1 OBJECT–ORIENTED DISTRIBUTED AND PARALLEL I/O STREAMS [1]

Andrew Dick and Eshrat Arjomandi

Department of Computer Science
York University, Toronto ON, Canada
{andrewd, eshrat}@cs.yorku.ca

Tim Brecht

Department of Computer Science
University of Waterloo, Waterloo ON, Canada
brecht@cs.uwaterloo.ca

Abstract: Writing programs for parallel and distributed computing environments can be significantly more complex than writing programs for their sequential counterparts. These complexities mainly arise from the additional synchronization and communication requirements imposed by such environments. These requirements also make debugging and maintaining such programs significantly more complicated. The problem of debugging and maintenance is further exacerbated by the lack of good debuggers and the lack of proper I/O support for such environments.

In this paper we describe the design and implementation of an object-oriented streams library (`piostream`) which provides convenient and extensible constructs for input and output in parallel and distributed programming environments. These environments include multi-threaded applications, multiprocessors, and distributed systems. Our design is based on the familiar C++ `iostream` library. Thus simplifying the use of I/O operations in parallel and distributed environments. A prototype implementation, which has been integrated within the ABC++ concurrent object-oriented library for C++, is

[1] APPEARS IN THE 13TH ANNUAL INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS (HPCS '99), KINGSTON, ONTARIO, PP. 253-268. JUNE 1999.
Arjomandi and Brecht are supported by the Natural Sciences and Engineering Research Council of Canada and the IBM Toronto Center for Advanced Studies.

described and used to demonstrate the feasibility of our design and the ease with which it can be used.

Keywords: Distributed, Parallel, Object-Oriented, I/O Streams

1.1 INTRODUCTION

Significant research efforts have been expended in recent years to improve the performance of I/O subsystems by using parallel techniques to transfer portions of data to and from several storage devices simultaneously. These efforts have concentrated almost exclusively on alleviating the I/O performance bottleneck by using multiple disk devices to perform file I/O (Gotwals et al., 1995b; Nieuwejaar and Kotz, 1995). Unfortunately, techniques for providing users with the ability to simply and easily perform input and output operations on multiple processors or hosts simultaneously for the purpose of debugging, executing and maintaining parallel and distributed programs have received little attention.

In this paper we address this shortcoming and focus on techniques that enable users to easily write parallel and distributed applications that perform I/O. We discuss the issues involved in designing and implementing the components for a parallel streams library called `piostream`. The `piostream` library is not intended to solve the I/O bottleneck, but instead is designed to provide simple and easy to use I/O constructs for the purpose of writing, debugging, executing, and maintaining distributed and parallel programs. The `piostream` library is comprised of `postream` for parallel output, `pistream` for parallel input, and `pfstream` for parallel file I/O. By using the `piostream` library, programs can execute input and output operations from any machine in a networked environment and the input and output data is transparently obtained from or sent to the main host (the host on which the program originally executes) by the underlying run-time system. Our target environments include multi-processors, networks of workstations, and distributed communicating workstations and PCs.

The primary advantages of our library and thus the main contributions of this paper are:

- It is based on the C++ `iostream` library (Teale, 1993) and as a result is familiar and straightforward to use as well as easy to extend. Since `piostream` constructs are high-level, users are not required to construct, send and receive elaborate messages. The users can easily input and output predefined and user-defined data types, by overloading `operator<<()` or `operator>>()`.

- It supports C++ I/O manipulator functions which allow precisely formatted output. Extensible user-defined manipulator functions are also supported.

- It is built using and is fully compatible with standard C++, which means that no special compiler support or language extensions are required.

- The library provides mechanisms for identifying the source of the output. For example, output being printed by tasks executing on several remote client machines is automatically and transparently collected, collated, and printed to a screen or file on the main host along with information identifying the host name, process id, and thread id of the task performing the output.

- Input constructs can be used on hosts that are potentially remote. The `piostream` library provides the facilities to transparently read input on remote hosts using shared or independent stream positions using an interface that is similar to the C++ `iostream` library.

- Synchronization is an integral part of the library and as a result the user is not required to synchronize access to input and output streams. The proposed file I/O constructs transparently provide mutually exclusive access to I/O streams.

- The proposed file I/O constructs allow the user to access the main host's file system from remote hosts for input and output.

We believe that our clean and simple interface is one of the main contributions of this paper. Many parallel programs are converted from their original sequential versions. As a result, converting a program from a sequential program to a parallel program can be complicated and time consuming. To illustrate the ease with which `piostream` can be used for I/O, Figure 1.1 provides an example in C++ using the C++ `iostream` interface for `cout`, `cerr`, `cin` and `fstream` and their `piostream` counterparts `pout`, `perr`, `pin`, and `pfstream`. The C++ `iostream` interface is illustrated in Figure 1.1A and the `piostream` interface is shown in Figure 1.1B.

In this paper much of our discussion of the `piostream` library will take place in the context of a parallel system that provides concurrency through an active object concurrency model (Arjomandi et al., 1995)[1] such as ABC++ (O'Farrell et al., 1995; Arjomandi et al., 1996). The proposed design and our library could easily be implemented using any object-oriented concurrency model.

The remainder of this paper will discuss the design and prototype implementation of the piostream library. Section 1.2 describes the design and implementation of the `postream` construct for output. The `pistream` construct for input is explained in Section 1.3 and Section 1.4 describes the proposed file stream constructs. Related work is presented in Section 1.5 and our conclusions and contributions are listed in Section 1.6.

[1] An active object is an object with its own thread of control. It encapsulates thread control, message queuing, and synchronization.

(A)

```
// Counter
int num_elements;
// Data
int element;
// Input file
ifstream in("input", ios::exist);
// Output file
ofstream out("output", ios::nocreate);
if ( (!in) || (!out) ) {
  cerr << "Error in input or "
       << "output file." << endl;
  exit(0);
}
// Header buffer
char header[256];

// Get number of input elements
cout << "Please input number of "
     << "elements:" << flush;
cin >> num_elements;

// Get header for output file
cout << "Please enter header "
cout << "for output file." << endl;
cin.getline(header, 256);

// Insert header into output file
out << header;

// Copy and convert decimal data from input file
// to hexadecimal in output file
for (int i = 0; i < num_elements; i++) {
  in >> ws >> element;
  out << hex << element;
}
```

(B)

```
// Counter
int num_elements;
// Data
int element;
// Input file
pifstream in("input", ios::exist);
// Output file
pofstream out("output", ios::nocreate);
if ( (!in) || (!out) ) {
  perr << "Error in input or "
       << "output file." << endl;
  exit(0);
}
// Header buffer
char header[256];

// Get number of input elements
pout << "Please input number of "
     << "elements:" << flush;
pin >> num_elements;

// Get header for output file
pout << "Please enter header "
pout << "for output file." << endl;
pin.getline(header, 256);

// Insert header into output file
out << header;

// Copy and convert decimal data from input file
// to hexadecimal in output file
for (int i = 0; i < num_elements; i++) {
  in >> ws >> element;
  out << hex << element;
}
```

Figure 1.1 This diagram provides a comparison of interfaces between the C++ `iostream` library and the `piostream` library.

1.2 POSTREAM

Our `piostream` library is designed to solve several problems encountered when using the `iostream` library in parallel and distributed environments. The central problem is that using standard techniques for creating threads of execution on remote hosts results in the linking of that thread's `stdin` and `stdout` to what are essentially incorrect devices for the purposes of performing input and output on the main host. If remote threads are created using an `rexec` system call, a socket is created and given to the `stdin` and `stdout` of the thread on the remote host (which is clearly not the correct device). Thus, using `cin` and `cout` would require the programmer to add special code to their program to coordinate with the main host. Our design of the `piostream` library alleviates the programmer from the burden of writing special code for performing I/O in such environments. In addition, our library also solves the problem of unsynchronized data interleaving (which can occur when access to the input and output streams is not properly synchronized) and the data origin identification problem (which means that the source of the output is not identified). Several user-level solutions are possible but result in added complexity to the user code which makes the program harder to port and maintain.

In our library, unique identification tags are prepended to each line of output. The identification tags consist of three variables: the host name, the process id,

and the thread id. The `piostream` library also transparently synchronizes access to the output stream thus greatly simplifying the user code. It is clear that a library providing such facilities would be invaluable for debugging purposes. Furthermore such a library contributes to user code simplicity and portability, hence lowering the cost of code maintenance.

In the remainder of this section we present issues concerning the design of the `postream` library. To demonstrate the feasibility of our design, we have built a prototype implementation of `postream` in a library which supports concurrency in C++. This prototype implementation is discussed in the last part of this section.

1.2.1 The Architecture of Postream

The `postream`'s architecture is primarily based on a client-server model. In this model, the server is a dedicated active object in charge of collecting output from remote hosts and printing the output to a local device. It handles data origin identification and synchronizes data interleaving. The clients are active objects which may be executing on remote hosts who may wish to use the `postream` construct. The `postream` construct transparently passes output from each client to the server. The creation of the dedicated server can be encapsulated within the startup routines of the parallel system (as is the case in our prototype).

Similar to the `cout` object of C++'s `ostream`, `postream` provides a parallel output object called `pout`. Each client has its own `pout` object. The client-server architecture of `postream` supports the same interface as `ostream`. It allows for the chaining of `operator<<()` functions and the output of user-defined objects. The data origin identifier is created by the run-time system the first time an active object uses `pout` and is used in subsequent uses of `pout`. The clients' requests for output are transparently queued in the message queue of the active object representing the dedicated server. The server then outputs the data to the proper output device.

A client object has one of two choices with respect to when to transfer data to the server. It can either transfer the data after a flush of the output stream, or after each invocation of the `operator<<()` function. Transferring the data after a flush of the output stream requires each object to store its output data until the flush occurs. On the other hand transferring the data after each invocation of the `operator<<()` function localizes the overhead in the server object but requires more message passing between the client and the server. Because message passing is usually expensive in a distributed system, our prototype implementation transfers data to the output server after flushing the output stream. Ghormley *et al.* (Ghormley et al., 1998) also used a centralized server design to implement their distributed operating system and found it was not a significant bottleneck when used in a network of workstations.

We considered alternative models in the design of the `postream` library. A possible alternative is the *pass to parent* model, which is based on passing the output data to the parent node and making the parent node responsible for

all of its descendant's output. A second possibility is a *shared lock* model in which a lock is used to synchronize each client's output operations. The pass to parent model was rejected due to the complications involved in maintaining a dynamically changing hierarchy of ancestors. The shared lock model was not used because of its dependence on direct client access to `stdout` and `stderr` on the main host.

1.2.2 Implementation

We have developed a prototype implementation of the `postream` library in C++ for the parallel class library, ABC++ (O'Farrell et al., 1995; Arjomandi et al., 1996). Before discussing the major components of the implementation, we briefly describe the implementation environment.

ABC++ is a class library for parallel programming in C++. It promotes code reuse through the abstraction and polymorphism facilities of the Object-Oriented Programming (OOP) paradigm. ABC++ is written in C++ and requires no preprocessing, compiler or language extensions. The library is portable and object-oriented with a concurrency model based on *active objects*. It presently runs on SUN workstations, IBM RISC System/6000 workstations and the IBM SP supercomputers. To allow active objects of a class to be created, the class must publicly inherit from the class `Pabc`. Active object communications are based on *synchronous* and *asynchronous* object interactions through a Remote Method Invocation (RMI) on both distributed and shared memory platforms. ABC++ encapsulates the work required to control threads and synchronize objects thus allowing the user to concentrate on the semantics of the program. The ABC++ library does not provide support for parallel and distributed I/O (O'Farrell et al., 1995).

As mentioned earlier, we use a client-server model in the design of the `postream` library. In our prototype implementation, the server is an active object, in charge of output. Any user active object, namely the client, wishing to perform output, will transfer its output data to the server object. All actions performed on behalf of the server and clients as related to I/O are transparently supported by the library and the underlying run-time system. The user program simply uses `pout` in the same fashion that `cout` is used.

The server component of the `postream` library is primarily a single class, `postream_server`. A single object of this class, `po_server` is instantiated to act as the output server. The server body continually accepts RMIs from clients until it receives a terminate RMI from the run-time system. The `postream_server` class contains a dynamically sized vector of buffers for storing output data, one per client. If the vector becomes full, it is doubled in size.

Each client is assigned a buffer when the run-time system first transfers the client's data to the output server. Each buffer is also dynamically sized with an initial size of zero. Each buffer has a size limit which if reached results in the data in the buffer being flushed to `stdout`. The buffer is used as a temporary storage facility in the event of an overflow of the buffer stored locally on the

client's host. The client's data origin identification information is stored with the client's respective buffer on the server. The `postream_server` supports the following methods:

- *Request key* (`Request_key()`) — this public method allows a client to obtain a unique identification key for communicating with the server. This method is invoked by the run-time system the first time output data is transferred to the server. This exchange can not be performed at object creation time because the active object communication mechanisms may not have been instantiated at this time. This method takes the client's identification information (host name, process id, and thread id) as parameters.

- *Transfer data* (`Transfer_data()`) — this public method is invoked by the run-time system to transfer data to the server for output to `stdout`. The client's index key and output data are provided as arguments.

The client component of the `postream` library consists of one class, `postream`. At first glance, extending the existing `ostream` class through inheritance seems to be the best approach for the design of the `postream` class. However, the extensive use of friend functions for the `iostream` interface (`operator<<()` and `operator<<()`) creates problems with the inheritance approach. Using the existing `operator<<()` functions creates the problem of upcasting the `postream` object to its base class, which interferes with the proper use of the manipulator functions used to encapsulate the transfer of data between client and server. In order to support a `piostream` interface that is as close to the existing `iostream` interface as possible, our `piostream` library uses the `flush` and `endl` manipulator functions to encapsulate the transfer of data between the client and the server. However, the use of these functions prevents the use of inheritance to extend the `iostream` classes.

An alternate approach, overloading the `operator<<()` functions, also does not work because two identical function prototypes are created (`postream` is an `ostream`) and they cannot be distinguished by the compiler. Hence inheritance cannot be used to extend the `iostream` library and therefore a new `postream` class must be implemented. Similarly the `pistream` and `pfstream` components cannot be extended from the existing `iostream` classes.

The use of the `operator<<()` and `operator>>()` interface creates a problem with supporting manipulator functions with arguments. `Operator<<()` is a binary operator and cannot pass the required information (manipulator pointer, output stream and manipulator argument) as arguments. The technique used by the `iostream` library to solve this problem is to construct a temporary class during the function chain[2]. This approach cannot be duplicated to create a second manipulator of the same name due to a class name or function conflict (depending on the particular `iostream` implementation). The approach used

[2]More information on this technique can be found in (Eckel, 1995) (pp. 171-178).

to bypass this problem in the prototype implementation of the `piostream` library is to re-implement the manipulator library (`iomanip.h`). The users must therefore include our `piostream` version of `iomanip.h` (called `piomanip.h`) in order to use manipulator functions with arguments.

Each client active object contains an object of the `postream` class named `pout`. This is implemented in the prototype by adding a data member, `pout`, to the root class in ABC++. Since all user active object classes inherit from ABC++'s root class, `Pabc`, every client contains a `pout` object as a data member. This object, `pout`, then acts as an interface between each active object and the `postream_server` object.

Data members of the `postream` class include the index key provided by the `postream_server` object, the handle of the `postream_server` object, and a storage buffer. The handle of the server is used to access the processor id, and the machine name or Internet address of the output server. The `postream` class supports both the output of user defined objects (through overloading) and manipulator functions.

The interface and behaviour of the `postream` methods are similar to their `ostream` counterparts. The `postream` class supports the following methods:

- *Set address of* `po_server` (`Set_POS_server()`) — this private method is invoked by the run-time system to set the `postream_server` handle which consists of the server run-time host and memory address.
- *Output operator for predefined types* (`operator<<` {`predefined type`}) — this public method allows the client to insert all predefined data types (except character strings) into the local buffer. Character strings are handled separately because of their arbitrary size.
- *Output operator for character strings* (`operator<<` {`character string`}) — this public method is used to insert character strings into the buffer. If the character string is too large to fit into the client buffer, it is divided and transmitted to the `postream_server` by remotely invoking the `postream_server` method `Transfer_data()` on the server.
- *Output operator for manipulator functions* (`operator<<` {`manipulator function`}) — this public method is used to invoke manipulator functions including `endl` and `flush` in a chained fashion. The manipulator function parameter is invoked with the `postream` object as an argument.
- *Manipulator functions* (`flush`, `endl`, `hex`, etc.) — these manipulator functions implement the parallel variants of the standard `iostream` manipulator functions. The parallel manipulator functions invoke their standard `iostream` counterparts which set the appropriate format fields on the underlying `postream` client buffer. These manipulator functions are invoked in a similar manner to their `iostream` counterparts.
- *Show / Hide data origin identification* (`Show_id/Hide_id`) — these manipulator functions allow the client to suppress the data origin identification that is prepended to each client's output data.

The `postream` library follows the design of the `ostream` library of C++ and the run-time system provides all the support for transparently performing data transfer, buffering, data origin identification and synchronized interleaving. At the user level, active objects simply use `pout` for transferring data to standard output on the main host. The `piostream` library also provides a second `postream` object, `perr`, for parallel and distributed output to `stderr` on the main host by remotely executing active objects.

1.3 PISTREAM

Like sequential programs, many parallel and distributed programs also require input to operate. In a distributed environment however, the use of `cin` by active objects residing on remote machines is often not meaningful because they are frequently not associated with `stdin` on the main host. The parallel input stream, `pistream`, is a component of the parallel streams library designed for performing input operations in a parallel or distributed environment. The `pistream` construct supports distributed input for remotely executing objects. We support the distribution of input to client objects using two different modes: broadcast and striped. The broadcast mode transmits each token of information to all active objects that perform input (i.e., that call `pin`). The striped mode transmits each token of information to a different active object, in a first-come first-serve fashion. The different approaches are not compatible, hence one distribution mode is specified on the command line for each program execution.

Traditionally, the user had to control this data distribution, and must also provide extra methods to marshal and transfer the information. Few existing systems are capable of supporting access to `stdin` on the main host for remote objects[3]. In most existing systems delivering input from a user or file to threads executing on several different hosts is simply not possible. Instead the user must first read and buffer the input data on the main host. The user must then co-ordinate the sending and receiving hosts, and somehow transmit the data to the desired hosts. Clearly this approach is not easy to use and makes porting and maintaining the user program more difficult. The `pistream` input construct provides an encapsulated, easy to use method of distributing input data to remote user objects. It allows remote objects to read data through the natural and familiar `istream` interface in either a striped or broadcast fashion.

1.3.1 The Architecture of Pistream

The `pistream` architecture is based on a client-server model similar to `postream`. It encapsulates the complexities of input data distribution. The server object reads input from `stdin`, buffers and serves client active object requests for data in either a broadcast or striped fashion. The server maintains

[3]Condor (Litzkow et al., 1988) for instance supports access through Remote Procedure Calls (Litzkow, 1987). However, the current implementation of Condor only supports batch executions and has no support for interprocess communication.

each participating client object's read position in the buffered input stream to support input broadcasting. To allow striped input, the server maintains a single shared stream position.

Similar to C++'s `istream`, `pistream` provides a parallel input object called `pin`. Each client will have its own `pin` object. The client-server architecture of `pistream` supports the same interface as in `istream`. It supports the chaining of `operator>>()`, user defined objects, manipulator functions and all `istream` methods. The run-time operations of the `pistream` library are fully encapsulated allowing the user to input data without additional distribution complexities. The run-time system encapsulates both the request for data at the client side and the distribution of data by the server. The server object encapsulates the distribution of data to requesting active object clients.

1.3.2 Implementation

In our prototype implementation, the server is an active object, in charge of input. Any user active object, namely a client, wishing to input data, will request input data from the server object. Below we describe an overview of the major components of the server and the clients.

The server side of the `pistream` library is primarily a single class, `pistream_server`. A single object of this class, `pi_server` is instantiated to act as a server. The data members of this class include a buffer containing the input data for the program. In striped mode, a single position is maintained for reading the buffer. In broadcast mode, a dynamically sized vector of positions, one per client, is maintained. Upon a data request the appropriate position is set, and the corresponding data is read. The server object supports the following methods which are similar to their `postream_server` counterparts:

- *Request key* (`Request_key()`), same as `postream`
- *Request data* (`Request_data()`) — this public method is invoked by the client run-time system to request data. The client's index key and a delimiting character are provided as arguments. The server provides a string of data, delimited by the specified character — with the default being the newline character. The conversion of data from a byte sequence to the appropriate C++ data type is handled on the client side of the `pistream` client-server model.
- *End of file* (`eof()`) — this public method is invoked by the run-time system and performs the same function as the `eof() istream` method. The method returns the condition of the `EOF` bit for the client's stream position in the server's data buffer.
- *Poll for data on* `stdin` (`Poll_available_data()`) — this private method is invoked by the server to determine if data is available to be read from `stdin`. The technique used can detect input from either the console or a redirected file.

The client portion of the `pistream` library consists of one main class, the `pistream` class. Each client owns an object of the `pistream` class, `pin`. This is implemented similarly to `postream`, by adding `pin` as a data member of the root class in ABC++. Data members of the `pistream` class are the index key provided by the input server and the handle of the server object. The `pistream` class supports both the input of user defined objects (through overloading) and the use of manipulator functions.

The methods outlined below are supported by the `pistream`, the duties of these methods are the same as their `postream` and `istream` counterparts.

- *Set address of* `pistream_server` (`Set_PIS_server()`),
- *Input operator for predefined types* (`operator>>` {`predefined type`}),
- *Input operator for manipulator functions* (`operator>>` {`manipulator function`}),
- *Input character(s)* (`get()`),
- *Input next line* (`get_line()`),
- *Examine next character* (`peek()`),
- *Manipulator functions* (`ws`, `hex`, etc.),
- *End of file* (`eof()`).

1.4 PFSTREAM

The proposed file stream component of the `piostream` library provides support for file I/O for remotely executing objects that may not have access to the main host file system. The Network File System (NFS) (Sandberg, 1985) provides support for the sharing of different machine's file systems. It is possible however that file systems cannot be mounted for security or administrative reasons. Another potential difficulty with NFS is the coordination required to support active object file I/O with a single shared file pointer. The statelessness of the NFS server requires any coordination to be performed explicitly by the user. For these reasons NFS is not a suitable solution for high-level distributed file I/O.

The `pfstream` component of the `piostream` library is composed of three distinct classes, `pofstream`, `pifstream`, and `pfstream`, which are based on their `fstream` counterparts. The `pfstream` constructs use the same client-server model as the `postream` and `pistream` constructs previously discussed. In addition to sharing many design issues with `postream` and `pistream`, issues relevant exclusively to the design of the `pfstream` include:

- The handling of conflicting modes when multiple active objects access the same file simultaneously. By simultaneously we mean that at least

two client objects overlap the `open()` and `close()` file operations. The `pfstream` library supports independent and shared file stream positions.

- The tracking and representation of open files required to support shared access to the same file by multiple clients.
- The simultaneous use of different access modes on the same file by different active objects.
- The opening of the same file multiple times by the same active object.

Together these facilities will make the `pfstream` constructs an important component of the `piostream` library. The high-level tools will encapsulate complicated sequencing of multiple remotely executing active object interactions with files on the main host file system. We have completed the preliminary design of the `pfstream` library and in the future hope to develop a prototype implementation using the ABC++ environment.

1.5 RELATED WORK

Parallel I/O that addresses the problems of debugging and maintenance has received little attention in the literature and even less research has been conducted on object-oriented solutions to this problem. In this section we first examine how I/O is supported by the popular PVM (Parallel Virtual Machine) system and the MPI (Message Passing Interface) standard and then describe existing object-oriented solutions for parallel I/O.

PVM (Geist et al., 1994), which is a widely used library for parallel programming, provides no access to `stdin` for remote tasks. Each task in PVM is provided with a `stdout` sink, a construct that inherits from its parent task. PVM supports data origin identification by tagging output from each task. Synchronized data interleaving is provided around each output statement. Support for user-defined objects and formatting manipulator functions is not provided because PVM is implemented in C. The main drawback of the approach used in PVM is that output must first be marshaled into a character array by the user before being output. The added complexity makes the user program more difficult to port and maintain.

MPI (Snir et al., 1996) is a standard message passing interface for parallel programming. The current specification of MPI does not fully address the support of I/O for all tasks (Snir et al., 1996) (pp. 287-289). The MPI-2 standards document (MPI Forum, 1997) has addressed this problem with a chapter on I/O supporting only C and Fortran77 interfaces. The problem with the support provided for output in PVM and MPI is that neither provides an easy to use and extensible interface that transparently supports I/O on arbitrary objects and manipulator functions.

Gotwals *et al.* (Gotwals et al., 1995a), explore the problem by implementing the `d/stream` construct in the parallel language `pC++` (Gotwals et al., 1995a). The `d/stream` construct is a language-independent abstraction that supports a number of simple primitives which allow I/O to be performed on distributed

arrays with arbitrary object elements. Conceptually, a `d/stream` is a buffer that is used as an intermediate storage step between the user and a file. Using a `d/stream`, a user is able to insert data into the buffer, then write it to a file at a later time; or conversely read data from the file into the buffer, from which it can be extracted into a distributed array. They accomplish this by using a compiler dependent feature called *collections*. A collection is defined as a distributed array of objects with additional underlying infrastructure that provides support for arbitrary data structures such as trees.

Gotwals *et al.* duplicate portions of the C++ `iostream` interface by supporting the use of the `operator<<()` and `operator>>()` methods which can be extended to support user defined objects. `pC++/streams`, the implementation of `d/streams` in the language pC++, is described for file I/O only, although extending it to implement support for standard I/O descriptors would likely be possible. A major limitation of `d/streams` which prevents it from being widely and easily used is that `pC++/streams` is based on the compiler dependent construct, collections. Moreover the `pC++/stream` construct requires the use of a parallel file system for data buffering and transmission in a distributed environment. The `pC++/streams` interface also lacks support for chaining I/O method invocations (the use of several input or output operations in the same C++ statement) and manipulator functions. As a result of these limitations `pC++/streams` deviates significantly from C++ `iostream` interface.

The `piostream` library presented in this paper is based on the interface of `iostream` which means that chained I/O and manipulator functions are both supported. The `piostream` library does not rely on any language extensions and therefore, depends only on the use of a standard C++ compiler. Lastly the `piostream` library file constructs allow remote objects to perform I/O on the main host file system without dependence on the use of a parallel file system unlike `pC++/streams`.

1.6 CONCLUSIONS

In this paper we describe the design and prototype implementation of a C++ streams library called `piostream` designed for use in parallel and distributed environments. Our design is based on the familiar C++ `iostream` library and as a result it is easy to learn, use and extend. The `piostream` library provides high-level constructs `pin`, `pout`, `perr`, and `pfstream` so that users are not required to construct, send and receive elaborate messages in order to perform I/O with active objects that may be executing on remote hosts. We support the chaining of I/O method invocations, manipulator functions and the overloading of operators to support user-defined data types. Unlike many existing systems, our prototype implementation is built using and is fully compatible with C++, which means that no special compiler support or preprocessing is required. Synchronization is an integral part of the `piostream` library and therefore, the user is not required to synchronize access between multiple objects and I/O streams or files on the main host.

The `piostream` library demonstrates that parallel and distributed I/O can be supported using a standard and familiar interface. Moreover, parallel and distributed I/O can be supported without the use of compiler and language extensions. The `piostream` library provides intuitive and powerful high-level object-oriented constructs that can offer significant benefits to programmers when writing, debugging, executing and maintaining parallel and distributed programs.

1.7 ACKNOWLEDGEMENTS

We thank Dr. William O'Farrell of IBM Canada and Dr. Gregory Wilson of Software Carpentry for their many helpful discussions related to this work.

References

Arjomandi, E., O'Farrell, W., Kalas, I., Koblents, G., Eigler, F., and Gao, G. (1995). ABC++: Concurrency and inheritance in C++. *IBM Systems Journal*, 34(1):120–136.

Arjomandi, E., O'Farrell, W., and Wilson, G. (1996). Smart messages: An object–oriented communication mechanism. In *2nd Conference on Object–Oriented Technologies and Systems (COOTS)*, pages 233–240, Toronto, Canada.

Eckel, B. (1995). *Thinking in C++*. Prentice Hall, New Jersey.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge Massachusetts.

Ghormley, D. P., Petrou, D., Rodrigues, S. H., Vahdat, A. M., and Anderson, T. E. (1998). GLUnix: a global layer Unix for a network of workstations. In *Software, Practice and Experience*.

Gotwals, J., Srinivas, S., and Gannon, D. (1995a). pC++/streams: a library for I/O on complex distributed data structures. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–19, Santa Barbara.

Gotwals, J., Srinivas, S., and Yang, S. (1995b). Parallel I/O from the user's perspective. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 129–137.

Litzkow, M. (1987). Remote Unix – turning idle workstations into cycle servers. In *Proceedings of Usenix Summer Conference*, pages 381–384.

Litzkow, M., Livny, M., and Mutka, M. W. (1988). Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111.

MPI Forum (1997). MPI-2: Extensions to the message–passing interface. Technical report, (http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html).

Nieuwejaar, N. and Kotz, D. (1995). Low–level interfaces for high–level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62.

O'Farrell, W., Eigler, F., Kalas, I., and Wilson, G. (1995). *An Introduction to the IBM Parallel Class Library for C++*. ABC++ Version 1, Release 1, IBM Canada.

Sandberg, R. (1985). The design and implementation of the Sun network file system. In *USENIX Association Conference Proceedings*, pages 119–130.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1996). *MPI: The Complete Reference*. The MIT Press, Cambridge Massachusetts.

Teale, S. (1993). *C++ IOStreams Handbook*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts.