A Framework for Optimizing Parallel I/O *

Robert Bennett Kelvin Bryant Alan Sussman Raja Das
Joel Saltz

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742
{robertb,ksb,als,raja,saltz}@cs.umd.edu

Abstract

There has been a great deal of recent interest in parallel I/O. This paper discusses issues in the design and implementation of a portable I/O library designed to optimize the performance of multiprocessor architectures that include multiple disks or disk arrays. The major emphasis of the paper is on optimizations that are made possible by the use of collective I/O, so that I/O requests for multiple processors can be combined to improve performance. Performance measurements from benchmarking our implementation of an I/O library that currently performs *collective local optimizations*, called Jovian, on three application templates are also presented.

1 Introduction

Recently, there has been a great deal of interest in parallel I/O. In this paper, we discuss issues for the design and implementation of a portable I/O library designed to optimize the performance of multiprocessor architectures that include multiple disks or disk arrays. We also present performance measurements obtained by linking three application templates with an optimizing I/O library and executing them on an IBM SP1 distributed memory parallel machine. The I/O library, called Jovian, performs optimizations for one form of collective I/O.

Processor speeds have increased dramatically over the last decade and to a lesser degree memory speeds and density have also increased. While I/O subsystem performance has also increased, it still lags far behind in terms of relative performance. Higher I/O bandwidths can be obtained by attaching disks or disk arrays to separate processors and declustering files among a number of disks or disk arrays. An architecture can be configured so that the disks are attached to processors which have primary responsibility for I/O or the architecture can be configured so that processors provide both computational and I/O support. A key objective of a library designed to optimize parallel I/O is to coalesce disk access requests in a way that makes it possible to present each secondary storage device with a minimal number of disk access requests, thereby reducing disk latency.

Many I/O intensive applications can benefit from an optimizing I/O library. Some of the applications areas that we are targeting include:

*This research was supported by the National Science Foundation under Grant #ASC 9318183, and NASA under Grant #NAG 11485 (ARPA Project #8874).

- Out-of-core algorithms for accessing large amounts of satellite data in spatial databases, in the area of **Geographic Information Systems**. These applications are being developed as part of the Grand Challenge project in Land Cover Dynamics at the University of Maryland [19].
- **Data mining** applications for categorizing objects in a database, currently being developed at the University of Maryland [11].
- Sparse, adaptive, and block-structured **scientific codes** with irregular data access patterns and non-contiguous accesses to large data sets on disk. Such codes come from research efforts in areas such as computational chemistry [14] and computational fluid dynamics [6, 7, 23].

A significant number of parallelized programs make use of a Single Process Multiple Data (SPMD) model of computation. SPMD programs frequently need to move large sets of data from secondary storage to each processor's primary memory. In the SPMD model, large data sets are distributed across a group of processors in some fashion to achieve parallelism. Also, the distribution of data over the parallel disk system must be taken into account when performing I/O. More often than not, the two distributions are not the same, possibly leading to a great deal of disk head movement. From I/O requests provided by the user, an optimizing I/O library can attempt to minimize the number of I/O requests to disk by coalescing them into larger contiguous requests. The library can make use of a varying number of *coalescing* processes (in effect, I/O processes that are fixed before program execution) to carry out this aggregation process. We will call this kind of optimization *collective local I/O optimization*, and the interface that is provided by such an I/O library is called a **distributed view**. This is illustrated in Figure 1(b). A one-dimensional data array is distributed cyclically across a set of application processes and by blocks across a set of I/O processes, with the I/O processes responsible for optimizing and satisfying requests to or from the disks. The library can analyze the requests and determine that the aggregate request is for logically contiguous data on disk. The aggregate request is partitioned among the I/O nodes, each of which can perform one read operation to retrieve the contiguous data that it is responsible for.

In some cases application programs have access to compact global descriptions that indicate what data needs to be transferred from secondary (*out-of-core*) storage to storage on the processors (*in-core*). For instance, this is likely to be the case when executing an out-of-core calculation involving a structured mesh or dense matrix. Therefore all application processes can share a common description that summarizes the distribution of data among the processors, and the data that needs to be transferred between secondary and primary storage. For arrays, such a *distributed array descriptor* can contain a logical description of the in-core and out-of-core data distributions and pointers to processor memory for the array. For irregularly distributed data, a structure similar to the CHAOS distributed translation table [9] can be employed to describe the data transfer. Access to compact global descriptors can allow an I/O library to skip the collective communication stage that is needed when performing collective local I/O optimization. The *collective global I/O optimizations* in such a library will be similar to the optimizations carried out by the Multiblock Parti library [1, 24] for interprocessor communication of regularly accessed data. The interface that is provided by an I/O library that implements these optimizations is called a **global view**. An example of an interface for a global view is illustrated in Figure 1(a). The I/O request is similar to the one illustrated in Figure 1(b), except that the library has access to descriptions of both the distributions of the target and destinations

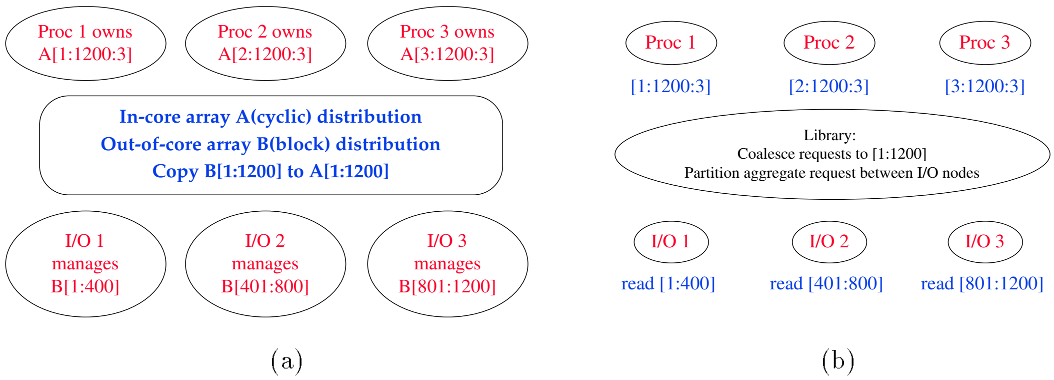

Figure 1: (a) Global and (b) Distributed Views

arrays and a compact description of the data to be transferred (a regular section copy). Each I/O process can use this information to determine what to read from disk and what application process to send the data to.

There are at least two other projects that perform collective local I/O optimizations. Choudhary [5, 10] assumes that each processor has access to either a physical or logical I/O device. They partition parallel I/O into two phases. The first phase reads data in a layout that corresponds to the logical disk layout; the second phase then performs in-memory permutations to lay out the data as required. The goal is to do the preprocessing needed so that disk requests access data in large contiguous chunks. Kotz [17] modifies this approach by having compute processors pass their disk access requests to I/O processors. In Kotz's approach, the I/O processors are responsible for coordinating the collective requests to the file system. This will make it possible for the I/O processors to carry out optimizations that require knowledge of how data is physically mapped to disk.

2 Programming Model - Accessing Out-of-Core data

2.1 Execution Model

We are interested in providing optimized I/O for programs that use the *single program multiple data* (SPMD) model. This model requires that the same program text be executed on each processor, but the input data to each of the programs may be different. This may cause the execution path taken in each program to be different. Following an SPMD model simplifies the development of both the I/O optimizing library and the applications that will use it.

We assume that an application which uses an optimizing library to perform I/O will execute in a *loosely synchronous* manner. The main feature of loosely synchronous computation is that all the participating processes alternate between phases of computation and I/O. The major implication of loosely synchronous execution within the context of the I/O library is that even if a particular process does not need to read (write) data it still has to participate in the I/O preprocessing phase. This participation may only involve

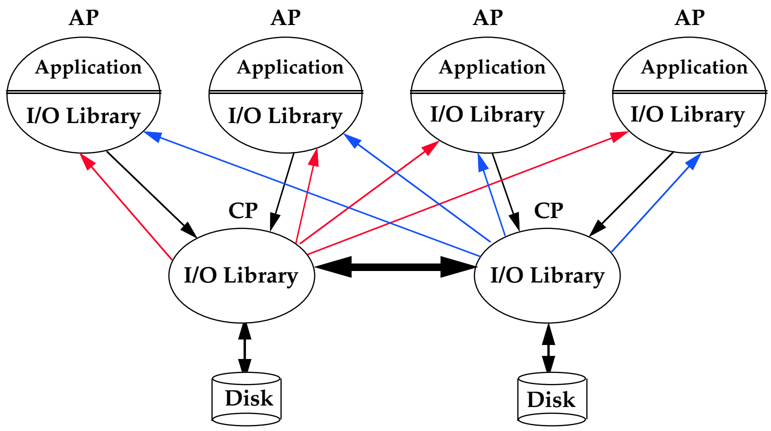

Figure 2: Application and coalescing processes, for a read request

sending a single null request, informing the I/O library that it does not require any data.

The assumption of a loosely synchronous, SPMD execution model implies that all processes that require I/O in a particular phase of the computation will implicitly synchronize their requests, much like *collective communication*. For collective communication, all processes that need to communicate at a given point in the program must execute matching communication operations. For communication, such matching means that all the sends specified by various processes must have corresponding receives posted by other processes (and for each receive there should be a corresponding send). In a similar vein, our I/O optimizations are applicable to *collective I/O* operations. By *collective I/O* we mean that every process participating in the computation must execute every I/O operation (read, write, open file, etc.).

The *collective interface* can place an unnecessary burden on application processes since it is a "barrier" at which processes synchronize. Therefore, this restriction must be relaxed such that an application that may be highly asynchronous in nature (i.e., every process requests data at different times) does not have to synchronize on a collective I/O operation. The optimizing library has to provide asynchronous I/O operations and attempt to optimize these requests. Such techniques are currently under investigation.

The model that has been used to develop the optimizing I/O library is illustrated in Figure 2. The model distinguishes between two types of processes: application processes and coalescing processes. The application processes consist of the user application and part of the I/O runtime library, and issue I/O requests, consisting of ranges of indices, to a statically assigned coalescing process. The mapping between application and coalescing processes can be one-to-one or many-to-one. The coalescing processes, which are in many aspects similar to server processes in database management systems (DBMS), are each responsible for a part of the global data structure in secondary storage. Theses processes can also be thought of as servers for persistent data (e.g. files) that are created specifically for the application.

2.2 Application Program Interface

The optimizing I/O library provides an application with a collective I/O interface that allows the library to optimize both regular and irregular access patterns to data distributed across a set of disks.

A significant portion of the resource utilization and overhead in such a library can be attributed to the preprocessing of the I/O requests from the application. That is, the structure of the request from the application can affect the overall performance of the library. In this section the Application Programming Interface (API) is described in detail. In addition, the API's impact on designing optimizing I/O libraries, in terms of memory usage, communication cost, performance and ease of integration into runtime support for parallel compilers is discussed in detail. Specific examples from the Jovian library [3] are provided to illustrate the issues for an existing design.

There are two complementary views for accessing an out-of-core data structure (residing on secondary storage), such as an array, from each process running a parallel program. With a *global* view, access to out-of-core data requires copying a globally specified subset of an out-of-core data structure distributed across disks from or to a globally specified subset of a data structure distributed across the processes (an *in-core* data structure). In order to implement such accesses, the library needs to know the distributions of both the out-of-core data structure and the in-core data structure, and also requires a description of the requested data transfer.

The in-core and out-of-core data distribution information can be defined compactly in either a single global data descriptor or multiple data descriptors. The format for such descriptors must be able to describe both regular and irregular distributions, and is in the process of being defined by the Parallel Compiler Runtime Consortium (PCRC). PCRC includes groups from Syracuse University, University of Indiana, Rice University and several other universities, in addition to our group at Maryland. The library should include methods for dealing with various descriptions of I/O requests, including entire arrays, irregular portions of arrays, and regular sections of arrays (with strides), as will be discussed in Section 3.

With a *distributed* view, each process effectively requests only the part of the data structure that it requires. In that case, the user program is responsible for translating local, in-core, data structure addresses into the correct global addresses for accessing the entire out-of-core data structure (perhaps through calls to a runtime library such as CHAOS [9] or Multiblock PARTI [1]).

A distributed view for out-of-core data structures greatly simplifies the design and implementation of a collective I/O optimizing library. With a distributed view, the user program is responsible for providing the I/O system with only the requests for that process, so the burden of translating local addresses for a distributed data structure into the corresponding global, out-of-core, addresses is on the user program. The I/O system is then only responsible for optimizing the accesses to maximize bandwidth and minimize latency to secondary storage. One way to optimize I/O requests from multiple processes using a distributed view is to restrict the requests to collective operations. In a loosely synchronous, SPMD execution model, this means that all processes must execute every I/O call, even if it is only for a process to tell the I/O system that it is not transferring any data for that request.

2.3 Impact on designing optimizing I/O libraries

The API, specifically the type of view used, is expected to have a measurable impact on the performance of an I/O library. There are several factors that must be taken into account when implementing global and distributed views. The main factors are communication overhead, memory usage and performance. Another factor that must be considered is the ease of integrating the library into runtime support for a parallelizing compiler, for example for data parallel languages such as High Performance Fortran [15] or pC++ [18].

2.3.1 Communication Overhead

Communication overheads will be incurred throughout an I/O library. This is to be expected, since the library uses the interprocessor interconnect to collect data from and distribute data to application processes during I/O operations. One of the basic assumptions driving the design of the library is that interprocessor communication is much faster than disk operations.

Preprocessing an application's data requests in an I/O library can contribute significantly to communication overhead. Depending on the structure of the requests, or more specifically, the view used, this preprocessing stage will vary in the amount of communication needed. The following discussion applies to both regular and irregular data accesses.

As described previously, an application using a distributed view can request data using global indices. There are (at least) two options for the library to handle the requests. Either the global indices can be directly forwarded to the coalescing processes responsible for optimizing the requests, or the global indices can be converted into an intermediate form and then forwarded to the coalescing processes. For example, in the Jovian library, a *block* is the intermediate form employed. *Blocks* are a unit of request that determine the amount of data read from disk and communicated to the application. With a distributed view, *blocks* must be generated in the application process part of the library and then communicated to a predetermined coalescing process. The number of *blocks* is a function of the amount of data requested by the application and the data access pattern; in general, larger requests require larger messages containing the *blocks* requested, resulting in higher communication overhead. Similarly, irregular data access patterns can increase the number of blocks required, with only a small amount of data in each block actually required by the application.

An alternative to specifying sets of global indices for requesting data is to specify regular sections (for arrays). The details of specifying regular sections are covered in Section 3. Regular sections offer a more compact format for describing a data transfer. For each dimension of the data structure being described, only *three* values are required; values for the low and high index and a stride. Using regular sections with a distributed view dictates that the regular section parameters are expressed in global terms for the out-of-core array. Once the regular section requests are passed on to the library in the application process (using a distributed view), they can be processed in the same manner as sets of global indices.

A global view provides a fixed size data structure, the distributed data descriptor, that compactly describes both the in-core and out-of-core data distributions. This data structure, along with a description of the data transfer, is forwarded to the respective coalescing process. This is all the information needed by the library to satisfy the requests. Unlike the distributed view, a global view offers the opportunity for minimizing communications overhead. The amount of savings that can be obtained depends on the format used to describe the data transfer.

The description of the data transfer can take on several forms in the context of a global view. For example, a structure similar to the CHAOS distributed translation table [9] can be used for specifying irregular accesses, and regular accesses can be specified by regular sections, global index ranges or even *blocks*. We would like to use the format that will provide the the smallest possible overhead. Regular sections are suitable for regular accesses. Since, in practice, most large structured data sets model two- or three-dimensional physical grids, the size of a regular section request will be fairly small. Therefore, the overhead of communicating a fixed size descriptor and regular section information should be relatively small across all request sizes.

Using a distributed view requires implementing an exchange phase between the coalescing processes. In the Jovian library, the application side of the I/O library sends its data requests to a statically assigned coalescing process. Ultimately, this process may not be able to satisfy all of the requests. Therefore, the requests that cannot be satisfied must be forwarded to the coalescing process that can satisfy them. All coalescing processes must participate in the exchange phase (i.e., an all-to-all communication phase). This phase is not needed for a global view. Since each coalescing process has access to the data distribution descriptor which describes both the in-core and out-of-core data distribution, each process needs only to look at the I/O request and the distribution descriptor to determine what subset of the requested data it is responsible for.

2.3.2 Memory Overhead

Another factor that has to be considered in the design of optimizing I/O libraries is memory overhead. Since applications and linkable runtime libraries are allocating memory from the same pool, memory allocation should be minimized in the library.

From our experience with the Jovian library, a distributed view needs several data structures for storing accounting information. For example, a Jovian read operation requires a structure to hold the index information for each *block* for easy retrieval during the phase when data from the coalescing processes is copied to the user data structure. For a write operation similar information must be stored. Therefore, for an application accessing a large out-of-core data structure, the memory overhead of storing accounting information (i.e., lists of indices for an irregular access pattern) for thousands of *blocks* can be significant.

For a global view, storing regular section information requires very little memory. It is possible to calculate the needed ranges of indices within the requested data (possibly *blocks* can be used internally) on an "as needed" basis, using the information provided by the global data descriptor and the regular section requests. Although this may increase the computation overhead in the I/O library, it can potentially save a significant amount of memory.

2.3.3 Performance

The total overhead incurred by the library is composed of three main components:

1. communication,
2. memory-to-memory copy operations, and
3. computation.

In the context of an I/O library, computation is a "catch-all" term for frequently performed operations such as sorting and searching lists of *blocks*, translating ranges of indices and regular section information into *blocks* (i.e., requiring integer division operations) and regular section analysis. Memory-to-memory copy operations are necessary and their cost should be the same independent of the view employed.

The view determines which of the overhead operations are performed within the library. For example, a global view would require that regular section analysis be performed to determine the ownership of the data to be read or written. This operation is currently implemented in Multiblock PARTI [1], and the associated computational overhead is not very high (linear in the number of dimensions of the distributed array).

During the disk I/O optimization phase in the Jovian library, in which the requested blocks are coalesced, each coalescing process attempts to convert requests for consecutive *blocks* into a single disk operation. To accomplish this, the total list must first be sorted and then searched. In this case a distributed view is assumed to have been used to generate the requests.

Unlike the optimizing phase for a distributed view, the same phase for a global view will not require sorting *blocks*. Since the participating coalescing processes can determine the data they are responsible for, as stated earlier, the *blocks* can be computed on an "as needed" basis and coalescing can be performed dynamically.

2.3.4 Runtime library support for compilers

Before attempting to discuss integrating an optimizing I/O library for out-of-core data into the runtime support for a parallelizing compiler, a general model for out-of-core compilation must be outlined.

The general model for out-of-core compilation relies on having the compiler generate calls to various runtime libraries to handle both I/O and interprocessor communication. For communication, our techniques have been validated for irregular, block structured and structured data access patterns [1, 8, 20, 21]. Extending these techniques to handle out-of-core data structures requires using the local memories as "caches" for out-of-core data. The problem then becomes keeping track of the data currently available in memory. For data that is irregularly distributed across processes, a data structure analogous to the CHAOS distributed translation table [9], can serve to store the locations (process and local address) of data currently available in the local memory of some process. We call this data structure an *out-of-core data access descriptor*.

Since irregular data access patterns usually cannot be analyzed at compile time, the compiler has to generate code to initialize and maintain the out-of-core data access descriptor at runtime and then generate code to use the descriptor to move data to the right place at the right time. Instead of generating low-level communication and I/O statements, the compiler can use high-level calls to both communication libraries (e.g. CHAOS or Multiblock PARTI) and I/O libraries to manage the data. The I/O library can be used by the compiler to move data from secondary storage into local memories (the equivalent of a gather, for data irregularly distributed across a distributed memory machine) and from the local memories back to secondary storage (the equivalent of scatter, for irregularly distributed data). For more regular data distributions, the compiler can generate the equivalent of regular section copies between local memories and secondary storage. This functionality is provided by a global view and can be based on existing implementations of this capability for moving data between local processor memories, such as that provide by CHAOS and Multiblock PARTI.

A global view of out-of-core data structures is the natural view for a compiler for a parallelizing compiler for languages such as HPF or Fortran D. Since a program written in such a language uses a shared address space model for all distributed data structures, any I/O statements in the source code can be directly translated into I/O statements in the code generated for each process. Global I/O routines require knowledge of the user specified data distributions to determine the portion of the data structure that is requested by each process. Since the source program to the compiler uses a global view for the data structures, the most straightforward way for the compiler generated parallel code to interact with the I/O system is through a distributed data descriptor using a global view. Since we are targeting HPF-like languages which support regular sections for describing array sections with regular distributions, much of that information can be passed directly down to the library via a global view without much work from the compiler.

3 Programmer Interface

In this section, we discuss the types of accesses to out-of-core data structures (usually arrays) that an optimizing I/O library can provide to the application program. All of the interfaces can be used from either a global or distributed view of the data, and the implications of which view is used are also discussed in this section. We assume that a file, either on a conventional (sequential) or parallel file system, has been opened and a file pointer returned, so that the processes that directly access the file (coalescing processes for the Jovian library) can perform reads and writes using the appropriate calls to the file system.

The first interface that is provided is for reading or writing a set of individual elements from or to the out-of-core data structure. This is the most general form of request, but requires the application program to specify each element separately, which can cause large overhead in an optimizing I/O library. For a distributed view, the interface requires that each application process provide the global index (or set of indexes) for each element it requires for an out-of-core array, or a direct file address for a non-array data structure. In addition, a local (on-processor) address must be provided for each element, to either place the data for a read, or get the data for a write. If the size of an out-of-core array is known, the optimizing I/O library can perform the translation of the global indexes into file addresses. On the other hand, for a global view of a distributed array each application process (or just one) specifies the global indexes (sets of indexes) for the elements required by *all* processes, along with a descriptor that defines the mapping of global indexes to processes and local indexes in the process that owns a particular global index. The descriptor provides all the information required to place data in memory for a read, or to obtain data from memory for a write. Optimizing disk accesses is much easier than for the distributed view, because all the data required by all processes is provided in a single request. For a non-array out-of-core data structure, the data descriptor must provide a way of mapping out-of-core objects (specified by file addresses) into an in-core data structure in the application processes.

The second interface is for contiguous ranges of indexes in a distributed array. This provides a compact form for specifying I/O for common data distributions across processes, and can significantly decrease both the overhead of making requests from the application to the I/O library and the overhead required in the library to optimize accesses to the file system. For a distributed view, each application process would provide the global index ranges that it requires to be read or written, along with a local address to store each range, and the I/O library would be responsible for translating the index ranges into file addresses. For a global view, the application processes would all make the same request to the I/O library for the set of ranges that *all* processes collectively require, and also provide a distributed array descriptor. The I/O library is responsible both for mapping the ranges into file addresses and for mapping the global request into separate requests for each application process using the distributed array descriptor, to either put the data into the local memory of the appropriate application processes for a read or obtain the data from the local memories for a write.

The least general, but most compact, interface allows for accessing a regular section [13] of an out-of-core distributed array, with non-unit strides allowed in each dimension. A range with stride (lower bound index, upper bound index, stride) is the one-dimensional case of a regular section. Regular sections compactly describe Fortran 90 array notations, and also allow for descriptions of local sections of distributed arrays for regular data distributions in languages such as High Performance Fortran (HPF) [15]. I/O requests based on regular sections can be very compact, requiring only three values for each array dimension (low, high, stride), so can minimize the cost of making requests to the I/O system. For a distributed view, each application process provides the regular sections that it requires, in global indexes, along with a local address for each regular section, and, as for the other interfaces, the I/O library is responsible for translating the requests into file addresses. With a global view, all the application processes would request the same set of regular sections, also providing a distributed array descriptor; however the I/O library is now responsible both for mapping the regular sections into file addresses and for mapping the request into the local memories of each application process. Regular section analysis has been studied in detail in the context of compilation for distributed memory machines [2, 4, 12, 16, 22], and such analysis can be applied to accessing out-of-core arrays as well. The analysis is inexpensive for the most common data distributions, so the regular section interface should provide good performance. The regular section interface, with a global view, is likely to be the most appropriate target for a compiler for data parallel languages such as HPF, since that is the way out-of-core distributed array accesses will be specified.

4 Experimental Results

In this section, we discuss the functionality of the Jovian library, and present results from measuring the performance of the library under three scenarios.

4.1 Jovian Library

As described in Section 2, the Jovian library consists of two types of processes: application processes and coalescing processes, henceforth referred to as A/Ps and C/Ps. Initially, when an application that has been linked to the library is run on a parallel system, there is no distinction between A/Ps and C/Ps. Jovian provides an initialization routine that employs a user specified map to determine which processes will run the application and which will perform coalescing of I/O requests. Jovian also provides a small set of routines for file management, such as opening and closing distributed files.

The library provides a set of routines for performing *collective* read and write operations using a distributed view. The implementation of such operations proceeds in several phases:

Request: During the *request* phase each A/P makes a Jovian I/O call to read or write data. Control is passed to the Jovian library module in the A/P. As described in the Section 3, the library creates the required block requests from the various types of I/O calls and forwards the block requests, on behalf of the A/Ps, to a statically assigned C/P.

Exchange: In the *exchange* phase, each C/P determines the requests that it can satisfy from the entire set of requests that it has received. The requests that cannot be satisfied locally are forwarded to the appropriate C/P in a collective communication phase (i.e., all C/Ps participate). In general, different in-core distributions will result in different communication patterns and overheads during this phase.

Read and Return Blocks: During a Jovian read operation, the C/Ps read the requested blocks from disk and send them directly to the requesting A/P. This process may require multiple messages to be sent, since the C/Ps coalesce block requests into sequential sets of contiguous block; therefore a single disk read could contain data destined for more than one A/P. Data is returned to A/Ps immediately after the C/P disk read buffer is filled.

Send and Write Blocks: During a Jovian write operation, C/Ps create communication schedules to receive data from A/Ps. C/Ps compute which blocks are to be written during each coalesced write operation. This information is broadcast from each C/P to every A/P. Using the schedule, A/Ps then asynchronously send data blocks to the appropriate C/Ps. More than one data block may be transferred per message. Each time a buffer is filled on the C/P, data is written to disk.

All of the file operations are done through the native file system interface supported on the target machine, so the Jovian library is portable across multiple distributed memory platforms, including networks of workstations.

4.2 Experiments

This section presents experimental results obtained on the 128 node IBM SP1 at Argonne National Laboratory. Three application templates, each modeling structured grid applications, were used to evaluate the performance and scalability of the Jovian library. For each application, the out-of-core data is a two-dimensional array striped across the disks (C/Ps) by blocks of rows (in HPF terms, with a *(block, *)* distribution). To test the scalability of our library, we fixed the number of A/Ps ($P_{A/P}$) and varied the number of C/Ps ($P_{C/P}$) up to the number of A/Ps. We performed extensive scalability tests for the first application described in Section 4.3; the A/P to C/P ratio was set to 1:1 in the benchmarks described in Sections 4.4 and 4.5. Each C/P had access to a *local* $\sim 1GB$ SCSI disk, with a maximum transfer rate of about 3MB/sec.

The structured grid applications templates were parallelized using the Multiblock PARTI runtime library [1]. This library provides the functionality to lay out distributed arrays in a user specified way. For two of the application templates, the in-core structured grid was partitioned across processors in both dimensions (in HPF terms, with a *(block,block)* distribution). This implies that an $N \times N$ in-core grid is partitioned into $P_{A/P}$ sub-grids each of size $N/\sqrt{P_{A/P}} \times N/\sqrt{P_{A/P}}$. In the third template, the $N \times N$ structured grid was divided into $N/P_{A/P} \times N$ sized subgrids by blocks of rows (an HPF *(block,*)* distribution). Multiblock PARTI also provided the functionality for translating local in-core indexes into the out-of-core global indexes required for the Jovian I/O routines; the Jovian routines provide a distributed view.

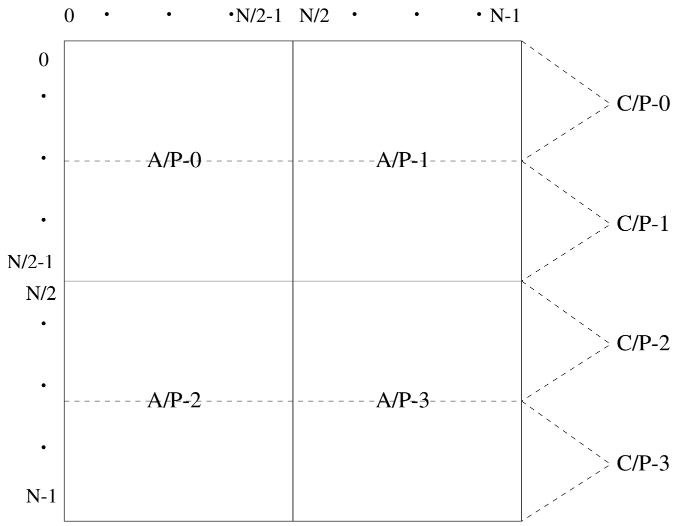

Figure 3: Distribution and Striping of Structured Grid - 8 A/Ps

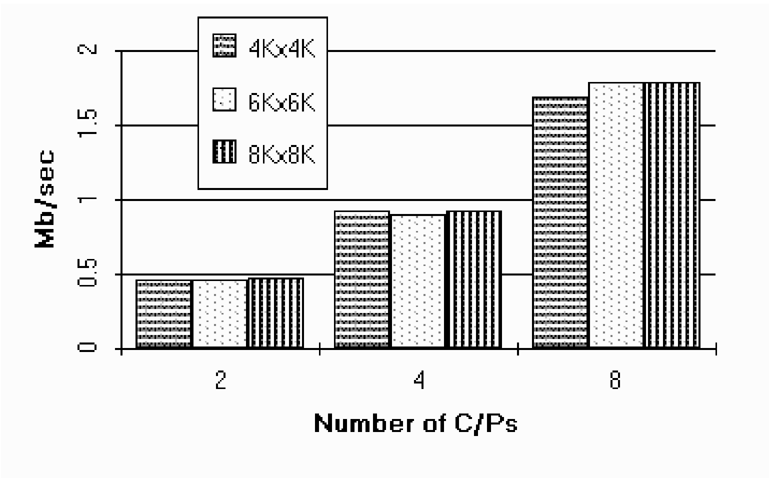

Figure 4: Non-Conforming Distributions, Minimum I/O Bandwith to an A/P - 8 A/Ps

4.3 Structured Grid Read: (BLOCK, BLOCK) In-Core Distribution

The first application template involved reading an $N \times N$, (block, block)-distributed, in-core structured grid from disk (stored by blocks of rows on the C/Ps) on 8, 16 and 32 application processes, with the number of coalescing processes ranging from 2 up to the number of A/Ps. An example of the distribution of the grid across A/Ps and C/Ps is illustrated in Figure 3 for four A/Ps ($P_{A/P} = 4$) and four C/Ps ($P_{C/P} = 4$). Note that each A/P requests its *entire* subgrid.

The out-of core array size was varied from $4K \times 4K$ double precision floating point numbers (128 MB) up to $8K \times 8K$ (512 MB). The performance results are shown in Figures 4, 5 and 6. The measured raw disk transfer rate for the C/Ps averaged about 2.0 MB/sec.

There are several observations that can be made from these results. First, as the number of C/Ps are increased, the *per* A/P transfer rate increases. This is to be expected because the data is evenly distributed

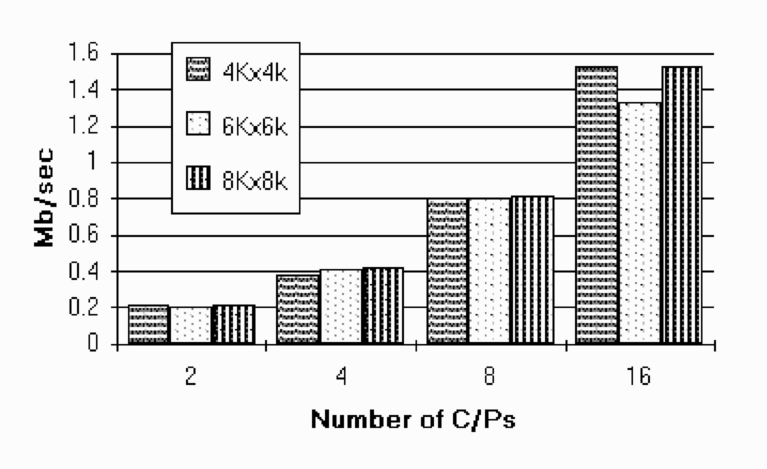

Figure 5: Non-Conforming Distributions, Minimum I/O Bandwith to an A/P - 16 A/Ps

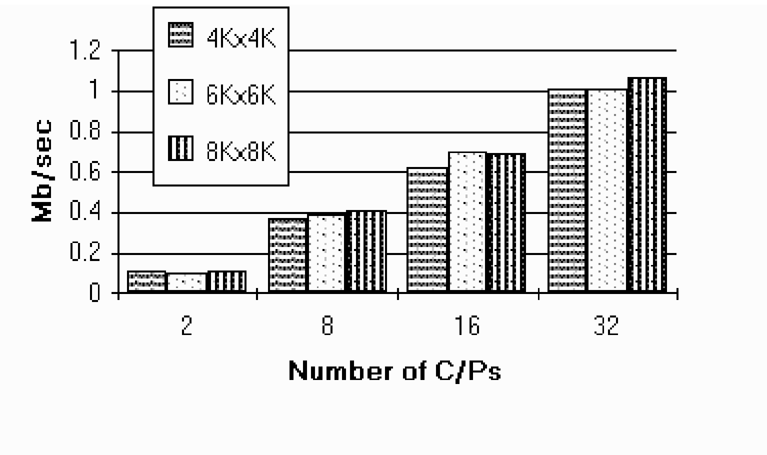

Figure 6: Non-Conforming Distributions, Minimum I/O Bandwith to an A/P - 32 A/Ps

Global Grid Size	#A/Ps	#C/Ps	Fastest Disk Read		Slowest Disk Read	
			Disk Read	Jovian Read	Disk Read	Jovian Read
$4K \times 4K$	8	8	2.2	1.8	1.9	1.6
	16	16	2.2	1.8	1.6	1.4
	32	32	2.2	1.6	1.8	1.3
$6K \times 6K$	8	8	2.2	1.9	2.1	1.8
	16	16	2.2	1.8	1.8	1.5
	32	32	2.2	1.8	1.7	1.5
$8K \times 8K$	8	8	2.2	1.9	2.2	1.9
	16	16	2.2	1.9	2.1	1.8
	32	32	2.2	1.8	1.8	1.5

Table 1: Performance results for (block, *) distributed grid. The numbers in cols 4-7 are in MB/sec

across a larger number of C/Ps, so the additional overhead of sending more messages (from more C/Ps) to satisfy the I/O requests is insignificant compared to the higher disk read performance provided by additional C/Ps. Also, the library achieves approximately the same transfer rates across all data set sizes for the same A/P-C/P configuration.

A final observation is that for process configurations in which 2 C/Ps serviced requests, there is a significant non-uniformity in the transfer rates for the A/Ps involved. For example, with 8 A/Ps and 2 C/Ps, four A/Ps had transfer rates that were a factor of 2 higher than the others. We believe that this performance difference can be attributed to the implementation of the library, which we are currently working on. For this reason, the transfer rates in the figures are the minimum rates achieved by any A/P, thus accounting for somewhat lower transfer rates for a very small number of C/Ps.

4.4 Structured Grid Read: (BLOCK, *) In-Core Distribution

This application template is similar to the one presented in Section 4.3, except it utilizes an in-core (*block,**) distribution. Thus the in-core array distribution matched the out-of-core distribution.

The performance results for this application are shown in Table 1. Configurations with ($P_{A/P} = P_{C/P}$) were executed to allow a one-to-one mapping between A/Ps and C/Ps. That is, each C/P serviced the requests of *one* A/P. There are two main implications of this configuration for a read operation (this should also hold true for a write operation). First, the overheads for the *exchange* phase are minimized since C/Ps can satisfy requests with a minimum amount of communication. Second, the data read from a single C/P read does not have to be sent to multiple A/Ps. This saves processing and communication overhead.

Minimum and maximum disk transfer rates and their associated total *per* A/P transfer rate are given in Table 1. While running the experiments, the disk transfer rates varied considerably across the C/Ps. The variability in the disk transfer rate can be attributed to the fact that, on the IBM SP1, a Unix filesystem is run on each local disk and there is no guarantee that the physical layout of the striped file will be the same even though the local file size is the same across all C/Ps. Factors such as fragmentation and rotational delay become more evident in the read times. By showing the minimum and maximum rates, we can see that variability in the total *per* A/P transfer rate is a strong function of the associated disk transfer rate. The overhead of the library, which causes the difference between the disk read rate and the total *per* A/P

Global Grid Size	A/P Write Rate	
	4 A/Ps, 4 C/Ps	8 A/Ps, 8 C/Ps
4Kx4K	0.89	0.84
6Kx6K	0.96	0.81

Table 2: A/P write for (block, block) distributed grid (MB/sec)

rate, shows only small variations.

Although the A/P read rates were higher for this distribution than the rates obtained from the (*block, block*) distribution presented in Section 4.3, they were not significantly higher. We can compare the last column of the Table 1 with the last group of bars in Figures 4 through 6 (the last group of bars are the cases where reads were performed with $P_{A/P} = P_{C/P}$). One of the goals of an optimizing I/O library is to minimize the effect of having different in-core and out-of-core distributions. Future experiments will test additional in-core and out-of-core distribution combinations.

4.5 Structured Grid Write: (BLOCK-BLOCK) In-Core Distribution

The third application template performed a collective write of a *(block, block)*-distributed structured grid. Because of space limitations, we only show test cases to characterize the general behavior of Jovian writes. Table 2 shows the performance results. Comparing the transfer rate of the write operations with the corresponding (1:1 ratio of A/Ps and C/Ps) read transfer rates from Section 4.3, the write performance is about half the read performance. This is due to the fact that the library performs a *read-modify-write* on the disk blocks. This could potentially save time when there are many non-contiguous block requests remaining after coalescing (as is the case for unstructured meshes). The test application made contiguous requests, which eliminated the need for the read operation in the *read-modify-write*. We are working on a revision of the library that will optimize the write process by only performing the read-modify-write if not all of a block is modified.

5 Conclusions

We have presented the design of an optimizing collective I/O library. The library is intended to optimize the performance of distributed memory parallel architectures that include multiple disks or disk arrays. We have described two distinct views for accessing an out-of-core data structure: a global view and a distributed view. The global view requires the I/O library to have access to both the in-core and out-of-core data distributions. Each application process requests I/O by providing the I/O library with a globally specified subset of the data structure. The distributed view requires more work by the application, because each application process must convert local in-core data indices into global out-of-core indices before making requests to the I/O library.

We have also presented measurements obtained from benchmarking the library using three applications templates on the IBM SP1. We varied both the ratio of application processes to coalescing processes and the size of the out-of-core data structure. There were other factors related to performance that we could not control (such as file layout on a disk); however, the benchmarks indicated that the library overhead was small and that high I/O bandwidth can be obtained using these methods. In addition, the I/O performance does not suffer greatly even if the in-core and out-of-core data distributions are not the same.

In the future, we plan to modify the Jovian library to provide a global view to ease the burden of both application programmers and parallelizing compilers for data parallel languages such as HPF. In addition, we are working with the Scalable I/O Consortium to define a standard parallel I/O interface, to increase the portability of applications across architectures.

Acknowledgements

This research was performed on the IBM SP1 at Argonne National Laboratory, on the Intel Paragon at Rice University and on the Intel Paragon operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to the Rice and CCSC Paragons was provided by the Center for Research on Parallel Computation. We wish to gratefully acknowledge the willingness of these organizations to provide access to their multiprocessors.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[2] Gagan Agrawal, Alan Sussman, and Joel Saltz. On efficient runtime support for multiblock and multigrid applications: Regular section analysis. Technical Report CS-TR-3140 and UMIACS-TR-93-92, University of Maryland, Department of Computer Science and UMIACS, September 1993.

[3] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1994.

[4] Siddhartha Chatterjee, John R. Gilbert, Fred J.E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 149–158, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[5] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, September 1994. Also available as CRPC Report CRPC-TR94483.

[6] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56. IEEE Computer Society Press, October 1993.

[7] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *AIAA Journal*, 32(3):489–496, March 1994.

[8] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[9] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994. Also available as University of Maryland Technical Report CS-TR-3163 and UMIACS-TR-93-109.

[10] Juan Miguel del Rosario and Alok N. Choudhary. High-performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[11] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD Conference*, May 1994. Also available as University of Maryland Technical Report CS-TR-3190 and UMIACS-TR-93-131.

[12] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed memory machines. In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[13] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[14] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodoscek. Parallelizing molecular dynamics programs for distributed memory machines: An application of the CHAOS runtime support library. Technical Report CS-TR-3374 and UMIACS-TR-94-125, University of Maryland, Department of Computer Science and UMIACS, November 1994. Submitted to IEEE Computational Science and Engineering.

[15] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[16] Charles Koelbel. Compile-time generation of regular communication patterns. In *Proceedings Supercomputing '91*. IEEE Computer Society Press, November 1991.

[17] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Department of Computer Science, Dartmouth College, July 1994.

[18] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings Supercomputing '93*, pages 588–597. IEEE Computer Society Press, November 1993.

[19] Rahul Parulekar, Larry Davis, Rama Chellappa, Joel Saltz, Alan Sussman, and John Towhshend. High performance computing for land cover dynamics. In *Proceedings of the International Joint Conference on Pattern Recognition*, September 1994.

[20] Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions in FORTRAN 90D/HPF compilers. Technical Report CS-TR-3268 and UMIACS-TR-94-57, University of Maryland, Department of Computer Science and UMIACS, May 1994. To appear in IEEE Parallel and Distributed Technology, Spring 1995.

[21] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Also available as University of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

[22] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[23] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992. Papers presented at the Symposium on High-Performance Computing for Flight Vehicles, December 1992.

[24] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.