIOFS version run directly, PIOFS version run through ADIO (PIOFS -> ADIO -> PIOFS), and the Intel PFS version run through ADIO (PFS -> ADIO -> PIOFS).

Direct PIOFS	PIOFS through ADIO		PFS through ADIO	
time (sec.)	time (sec.)	ovhd.	time (sec.)	ovhd.
11.22	11.47	2.23%	11.68	4.10%

Table 4: I/O time on the Paragon for the production application. The timings shown are for three cases of the application: PFS version run directly, PFS version run through ADIO (PFS -> ADIO -> PFS), and the IBM PIOFS version run through ADIO (PIOFS -> ADIO -> PFS).

Direct PFS	PFS through ADIO		PIOFS through ADIO	
time (sec.)	time (sec.)	ovhd.	time (sec.)	ovhd.
22.28	22.78	2.24%	22.92	2.87%

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 size of $128 \times 128 \times 128$. 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 Summary and Future Work

We have described the ADIO concept for implementing portable parallel-I/O interfaces. 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 portability and lack of a standard API for parallel I/O. Therefore, we view the work described in this paper as only the beginning of a large project. We are actively expanding our implementation to include other APIs and file systems. We intend to distribute our code freely together with the MPICH implementation of MPI [8].

We are currently collaborating with several vendors to refine the ADIO interface to the most appropriate one for all systems. Therefore, 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.

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] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[4] P. Corbett, J. Prost, J. Zelenka, C. Demetriou, E. Riedel, and G. Gibson. Proposal for a Common Parallel File System Programming Interface, Version 0.52, March 1996.

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

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

[7] G. Gibson, D. Stodolsky, P. Chang, W. Courtwright II, C. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. Patterson, J. Su, R. Youssef, and J. Zelenka. The Scotch Parallel Storage Systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, Spring 1995.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. Technical Report MCS–P567–0296, Mathematics and Computer Science Division, Argonne National Laboratory, February 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] E. Miller and R. Katz. RAMA: Easy Access to a High-Bandwidth Massively Parallel File System. In *Proceedings of the 1995 Winter USENIX Conference*, pages 59–70, January 1995.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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 November 1995. Available at `http:/www.mcs.anl.gov/home/thakur/cacr-103.ps`.

[23] 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 Int'l Conf. of the Austrian Center for Parallel Computation (ACPC) with special emphasis on Parallel Databases and Parallel I/O*, September 1996. To appear.

[24] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. On the World-Wide Web at `http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps`, April 1996.

[25] 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.