Applications-Driven Parallel I/O *

N. Galbreath[†] W. Gropp D. Levine

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4801

Abstract

We investigate the needs of some massively parallel applications running on distributed-memory parallel computers at Argonne National Laboratory and identify some common parallel I/O operations. For these operations, routines were developed that hide the details of the actual implementation (such as the number of parallel disks) from the application, while providing good performance. An important feature is the ability for the application programmer to specify that a file be accessed either as a high-performance parallel file or as a conventional Unix file, simply by changing the value of a parameter on the file open call. These routines are examples of a parallel I/O abstraction that can enhance development, portability, and performance of I/O operations in applications. Some of the specific issues in their design and implementation in a distributed-memory toolset are discussed.

1 Introduction

In order to run Grand Challenge computational science applications on massively parallel processor (MPP) systems in a production mode, both high-performance parallel I/O systems and programmer-friendly software will be required. Unfortunately, I/O systems and software have been largely neglected by both high-performance computing researchers and vendors. Issues such as I/O as a bottleneck, the scalability of I/O architectures, and ease of use of I/O software remain largely unexplored. There has been little experimental study of I/O access patterns and types of usage, particularly in the context of scientific computing. We believe the first step in developing appropriate parallel I/O (PIO) software is an analysis of the I/O requirements of a few key application domains. An equally important second step is the evaluation of different implementation strategies. In this paper we report on the results of a study of parallel I/O requirements at Argonne National Laboratory (ANL) and a distributed-memory system we have developed to support these requirements. In order to meet the requirements, this system has been designed to provide high-level I/O operations, such as "write array to parallel file," rather than the more elemental I/O operations, such as "write byte stream to parallel file." This approach has the advantage that, to the applications programmer, parallel I/O does not appear very different from sequential I/O. This simplifies the process of developing and debugging a program, as well as investigating various algorithms for implementing the high-level parallel I/O operation. Del Rosario and Choudhary [?] provide a recent overview of many issues in high-performance parallel I/O.

This paper is laid out as follows. In Section 2 we discuss the types of I/O being done in some computational science applications at ANL and the methods we use to abstract out the main ideas. In Section 3 we describe the `PETSc/Chameleon` package used in our implementation and the parallel I/O routines we have developed. Finally, in Section 4 we make some concluding remarks and discuss our future plans.

2 I/O Requirements of Computational Science Applications

2.1 Types of I/O Usage

To decide what types of parallel I/O functionality we wanted in our system, we began by examining the I/O requirements of various Grand Challenge computational science applications at ANL that were running on distributed-memory computers. We chose to

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[†]Current address: Boston University.

focus on the I/O needs of the applications rather than the expression of those needs in current code so as to avoid being biased by code written for existing I/O interfaces.

Applications perform I/O for various reasons. The most obvious is to output the results the program computes. Some other reasons are to initialize the program, to hold in temporary scratch files data structures that do not otherwise fit in main memory, and to save the state of a job so that it can be restarted later. While each of these requirements is routinely handled in a sequential program, MPP systems raise a number of questions. Among these are the following: What programming model is being used? Should all processors write to the same file, or each to its own file? In what format should the data be stored (e.g., different formats may be needed depending on whether the program is being debugged or run in production mode)? Below we provide more detail about the I/O requirements of computational science applications at ANL.

2.1.1 Input

Most programs need to read some data to initialize a computation. The input data varies in size from small files containing a few important parameters to large data files that initialize key arrays and databases. Also, some applications require periodic input, such as boundary information or other datasets that occur at some interval for the duration of a run, *not* just initially.

Key questions include whether a replicated or distributed data structure is being read, and if a distributed data structure is being read, which data to map to which processors, and whether each processor executes the read or whether only one processor executes the read (and broadcasts to all others, in the case of a distributed-memory programming model).

One application at Argonne is mesoscale climate modeling. Future work with this model will investigate four-dimensional data assimilation. This involves input of data from observations and potentially also weather radar. This data may be obtained in real time or from tape. It is quite likely that the speed and frequency of this data acquisition will become the limiting step in the calculations unless the I/O capabilities of the MPP machines being used can keep pace with CPU performance.

2.1.2 Debugging

A common need for performing I/O arises when debugging a parallel program. We note in particular the somewhat complicated case where the parallel calculation of a distributed data structure is being compared with the same calculation done by the original sequential program. Questions arise as to how to write the parallel data structure so that it may "easily" be compared with the sequential calculations and so that the differences may be "easily" isolated.

As one example, an electromagnetics code in use at ANL solves a nonlinear problem by repeatedly solving a system of linear equations. In the parallel program the matrix is distributed by rows. At each nonlinear iteration, an iterative parallel linear solver is called to solve the current system of equations. From the solution a new matrix is generated and solved, and this process continues until convergence. Debugging the parallel code required comparing of the sequential and parallel versions of the matrix to see whether they were the same. In a second example, subtle boundary condition errors were occurring on the faces of a cube in a three-dimensional superconductivity code. Again, to isolate the parallel bug required comparisons of the sequential and parallel values of those faces, which changed each time step.

It is typical in debugging a sequential program to compare results before and after a "fix". On Unix systems a common way to do this is using the `diff` command to test for differences. An aim of our system is to provide the user the capability when debugging to have a parallel file written in a conventional Unix format so that traditional debugging methods may be used.

2.1.3 Scratch Files

Scratch files are often used to hold data structures containing intermediate calculations that do not fit into main memory or that a user does not wish to recompute. Speed of access and file size are usually the important considerations for these files.

For example, at the heart of a computational chemistry program is an iterative algorithm to compute the lowest eigenvector and eigenvalue of a large, sparse symmetric matrix. The algorithm requires the computation of matrix-vector products of the matrix with various trial vectors. The matrix is not stored, but rather the matrix-vector products are computed using knowledge of the underlying structure of the matrix. However, the trial vectors and corresponding matrix-vector products must be stored during the iterative

procedure. These vectors are stored by distributing them across the memories of the processors. In order to study larger molecular systems, an efficient means to handle larger vectors must be developed. This will require paging of the distributed vectors to secondary storage, and caching in main memory only a small fraction of these vectors at each node. For such a scheme to work, the algorithms, software, and hardware involved must be efficient and robust so that the I/O activity involved at each iteration does not become an overwhelming bottleneck.

A similar storage requirement arises in the n-body portion of a computational biology application. A portion of the calculation requires the determination of forces on the atoms resulting from quantum mechanical effects. The calculation of the quantum mechanical effects can require the evaluation of tens of millions of integrals. Since the value of these integrals remains constant for a particular configuration of atoms, a tradeoff is to either store the integrals to disk or recalculate them in order to avoid expensive I/O accesses. The number of integrals required increases as the cube of the number of electrons. The ability to access these integrals from main memory can make a difference in calculation speed by at least one to two orders of magnitude. Hence, the type of calculations that are feasible is partly limited by the I/O access speed.

2.1.4 Checkpoint/Restart

For long-running production codes, it is desirable to have the ability to save the state of the computation in order to continue computing from that point at a later date. Most operating systems on traditional, sequential high-performance computers provide such a capability at the operating system level. Sometimes an application may have such a facility built into the logic of the program.

In a distributed-memory context, a checkpoint requires saving one or more distributed arrays as well as (usually replicated) scalars. A key question is whether to save to one file or many and, if so, in what order. Another consideration is, whether the restart can be done *independently* of the number of processors the checkpoint was taken on. This is the most general case. However, it requires that only one checkpoint file be written. This in turn requires that the distributed data structures be written to a single file in their natural sequential format. Most users implementing their own parallel checkpoint/restart capabilities have tried to avoid this single file approach because of the code complexity involved in having each processor calculate the global location(s) for its data values. As a result, most current checkpoint/restart efforts *are* dependent upon the number of processors used. Finally, an additional complication arises if checkpoint/restart capabilities are already designed and deeply incorporated within an existing sequential code. Here, the parallel programmer is faced with the prospect of ripping out all the old sequential code and designing a new parallel version.

2.1.5 Output

Output from a program takes many forms; it can be a small file with a few results or a large file containing all the values from several arrays. Sometimes the output is postprocessed and only a small subset of the data actually used. Often time-dependent data is involved, and large files are generated periodically.

As examples, two codes used to model high-temperature superconductivity and global climate change, respectively, at ANL are time-dependent and three-dimensional. In both codes, the algorithmic kernel is the numerical solution of a time-dependent partial differential equation. The main computational data structure is a grid, and users are interested in the value of one or more parameters defined at each grid point.

These codes produce several types of output: (1) "small" result files containing the values of a few "interesting" parameters every k time steps; (2) files containing those "interesting" parameters sampled more frequently and output in a form for use in a postprocessing tool that plots the value of the parameter as a function of the time step; and (3) files that hold results for each grid point for several different parameters of interest. An individual file is created for each discrete time step, and the results are analyzed with the aid of a postprocessing graphics package.

2.2 I/O Abstractions

Abstraction is a key idea in contemporary computer science. The idea is to use a series of layers so that the higher layers mask the inconvenience of working directly with the lower layers. A common example is the translation of high-level languages to assembly language and then to machine code. Abstractions are also commonly used in I/O where, for the high-level language programmer, they can hide details such as buffering, actual layout of a file on a device, or even the type of storage media being used.

We believe abstractions to be even more important for parallel I/O because of the additional complica-

tions. First, there are hardware considerations. Modern parallel systems have a wide variety of storage devices intended to provide both high-performance and large capacity. Currently, there are as many approaches to parallel I/O on MPP systems as there are MPP vendors. Second are the software considerations. Parallel computing introduces a number of new concepts such as distributed and replicated data structures. This situation is further complicated by the wide variety of parallel programming models in current use.

One decision that must be made early in the design of a parallel file system is how the bytes are mapped from the application to the file. For example, a simple striping system to eight disks may say that the jth byte of the file is placed on disk $(j/blocksize) \bmod 8$. This mapping is determined at the time a file is opened; some proposals for parallel file systems provide ways for the user to control the mapping on a file basis. We have observed that often it is more convenient if the mapping is based on the object rather than a file. For example, two arrays written to the same file may have different "natural" mappings. An example of such a case includes adaptive computations where the size of the data may vary during the computation. Crockett [?] classifies a number of parallel file organizations.

Our aim is to analyze many computational science applications at ANL and abstract out of them common hardware and software requirements. In the case of hardware, our approach is to give the user a common set of calls that will work across all or most commercial MPP machines and workstation clusters. The user need not be concerned whether the underlying implementation stripes a file across parallel disks or uses high-speed tape. Also, since the calls are portable across systems, a particular vendor's hardware is hidden from the user.

In the case of software, analysis of the types of I/O usage in computational science applications identified several common themes, including reading and writing a replicated value (scalar or array), and reading and writing a partitioned and distributed array. Accordingly, it is this functionality that we seek in our system by providing subroutine calls that allow the user not to have to be concerned with issues such as deciding which processor will write a replicated value, calculating the location in a file for each distributed array element, and handling processor synchronization. We also wish to provide the user the ability to open a parallel file as if it were a regular sequential file. Finally, by providing a portable set of calls, we free the user from having to learn each parallel computer vendor's parallel I/O interface.

3 Parallel I/O Programming Interface

3.1 Distributed Data Structures

Our initial implementation of this work has been done in the context of the distributed-memory programming model. In this model a global data structure is decomposed among the memories of the individual processors. The individual parts of this distributed data structure are then operated on in parallel. A common instance of this in many scientific and engineering applications is the single program, multiple data (SPMD) model. In the SPMD model each processor executes a copy of the same program on the data in its memory.

We assume there are several kinds of data of interest. For example, there is replicated data, such as scalar values or small arrays that exist in duplicate in each processor's memory. Also, there are arrays that have been partitioned and distributed across the processor memories. Another type of data is where each processor has a large block of contiguous data. Additional kinds of data include arrays on a subset of the processors, more general data layouts (such as data from an irregular mesh), and more complicated data elements (such as structures rather than individual reals or integers). Rather than try to predict all possible data arrangements, we have chosen to support, at a high level, those operations needed in our applications. Our implementations of the low-level I/O operations are intended to be more general, but as a result are significantly more difficult to use.

3.2 The `PETSc/Chameleon` Package

Our parallel I/O system has been implemented in the context of the `PETSc/Chameleon` package. The `PETSc/Chameleon` [?] package is a collection of routines that provide a hierarchy of models for parallel programming on distributed-memory parallel computers. These routines are intended to provide a consistent, easy-to-use model of message-passing that enables access to all of the power of a distributed-memory computer. This package supports both native (vendor) communications libraries and several popular "portable" communications packages. The "portable" packages supported include `p4`, PICL, and PVM; The `PETSc/Chameleon` package runs on any system that these packages support. Of these, `p4` [?] supports the

widest variety of systems, including workstations and massively parallel computers.

The `PETSc/Chameleon` package currently comprises hundreds of routines, mostly in C. It provides support for a variety of common message-passing primitives and provides routines that support collective operations that involve a collection of processors including all processors and subsets of all processors.

3.3 Overview of Parallel I/O Routines

All of the parallel I/O routines are organized as follows. First, a parallel file is opened with `PIFopen`; this specifies the kind of parallel file (see below) and returns a file descriptor (or context) that is used in all of the other parallel I/O routines. This corresponds to an `fopen` in C or, more approximately, an `OPEN` in Fortran. At this point, the action of the parallel routines may be modified in much the same way that C permits a `FILE` to be modified with routines such as `setbuf`. Next, actual input and output are performed by using routines for specific operations such as "output a distributed array" or "read in a scalar." Finally, the parallel file is closed with `PIFclose`.

3.3.1 Function Calls

The `PIFopen` call opens a parallel file for future use. It must be called before any parallel I/O commands are issued. It returns a file pointer to be used in all subsequent PIO calls. Two important parameters are `processors`, which allows the specification of which set of processors will be involved in any subsequent PIO calls (typically, but not always, this will be all processors), and `ftype`, which (like low-level C open commands) allows specification of what mode the file should be open in. Our goal with the `ftype` parameter is to provide a "parallel" mode where the parallel file will be opened for access in whatever way provides the highest possible I/O performance. The actual implementation will be highly vendor-specific. In the "as-sequential" mode the parallel file will be a conventional Unix file. The `PIFclose` call closes a file and frees any buffers allocated to it.

The `PIFWriteCommon` call writes a nondistributed (i.e., a global or single) variable *once* to a file. The implementation is responsible for deciding which processor will actually write the variable to the file. The `PIFReadCommon` call reads a data value and scatters that value to all of the processors. The implementation is responsible for deciding which processor will read the value. The parameter `fmat` is a string that specifies the format the data will be read or written in. (It is similar to the C `printf` command specifiers with some restrictions.) The data is written unformatted if `fmat` is not specified. Other parameters describe the data itself (location, data type, and size).

The two functions `PIFWriteDistributedArray` and `PIFReadDistributedArray` are used for reading and writing a partitioned and distributed array. `PIFWriteDistributedArray` writes a distributed array to a *single* file. `PIFReadDistributedArray` reads an array from a single file into a distributed-memory environment and partitions the data among the processors according to the descriptor array `sz` of type `PIFArrayPart`. Array `sz` contains the total number of elements in the entire array, the starting indices in each dimension for the process, the ending indices in each dimension for the process, and the index of the *first* global array element. The th dimension of a distributed array is defined by the ith element `sz[i]` of `sz`. Besides `sz`, the other unique parameter is `nd`, which specifies the number of dimensions in the array. The other parameters are the same as those used in `PIFWriteCommon` and `PIFReadCommon`.

The two functions `PIFread` and `PIFwrite` are used by each processor to read/write a block of data to a file. Important parameters are the length of the block of data and the data type. In the "as-sequential" mode, the blocks are read/written in processor rank order. In the "parallel" mode, many tasks *may* write simultaneously. `PIFwrite` and `PIFread` are optimized for the case where each processor may have a single large *contiguous* block of data that it wishes to read/write.

3.3.2 Implementation Details

The implementation is very important for performance. In this section we briefly describe some of the issues and two different implementations. Our abstraction-oriented approach has allowed us to experiment with a number of ideas without changing the source code of the applications, with the result that we have been able to speed up some applications by simply relinking them.

Some issues. Any parallel I/O system provides one or more disk systems attached in some way to one or more parallel nodes. For example, a disk may be placed at every parallel node. In another case, all the parallel nodes may be connected to a set of "service" nodes that manage the actual I/O traffic between the parallel nodes and the disk(s). Still others may connect the disk system to an external network. This wide range of options makes it nearly impossible to specify

a single low-level approach that is both portable and efficient.

Figure 1 is a diagram of how the PIO system is implemented on a "generic" parallel I/O system. Above the dotted line are the parts visible to the user: the parallel nodes and their memories. The darkened areas in the memories represent blocks of data that are to be written by each processor.

Beneath the dotted line are the parts managed by the PIO system or provided by the underlying hardware. The "aggregated data" is managed by the PIO routines. The PIO routines receive the data from the user and may pack or shuffle it before passing it on to the exchange network. The exchange network is an abstraction which *may* have a direct hardware realization. It is used to pass the aggregated data to the nodes that interface directly with the I/O system. For example, in the Intel DELTA the exchange network is the mesh that connects the parallel compute nodes to the I/O service nodes. In the IBM SP-1, where each compute node is directly attached to its own local disk, the exchange network is a no-op, since the "aggregated data" is written directly to the processor's local disk. The output of the exchange network is passed to the nodes that interface directly to the physical I/O network. For example, in a Sun workstation network these "nodes" might be a single workstation (in the "as-sequential" case). The IBM SP-1 at ANL is unique in that there exists a second tier in the I/O system. Here, the I/O nodes are every fourth parallel node in the system which has a fiber channel connection to a RAID disk.

Two issues that need to be considered in writing a parallel file are how many processors request the operating system to write to the disks and what buffer sizes are used. The first of these issues is fairly obvious; we would like to use exactly as many processors as are directly connected to the disks so that the operating system does not try to rearrange the data that we wish to write, generating a communications bottleneck in moving the data between processors. The second is more subtle. If the library or application chooses buffer sizes that do not match those of the disks, the impact on performance can be much greater than a simple factor of two or three (caused by a read-modify-write cycle where a single write would have sufficed). Depending on how the operating system is managing the parallel file system, a write of data that spans two disk buffers may actually span two disks, causing additional communication traffic and loss of performance.

Two implementations. For writing a file in the "as-sequential" mode, we experimented with two different implementations. Both of these use a single process to write to the file; this simplifies the interface with the operating system at some cost in performance (though on workstations the cost is slight since, eventually, a single process is writing the file). The first implementation is unbuffered: each processor generates the data that it wants written and sends it to the master process, along with information on where to write it. The second implementation divides the output into buffers; each processor contributes to the buffer, and then the buffers are gathered up and written out by the master processor. The first approach minimizes the amount of data that is sent between processors; the second uses fewer communications and disk operations. In our experiments, the second approach was significantly faster. Note that both approaches were used with the same application program without changing the application code.

For the parallel implementation for systems with a disk per node, each processor opens a file on the local node (the name of this file system is provided by the implementation, allowing the user to use the same relative file name independent of the system). On systems with vendor-supplied parallel file systems, a parallel file is opened. In this case, the implementation can use the local network to move data around so that reads/writes from the parallel file system are as fast as possible (by mapping bytes from the application's natural format to the file system's natural format).

We plan to try additional implementations. For example, we can relax the requirement that the "as-sequential" file be written in that form until the `PIFclose` is called. This will allow us to write the file in a parallel format, taking advantage of the parallelism in the disk system, followed by a single merge of that parallel file into a single file. Other implementations could exercise more care in matching the buffers used by the `PIF` routines to the operating system.

3.4 Programming Examples

The program fragment in Figure 2 is taken from a superconductivity code. It writes the distributed, three-dimensional array `ps` and the scalars `nx, ny, nz` (number of grid points) and `hlx, hly, hlz` (grid spacing) to a file for postprocessing graphics. The `sz` descriptor array defines the six fields `mdim, ndim, start, end, gstart`, and `gend` for each dimension of `ps`. For each dimension these fields contain the global array size, the local array size, the starting and ending indices for the local part of the array, and the global

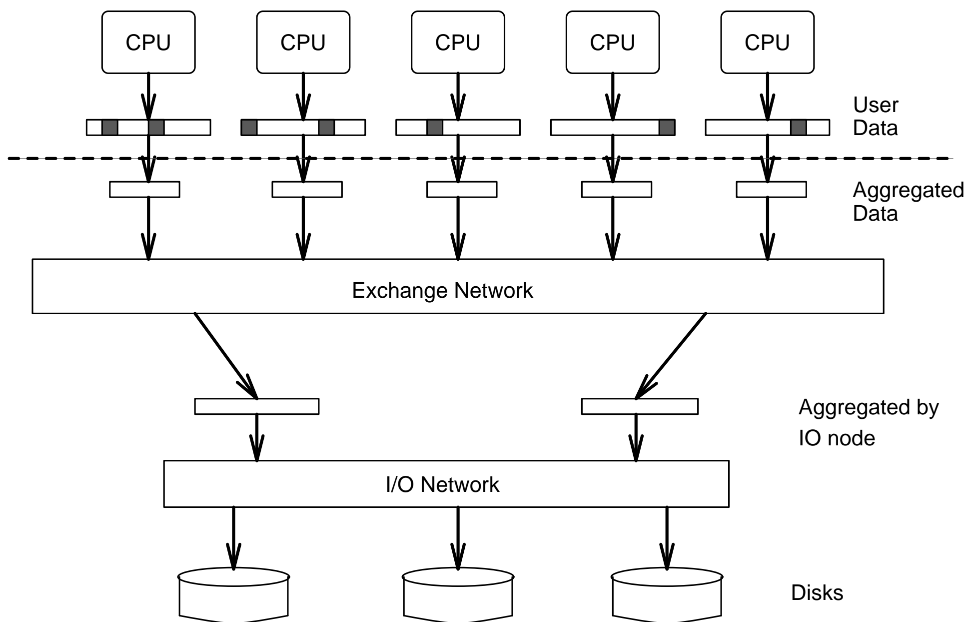

Figure 1: A generic parallel I/O system

indices for the local part of the array.

First, the `sz` descriptor is defined. Second, the parallel file "outfile" is opened. Third, all processors make the `PIFWriteCommon` call to write the scalars `nx, ny, nz, hlx, hly`, and `hlz`. Fourth, all processors call `PIFWriteDistributedArray` to write their piece of the `ps` array. Finally, all processors flush and close the parallel file. An important point is that the "ghost points" (the extra memory allocated by each processor to hold neighboring values of its distributed array) are *not* written to the parallel file.

3.5 Performance

Our primary intent in this section is to show that improved performance may be achieved using the "parallel" file format when compared to the "as sequential" file format. We caution that the results presented here were not run on a standalone system and the read and write speeds can reflect disk cache access time, and so should *not* be considered benchmark data. They are presented strictly to demonstrate the flexibility of the PIO system and the different levels of performance that may be achieved.

The experiments were performed as follows. To measure the input (read) rate of a parallel file access, a parallel file was opened, read from using `PIFread`, and closed. Only the latter two steps were timed. To measure the output (write) rate of a parallel file access, a parallel file was opened, written to using `PIFwrite`, and closed. Again, only the latter two steps were timed. Since the timing includes closing the file(s), the writes are flushed out of the application (though not necessarily out of the OS onto the disks). Our experiments were performed on two Sun Sparc workstations connected over an Ethernet network, and on a 32-node IBM SP-1. All reads and writes were done using `PIFread` and `PIFwrite` in an unformatted mode. `p4` was used for the underlying message-passing system.

Table 1 contains results from the Sun Sparc workstation experiments. In the "parallel" implementation each workstation writes to a local disk. In the "as-sequential" implementation one processor manages all the I/O, with the other processor sending or receiving data to and from the managing processor via messages. Here, the file resides on a single disk that is accessed over an Ethernet network. In all examples the file size used was 64k.

The main point we wish to make in Table 1 is the performance difference achieved using the "parallel" file format as opposed to the "as sequential" case. For the application programmer this performance difference is achieved simply by switching the choice of file format in the `PIFopen` call.

Table 2 contains results from the IBM SP-1 experiments. In all cases the "parallel" implementation was used and the file size was fixed at 16 MB. In the IBM SP-1 "parallel" implementation, each processor

```
void write_graphics (ps, nx, ny, nz, hlx, hly, hlz,
                     sx,ex,sxgp,exgp,sy,ey,sygp,eygp,sz,ez,szgp,ezgp)
{
    int   *nx, *ny, *nz;
    int   *sx, *sy, *sz, *ex, *ey, *ez;
    int   *sxgp, *sygp, *szgp, *exgp, *eygp, *ezgp;
    double *hlx, *hly, *hlz, *ps;

    PIFILE *fp;
    PIFArrayPart sz[3];

    sz[0].mdim   = nx;
    sz[0].ndim   = ex + exgp - sx + sxgp + 1;
    sz[0].start  = sxgp;
    sz[0].end    = sxgp + ex - sx ;
    sz[0].gstart = sx - sxgp;
    sz[0].gend   = ex + exgp;

    sz[1].mdim   = ny;
    sz[1].ndim   = ey + eygp - sy + sygp + 1;
    sz[1].start  = sygp;
    sz[1].end    = sygp + ey - sy;
    sz[1].gstart = sy - sygp;
    sz[1].gend   = ey + eygp;

    sz[2].mdim   = nz;
    sz[2].ndim   = ez + ezgp - sz + szgp + 1;
    sz[2].start  = szgp;
    sz[2].end    = szgp + ez - sz;
    sz[2].gstart = sz - szgp;
    sz[2].gend   = ez + ezgp;

    fp = PIFopen("outfile", ALLPROCS, O_WRONLY | O_CREAT, 0);

    PIFWriteCommon(fp, NULL, sizeof(int),    nx,  1, MSG_INT);
    PIFWriteCommon(fp, NULL, sizeof(int),    ny,  1, MSG_INT);
    PIFWriteCommon(fp, NULL, sizeof(int),    nz,  1, MSG_INT);
    PIFWriteCommon(fp, NULL, sizeof(double), hlx, 1, MSG_DBL);
    PIFWriteCommon(fp, NULL, sizeof(double), hlz, 1, MSG_DBL);
    PIFWriteCommon(fp, NULL, sizeof(double), hlz, 1, MSG_DBL);
    PIFWriteDistributedArray(fp, NULL, sizeof(double), sz, 4, ps, MSG_DBL);

    PIFflush(fp);
    PIFclose(fp);
}
```

Figure 2: Example program fragment

Table 1: Sun Sparc Workstation Results

File Format	No. Proc.	Read MB/s	Write MB/s
"parallel"	2	6.8	3.7
"as-sequential"	2	.8	.1

Table 2: IBM SP-1 Results for a 16 MB File

No. Proc.	Blocksize	Read MB/s	Write MB/s
4	4MB/8k	6	26
8	2MB/64k	12	40
16	1MB/16k	26	30

wrote directly to its own local disk. The parameters that were varied were the number of processors used and the "blocksize"—the amount of data read/written with a single call to `PIFread/PIFwrite`. The first column is the number of processors used; the second column is the blocksize at which the read or write rate was a minimum. The third and fourth columns contain the *minimum* read or write rates achieved over all blocksizes tested.

We emphasize that disk cache effects should not be discounted. Increasing numbers of processors lead to smaller subdivisions of the parallel file and possibly better chances of disk cache hits. The drop in write rate for sixteen processors is anomalous and may simply reflect a load imbalance caused by other tasks.

Again, the real point we wish to stress in this section is that our system provides an interface design that hides the underlying implementation from the user. It permits *both* high-performance, nonsynchronous parallel I/O using the "parallel" file format and the ability to write a parallel file in a sequential format for debugging—with only a single change by the user: a different value for the file format parameter on the `PIFopen` call.

4 Conclusions and Future Work

In this paper we have described the I/O requirements of some computational science applications at Argonne National Laboratory and discussed a distributed-memory system we have developed to support these requirements on MPP systems. Two key points in our work are the use of high-level abstractions and portability among MPP systems.

We believe the ability to use abstractions is very important. An applications developer goes through a number of steps such as development, debugging, and production to get a (parallel) code working. Each of these phase has certain unique requirements associated with it. With our approach, the abstraction of file type allows a parallel programmer to easily switch between the requirements of different phases. For example, in the debugging phases a programmer may desire output in the form of contiguous ASCII files on his "home" file system, while unformatted and stripped files may be desirable for production computing. With the file type option to the `PIFOpen`, our system supports all of these without requiring any code changes by the user.

Portability is also a key aspect of our work. By integrating our system in the context of the `PETSc/Chameleon` package we achieve portability among the large number of systems this package runs on. Somewhat optimized implementations of our system currently exist for the IBM SP-1, Intel MPP systems, and Sun workstation networks.

We remark in passing that the operations described here are natural ones to support in languages such as High Performance Fortran. We fully expect that, as more real applications are run on parallel computers, there will be enhanced support for parallel I/O. We hope that our work will (a) allow applications to use parallel I/O in the short term and (b) provide a testbed for implementations and program annotations (guides to the runtime system about the parallel I/O). In addition, we expect to add additional abstractions, such as those for unstructured data, that are beyond the scope of current language standards.

Availability and Documentation

The `PETSc/Chameleon` system is in the public domain. The complete distribution can be obtained by anonymous ftp from info.mcs.anl.gov. Take the file chameleon.tar.Z from the directory pub/pdetools. If you wish to use the `PETSc/Chameleon` with `p4`, take the file pub/p4/p4-1.3.tar.Z The `PETSc/Chameleon` distribution contains all source code, installation instructions, a users guide in both ASCII text and latexinfo format, and a collection of examples in both C and Fortran.

Acknowledgments

We thank Paul Bash, John Michalakes, and Ron Shepard for helpful discussions about their applications.

References

[1] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Journal of Parallel Computing*. to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[2] T. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing'89*, pages 574–579, 1989.

[3] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. Preprint, 1993.

[4] W. Gropp and B. Smith. Users manual for the Chameleon parallel programming tools. Technical Report ANL-93/23, Argonne National Laboratory, 1993.