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

In this paper we describe *d/streams*, a language-independent abstraction with a small set of simple primitives for buffered I/O on distributed data-structures. We describe 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. We present performance results on the Intel Paragon and SGI Challenge which show that d/streams can be implemented efficiently and portably. pC++/streams is intended for developers of parallel programs requiring efficient high-level I/O abstractions for checkpointing, scientific visualization, and debugging.

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 which we address in this paper.

We first identify parallel programming tasks for which the use of high-level I/O primitives is appropriate. Next

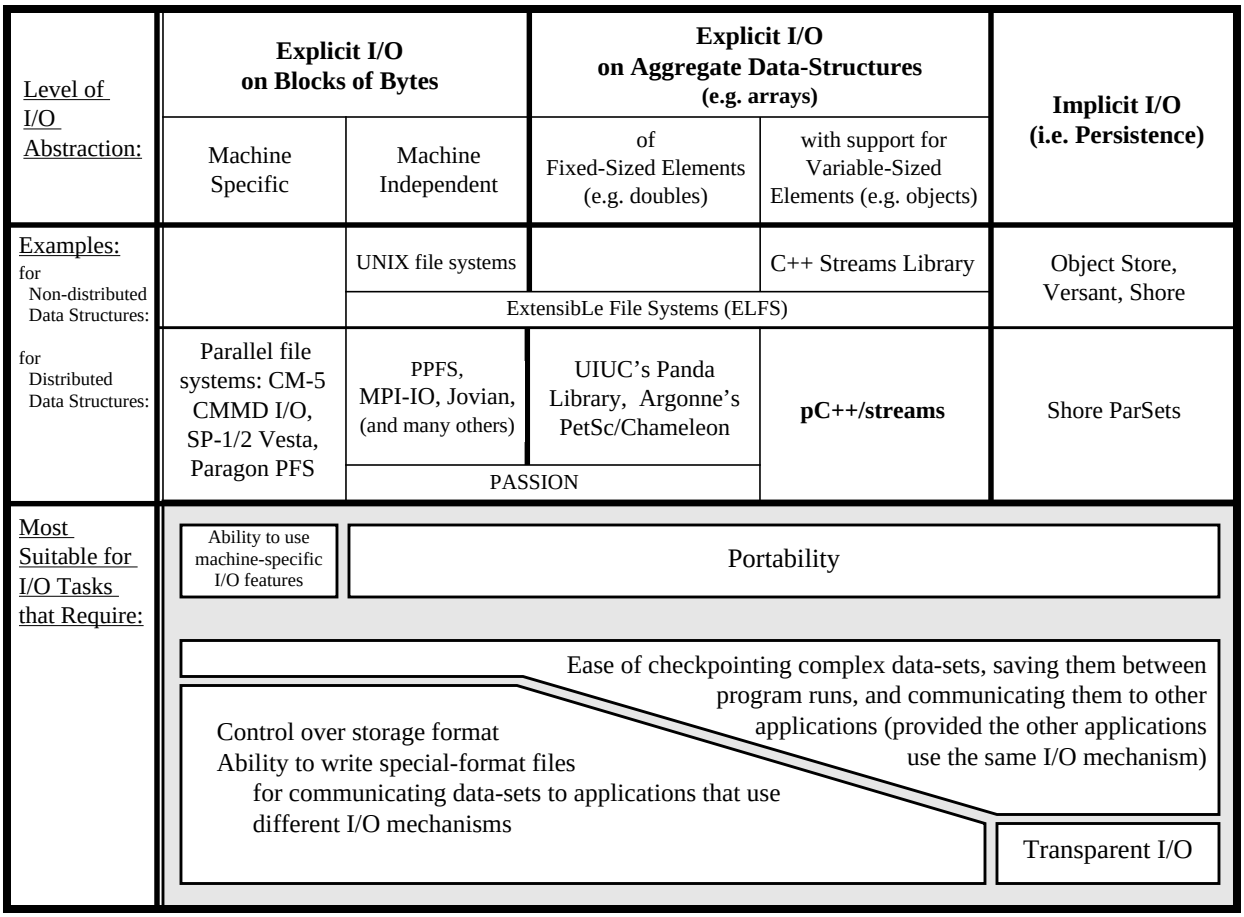


Figure 1. A RANGE OF I/O MECHANISMS: Those mechanisms offering a higher degree of control over low-level I/O details are grouped toward the left; those offering greater ease of use and supporting I/O primitives operating at a higher level of abstraction are grouped toward the right. I/O libraries such 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.)

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++.

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 such as 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.

Persistence is another I/O mechanism that can prove useful for such tasks. In addition, persistence can serve as an interface to object-oriented databases. However, explicit I/O on complex distributed data-structures is a more appropriate mechanism when a higher degree of control over the I/O process and storage format are desired or when the maintainance of a persistent database would entail complexity disproportionate to the size of the I/O task at hand. The integration of persistence with parallel languages, machines, and programming environments is currently an ongoing research problem.

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 great majority of developers of parallel programs 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 containing variable-sized elements. 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**.
- Each **extract**ed array must have the same size, number of dimensions, and element type as the corresponding array that was **insert**ed.

D/streams are intended to support *interleaving* [22], 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

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 of variable-sized elements. 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

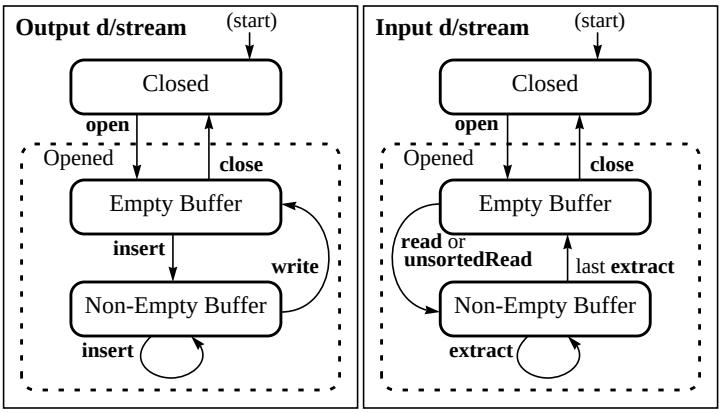
insert(distributedArray):
Inserts a distributed array into an output d/stream's buffer - may be performed multiple times before writing

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 file



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 multicomputers having parallel file systems, such as the Intel Paragon and Thinking Machines CM-5. The implementation for shared-memory multicomputers is somewhat simpler; depending on the capabilities of the system-provided 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:

```
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. A d/stream is **close**d automatically when the program block containing the d/stream is exited.

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
  s << g.numberOfParticles;   // from each element

  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
  s >> g.numberOfParticles;  // into each element
}
```

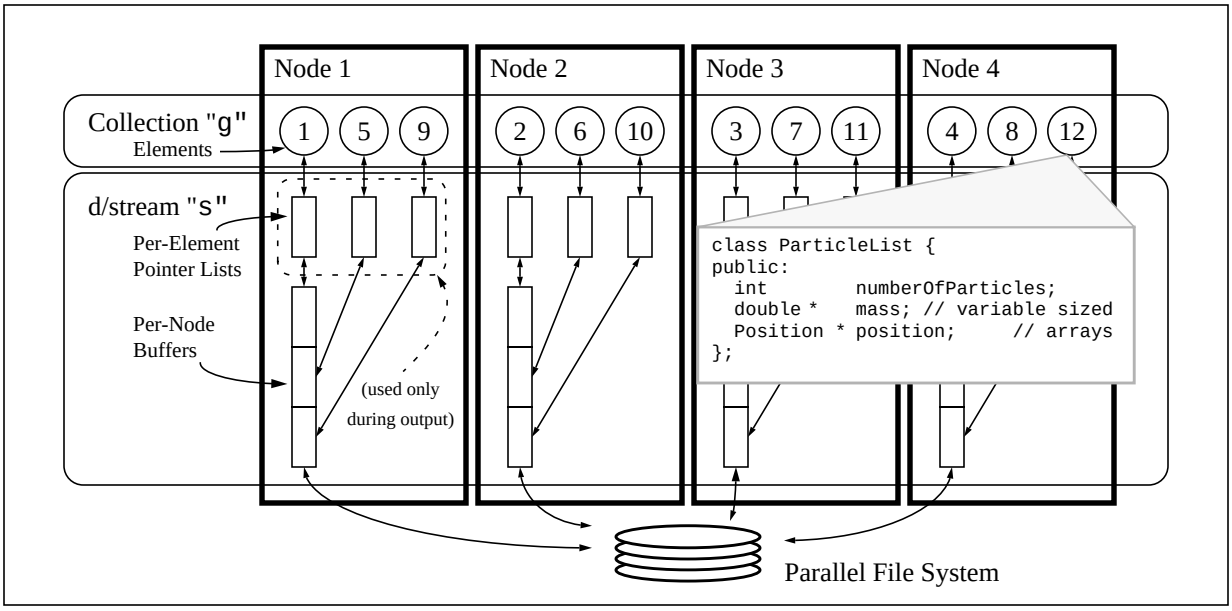


Figure 4. D/STREAM IMPLEMENTATION IN pC++: This diagram shows the internal structure of the pC++ d/stream implementation used for distributed-memory multicomputers with parallel file systems, such as the Intel Paragon and Thinking Machines CM-5. The diagram also depicts `g`, an example collection containing 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 system-provided file system using parallel I/O, as described in Section 4.1. Invocation of the *read* primitive with `s.read()` inputs data from the file system into the per-node buffers, then invoking the *extract* primitive (using `s >> g` or `s >> g.numberOfParticles`, for example) causes that data to be transferred into collection `g`.

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:

```
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**: First, 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 is written to the file in a single parallel write.
2) **Writing the actual data**: Next the actual data is written. The per-element pointer lists are traversed, the data referenced (which the programmer previously inserted) is marshaled into the per-node buffers, and the per-node buffers are written in a single parallel write.

Each parallel write operation described above transfers a single block of data from each compute node to the file system concurrently and writes those blocks to the file in node order, using the I/O primitives of the underlying parallel file system.

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 were stored in the file in step one above in one parallel read operation, then reads the actual data into the per-node buffers in 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 dynamic types containing pointers *stream-gen* generates a comment statement allowing the programmer to specify exactly how the pointer should be handled. *Stream-gen* is written using the Sage++ compiler toolkit [8].

Additionally, the pC++ compiler supports I/O on local data that is replicated on each node, by transforming programs to insure that local data is output and input by only one node. Also for input, the data is broadcast to the rest of the nodes after it is read. This is similar to what is done in PetSc/Chameleon [11], but they do not have compiler support and the programmer has to explicitly distinguish between I/O on non-distributed and distributed data-structures.

4.3 Performance

The Benchmark

We developed a benchmark that contains the I/O skeleton from a Grand Challenge application[1] 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 co-ordinates of the particles, their x, y, and z velocities, and their masses.

In the SCF code particle data is periodically saved for later analysis (to visualize how the particles interact as well as to compare 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 code the I/O in 3 ways: using the pC++/streams library, using operating system I/O primitives directly with buffering, and using operating system I/O primitives without buffering.

[1] "Grand Challenge Computational Cosmology"

Performance results

Figure 5 shows preliminary performance results for implementations of pC++/streams on the Intel Paragon and the SGI Challenge. The library also runs on the CM-5[1] [15] and a number of workstation platforms. We are debugging the library on the KSR-1 and will present performance results of the library running on it in the final paper. We are also in the process of porting the library to the IBM SP-2 using the Vesta file system and the Cray T3D.

Our preliminary performance results indicate that the library is competitive with hand-optimized buffered I/O and out-performs the unbuffered I/O. Application developers often use unbuffered I/O - which was the case in the SCF code - to avoid the extra bookkeeping involved in buffering, and this can lead to I/O performance problems. The results also show that the overhead introduced by the library decreases as the total number of elements increase - that is the performance of the library scales well with problem size.

The results show that abstractions for high-level I/O on distributed data-structures can be implemented without substantial performance penalties, in a portable fashion.

Figure 5. pC++/STREAMS PERFORMANCE: The timings below are in seconds and the measurements are for a collection output operation followed by an input operation. The total number of elements in the collections (i.e., the number of segments) is varied to determine the performance of the library using unsortedRead for various I/O sizes. The I/O sizes (in megabytes) for the various segment sizes are given in parentheses in the first row of each table. Each Segment contains particle list data of size 700 doubles (i.e. 5600 bytes). The final row in every table gives the library's I/O rate as a percentage of the rate obtained with buffered I/O method. This indicates the overhead of storing distribution and size information in the file in the current implementation of pC+/streams.

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

Number of Segments	256 (1.4MB)	512 (2.8MB)	1000 (5.6MB)	2000 (11.2MB)
Library	3.76 sec	3.98	11.07	57.08
Buffered I/O	1.99	3.08	7.00	45.54
Unbuffered I/O	7.13	14.73	283.00	556.78
Library speed	53%	77%	63%	80%

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

Number of Segments	256 (1.4 MB)	512 (2.8 MB)	1000 (5.6MB)	2000 (11.2 MB)
Library	3.77	4.95	8.25	15.77
Buffered I/O	2.95	4.12	7.04	14.48
Unbuffered I/O	7.53	14.47	273.77	561.72
Library speed	78%	83%	85%	91%

Table 3: Benchmark Results on Uniprocessor SGI Challenge

Number of Segments	1000 (5.6 MB)	2000 (11.2 MB)	20000 (112MB)
Library	1.32	2.71	21.84
Buffered I/O	1.05	2.13	20.9
Unbuffered I/O	1.68	3.42	32.20
Library speed	79%	78%	95%

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

Number of Segments	1000 (5.6MB)	2000 (11.2MB)	8000 (44.8MB)
Library	0.39	0.75	2.65
Buffered I/O	0.22	0.34	2.38
Unbuffered I/O	0.55	1.10	4.95
Library speed	56%	45%	89%

[1]On the CM-5 the wall clock time has to be used for measuring I/O since the CMMD timers do not account for I/O and so we do not give detailed performance results since we do not have access to dedicated time on the CM-5 to run our benchmark

5 Related Work

The libraries PetSc/Chameleon [11] from Argonne and Panda [22] [21] from UIUC, and the software system PASSION [6] from Syracuse, provide support for 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 implementation of the pC++/streams **read** primitive. Language extensions for parallel I/O on distributed arrays have been discussed in the HPF Forum [10].

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.