Usability of Parallel I/O Templates

Ian Parsons, Ron Unrau, Jonathan Schaeffer, and Duane Szafron
University of Alberta
Department of Computing Science
{ian,ron,jonathan,duane,}@cs.ualberta.ca

Keywords: parallel I/O, templates, parallel programming systems, Enterprise, PIOUS

Abstract

This paper presents an alternative high-level approach to parallelizing file I/O. Each parallel file descriptor is annotated with a high-level specification, or template, of the expected parallel behaviour. The annotations are external to and independent of the source code. At run-time, all I/O using a template file descriptor adheres to the semantics of the selected template. By separating the parallel I/O behaviour from the code, a user can quickly change I/O semantics without rewriting code. Templates can be combined hierarchically to allow the simple construction of complex access patterns.

Two sample parallel programs using these templates are compared against versions implemented in an existing parallel I/O system (PIOUS). The sample programs show that the use of parallel I/O templates are beneficial from both the performance and software engineering points of view.

1 Introduction

The development of parallel applications has focused on computational parallelism. However, the corresponding growth in parallel input and output (I/O) implementation techniques has not kept pace. If an application is to perform parallel I/O operations, the user must explicitly differentiate between parallel and sequential I/O streams at the source level, and often import or export files into or from specialized file systems. As well, the computational parallelism may have to be re-implemented to work with the communication system used to build the parallel I/O library. This results in a lack of portability between different operating systems, architectures, and even changes in the physical layout of the files.

This paper proposes a design for high-level parallel I/O templates within the auspices of a parallel programming system (PPS). Examples of these systems can be found in [1-3, 7, 12, 13, 15, 23, 31]. A PPS could use these parallel I/O templates along with templates for parallel computation to implement the desired parallel behaviour. The PPS integrates all components of developing, compiling, running, debugging, and evaluating the performance of a parallel application. That is, the implementation of the parallelism is handled by the PPS. The user chooses the computational and I/O templates that give the best performance.

A study [32] examined the usability of several parallel programming systems. The study found that using computational templates to create parallel applications is beneficial. The user code is significantly reduced and the application is up and running much sooner since the templates are correctly implemented for the selected parallel behaviour. The drawback to templates is that there can be a slight performance penalty (less speedup). The work presented in this paper extends these results from computational templates to I/O templates and provides experimental validation.

Parallel libraries are an improvement over implementing the desired functionality using low-level functions offered by operating systems. Even if a parallel I/O library or system is used [4-9, 14, 17, 19-22, 24, 26-28, 33, 36], the user specifies the parallel I/O by using a package of specially designed parallel I/O library calls (typically highly tuned to one or a few architectures). The user differentiates between sequential and parallel I/O streams and specifies how the data is to be subdivided, synchronized, and merged.

When libraries are used, it is important to note that the parallel behaviour is still directly coded into the program. Any changes to the I/O or the parallel computation behaviour is reflected by modifications to the code. Thus, something as simple as integrating a new release of the I/O library could introduce errors. Since the code is designed to use one I/O library, if a decision is made to use another I/O library (possibly due to moving the code to a different system), modification to the source code is required even though the parallel behaviour has remained the same. Unfortunately, to experiment with different parallel I/O access patterns or behaviours, many lines of code must be rewritten.

An alternative to embedding the parallel behaviour directly into the application is a high-level abstraction, or template, that separates the parallel behaviour from the code. High-level abstractions such as these are intended to work within the framework of a parallel programming system. For example, it would be ideal to designate an I/O stream as segmented and have the system parallelize all the sequential I/O calls that use that stream correctly. These abstraction mechanisms are beneficial since:

- Parallel I/O and computational behaviours are encapsulated into an easy to understand set of templates.
- The user specifies the parallelism that is wanted while the template offers different solutions for the same parallel behaviour, depending on the underlying architecture or low-level software libraries.
- Parallel behaviour can be changed with minimal (or no) changes to the user code.
- Because the computational and I/O templates are integrated, optimizations between the different parallel behaviours are possible at both compile and run time.
- Templates provide a quick first-draft of a solution which can be incrementally refined depending on the user's expertise.
- Correct parallel behaviour and implementation for the template are guaranteed.
- The performance of templates can be comparable to hand-coded solutions.

The programmer uses the PPS to produce a parallel application by supplying the sequential code for the parallel algorithm. The parallelism is described by selecting templates of predefined parallel behaviours for parallel computation and I/O and associating specific functions or variables to different templates. The PPS stores these templates separately from the user’s code. The templates and the user’s code are then processed by the PPS to generate code to perform the parallel behaviour. This machine-generated code is linked with the necessary run-time support libraries to generate an executable for a specific target architecture. This is repeated if more than one type of architecture is being used (different I/O implementations could be used that are transparent to the program). At run-time, the PPS is responsible for starting, monitoring, and terminating the parallel application.

For example, consider an application that has one of its I/O descriptors annotated to use a segmented behaviour. The PPS analyses the source code for instances of the parallel file descriptor and modifies any code necessary to ensure the correct parallel I/O semantics (as defined by the template). If the user wishes to change the parallel I/O behaviour, a different template is specified and the PPS regenerates the code to implement the new behaviour. The strength of this approach is that different parallel I/O behaviours are specified by changing templates -- not user code.

There are two perceived disadvantages to using such a high-level abstraction mechanism. First, there is the loss of direct control by the user since a high-level abstraction is supposed to shield the user from many of the low-level details. Second, the performance of the application might not be as good as the hand-crafted application since the abstraction deals with the general rather than the specific details of the problem.

This first point is addressed by creating a base set of templates that can be composed into more complex behaviours. If users require more hands-on control, they can change the attributes of the template (but not the code) to customize it for their application. The simple programming model, the short time to draft a working application and the independence from implementation details typically outweighs the restrictions imposed by working within a template framework

The second concern is more serious, since to many people performance is the only evaluation metric. While this paper primarily addresses the software engineering benefits of template I/O, the performance is shown to be comparable to hand-coded, tuned implementations. Since template I/O offers significant software engineering benefits, the user should only consider hand-coded solutions if they are convinced that additional performance gains are possible. The possible performance gains are offset by the cost of the additional effort required to implement, debug and test their custom solution. An alternative approach for the advanced user could be to tune and modify the code generated by the PPS since many PPSs use source-to-source translation.

This paper is divided into several sections. Section 2 examines some related work. In Section 3, a simple example showing the difficulties of moving from the sequential to the parallel I/O domain is discussed. Section 4 presents the parallel I/O templates proposed by this paper. Section 5 compares the conversion steps in developing a parallel application against an existing parallel I/O system (PIOUS [27]). Section 6 presents the actual performance of two applications. The first application is the one presented in Section 5 which has fine-grained I/O spread throughout the application. The second application uses large-grained I/O operations. Finally, the conclusions are presented.

2 Related Work

There have been several studies of parallel I/O for applications using real data. Some examples of these are found in [10, 25, 29, 30, 34, 35]. These studies look at traces of actual parallel applications doing I/O using specific parallel I/O libraries and architectures. Characterizing well-understood parallel programs under controlled conditions facilitates development of optimization techniques.

Parallel I/O systems such as PIOUS[27], MPI-IO [8], and ELFS [19] have abstracted parallel I/O semantics so that there are two types of I/O streams -- sequential and parallel. Additionally, these I/O systems are designed to use the parent communication system (PVM [16], MPI [37], and Mentat [18], respectively).

This approach of differentiating between parallel and sequential I/O streams both complicates and simplifies the programmer’s coding strategy. It simplifies the problem since only the parallel I/O is converted. The complication is that the user must choose which files to be parallelize, and then decide on the parallel I/O model and its implementation (library) before starting to write code. Templates allow the user to switch between sequential and parallel I/O at any time, independently of the code. This leads to more portable and maintainable code.

A template approach can use any low-level parallel I/O implementation that supports the expressed parallel behaviour of the template. The basic types of parallel I/O are still the same as when Crockett [11] first characterized them -- global, segmented, and independent. How they are implemented, either as a library for a specialized file system, as an operating system module, or even as hardware, is strictly a matter of efficiency. The interface to the user must be simple enough to use, but flexible enough to allow performance tuning for specific applications.

3 A Simple Example

This section presents a simple example that illustrates some of the obstacles that are fundamental to parallelizing sequential I/O. The parallel program that is derived in this section is not an example of how the parallelization would be accomplished using templates. The example is intended to show what kind of code the user would need to provide if the parallelization was done by hand. Alternatively, it shows what kind of code must be generated if templates are used.

Figure 1 shows the sequential C code for this example, together with a sample input file and its output. The sequential program opens two files, one for reading and one for writing. The program reads integers from the input file and for each integer, outputs a line to the output file containing multiple copies of that integer. The input file consists of a series of ASCII character representations of integers, separated by new-line characters and terminated by an end-of-file marker. The output file can be viewed as a series of variable length character records, separated by new-line characters.

```
#include <stdio.h>

Parent( int argc, char **argv )
{
    FILE *fin, *fout ;
    fin = fopen( argv[1], “r” ) ;
    fout = fopen( argv[2], “w” ) ;
    while ( ! feof( fin ) ) {
       Child( fin, fout ) ;
    }
    fclose( fin ) ;
    fclose( fout ) ;
}

Child( FILE *fin, FILE *fout )
{
    int i, num ;
    fscanf( fin, “%d”, &num ) ;
    for ( i = 0; i < num; i++ ) {
       fprintf( fout, “%d ”, num ) ;
    }
    fprintf( fout, “\n” ) ;
}
```

```
Sample input file:

   3
   6
  12
   9
```

```
Sequential output file:

3 3 3
6 6 6 6 6 6
12 12 12 12 12 12 12 12 12 12 12 12
9 9 9 9 9 9 9 9 9
```

Figure 1 - Example program and the sequential input and output files.

This example is a simple one, but it illustrates that the following basic considerations must be made when converting from sequential to parallel I/O:

- When a file is opened by multiple processes, an access mechanism must be specified. The three most common access mechanisms are: shared, independent or segmented. Shared access means that movement of the file pointer by one process affects the file pointers of the other processes. Independent access means that each process has its own independent file pointer. Segmented access means that the processes access mutually exclusive regions of the file with their own file pointers. The code must be changed so that the access mechanism is explicit when a file is opened.
- For each parallel access mechanism, there are different criteria for checking the end-of-file condition and different actions must be taken to close the parallel file. These differences must be reflected in the code.
- Access synchronization must be specified. For example, in order to prevent unwanted interleaving of I/O operations by different processes, atomic I/O

transactions must be identified in the code. In addition, some synchronization may be necessary between transactions.

- The format of a file may need to be changed to support a particular parallel access mechanism.

These considerations are not intended to be exhaustive. They are given here to show that even a simple program requires extensive modifications when its I/O is parallelized. The goal is to generate these modifications automatically using parallel I/O templates.

A natural parallelization of the program in Figure 1 has the **Parent** function and multiple copies of the function named **Child** each executed by its own process. Figure 2 shows a parallel version of the code that accomplishes this. A boldface font is used to identify changes to the code.[1] Only two constraints are placed on the parallelization. The input file may only be read once and the output of each **Child** function may not be interleaved with the output from any other. For example, it is not necessary for the 3's to be printed before the 6's. However, it is necessary for the 3's to appear on a separate line from the 6's.

```
#include <stdio.h>

Parent( int argc, char **argv )
{
    par_FILE *fin, *fout ;
    fin = par_fopen( argv[1], “r”, parMode, parGroup ) ;
    fout = par_fopen( argv[2], “w”, parMode, parGroup ) ;
    while ( ! par_feof( fin ) ) {
        /* Wrapper function to send message to remote process executing Child */
        par_Child( fin, fout ) ;
    }
    par_fclose( fin ) ;
    par_fclose( fout ) ;
}

Child( par_FILE *fin, par_FILE *fout )
{
    int i, num ;
    par_fscanf( fin, “%d”, &num ) ;
    par_IOstart( fout ) ;                          /* Start I/O transaction */
    for ( i = 0; i < num; i++ ) {
        par_fprintf( fout, “%d ”, num ) ;
    }
    par_fprintf( fout, “\n” ) ;
    par_IOend( fout ) ;                             /* Stop I/O transaction */
}
```

Figure 2 - A parallel version of the example sequential code.

The **Parent** process opens the input and output files using a generic parallel library function **par_fopen**. The extra parameters indicate the parallel access mode of the file (**parMode**) and the processes that will collectively share this parallel file (**parGroup**). The **par_feof** function uses the parallel access mode set in the **par_fopen** function to determine whether the end-of-file condition has been met. For example, if shared file access was selected, then **par_feof** will be true whenever any **Child** process encounters an end-of-file condition. If independent file access was selected, then **par_feof** will be true only when the **Parent**'s file pointer reaches the end-of-file mark. In this program, that will never occur since the **Parent** never moves its file pointer. If segmented access is selected then as **Parent** calls its children, **Parent** moves its own file pointer forward, one

[1]For brevity, the code for spawning remote processes, marshalling and demarshalling of parameters and explicit process communication is not shown.

segment at a time. In this program, **par_feof** will be true when it passes the last segment to a child.

The **par_Child** function is a *glue* function that contacts a remote process to execute the **Child** functions. This function passes the parallel file descriptors to the remote **Child** processes. Finally, the **par_fclose** function closes the file using the correct parallel access mode to dispose of the appropriate file pointers.

The fundamental problem of parallel I/O programming is that multiple processes share a common resource. One of the consequences of this is that a user cannot assume a consistent I/O state between successive operations unless accesses are synchronized. Even using a parallel I/O library, a series of output operations would be interleaved unless the I/O library is informed that a succession of I/O actions are to be done as one *transaction*. The output operations in the **Child** function are a perfect example of this situation since the user wants all of the 3's to be output together on a line with all of the 6's on a different line. There are four approaches to solving this transaction problem. In each case, we assume that a single parallel I/O operation is atomic and we wish to build these into a larger atomic transaction.

In the first approach, each line is printed in a single I/O statement. However, since the number of output operations for each line is variable, we must explicitly write to a memory buffer each time through the **for** loop and then explicitly write the buffer to the file at the end of the loop.

In the second approach, an atomic block of output operations is explicitly identified to the parallel I/O system. This choice is seen in Figure 2 by the placement of **par_IOstart** and **par_IOend** functions around the atomic I/O operation.

In the third approach, each remote process gets a block of the file to which it has exclusive access. However, this approach does not support variable length output records since the block size cannot be easily determined in advance.

In the fourth approach, each remote process writes to a local file and after the transaction is finished, the file is returned to the parent to be integrated into the master file. This approach is similar to the first approach, except that it is intended to be managed by a parallel I/O system instead of being the explicit responsibility of the user.

In addition to a mechanism to delimit atomic I/O transactions, it is often necessary to specify the synchronization of I/O primitives themselves. For example, the **par_fclose** function cannot actually close the file until all **Child** functions have finished with the file. Code must be written in the **par_fclose** function to perform this synchronization.

Sometimes the structure of files must be changed to support a parallel access mode. For example, if we wished to use segmented access of the input file for the program in Figure 2, then fixed length records would be easiest to support. One way to do this would be to store the integers in binary format instead of ASCII format. Alternately, if ASCII format is necessary, then a fixed number of characters must be specified for each integer. This has the disadvantage of restricting the range of the input data, say from -999 to 9999, if four characters are used. Similarly, if segmented access to the output file is used, a fixed size line for the output file would be required, and padding would need to be done.

This section has shown that even a very simple program requires extensive modifications to parallelize the I/O operations. As the next two sections will show, templates provide a good mechanism for generating much of this tedious code automatically.

4 Parallel I/O Templates

Section 3 points out some of the complications of parallelizing code and I/O. Section 4.1 describes the parallel I/O templates proposed by this paper. Section 4.2 uses an example to show how templates can be used to specify more complex parallel I/O access patterns. Section 4.3 delves into the implementation algorithms used to develop the templates.

4.1 Description of Parallel I/O Templates

There are five basic parallel I/O templates proposed in this paper. The hierarchical tree (Figure 3) is similar to Crockett's proposal of global, independent, and segmented file I/O [11]. Each I/O template (the shaded nodes in Figure 3) describes a simple parallel I/O behaviour. There is also a sequential I/O class distinct from the parallel I/O templates. Sequential I/O has no parallel behaviour.

An important aspect of any template is its ease of comprehension as to what the template represents. These five templates are intended to be integrated with the parallel computational templates to create a parallel application.

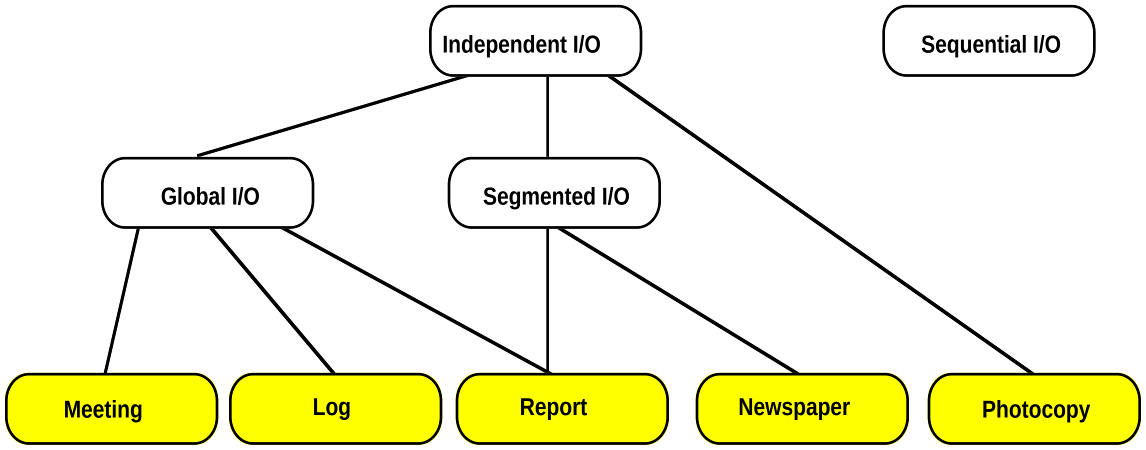

Figure 3 - Parallel I/O template hierarchy.

Meeting

The analogy comes from a meeting where only one person has control of the floor at a time. The **meeting** template uses a global file pointer and all processes using it must synchronize and coordinate access to the file. A **meeting** has both global read and write capabilities. However, only the process that has control of the file may read or write at any time.

Log

The analogy comes from maintaining a record of events. The **log** template uses a global file pointer with the restriction that all write activity takes place at the end of the file. After a write takes place, the global file pointer is left at the end of the file.

Report

Having a committee write a report usually involves the members collectively reading several other sections prior, during or after writing their own section. As well, comments may be written to other sections of the report which may or may not be incorporated into these remote sections. A **report** template has both global and segmented file properties. However, no segment has a fixed owner. A process must obtain read or write permissions for the desired segment from a file manager. The size of the segment is determined at run-time by a function supplied by the user.

Newspaper

A newspaper is composed of sections that can be read (or written) independently. The **newspaper** template is a means to segment a file into independent pieces. Each process gets exclusive access to a portion of the file. Any process that reaches the end of the segment has reached its version of the end-of-file. Like the **report**, the size of a given segment is determined at run-time by a function supplied by the user.

Photocopy

A familiar situation happens when an author distributes copies of a paper for review. After the reviewers have made their comments on their private copy, the author integrates all or some of the changes back into the original document. This may take several iterations. A **photocopy** template is intended for independent file access. Because multiple processes read the same file, one optimization is to selectively replicate the file so that the I/O operations become local I/O instead of networked I/O. However, a **photocopy** has the property that any write operation must be verified by the owner or controller of the file before becoming visible to any other processes.

In addition to the base semantics, each template can have several attributes which refine the base implementation behaviour. One attribute of all the parallel I/O templates presented here is the ordering of I/O operations. Both the read and write operations can have separately defined ordering. That is, the order in which a collection of processes communicate with each other defines the access sequence and when updates become visible. There are three possibilities: **ordered**, **relaxed**, and **chaotic**.

For example, assume processes perform blocks of I/O in the order: α_1, α_2, α_3, β_1, β_2, β_3, using two loops. With **ordered** I/O, the I/O will be done in this identical order. With **relaxed** I/O, the order is relaxed somewhat. Any process executing a β_i operation will wait until all of the α_i are finished, but the α_i and the β_i themselves may be executed in any order. With **chaotic** I/O, the ordering is completely relaxed so that any process can have access to the file at any time. The ordering attribute does not change the base behaviour unless synchronization was involved. For example, the **meeting** template still means one process has access at a time. Depending on the type of parallelism chosen, a process may or may not have to give up access or wait for access to a given file descriptor.

4.2 Composing Parallel I/O Templates

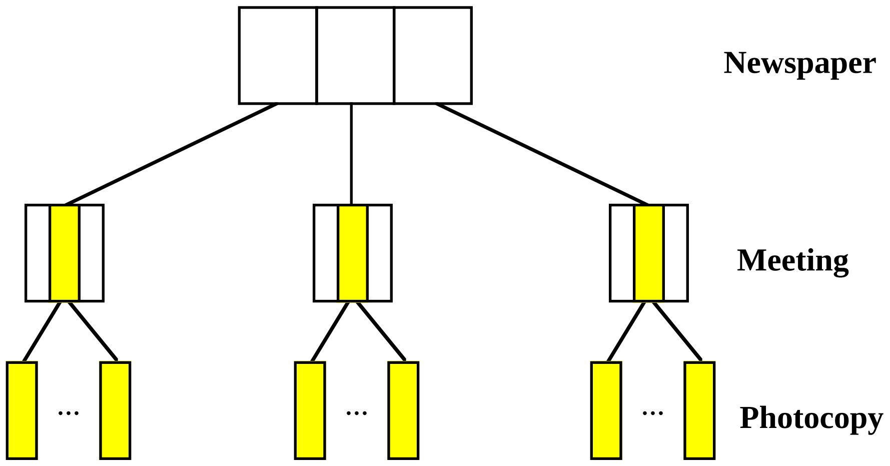

Figure 4 - Composing using parallel I/O templates.

The power of templates is that they can be arbitrarily composed to give more complex behaviours. Figure 4 shows a more complex example which benefits from this approach. In this example, the file is segmented so that concurrent processes are accessing different portions of the file. However, within a given segment, a portion of that segment is independently read by several other processes. The file is broken into three segments using a **newspaper** template. Each segment is treated as a **meeting** (global file) until a particular portion of the segment is reached. At that point, several processes are granted independent access as **photocopies**. After the independent operations are finished, the file access reverts back to a **meeting** (the global file pointer forms a barrier). A pipeline model of computation could yield such an access pattern.

If the user tries to code all of this by hand, the amount of specialized code increases at each level along with the chances of introducing errors. If the computational parallelism changes, the restructuring of the code to reflect this is a potential source of errors. Suppose this pipeline example has sufficient granularity to run efficiently on a shared-memory multiprocessor system. If the code is ported to run on a network of workstations, the granularity of the application may mean that one stage of the pipeline should be collapsed. This is achieved by dropping the **photocopy** template along with the parallel computational behaviour of the associated stage of the pipeline. The strength of a template approach for both the computational and I/O parallelism is that any changes are quickly and correctly implemented.

4.3 Implementation of Templates

This subsection describes how the templates were implemented. Users of the templates do not need to know about this implementation and alternate implementations can be used without affecting user programs.

The templates have been implemented to parallelize the standard C stream I/O library (**fopen**, **fclose**, **freopen**, **feof**, **fprintf**, **fscanf**, **fwrite**, **fread**, **fseek** and the variants using these functions such as **rewind** or **printf**). However, there is no reason that they cannot be implemented to replace low-level I/O calls (**open**, **write**, **close**, **lseek**). The Enterprise [31] PPS was used test these templates, however there is nothing Enterprise specific in their design. Enterprise merely provided the necessary computational framework to implement this design.

The templates are intended to work within a computational parallel hierarchy. Any process that makes a remote call to another process creates a parallel computational hierarchy. The templates are implemented using a client-server model that is distinct from the computational model. If the remote call includes a file pointer, the file pointer is treated as a parallel file pointer. If the user does not specify the parallel behaviour of the passed file pointer, it is considered an error. Or, if a file is collectively opened by a group of processes, this is also treated as a parallel I/O object if it is identified as such by the user. The manager (server) of an I/O object is the process that actually coordinates the file access pattern in the user's source code. In the case of multiple or replicated processes concurrently opening the same file descriptor, only one of the replicated processes becomes the manager.

There are several ways to select the manager: the first process that opens the file becomes the manager, the user designates a specific client process (the process is local to the disk file), or the PPS spawns a new manager process. The other processes become clients for I/O purposes only. It is also important to realize that a client becomes a "branch" manager when the user’s process (which contains the I/O client), in turn, makes remote calls to other processes.

All template I/O operations involve two processes -- the client and the manger. The manager is responsible for synchronizing accesses and merging results. The client must recognize when access permissions are required, execute the user’s code, and end the I/O transactions properly.

The client starts when the user makes an I/O call using a file pointer. The client determines if the file pointer is to be treated as a parallel I/O file pointer. If the file pointer is for sequential I/O, the I/O operation proceeds normally and control is returned to the user's code when the I/O operation finishes. If the file pointer is considered to be parallel, the client determines if it has access to the file. If it does not, the manger of the I/O object is sent a message requesting access.

When the manager receives a request for access, it searches its **order** queue to determine if access can be granted. This queue is filled as the manager process invokes remote functions passing I/O objects as arguments. The access permissions for that I/O variable are determined by the ordering selected by the user. If access is not allowed at this point, the request is queued at the manager in its **pending** queue until the request can be satisfied. If the access can be granted, the manager marks the request in the **order** queue as **active**, updates the parallel I/O data structure, and sends a message containing the access permission along with any new global information to the client. When the client receives the manager's message, the client's file data structure is updated to reflect the new global information.

When access is allowed at the client process, the I/O operation is verified, performed and the parallel file pointer is locally updated. If the atomicity of the parallel I/O operations is set to be a single I/O operation, the client surrenders control by sending the manager a message containing the surrendered access permission and the updated information for that parallel I/O object. If the I/O operation takes place within a defined transaction, control is retained until the transaction is finished. Finally, the client process returns to the user's code. When the manager receives the access surrender message from the client, the manager searches the **order** queue for the active I/O object and removes it. The manager then searches the **pending** queue for the next I/O request that can be satisfied.

The global file pointer templates have I/O stream behaviour similar to the sequential behaviour. There are differences when **fclose** and **freopen** are done by a client or when collective or group **fopen** occurs. Closing a file on the client side causes the manager to invalidate all remaining I/O requests left on the pending queue for that particular I/O object. Reopening the file causes all subsequent I/O accesses to use the new file. When a collective **fopen** is done, as pointed out earlier, one process is designated as the manager to control access to the file pointer. The ordering attribute for the template (Section 4.1) defines which process next gets access to the file.

The segmented file pointer templates are different in that a client receives access permissions for a file pointer that lies within a range specified by the **base** or starting point in the file and the **extent** or the number of bytes that define the limit of the segment. At run-time, the base and extent for the client are determined by the manager using the user's supplied function with the manager advancing its file descriptor to point past the last segment boundary. The client uses a local copy of the segment if the file is opened in write or update mode. One attribute allows segmented files opened in read-only mode to be copied if so desired by the user. If fixed-length records are specified, the segments are updated as-received.

If a **newspaper** template that uses a **defined-length** extent (greater than zero) is selected, each process must stay between these two limits (**base** to **base+extent**). **Defined-length** segments do not mean fixed-length records. Currently the file is segmented by a user-supplied function at run-time. In the future, we hope to derive the segment size by analyzing the code. The user can specify **unknown-length** records by supplying an extent of 0. This unknown-length extent is only useful if the file is opened using write-only mode. One side-effect of using unknown-length extents is that both testing for end-of-file or any read or write operation now block until the outstanding segments have been processed and reassembled in the file.

The order attribute indicates how the file will be reassembled when a client is finished with a segment. For example, if **ordered** I/O is specified, the segments are integrated as specified by the call ordering. If **relaxed** I/O is used, segments representing similar work (α type) are assembled in an as-received order with the other segments (β type) blocked from being committed to disk until all α segments are finished. **Chaotic** I/O allows any segment to be re-integrated into the manager's file in an as-received fashion.

If a **report** template is used, unknown-length extents are not appropriate. When a **report** I/O operation crosses a segment boundary (either less than base or greater than extent), the client requests permission from the manager to access the new segment. The manager waits until the requested segment is free or it asks the client that owns the requested segment to temporarily give the requested read or write permission to the segment. If the segment is free, the manager passes the new segment on to the client. To prevent deadlock, the client gives up its current segment before receiving the new segment.

To ensure that multiple processes do not have access to the same segment, the manager blocks if the user attempts to **fseek** back in the file when there are still segments left in the **pending** queue. Seeking forward is not an error and does not block unless the manager exceeds its defined boundaries. Similarly, the manager blocks on an **fclose** until the outstanding segments are consumed. The **freopen** does not affect processes currently working with a particular segment but any outstanding segments waiting for a process are modified to point to the new file.

When a client is finished with a file segment, it sends a message to the manager that it is finished. If the segment has been modified, the message also contains the modified segment. The manager processes the client's message and updates its file appropriately

The independent templates treat files similar to the sequential stream I/O except that write operations are visible only to the local client. When the client is finished processing the file, the manager gets the updated file. The ordering of the write operations determines when changes to the manager's file become visible to the collective.

5 Programming with Parallel I/O Templates

This section examines in detail the parallelization of a real problem to illustrate the software engineering advantages of template I/O. The program is derived from a molecular docking problem in biochemistry. In Figure 5 the application specifics have been abstracted out, leaving the high-level I/O view of the program. The code looks similar to that in Figure 1, but the Child functions, of course, are different. As well, the **rewind** introduces new considerations. A template I/O (Enterprise [31]) version of this program will be compared to a hand-coded (PIOUS [27]) implementation.

```
#include <stdio.h>

main( int argc, char **argv )
{
   FILE *fin, *fout ;                    /* Input and output file descriptors */

   fin = fopen( argv[1], "r" ) ;                          /* Open the input file */
   fout = fopen( argv[2], "w+" ) ;                       /* Open the output file */
   while ( ! feof( fin ) ) {                            /* Until end of file work */
     Child( fin, fout ) ;
   }
   fclose( fin ) ;                                       /* Close the input file */
   rewind( fout ) ;                   /* Rewind the output file to the beginning */
   Stats( fout ) ;                       /* Perform summary statistics on output */
   fclose( fout ) ;                                     /* Close the output file */
   return 0 ;
}
```

Figure 5 - Sequential code for fine-grained I/O test program.

In the sequential version, the **Child** reads from a file (**fin**) and performs calculations, with the results going to an output file (**fout**). Once the input is exhausted, the main program rereads the output file to analyze the results (**Stats**).

The input and output files contain data objects within data objects within data objects. Each object has its own specific read and write functions and knows how many immediate sub-objects it contains. All I/O is spread throughout the code and is quite fine-grained (one to several hundred bytes at most for any individual I/O operation). In the real application, the data objects are all variable length but to keep this example simple, the records were all set to a fixed length.

5.1 Parallel Design Considerations

Since each **Child** computation is independent of the others, multiple **Child** processes can run concurrently. They need only coordinate reading from the input file and writing to the output file. There is no need to preserve the correlation between the input file order and the output file order.

Coordination of the input file must guarantee that each input datum is read precisely once. Since it does not matter which **Child** does which piece of work, segmenting the input file avoids the inefficiency of having to synchronize file access. Each **Child** process reads a contiguous disjoint interval in the file. The program has been set to use a segment size of roughly 300K bytes. Output file access also needs to be synchronized. The sequential program appends to the end of the output file. Since the output data is a fixed size for each piece of input data, the output file can also be segmented.

Segmenting both the input and output files eliminates the need for **Child** processes to synchronize their concurrent activities. However, they must synchronize before the sequential **Stats** function can be called. A barrier is necessary to guarantee that all the results are in the output file. The barrier is found in the **rewind** function since this function puts the **Parent**'s file pointer in a position that potentially allows two processes access to the same segment. **Stats** does a sequential read of the output file, summarizing each record. Note that the parallel programmer must be careful with the output file, since the **Child** function treats it as parallel I/O, while **Stats** treats it as sequential I/O.

Since there are few constraints on the ordering of input and output, it allows us to experiment with a variety of parallel I/O implementations.

5.2 Template I/O in Enterprise

```
#include <stdio.h>
Parent( int argc, char **argv )
{
   FILE *fin, *fout ;                  /* input and output file descriptors */

   fin = fopen( argv[1], "r" ) ;                       /* Open the input file */
   fout = fopen( argv[2], "w+" ) ;                    /* Open the output file */
   while ( ! feof( fin ) ) {                        /* Until end of file work */
         Child( fin, 1, fout, 1 ) ;
   }
   fclose( fin ) ;                                    /* Close the input file */
   rewind( fout ) ;              /* Rewind the output file to the beginning */
   Stats( fout ) ;                  /* Perform summary statistics on output */
   fclose( fout ) ;                                  /* Close the output file */
   return 0;
}
```

Figure 6 - Modifications to sequential code for Enterprise.

Using a graphical interface, the programmer specifies that one process, called **Parent**, can call multiple instances of the **Child** process. To have this program run correctly under Enterprise, the user must make a number of small changes, as shown in bold in Figure 6. All the changes to the user code are Enterprise-specific (for data marshalling purposes) and have nothing to do with parallel I/O. In the implementation generated by Enterprise, all calls to **Child** will be translated to a message sent to a remote process. The Enterprise run-time system takes care of the spawning of processes, communication (sending, receiving, marshaling/demarshalling of data), synchronization and program termination.

The application parallelism is specified graphically in Enterprise and saved in a file separate from the sequential source code (the *graph* file). For this example, to specify the **newspaper** template add the following parallel I/O annotation in the graph file for the **Parent**:

fin newspaper fout newspaper

The Enterprise compiler will ensure that all occurrences of these file pointers in **Parent** and **Child** will have the appropriate parallel I/O semantics enforced.

A **newspaper** (segmented file) requires a segment size. Ideally, this consideration should be transparent to the user but, unfortunately, it is difficult to automatically choose a good segment size. The user knows best how the I/O is to be accessed, so for segmented files, the user must provide a call-back function that specifies the segment offsets. Figure 7 shows an example of the function appropriate for this application. The bolded code indicates the user-supplied portion, while the rest indicates the Enterprise predefined code that the user does not modify.

```
unsigned long
Enterprise_SegmentFnc( FILE *fp, char *AssetName, char *VariableName )
{
   /*  Based on the process name and file descriptor, return the */
   /*  file's segment size.  It is possible to use the file      */
   /*  pointer to derive a dynamic segment size but it is not    */
   /* necessary for this example.                                */
   /*  For clarity, the following is in pseudo-code              */
   if ( AssetName == "Child"   && VariableName == "fin"   )
       return  ChildInBufferSize ;
   if ( AssetName == "Child"   && VariableName == "fout"  )
       return  ChildOutBufferSize ;
   if ( AssetName == "Parent" && VariableName == "fin"   )
       return  ParentInBufferSize  ;
   if ( AssetName == "Parent" && VariableName == "fout"  )
       return  ParentOutBufferSize  ;
   return 0 ;
}
```

Figure 7 - I/O segmentation function used by template I/O in Enterprise.

Enterprise uses a source-to-source translation to insert the correct code to do message communication and synchronization. The translator has been modified to look for parallel I/O file descriptors (as identified in the graph file) and replace them with calls to parallel I/O functions. The machine-generated source code is then conventionally compiled and linked for a target architecture. The Enterprise run-time library uses the graph file and run-time computational behaviors to implement the parallel I/O operations. Since the I/O behavior is interpreted at run-time, the user can change the I/O templates without having to recompile the program. For example, the parallel template for **fin** can be changed from **newspaper** to **meeting** and the program immediately re-run. As well, **fin** could be converted back to sequential I/O without any additional effort by the user. This makes it easy for the user to experiment with different types of I/O (and computational) parallelism. Note that in any other system, changing the I/O behavior would necessitate many changes to the source code.

5.3 PIOUS Implementation

This section describes our experiences in implementing the program of Figure 5 using PIOUS [27]. First, the MPI-IO alpha versions had just been released at the time of testing. Testing and comparing alpha software is not appropriate or fair to either system. PIOUS has been available for about a year and seems reasonably stable. Second, PIOUS and Enterprise both work well using PVM. By keeping the hardware and the communications software constant, more meaningful comparisons can be drawn.

Comparing PIOUS with template I/O is not intended as a critique of PIOUS or of any other parallel I/O system. Rather, it is intended as an experiment to see if parallel I/O templates are viable. It is assumed that the low-level libraries and systems would be integrated with these high-level templates in a fashion similar to what Enterprise has demonstrated with computational parallelism.

Several PIOUS implementations of the example application were built. Any PIOUS application must to import a file into the PIOUS file system before it can be accessed. Similarly, the output file must be exported back to the file system. The user needs to write these conversion routines.

The first PIOUS version used global file pointers. Because the ordering of the input and output file is not required for this application, the input and output files could be treated as globally shared resources. Globally shared files effectively have one global file descriptor, for which all processes have to synchronize their access. (This is similar to the **meeting** or **log** templates.) The program retrieved an entire data segment as one big block I/O operation and cached the block on local disk storage (**/tmp**). The locally cached data was processed using the conventional stream I/O (**Child** code was not touched) with the output again going to local disk storage. After processing, the results were exported as another big block I/O operation. When the end of the input file is reached, all the **Child** processes notify the **Parent**. When all the children have reported in, the **Parent** continues on to the sequential part of the computation.

This approach proved to be the easiest to implement since most of the explicit parallelism is hidden by the global shared file synchronization. It allowed minimal impact on the existing user's code by using the standard I/O operations to read the local file and then create the output data segment.

A second implementation involved importing the input file into PIOUS as a list of segments and creating a corresponding list of empty output file segments. (This is similar to the **newspaper** or **report** templates.) The user had to write additional code to distribute the input segments as they were requested by idle **Child** processes. Initially, the **Parent** allocates one segment to each **Child**, but as a **Child** completes its work, the **Parent** is responsible for allocating it a new segment.

Each **Child** process opens the appropriate input and output file segments, copies the local segment of work to a temporary file in one I/O operation, opens the temporary output file, performs the work and then exports the local output file back to the parallel output file (again in one operation). This repeats until all segments are distributed. The **Parent** process is then informed and the **Child** process exits after cleaning up the temporary files. The advantage of this method is that the output is in the same order as the sequential version. Again, the **Child** code was not touched.

The final method was to write a pure PIOUS application. It used the PIOUS segmented file capabilities. However, instead of importing or exporting a block of work to local storage, all parallel I/O operations were identified and replaced with the appropriate PIOUS function calls. This was the most intrusive solution as significant portions of the **Child** code needed modifications.

Each approach requires a significant amount of new code. This would also be true when using any other low-level parallel I/O library. The caching version of the program using the global file pointers is given in Appendix A (note that much of the implementation has been abstracted into subroutines that, for brevity, are not included). The original sequential version is about 530 lines of code. The parallel version is approximately 350 lines longer. Any changes in the I/O functionality of the program must be reflected in the source code. For example, if the user wants to do the equivalent of changing a

newspaper to a **meeting**, a considerable number of changes have to made to the source code, with the resulting overhead of testing and debugging the changes.

6 Performance

Section 5 showed the differences in implementing a parallel I/O algorithm in PIOUS and Enterprise. If the number of lines of extra code is used as a metric, it appears that templates are a better choice. However, is the performance of a template approach comparable to a hand-crafted version?

This section presents the performance results of two applications. The first is the fine-grained I/O application discussed in Section 5; the second is disk-based matrix multiply which is implemented as a large-grained I/O application. Both applications have similar parallel computational behaviours. However, they are quite different in their I/O behaviours. Both PIOUS and Enterprise use PVM as the message passing library; all applications were compiled to the same level of optimization (**-O2**).

The eleven processors used were a Sparc 5 (32MB) with 2GB disk, two SUN Classics (32MB), four SUN ELCs (one 24MB and three 16MB), and four SUN SLCs (16MB). One SUN Classic has a 2GB local disk attached to it as well. All processors used local disks for swap and temporary files (**/tmp**). The network consisted of two Ethernet subnets. One of the subnets is further segmented to get more concurrent usage of the network. Where possible, machines that were physically on one segment or subnet were used. However, some configurations did cross net boundaries since the gain in processor speed overcame any network delays. It should be noted that when network boundaries were crossed, the deviation of results grew larger, as did the impact of the application on other users of the network.

It is difficult to get meaningful sequential times. The two processors that have the local disks are the most obvious ones to use. The fastest processor of the group used for testing is also one of the file servers and was eventually used to generate the base sequential time. However, there is a significant difference in the processor speeds in the cluster. Therefore the values presented here should be compared for their relative performances and not as absolute speedups. All times reported are the best of at least five runs. The best time is used instead of an average time because exclusive access to both the network and workstations was not an option.

6.1 Fine Grained I/O

This fine-grained I/O application has been discussed in Section 5. A total of seven PIOUS versions were developed and three of them are presented along with the segmented I/O Enterprise version. Each of the other PIOUS versions gave similar results to one of the three that is presented. The versions not reported used large system buffers to try to reduce the number of physical disk accesses or used low-level I/O calls.

The first PIOUS version uses global file semantics with local file caching of file segments (Global Stream PIOUS, or GSP for short). That is, a large PIOUS I/O operation is done and the resulting block is cached on a local disk. The user's code reads from this local file while writing to another local file. After the work is finished for this segment, the local output file is read in and written to the PIOUS file in one operation. The second version uses a similar approach except that the files are segmented by PIOUS rather than the user (Stream Segmented PIOUS, SSP). However, the user is responsible for distributing the segments. In the third implementation, all the I/O is done in a segmented file system using PIOUS function calls, without any caching (Pure Segmented PIOUS, PSP). From the programming perspective, this version required the most number of code changes.

Enterprise has one version that gives acceptable parallel I/O performance: both the input and output files are segmented using the **newspaper** template. Another version uses the **newspaper** template for the input and the **log** template for the output. This did not give good performance because the output file was locked until all the write operations for a given **Child** process were finished. The other **Child** processes were blocked waiting for access. The times for this inferior version are not shown.

Children used	Enterprise	PIOUS		
		Pure Segmented (PSP)	Stream Segmented (SSP)	Global Stream (GSP)
2	1484	1917	1409	1322
5	813	1040	704	699
10	513	799	509	505

Table 1 - Elapsed times in seconds for Enterprise and PIOUS (PSP, SSP and GSP). Import and export times are not included. Sequential time was 1914 seconds.

Table 1 shows the results for the small-grained I/O tests. For both systems, the time for spawning the remote processes is ignored. The cost of the PIOUS import operation (30-60 seconds depending on the segmentation factor) is ignored as this could be considered a one-time cost if the input file was generated *in situ*. Similarly, the costs of creating and exporting the output file back to the network file system are ignored (5-10 seconds).

The results show the effect of using two file systems for the physical storage. In the case of Enterprise, the input file was on one file system and the output was on the other. PIOUS distributes files between the two file systems. In all instances, wherever possible, the effect of the network was minimized. The number of processors used was one more than the number of children. No processor ran more than one **Child** process.

The Enterprise performance, although comparable, was slightly inferior to the GSP and SSP versions. Even though it used a similar design in its implementation, the cost of using templates to abstract the parallel I/O diminished only with a larger replication of the workers. The Enterprise implementation checks every I/O operation if the file pointer has parallel behaviour. If there are many I/O operations, this cost becomes more significant. Clearly, for this example there is a small performance cost to using templates. However, the Enterprise application still shows improved performance compared with the sequential time. Future work on optimization using pre-fetching and compiler code analysis to order I/O operations will only improve the template performance.

The benefits of templates are seen in the amount of modification to the user's code and the ease of changing parallel behaviours. Each PIOUS version took several hours to modify and debug. For the Enterprise version, the changes to the sequential code, as specified in Section 5.2, were done and the application was generated. This took about twenty minutes from starting with the sequential code until the first test run. The application was first tested using a **meeting** template for **fin** and a **log** template for **fout**. Performance runs were generated in **newspaper** mode simply by changing the parallel behaviour type for both file descriptors and making no changes to the code! No recompilation was necessary as the segmentation function had already been written in anticipation of using the **newspaper** template. Any performance penalty for using templates should be weighed against the potential benefits of quickly getting the parallel application up and running.

The PSP implementation uses PIOUS to perform significant numbers of fine-grained I/O operations. This is very expensive as each I/O operation is converted to a message. However, it does show a performance gain over the sequential version but not as much as the other two implementations. The GSP version using global file pointers shows little difference from the SSP version except when only two child processes were used. Possibly, the cost of segmentation was finally showing.

It is interesting that by using the global synchronization offered by PIOUS with the caching of input and output segments to allow stream I/O operations, this application shows the best performance. However, would this be the case if the application only does large-grained I/O?

6.2 Large Grained I/O

The large-grained I/O application chosen was disk-based matrix multiplication,

$$\mathbf{C} = \mathbf{A} * \mathbf{B}.$$

This application is simple to code and can be done using large-grained I/O operations. The **A** and **C** matrices were segmented into user-specified stripes with the **B** matrix independently read by each processor. The **B** matrix was transposed on disk to make data input faster.

The same parent-child computational parallelism used by the fine-grained I/O application was used. In this case, the application had three parallel file pointers (the A , B, and C matrix data files). The PIOUS version took about 375 extra lines of code to implement both the computational and I/O parallelism. The Enterprise version had to add about 20 extra lines for the segmentation function in addition to the modifications for passing the file pointers.

Children	50 rows per stripe	
used	Enterprise	PIOUS
2	2225	2684
5	1473	1662
10	1598	1580

Table 2 - Disk-based matrix multiply timings in seconds for 2000 by 2000 matrix of doubles (reals) using Enterprise and PIOUS (input and export times not included). Sequential time is 2352 seconds for stream I/O.

Table 2 shows the results for Enterprise and a PSP implementation multiplying two 2000 by 2000 matrices of doubles (reals) stored in binary format and using a striping factor of 50 rows. Again, the cost of importing and exporting the files into and out of PIOUS (180 seconds) is not included in the test results. Preliminary experiments showed that using this striping factor gave better performance than using 100 or 25 rows per stripe. This is due to the ratio of work to message size and the different CPU speeds for the given network configuration.

The Enterprise results are better than PIOUS when using fewer child processes. This was unexpected but one explanation is offered. PIOUS uses direct process-to-process TCP/IP message-passing for parallel I/O, thereby by-passing the PVM daemons. On the other hand, Enterprise uses both the network file system (on-demand small messages) and default routing through the PVM daemons to communicate messages and file information. The differences can be attributed to the cost of using TCP/IP instead of UDP to transport data across the network. These differences are magnified because of the amount of data being accessed (1344 Mbytes). Using a replication factor of ten shows comparable results for either system. This is likely due to the network becoming saturated (measurements showed the network to be between 81% and 87% of maximum utilization).

These results highlight the difficulty of using performance as the determining metric for deciding the effectiveness of a given parallel I/O system. Just because Enterprise uses the network file system, which in turn uses a different protocol for transmitting data, Enterprise performs better for this particular example. In contrast, the previous example shows that PIOUS is somewhat better than Enterprise. The observed performance has little to do with the actual implementation of the I/O templates in Enterprise but rather the implementation of the network file system. Nevertheless, templates once again yield comparable performance.

Ultimately, it is the network availability that determines the overall performance of a parallel I/O application. Being able to experiment with different implementations that exhibit the same parallel behaviour gives more flexibility in tuning an application to a specific network, processor, and loading. Templates offers this flexibility with little cost.

7 Conclusions

The experiments demonstrate that template I/O is competitive in performance to the hand-coded alternative. The templates provide acceptable performance in return for minimal programming effort. Template I/O allows the user to quickly experiment and modify the parallel I/O characteristics with little or no modifications to the sequential source code. Changing the parallelism is a matter of changing either the computational or I/O templates. Because of the integration provided by the PPS, any changes to one aspect of the parallelism will be reflected (if necessary) in all other aspects.

Keying a parallel implementation to a specific underlying communication and parallel I/O library has positive and negative aspects. One positive benefit is that the best performance, tuned for specific architectures and systems, is usually possible. The negative aspects can be seen in the number of additional lines of code that are needed to be written. There is the disruption of the original sequential code to implement the parallel mode(s). As well, there is the difficulty of changing the parallelism to reflect modifications in the original application, changing system libraries, or the hardware used to run the parallel application. Finally, better performance may be achieved using a different parallel I/O system.

The benefits of using parallel I/O templates in a parallel programming system are:

- It is easy to change the I/O parallelism.
- The templates are simple and can be combined to create more complicated parallel I/O abstractions.
- The computational and file I/O parallelisms are integrated.
- Correct code is generated for the chosen parallel behaviour.

The benefits of achieving high performance, hand-tuned parallel I/O must be amortized against the cost of developing, debugging and testing the custom code. For many applications, the performance gains possible from a low-level implementation do not justify the additional effort.

Acknowledgments

Enterprise is a large team project and very little of this could be accomplished without the efforts of many graduate students and researchers. We would specifically like to thank Diego Novillo, Steve MacDonald, Randal Kornelsen, and Paul Iglinski for their contributions to this project. As well, we would like to thank Steve Moyer for his advice on the PIOUS implementations and discussions on the test results. This research was supported by research grants from NSERC and a grant from IBM Canada.

Bibliography

[1] Ö. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli, "Paralex: An Environment for Parallel Programming in Distributed Systems," University of Bologna, Italy, Technical Report UP-LCS-91-01, February 1991.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore, "HeNCE: A Heterogeneous Network Computing Environment," University of Tennessee, Technical Report CS-93-205, August 1993.

[3] M. Beltrametti, K. Bobey, R. Manson, M. Walker, and D. Wilson, “PAMS/SPS-2 System Overview,” In proceedings of Supercomputer Symposium, pp. 63-71, Ontario, Canada, 1989.

[4] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz, “Jovian: A Framework for Optimizing Parallel I/O,” In proceedings of Scalable Parallel Libraries Conference, pp. 10-20, Mississippi State, Mississippi, 1994.

[5] R. Bordawaekar and A. Choudhary, “Language and Compiler Support for Parallel I/O,” In proceedings of IFIP WG 10.3 Programming Environments for Massively Parallel Distributed Systems, pp. 26.1-26.8, Monte Verità, Ascona, Switzerland, 1994.

[6] R. Bordawekar and A. Choudhary, “Communication Strategies for Out-of-core Programs on Distributed Memory Machines,” Syracuse University, Syracuse, NY 13244, USA, NPAC Technical Report SCCS-667, December 1994.

[7] R. Bruce, S. Chapple, N. MacDonald, and A. Trew, “CHIMP and PUL: Support for Portable Parallel Computing,” Edinburgh Parallel Computing Centre, The University of Edinburgh, The King’s Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., Technical Report EPCC-TR93-07, March 1993.

[8] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong, “Overview of the MPI-IO Parallel I/O Interface,” In proceedings of Third Annual Workshop on Input/Output in Parallel Distributed Systems, pp. 1-15, Santa Barbara, California, 1995.

[9] P. F. Corbett, S. J. Baylor, and D. G. Feitelson, “Overview of the Vesta Parallel File System,” In proceedings of IPPS ‘93 Workshop on Input/Output in Parallel Computer Systems, pp. 1-16, Newport Beach, CA, 1993.

[10] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, “Input/Output Characteristics of Scalable Parallel Applications,” In proceedings of Supercomputing’95, pp. , San Diego, CA, 1995.

[11] T. W. Crockett, “File Concepts For Parallel I/O,” In proceedings of Supercomputing’89, pp. 574-579, Reno Nevada, USA, 1989.

[12] D. L. Eager and J. Zahorjan, “Chores: Enhanced Run-time Support for Shared Memory Parallel Computing,” *ACM Transactions on Computer Systems*, 11(1), pp. 1-32, 1993.

[13] W. Fenton, B. Rankumar, V. A. Salctore, A. B. Sinha, and L. V. Kale, “Supporting Machine Independent Parallel Programming on Diverse Architectures,” In proceedings of 1991 International Conference on Parallel Processing, pp. II-193-201, Boca Raton, Florida, 1991.

[14] J. Flower and A. Kolawa, “Express is not just a message passing system: Current and future directions in Express,” *Parallel Computing*, 20(4), pp. 597-614, 1994.

[15] I. Foster, C. Kesselman, and S. Tuecke, “Nexus: Runtime Support for Task Parallel Programming Languages,” Argonne National Laboratory, Technical Report ANL/MCS TM 205, February 1995.

[16] G. Geist and V. Sunderam, “Network-Based Concurrent Computing on the PVM System,” *Concurrency: Practice and Experience*, 4(4), pp. 293-311, 1992.

[17] K. Goldman, M. Anderson, and B. Swaminathan, “The Programmers’ Playground: I/O Abstraction for Heterogeneous Distributed Systems,” Department of Computer Science, Washington University, Saint Louis, MO 63130-4899, Technical Report WUCS-93-29, June 1993.

[18] A. S. Grimshaw, “Easy-to-Use Object-Oriented Parallel Processing with Mentat,” *Computer*, 26(5), pp. 39-51, 1993.

[19] A. S. Grimshaw and E. C. Loyot Jr., “ELFS: Object-Oriented Extensible File Systems,” University of Virginia, Computer Science Report TR-91-14, July 1991.

[20] M. Harry, J. M. del Rosario, and A. Choudhary, “The Design of VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing,” *Operating Systems Review*, 23(3), pp. 35-48, 1995.

[21] J. H. Hartman and J. K. Ousterhout, “The Zebra Striped Network File System,” *ACM Transactions on Computer Systems*, 13(3), pp. 274-310, 1995.

[22] M. Henderson, B. Nickless, and R. Stevens, “A Scalable High-performance I/O System,” In proceedings of Scalable High-Performance Computing Conference, pp. 79-86, Knoxville, Tennessee, 1994.

[23] V. Karamcheti and A. Chien, “Concert - Efficient Runtime Support for Concurrent Object Oriented Programming Languages on Stock Hardware,” In proceedings of Supercomputing’93, pp. 598-607, Portland, Oregon, 1993.

[24] D. Kotz, “Interfaces for Disk-Directed I/O,” Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, Technical Report PCS-TR95-270, September 1995.

[25] D. Kotz and N. Nieuwejaar, “Dynamic File-Access Characteristics of a Production Parallel Scientific Workload,” In proceedings of Supercomputing ‘94, pp. 640-649, Washington, DC, 1994.

[26] 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, pp. 95-108, Philadelphia, 1996.

[27] S. A. Moyer and V. S. Sunderam, “Scalable Concurrency Control for Parallel File Systems,” In proceedings of Third Annual Workshop on Input/Output in Parallel and Distributed Systems, pp. 90-106, Santa Barbara, CA, 1995.

[28] N. Nieuwejaar and D. Kotz, “Low-level Interfaces for High-level Parallel I/O,” In proceedings of Third Annual Workshop in Input/Output in Parallel and Distributed Systems, pp. 47-62, Santa Barbara, CA., 1995.

[29] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Schlatter Ellis, and M. Best, “File-Access Characteristics of Parallel Scientific Workloads,” *IEEE Transactions on Parallel and Distributed Systems*, 3(1), pp. 51-60, 1995.

[30] A. Purakayastha, C. Schlatter Ellis, D. Kotz, N. Nieuwejaar, and M. Best, “Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor,” In proceedings of Ninth International Parallel Processing Symposium, pp. 165-172, Santa Barbara, CA, 1995.

[31] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, “The Enterprise Model for Developing Distributed Applications,” *IEEE Parallel & Distributed Technology*, 1(3), pp. 85-96, 1993.

[32] D. Szafron and J. Schaeffer, “Experimentally assessing the Usability of Parallel Programming Systems,” In proceedings of IFIP WG10.3 Programming Environments for Massively Parallel Distributed Systems, pp. 19.1-19.7, Monte Verità, Ascona, Switzerland, 1994.

[33] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh, “PASSION Runtime Library for Parallel I/O,” In proceedings of Scalable Parallel Libraries Conference, pp. 119-128, Mississippi State, Mississippi, 1994.

[34] 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,” Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA, Technical Report MCS-P569-0296, February 1996.

[35] R. Thakur, E. Lusk, and W. Gropp, “I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon,” Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, Technical Report MCS-P534-0895, August 1995.

[36] D. E. Vengroff and J. S. Vitter, “I/O-Efficient Scientific Computation Using TPIE,” In proceedings of 1995 IEEE Symposium on Parallel and Distributed Processing, pp. 74-77, San Antonio, TX, 1995.

[37] D. W. Walker, “The Design Of A Standard Message Passing Interface For Distributed Memory Concurrent Computers,” *Parallel Computing*, 20(4), pp. 657-673, 1994.

APPENDIX A: A PIOUS Implementation

This appendix contains the PVM/PIOUS implementation for the small-grained I/O example program. From the original sequential code, there is a significant amount of new code that needs to be written. The original version is about 530 lines of code. Converting to parallel increases the program size by approximately 350 lines. The bolded lines in the code represent additional parallel code that the user had to write. For this example, the sequential code for **Child** did not have to be modified. As much as possible, the PVM code for the computational parallelism has been hidden away.

There are several functions that the **Parent** process calls to create (**CreateFileInPIOUS**), import files into (**ImportFileToPIOUS**), or export files from (**ExportFileFromPIOUS**) the PIOUS files system. These are tuned to the application granularity of an I/O segment and are specific to the application.

The **Parent** spawns all the **Child** processes (**CreatePVMChildren**) and waits for all the **Child** processes to finish (**WaitForChildrenToFinish**) before proceeding with the summary statistics (**Stats**). In the case of the segmented I/O, the **Parent** must also coordinate the access of the segments by the **Child** processes to ensure that all the segments are read and written only once. Note the distinction between the parallel and sequential I/O for the same file.

A **Child** process opens the global input and output files, copies the local segment of work (in one operation) to a temporary file, opens the temporary output file, performs the work and then exports the local output file back to the parallel output file (again one operation). This repeats until the global input file is exhausted. The **Parent** process is then informed and the **Child** process gracefully exits after cleaning up the temporary files.

```
#include  <pvm3.h>
#include  <pious1.h>
#include <stdio.h>

#define  GROUP  “iog”
#define  MYMESSAGE  1234
#define  MAXPATHLEN  1024
#define  INBUFSIZE  352108
#define  OUTBUFSIZE  18050
#define  REGMODE   ( (PIOUS_modet) (PIOUS_IRUSR | PIOUS_IWUSR | \
                                    PIOUS_IRGRP | PIOUS_IROTH ) )
main( int argc, char **argv )
{
   int myTID ;                                     /* my PVM handle */
   int myParentTID ;                      /* My parent’s PVM handle */
   int *childTID ;                    /* List of children’s PVM handles */
   int dscnt ;                                      /* PIOUS handle */
   int nchild ;                         /* Number of children wanted */
   int infd, outfd ;                   /* The parallel file descriptors */
   int i ;                                              /* A counter */
   int bufid, status;                   /* PVM buffer handle and status */
   FILE *fp ;                     /* Local segment’s input file handle */
   FILE *ofp ;                   /* Local segment’s output file handle */
   char  ibuffer[INBUFSIZE] ;                       /* Input buffer */
   char  obuffer[OUTBUFSIZE] ;                     /* Output buffer */
   char  inFile[ MAXPATHLEN ] ;                /* Global input and */
   char  outFile[ MAXPATHLEN ] ;               /* Output file path */
   char  myTmpInFile[ MAXPATHLEN ] ;         /* Temporary input and */
   char  myTmpOutFile[ MAXPATHLEN ] ;          /* Output file paths */

   if ( ( myParentTID = pvm_parent() ) == PvmNoParent ) {
     /* Parent -- spawn child processes                            */
     /* argv[0]:  process name      argv[1]:  input file name      */
     /* argv[2]:  output filename   argv[3]:  # of child processes */
```

```
    nchild = atoi( argv[3] ) ;
    ImportFileToPIOUS( argv[1] ) ;
    CreateFileInPIOUS( argv[2] ) ;
    childTID = (int *)malloc( nchild * sizeof( int ) ) ;
    CreatePVMChildren( childTID, argc, argv ) ;
    WaitForChildrenToFinish( nchild ) ;
    free( childTID ) ;
    ExportFileFromPIOUS( argv[2] ) ;
    fp = fopen( argv[2], "r" ) ;             /* Open sequential file */
    Stats( fp ) ;                                /* Calculate the totals */
    pvm_exit() ;                                         /* Leave PVM */
  } else {                                      /* I'm a child process */

    /* Child process -- wait for names of files to open */
    bufid = pvm_recv( myParentTID, MYMESSAGE ) ;
    status = pvm_upkstr( inFile ) ;                  /* The input file */
    status = pvm_upkstr( outFile ) ;                /* The output file */

    /* Ask for PIOUS default information */
    dscnt = pious_sysinfo( PIOUS_DS_DFLT ) ;

    /* Open the pious input and output files */
    infd = pious_popen( GROUP, inFile, PIOUS_GLOBAL, INBUFSIZE,
    PIOUS_VOLATILE, PIOUS_RDONLY, PIOUS_IRUSR, dscnt ) ;
    if ( infd < 0 )
    printError( status, "Opening input file: child" ) ;
    outfd = pious_popen( GROUP, outFile, PIOUS_GLOBAL, OUTBUFSIZE,
                         PIOUS_VOLATILE, PIOUS_RDWR | PIOUS_CREATE |
                         PIOUS_TRUNC, REGMODE, dscnt ) ;
    if ( outfd < 0 )
      printError( status, "Opening output file: child" ) ;

    /* Create local copy of input/output files for this segment */
    sprintf( myTmpInFile, "/tmp/in.%x", pvm_mytid() ) ;
    sprintf( myTmpOutFile, "/tmp/out.%x", pvm_mytid() ) ;
    while (1) {

      /* Read in the next block of work */
      status = pious_read( infd, ibuffer, INBUFSIZE ) ;
      if ( status < 0 ) {                                     /* Error */
        printError( status, "Reading input: child" ) ;
      } else if ( status == 0 ) {                          /* All done */
        break ;
      } else if ( status > 0 ) {                   /* Normal situation */

        /* Open local input file */
        fp = fopen( myTmpInFile, "w+" ) ;

        /* Fill it up */
        fwrite( ibuffer, sizeof(char), INBUFSIZE, fp ) ;

        /* Get it ready for the user's code */
        rewind( fp ) ;

        /* Open the local output file */
        ofp = fopen( myTmpOutFile, "w+" ) ;

        Child( fp, ofp ) ;                          /* Call user's code */

        /* Export the local output file to the global file */
        rewind( ofp ) ;
        status = fread( obuffer, sizeof(char), OUTBUFSIZE, ofp ) ;
        status = pious_write( outfd, obuffer, status * sizeof(char) ) ;

        /* Close the local input and output files */
        fclose( fp ) ;
        fclose( ofp ) ;
```

```
            }
          }
          /* Shutdown this child and let the parent know */
          bufid = pvm_initsend( PvmDataRaw );           /* A buffer please */
          status = pvm_send( myParentTID, MYMESSAGE ) ;     /* Tell parent */
          status = pious_close( infd ) ;        /* Close PIOUS input file */
          status = pious_close( outfd ) ;       /* Close PIOUS output file */
          pvm_exit() ;                                          /* Exit pvm */
          unlink ( myTmpInFile ) ;          /* Remove the local input file */
          unlink ( myTmpOutFile ) ;        /* Remove the local output file */
       }
       return 0 ;
    }
```