# pC++/streams: a Library for I/O on Complex Distributed Data Structures

**Jacob Gotwals  Suresh Srinivas  Dennis Gannon**

Department of Computer Science, Lindley Hall 215,
Indiana University, Bloomington, IN 47405.

{jgotwals, ssriniva, gannon}@cs.indiana.edu

## Abstract

The design and implementation of portable, efficient, and expressive mechanisms for I/O on complex distributed data structures - such as found in adaptive parallel applications - is a challenging problem that we address in this paper.

We describe the design, programmer interface, implementation, and performance of *pC++/streams*, a library that provides an expressive mechanism for I/O on distributed arrays of variable-sized objects in pC++, an object-parallel language. pC++/streams is intended for developers of parallel programs requiring efficient high-level I/O abstractions for checkpointing, scientific visualization, and debugging.

pC++/streams is an implementation of *d/streams*, a language-independent abstraction for buffered I/O on distributed data structures. We describe the d/streams abstraction and present performance results on the Intel Paragon and SGI Challenge showing that d/streams can be implemented efficiently and portably.

## 1 Introduction

Operating systems provide I/O primitives that allow the programmer to read and write blocks of bytes. I/O libraries on the other hand can provide I/O primitives that allow the programmer to work at higher levels of abstraction. For example, the C standard I/O library allows the programmer to perform I/O directly on integer, real, and string variables and the C++ streams library provides primitives for working at an even higher level of abstraction: I/O on arbitrary objects.

Distributed arrays (as found in HPF [10]) are a common data structure for parallel programming. Recently I/O libraries have been developed that provide primitives supporting I/O on distributed arrays of fixed-size elements (e.g. distributed arrays of reals).

Adaptive parallel applications using dynamic distributed data structures of variable-sized elements (e.g. distributed grids of variable density) are now emerging. In addition, parallel object-oriented programming languages supporting complex distributed data structures (e.g. distributed arrays of variable-sized objects) are now becoming available. The design and implementation of portable, efficient I/O libraries providing expressive I/O primitives that support I/O on these complex distributed data structures is a challenging problem that we address in this paper.

We first identify parallel programming tasks for which the use of high-level I/O primitives is appropriate. Next we describe d/streams, a language-independent abstraction for I/O on distributed arrays. We then discuss the interface, implementation, and performance of pC++/streams, an implementation of d/streams supporting I/O on complex distributed data structures with variable-sized elements in the object-parallel language pC++.



PPOPP '95 Santa Clara, CA USA
© 1995 ACM 0-89791-701-6/95/0007...$3.50

## 2 Parallel Programming Tasks Requiring High-Level I/O Mechanisms

I/O mechanisms operating at different levels of abstraction are appropriate for different I/O tasks (see **Figure 1**). In general, the higher the level of abstraction at which a set of I/O primitives operates, the less control it gives the programmer over the details of the I/O process. I/O libraries such as pC++/streams that provide the programmer with primitives for explicit I/O at a high level of abstraction are appropriate for a wide range of parallel programming tasks in which ease of coding, portability, and performance are important factors and where a high degree of programmer control over low-level I/O details is not required. Such tasks include:

- **Communicating** partial and final results to other applications and to tools (e.g., scientific visualization tools).
- **Checkpointing**: Many long-running parallel applications need to save the state of complex distributed data-sets periodically so that computation can be resumed at a later point. Periodically saving data-sets provides insurance against program termination by software bugs and job-control facilities.
- **Debugging**: Many parallel applications originate from sequential versions of the same program. During the parallelization process application developers often need to compare results of parallel and sequential runs on the same problem, to confirm that parallelization has not introduced bugs. This frequently involves output of large distributed data structures from the parallel program.

Object-oriented database management systems (*OODBMS*) provide another type of I/O mechanism that may prove useful for such tasks. However, explicit I/O is a more appropriate

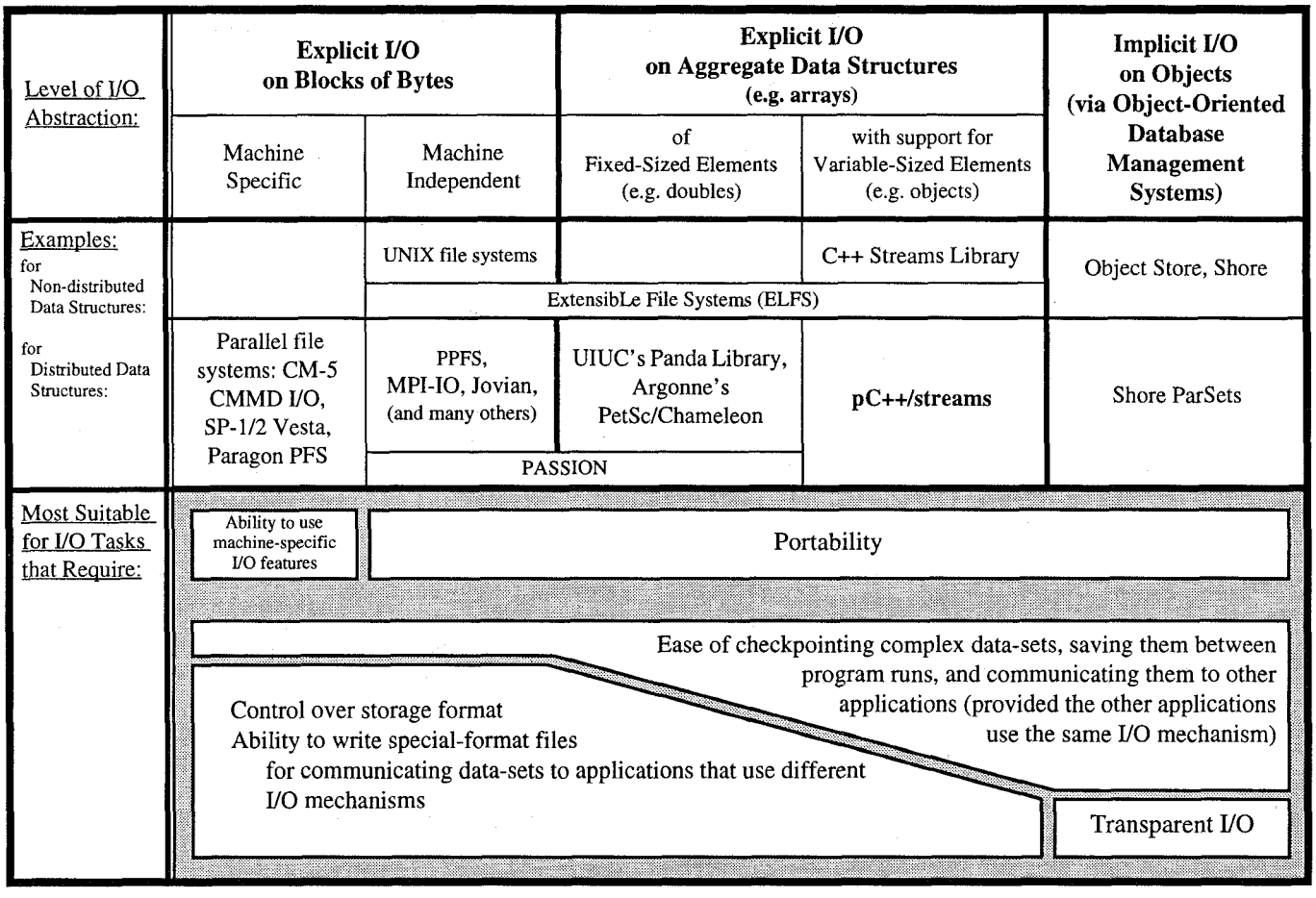



**Figure 1. A RANGE OF I/O MECHANISMS: Those mechanisms offering a higher degree of control over low-level I/O details are grouped toward the left; those offering greater ease of use and supporting I/O primitives operating at a higher level of abstraction are grouped toward the right. I/O libraries such as pC++/streams are appropriate for a wide range of I/O tasks common in many parallel applications, where ease of coding, portability and performance are important; for example, checkpointing complex data-sets, saving them between application runs, and communicating them to other applications and tools. (The thickness of the shapes under a given I/O mechanism in this figure is intended to represent the suitability of that mechanism for the indicated I/O task.)**

mechanism when a higher degree of control over the I/O process and storage format are desired or when the maintainance of an OODBMS would entail complexity disproportionate to the size of the I/O task at hand.

Parallel platforms offer low-level abstractions for I/O to secondary storage through diverse and often complex interfaces. Obtaining high I/O performance using these interfaces often requires a knowledge of parallel I/O, disk striping, and memory alignment of I/O buffers. Higher-level I/O libraries can be used to encapsulate this low-level I/O complexity. This is beneficial for the majority of parallel program developers who generally prefer not to delve into the low-level details of I/O optimization.

## 3 d/streams: An Abstraction for Buffered I/O on Distributed Arrays

A *d/stream* is a language-independent abstraction with a small number of simple primitives to be used for buffered I/O on distributed arrays. In this section we define an interface for d/streams, and in the section following we discuss the interface, implementation, and performance of an actual d/streams implementation.

Conceptually a d/stream is a buffer associated with a file. Data can be inserted from distributed arrays into an output d/stream's buffer and later written to the file; data can be read from the file into an input d/stream's buffer and later extracted into distributed arrays. Refer to **Figure 2**, which gives a language-independent description of the primitives used to perform these operations.

The state diagrams in Figure 2 show the order in which the primitives are called. Several constraints on the use of the primitives that cannot be indicated in the state diagrams so we discuss them here. Data written must be read back in the same order. More specifically:

- When a file written by an output d/stream is read by an input d/stream, every **read** or **unsortedRead** must correspond to a **write** that occurred when the file was written, and every **extract** must have a corresponding **insert**.

**Figure 2. D/STREAM INTERFACE: A d/stream is a simple abstraction with a small number of primitives to be used to express I/O operations on distributed arrays. A d/stream provides a high-level interface for buffered I/O to files, so the use of d/streams is similar to the use of lower-level file interfaces (e.g., data is read in the same order as it was written). The d/stream primitives are listed below. The state diagrams below to the right specify the order in which the primitives are intended to be used. Both *read* and *unsortedRead* transfer a block of data (written by a corresponding *write*) from a file into an input d/stream's buffer. *UnsortedRead* yields maximum input performance when reading un-ordered array data, as discussed in Section 3.**

**open(filename):**
Opens a d/stream file for output or input.

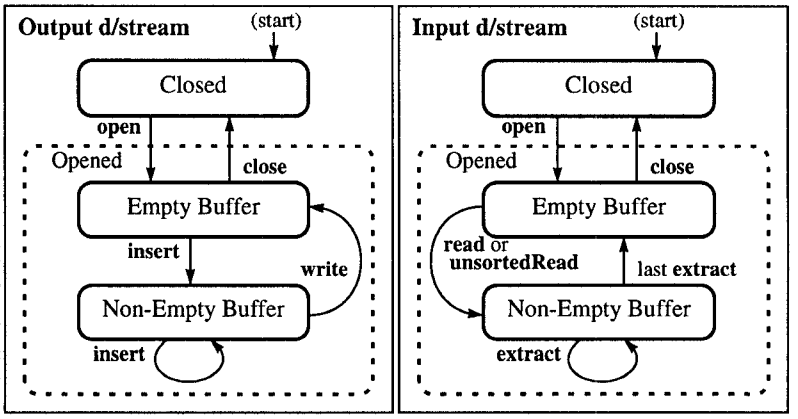
**insert(distributedArray):**
Inserts a distributed array into an output d/stream's buffer - may be performed multiple times before writing.

**write():**
Writes the buffer to the file, using parallel I/O if possible.

**read()** and **unsortedRead()**:
Reads a block of data (which must have been written by a corresponding write()) from the file into the d/stream's buffer.

**extract(distributedArray):**
Extracts a distributed array from a d/stream's buffer.

**close():**
Closes the d/stream file.




- Each **extract**ed array must have the same size and element type as the corresponding array that was **insert**ed.

D/streams are intended to support *interleaving* [23], in which data from corresponding elements of separate arrays can be written contiguously in the file even if the corresponding elements are not contiguous in memory. The intended implementation of interleaving is one where arrays **insert**ed into an output d/stream consecutively, with no intervening **write**, will have their elements interleaved in the file. To support this, we require that if more than one array is **insert**ed before a **write**, then those arrays must have the same size and number of dimensions.

Both **read** and **unsortedRead** transfer a block of data (written by a corresponding **write**) from the file into the d/stream's buffer, to be **extract**ed into distributed arrays later. When **read** is used, then elements of the extracted arrays will be in exactly the same order as the elements of the originally **insert**ed arrays. This may require interprocessor communication by the d/stream implementation on distributed-memory parallel machines with Paragon-style parallel I/O systems.

**UnsortedRead** is intended to be used to read array data in which the element indices perform no important role in the computation. When **unsortedRead** is used, no guarantee is made about the order in which the element data is extracted into elements of the receiving array, so the interprocessor communication can be avoided, resulting in higher performance.

# 4 pC++/streams: A Library Implementing d/streams in an Object-Parallel Language

*pC++* [2] is a portable object-parallel programming language for both shared-memory and distributed-memory parallel systems. Traditional data-parallel systems are defined in terms of the parallel action of primitive operators on distributed arrays. *Object-parallelism* extends that model to the object-oriented domain by allowing the concurrent application of arbitrary functions to the elements of more complex distributed data structures. This allows the construction of parallel applications having complex dynamic distributed data structures, within an object-oriented framework. pC++ is based on a simple extension to C++ that provides parallelism via the *collection* construct. A collection is a distributed array of objects with additional infrastructure supporting the implementation of arbitrary distributed data structures (e.g. distributed trees of objects) over the distributed array base. pC++ provides facilities for specifying HPF-style distribution and alignment of collections.

The I/O library *pC++/streams* is a portable implementation of d/streams supporting parallel I/O on pC++ collections. **Figure 3** gives a simple example of how d/streams are used in pC++, and **Figure 4** sketches the internal structure of the implementation of pC++/streams for distributed-memory parallel machines having parallel file systems, such as the Intel Paragon and Thinking Machines CM-5. The implementation for shared-memory multiprocessors is somewhat simpler; depending on the capabilities of the underlying file system, the "per-node" d/stream buffers can be reduced to one or eliminated.

## 4.1 pC++/streams Implementation

### Implementation of *open* and *close*

The pC++/streams library implements d/streams as collections having the same alignment and number of elements as the collection(s) on which I/O is to be performed. Using the pC++/streams library, the programmer sets up a d/stream and invokes the **open** primitive by declaring a d/stream object. An output d/stream "`s`" is declared as follows:

**Figure 3. A SIMPLE pC++ D/STREAMS EXAMPLE: pC++ supports *collections*, complex data structures based on distributed arrays of arbitrary objects. We have implemented d/streams for pC++ to support I/O on collections. The two pC++ programs below demonstrate how d/streams can be used in pC++ to output and then later input a distributed grid of objects which hold lists of particles. The declarations to the right are included in both programs below.**

Declarations:

```
class Position {
  double x, y, z;
};

class ParticleList {          // the element class
  int        numberOfParticles;
  double *   mass;       // variable sized
  Position * position;   // arrays
};

Collection DistributedParticleGrid {
  updateParticles();  // could be used to move the
};                    // particles over the grid
```

Output Program:

```
#include "declarations.h"
Processor_Main {
  Processors      P;
  Distribution    d(12, &P, CYCLIC);
  Align           a(12,"[ALIGN(dummy[i], d[i])]");

 // defining a distributed grid of ParticleLists g
  DistributedParticleGrid <ParticleList> g(&d,&a);

  // defining an output d/stream s:
  oStream s(&d, &a, "wholeGridFile");

  // to insert the entire collection g:
  s << g;
  // to insert only the numberOfParticles field
  //   from each element
  s << g.numberOfParticles;

  s.write();
}
```

Input Program:

```
#include "declarations.h"
Processor_Main {
  Processors      P;
  Distribution    d(12, &P, CYCLIC);
  Align           a(12,"[ALIGN(dummy[i], d[i])]");

  // defining a distributed grid of ParticleLists g
  DistributedParticleGrid <ParticleList> g(&d,&a);

  // defining an input d/stream s:
  iStream s(&d, &a, "wholeGridFile");

  s.read();

  // extracting the entire collection g:
  s >> g;
  // extracting only the numberOfParticles field
  //   into each element
  s >> g.numberOfParticles;
}
```

```
oStream s(&distribution, &alignment,
                          "filename");
```

where `distribution` and `alignment` are objects which specify the distribution and alignment of the collection(s) to be output, and `filename` is the name of the file in which the data to be output is to be stored. An input d/stream is declared the same way, except "`iStream`" is substituted for "`oStream`". Multiple d/streams may be set up and connected to the same file if collections with differing distributions and alignments are to be output. The d/stream **close** primitive is implemented in the destructor for `istream` and `ostream`, so **close** is automatically called when a program block in which a d/stream is declared is exited.

## Implementation of *insert* and *extract*

pC++ allows a particularly elegant programmer interface for the d/stream **insert** and **extract** primitives. In pC++, parallel operations on collections can be expressed using a data-parallel style syntax: if `a` and `b` are collections and if `*` is an operator on the element types of `a` and `b`, then `a * b` specifies the concurrent application of `*` to the corresponding elements of `a` and `b`. As in C++, any binary operator can be overloaded by the programmer to implement any binary function.

The pC++/streams library implements the d/stream **insert** primitive by overloading the operator <<. This lets the programmer insert an entire collection `g` into a d/stream `s` in parallel with a single line of code:

```
s << g;
```

which concurrently applies the operator << to the corresponding elements of `s` and `g`. This syntax is similar to that used for formatted ASCII I/O in C++, implemented by the C++ iostream library [3].

pC++/streams defines the << insertion operator for each of the fundamental pC++ types (such as integers and doubles) and for arrays of the fundamental types. These operators insert pointers to the data to be output into the per-element pointer lists, as described in **Figure 4**. Additionally, pC++/streams provides a straightforward means for the programmer to indicate the way in which complex programmer-defined types are to be inserted. The programmer can specify *insertion functions* to decompose the insertion of any complex type in terms of simpler insertions of the fields of that type. For example, for the type `ParticleList` described in Figures 3 and 4, which contains the dynamic arrays `mass` and `position`, the programmer could define an insertion function as follows:

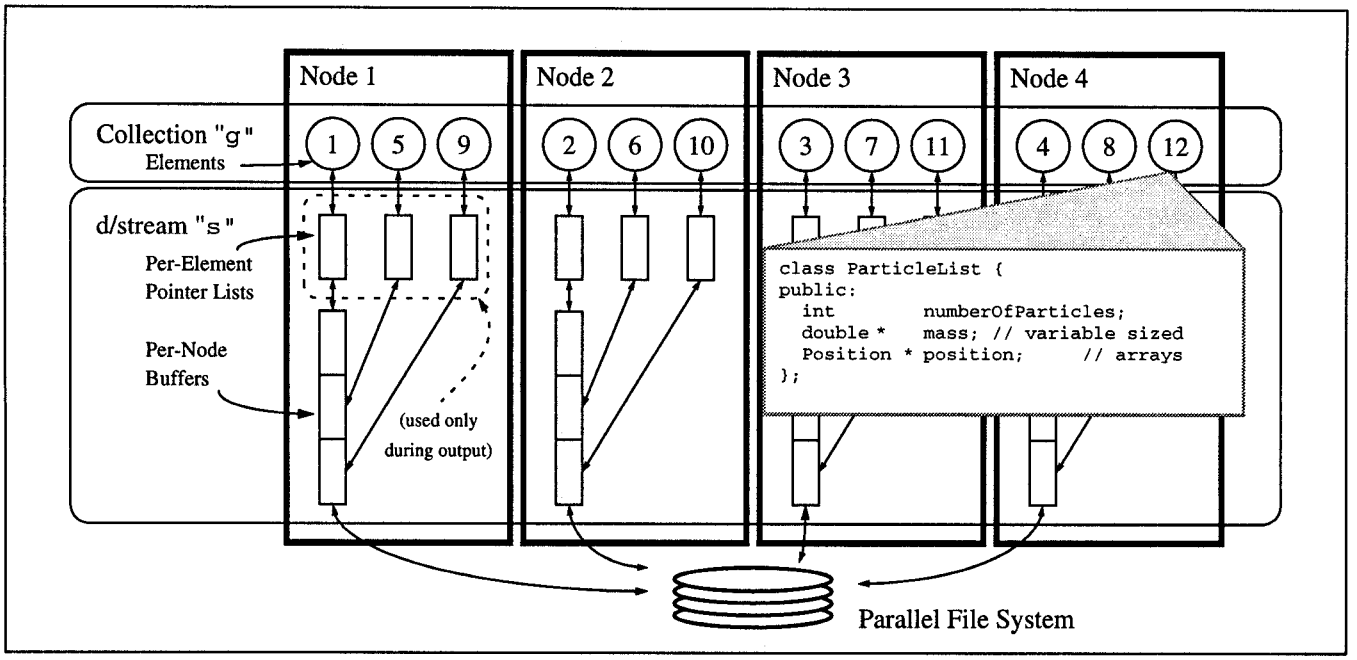



**Figure 4. D/STREAM IMPLEMENTATION IN pC++: This diagram shows the internal structure of the pC++ d/stream implementation used for distributed-memory parallel machines with parallel file systems, such as the Intel Paragon and Thinking Machines CM-5. The diagram also depicts `g`, an example collection containing a one-dimensional distributed array of 12 `ParticleList` objects. pC++ allows an elegant programmer interface to the d/stream primitives. The programmer can *insert* the entire collection `g` into the d/stream `s` with a single line of code: `s << g`. Any subfield of the elements of collection `g` can be inserted in a single line as well, for example: `s << g.numberOfParticles`. Invocation of the *insert* primitive causes a pointer to the inserted data to be added to each of the per-element pointer lists in the d/stream. Invocation of the *write* primitive with `s.write()` causes the pointer lists to be traversed and the corresponding data from `g` to be output to the underlying (system-provided) parallel file system, as described in Section 4.1. Invocation of the *read* primitive with `s.read()` inputs data from the parallel file system into the per-node buffers, then invocation of the *extract* primitive (using `s >> g` or `s >> g.numberOfParticles`, for example) causes that data to be transferred into collection `g`.**

```
declareStreamInserter(ParticleList &p) {
  // Insert the numberOfParticles field
  // of p (an integer):
  s << p.numberOfParticles;

  // Insert the mass field,
  // a variable-sized array
  // of size numberOfParticles:
  s << array(p.mass, p.numberOfParticles);

  // Similarly, insert the position field
  s << array(p.position,
             p.numberOfParticles);
}
```

(`declareStreamInserter` and `array` are macros defined by the pC++/streams library.) This insertion function would have to be defined before the programmer could insert a collection of ParticleLists (with `s << g` for example). Recursively structured data types such as trees can be output naturally using recursive insertion functions.

In addition to inserting entire collections in a single line, pC++/streams allows the programmer to insert any single field of the elements of a collection in a single line. For example:

```
s << g.numberOfParticles;
```

Assume `g2` is a second collection aligned with `g` and containing a double precision field `particleDensity`. The programmer can invoke

```
s << g.numberOfParticles;
s << g2.particleDensity;
s.write();
```

which will cause the corresponding `numberOfParticles` and `particleDensity` fields of `g` and `g2` to be written contiguously in the file, even if they are not contiguous in memory. This feature, called *interleaving*, is useful for writing files for communication with many visualization tools which require related data to be written contiguously.

The d/stream **extract** primitive is implemented similarly to the **insert** primitive. The programmer can extract data from an input d/stream into an entire collection using `s >> g`, or into a single field of a collection using `s >> g.numberOfElements`. As with insertion, the programmer can define extraction functions to define how complex types are to be extracted.

### Implementation of *write*, *read*, and *unsortedRead*

pC++/streams allows the programmer to invoke the **write** primitive on an output stream `s` by calling `s.write()`, which initiates the following two output steps:

1) **Writing distribution and size information**: Information about the distribution of the collection `s` (and thus the distribution of all collections that could have inserted data into `s`) and about the size of the data to be output from each element needs to be written to the file prior to the actual data, so that the implementation of **read** will know how much data is to be read on input and where the data belongs. For collections having a large number of elements (and thus a large amount of distribution and size information), this information can be written from all the nodes concurrently using a parallel write operation. However, for collections having a small number of elements, the latency involved in this parallel write may be greater than the time that would be required to communicate the information to node zero. In this case, instead of writing the distribution and size information separately, it should be collected into node zero and placed at the head of the "per-node" buffer on that node so that it can be written with the actual data in step 2 below.

2) **Writing the actual data**: Next the per-element pointer lists are traversed, the data referenced in those lists (data the programmer previously inserted) is copied into the per-node buffers, and the per-node buffers are written in a single parallel write operation, using the parallel I/O primitives of the underlying parallel file system. On the Intel Paragon and Thinking Machines CM-5, we use parallel I/O primitives which transfer a contiguous block of data from each compute node to the file system simultaneously and write those blocks to the file in node order.

The programmer invokes the d/stream **read** primitive on an input stream `s` by calling `s.read()`, and the **unsortedRead** primitive by calling `s.unsortedRead()`. When either primitive is invoked, the input stream reads the distribution and size information (which appear ahead of the actual data in the file) using one parallel read operation, then reads the actual data into the per-node buffers using a second parallel read. If **unsortedRead** was invoked, then the actual data is ready to be extracted directly from the per-node buffers by the programmer. If **read** was invoked, then the actual data may need to first be sorted and sent to the owner nodes by the library if the number of processors or distribution has changed since the data was written.

Note that no information about the distribution or size of the data to be read needs to be passed to the library by the programmer when reading, since that information is stored directly in the file, preceding the data itself. The programmer simply invokes `s.read()` and the library does the paperwork involved in determining the structure of the data that was written, reading it in correctly regardless of differences in the number of processors and distribution of the reading and writing arrays.

## 4.2 Compiler Support

In Section 4.1 we mention that the programmer can write inserter and extractor operators to specify exactly how programmer-defined types are to be inserted and extracted. We have developed a simple tool (*stream-gen*) that analyzes pC++ programs and generates the inserter and extractor operators for all programmer-defined types. The library can be used without the tool but the tool makes the programmer's job easier. In inserters and extractors for dynamic types containing pointers *stream-gen* generates comment statements allowing the programmer to specify exactly how the pointers should be handled. *Stream-gen* was written using the Sage++ compiler toolkit [8].

Additionally, I/O on local data that is replicated on all nodes of a distributed-memory machine is supported in pC++ through a facility similar to the C standard I/O library. The pC++ compiler automatically transforms programs to insure that local data is output and input by only one node. For input, the data is broadcast to the rest of the nodes after it is read. PetSc/Chameleon [11] provides similar functionality.

## 4.3 Performance

### The Benchmark

We developed a simple benchmark that contains the I/O skeleton from a Grand Challenge Computational Cosmology application written in pC++, the Self Consistent Field (SCF) code [12] [9]. SCF is an N-body code in which the primary data structure is a one dimensional collection of `Segments` where each segment stores data corresponding to several `particles`. There could be several segments on a given processor. Per-particle information includes the x, y, and z coordinates of the particles, their x, y, and z velocities, and their masses.

In the SCF code particle data is periodically saved for later analysis (for visualization of how the particles interact as well as for comparing the results to the sequential algorithm). The SCF code is mostly an "output only" application, but in our benchmark we perform both input and output on the particle data. We coded the I/O and measured its performance in 2 ways: using the pC++/streams library, and using operating system I/O primitives directly with no buffering. Application developers often use unbuffered I/O to avoid the extra code required for buffering, and this can lead to less than optimal I/O performance, since buffering reduces total I/O latency time.

In addition, we measured the overhead of the automatic bookkeeping of distribution and element size information supported by pC++/streams. pC++/streams stores size and distribution information for each element in the file. In applications where element sizes do not vary or where element sizes can be computed, a programmer using manual buffering with operating system primitives might not store as much per-element information in the file as pC+/streams. Therefore we also present pC++/streams' performance as a percentage of the performance that could be attained for such applications by using manual buffering, storing no element size or distribution information in the file.

### Performance results

**Figure 5** shows performance results for implementations of pC++/streams on the Intel Paragon and the SGI Challenge. The library also runs on the CM-5[1] and a number of workstation platforms. The **unsortedRead** d/streams primitive was used for input in these measurements.

The results indicate that pC++/streams performance is competitive with manually buffered I/O. The results show that the overhead introduced by the library decreases as the I/O size

[1]On the CM-5 wall clock time has to be used for measuring I/O performance since the CMMD timers do not account for I/O [16].

**Table 1: Benchmark Results on Intel Paragon (4 processors) (in seconds)**

| I/O Size (# of Segments) | 1.4 MB (256) | 2.8 MB (512) | 5.6MB (1000) | 11.2MB (2000) |
|---|---|---|---|---|
| Unbuffered I/O | 7.13 sec. | 14.73 | 283.00 | 556.78 |
| Manual Buffering | 2.14 | 3.04 | 5.42 | 54.17 |
| pC++/streams | 2.47 | 3.31 | 5.71 | 55.00 |
| % of Manual Buf. | 86.7% | 91.9% | 95.0% | 98.5% |

**Table 2: Benchmark Results on Intel Paragon (8 processors)**

| I/O Size (# of Segments) | 1.4 MB (256) | 2.8 MB (512) | 5.6 MB (1000) | 11.2 MB (2000) |
|---|---|---|---|---|
| Unbuffered I/O | 7.53 sec. | 14.47 | 273.77 | 561.72 |
| Manual Buffering | 2.91 | 3.75 | 5.72 | 9.69 |
| pC++/streams | 3.36 | 4.20 | 6.16 | 10.19 |
| % of Manual Buf. | 86.5% | 89.3% | 93% | 95.1% |

**Table 3: Benchmark Results on Uniprocessor SGI Challenge (preliminary)**

| I/O Size (# of Segments) | 5.6 MB (1000) | 11.2 MB (2000) | 112MB (20000) |
|---|---|---|---|
| Unbuffered I/O | 1.68 sec. | 3.42 | 32.20 |
| Manual Buffering | 1.05 | 2.13 | 20.9 |
| pC++/streams | 1.32 | 2.71 | 21.84 |
| % of Manual Buf. | 79% | 78% | 95% |

**Table 4: Benchmark Results on Multiprocessor SGI Challenge (8 processors) (preliminary)**

| I/O Size (# of Segments) | 5.6 MB (1000) | 11.2 MB (2000) | 44.8 MB (8000) |
|---|---|---|---|
| Unbuffered I/O | 0.55 sec. | 1.10 | 4.95 |
| Manual Buffering | 0.22 | 0.34 | 2.38 |
| pC++/streams | 0.39 | 0.75 | 2.65 |
| % of Manual Buf. | 56% | 45% | 89% |

**Figure 5. pC++/STREAMS PERFORMANCE: The timings above are in seconds and the measurements are for an output operation followed by an input operation on a distributed data structure. The total number of elements in the data structure (i.e., the number of *segments*) is varied to determine pC++/streams' performance for various I/O sizes. The d/streams unsortedRead primitive is used for input.**

**The final row in every table gives pC++/streams' performance as a percentage of the performance obtained using manually buffered I/O (indicating the overhead of pC++/streams' automatic bookkeeping of element size and distribution information). These results indicate that pC++/streams performance is competitive with manually buffered I/O. As expected, the buffered I/O supported by pC++/streams outperforms unbuffered I/O.**

increases - that is the performance of the library scales well with problem size. As expected, the buffered I/O supported by pC++/streams outperforms unbuffered I/O.

These results show that d/streams, an abstraction for high-level I/O on distributed data structures, can be implemented without substantial performance penalties in a portable fashion.

## 5 Related Work

The libraries PetSc/Chameleon [11] from Argonne and Panda [23] [22] from UIUC, and the software system PASSION [6] from Syracuse, support I/O on distributed arrays of fixed-sized elements in the context of the distributed-memory data-parallel programming model. pC++/streams differs from these systems in that it supports buffered I/O on distributed arrays of objects whose size may vary over the array itself, for example distributed arrays of variable-density grids or distributed arrays of lists of particles, in the context of a portable object-parallel language. PetSc/Chameleon supports I/O on block-distributed arrays. Panda supports more general HPF-style array distributions and interleaving, as does pC++/streams. PASSION is a large effort at Syracuse which provides support at the language, compiler, runtime, and file system level for I/O on distributed arrays as well as for computing over out-of-core distributed arrays. It should be noted that pC++/streams is not intended to be used for out-of-core computation. The two-phase access strategy described in PASSION for parallel access to files, where data is first read in a manner conforming to the distribution on disk and then redistributed among the processors, is similar to the design of the pC++/streams **read** primitive. Language extensions for parallel I/O on distributed arrays have been discussed in the HPF Forum [13].

## 6 Conclusions

This paper makes several contributions:

- It describes *d/streams*, a language-independent abstraction with a small set of simple primitives for buffered I/O on distributed data structures, which can be implemented in I/O libraries.
- It describes the interface and implementation of *pC++/streams*, a library that implements d/streams in the object-parallel language pC++ to provide simple and expressive primitives for I/O on distributed arrays of arbitrary variable-sized objects.
- It presents performance results which show that d/streams can be implemented efficiently and portably on a range of distributed-memory and shared-memory parallel machines.
- It shows that compiler support can be used to ease the coding of I/O.

- It provides a good picture of the landscape of mechanisms for I/O on distributed data structures.

pC++/streams is intended for developers of parallel programs requiring efficient high-level I/O abstractions for checkpointing, scientific visualization, and debugging. It uses parallel I/O primitives on machines that provide them, to implement the abstractions efficiently.

## Acknowledgements

This research is supported in part by ARPA under contract AF 30602-92-C-0135, and the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616. We would also like to acknowledge the anonymous referees for their comments, which helped us improve the paper.

## References

[1] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, Oct 1994. Available on the WWW at `http://www.cs.umd.edu/projects/hpsl/io/io.html`.

[2] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[3] B.Stroustroup. *The C++ programming language*. Addison Wesley, 1986.

[4] Michael Carey, David Dewitt, et al. Shoring up persistent applications. In *Proceedings of ACM-SIGMOD Conference on the Management of Data 1994*, May 1994. Available on the World Wide Web at `http://www.cs.wisc.edu/p/shore/html/shore.home.html`

[5] Alok Choudhary. Parallel I/O systems: Guest editor's introduction. *Journal of Parallel and Distributed Computing*, 17:1--3, 1993.

[6] Alok Choudhary et al. PASSION: Parallel And Scalable Software for I/O. Technical Report NPAC SCCS-636, Syracuse University, September 1994.

[7] Peter Corbett and Dror Feitelson. Design and implementation of the vesta parallel file system. In *Proceedings of Scalable High Performance Computing Conference*, SHPCC94, May 1994.

[8] D.Gannon, P.Beckman, F.Bodin, J.Gotwals, S.Narayana, S.Srinivas, and B.Winnika. Sage++: An object oriented toolkit for program transformations. In *Proceedings of Oonski 94*, April 1994. Available on WWW at `http://www.extreme.indiana.edu/sage/docs.html`

[9] D.Gannon, S.Yang, S.Srinivas, V.Menkov, and P.Bode. Object-oriented methods for parallel execution of astrophysics simulations. In *Proceedings of Mardigras94*, February 1994. Available from gannon@cs.indiana.edu.

[10] D.Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, 1:25--42, 1993. The HPF Forum archive for HPF related material is anonymous ftp from `titan.cs.rice.edu:pub/HPFF`.

[11] N. Galbreath, W. Gropp, and D. Levine. Application driven parallel I/O. In *Proceedings of Supercomputing 93*, November 1993. Also Argonne technical report MCS-P381-0893.

[12] L. Hernquist and J. P. Ostriker. A self-consistent field method for galactic dynamics. *The Astrophysical Journal*, 386:375--397, 1992.

[13] High Performance Fortran Forum. On the WWW at `http://www.erc.msstate.edu/hpff/home.html`.

[14] Intel Supercomputing System Division. *Paragon Users Guide*. Chapter 5 and Chapter 8 available online from `http://www.ssd.intel.com/`.

[15] John Karpovich, Andrew Grimshaw, and James French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of OOPSLA 94, 9th conference on object-oriented programming systems, languages, and applications*, October 1994. Available on the WWW at `http://uvacs.cs.virginia.edu`.

[16] Thomas Kwan and Daniel Reed. Performance of the CM-5 scalable file system. In *Proceedings of International Conference on Supercomputing 94*, July 1994. Also accessible on the WWW at `http://bugle.cs.uiuc.edu/`.

[17] Lamb et al. The ObjectStore database system. *Communications of the ACM*, October 1991.

[18] MPI-IO: A parallel file I/O for MPI. Available on WWW at `http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html`.

[19] Portable Parallel File System project. Available at `http://www-picasso.cs.uiuc.edu/Projects/PPFS/`.

[20] Juan Rosario and Alok Choudhary. High-performance I/O for massively parallel computers: Problems and Prospects. *IEEE Computer*, pages 59--68, March 1994.

[21] Documents from the scalable I/O initiative. Available on the World Wide Web at `http://ccsf.caltech.edu/SIO/SIO.html`.

[22] Kent Seamons and Marianne Winslett. An efficient abstract interface for multidimensional I/O. In *Proceedings of Supercomputing 1994*, November 1994. Accessible on the WWW at `http://www.computer.org/p3/sc94home.html`.

[23] Kent Seamons and Marianne Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, September 1994. Accessible on the WWW at `http://bunny.cs.uiuc.edu/CADR/winslett/arrays.html`.

[24] The parallel i/o archive. Available on the World Wide Web at `http://www.cs.dartmouth.edu/pario.html`.

[25] Thinking Machines Corporation. *CMMD Reference Manual*. Chapter 12.