Efficient Data-Parallel Files via Automatic Mode Detection

Jason A. Moore, † Philip J. Hatcher,†† and Michael J. Quinn†

†Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202
(503) 737-5128, FAX: (503) 737-3014
moorej,quinn@cs.orst.edu

††Department of Computer Science
University of New Hampshire
Durham, NH 03824
(603) 862-2678
pjh@cs.unh.edu

Abstract

Parallel languages rarely specify parallel I/O constructs, and existing commercial systems provide the programmer with a low-level I/O interface. We present design principles for integrating I/O into languages and show how these principles are applied to a virtual-processor-oriented language. We show how *machine-independent modes* are used to support both high performance and generality. We describe an automatic mode detection technique that saves the programmer from extra syntax and low-level file system details. We show how virtual processor file operations, typically small by themselves, are combined into efficient large-scale file system calls. Finally, we present a variety of benchmark results detailing design tradeoffs and the performance of various modes.

1 Introduction

Parallel computing offers great potential for speeding up a wide variety of applications. Unfortunately, parallel software has not matured enough to support mainstream developers and users. They will not embrace parallelism until the tedious low-level details of parallel programming are abstracted away from the programmer and high-level languages and environments support the construction of portable, high-performance applications. To this end, many parallel languages have been proposed. Few of these languages have been adopted by applications programmers.

One significant reason parallel languages have not been accepted is their lack of support for parallel I/O, which is critical for real applications. Parallel language designers must incorporate into their languages I/O operations which support high-level I/O integrated with the language's style of computation, which can be implemented with high performance, and which are portable. Although many portable parallel file systems have been proposed in recent years, their interfaces are relatively low-level, and their details require significant programmer effort to master. A more abstract interface, perhaps built on top of one of these systems and utilizing its optimizations, is needed.

In this paper, we describe a high-level and intuitive file interface for virtual-processor-oriented languages. The interface is presented in the context of the SIMD language C* [22]. The C* user's view is of an abstract parallel machine consisting of a front-end scalar processor combined with a back-end collection of virtual processors (VPs). The number of VPs matches the data-parallelism of a given program; each VP maintains its own element of every parallel variable. Parallel operations are programmed from the viewpoint of what each VP does with its data. Using the VPs' view makes parallel programming easier than less abstract methods [10]. We apply the same view to parallel files: a parallel file consists of one stream per VP, and each VP operates on its own stream within a file, as shown in Figure 1. Therefore, a parallel file may contain millions of streams, each under the control of a different VP. We call our implementation of this abstraction Stream* ("Stream-star"). In addition to maintaining consistency of the C* programmer's view, Stream* enhances programmability through its interface, which consists of parallel versions of familiar C file operations. At the VP level, these operations have the same semantics as their sequential counterparts.

The use of C file operations as the Stream* interface provides the programmer with great flexibility. Therefore, our implementation must support general file operations. However, parallel computers are used for speed, and support for general operations must not hamper the performance of frequently occurring, structured operations. Stream* addresses this dichotomy by accessing parallel files in three *modes* which use different techniques for managing VP file data. Two modes support simple, regular I/O operations and have little overhead, while a third mode supports complex, irregular I/O operations.

The selection of modes and the Stream* design were guided by the following principles, which can be applied to languages other than C* as well.

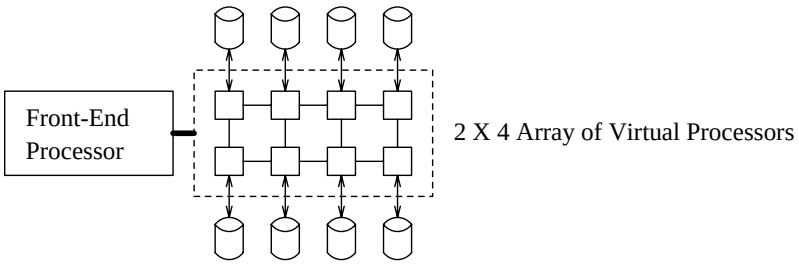

Figure 1: The C* model of computation combined with the data-parallel file abstraction in which each virtual processor in the SIMD machine's back end has its own I/O stream.

- Automatic mode detection by the run-time system maintains a high-level interface—the programmer is not burdened with specification of modes.
- Modes depend only on the program, not the underlying file system, so their management is completely portable.
- Modes change the performance of file operations, not their semantics.
- Common operations are performed using high-performance modes.
- High-performance modes are designed so they can exploit redistribution optimizations such as disk-directed I/O [13] and the two phase access strategy [6], where available.
- Whenever possible, fine-grained VP file operations are combined into large-grained file system operations.
- The system must be able to read and write external files (e.g., from sequential programs or other data-parallel programs).

The remainder of the paper describes how we have applied these principles to the implementation of Stream*. We first discuss related work and follow that with a brief discussion of C*. Next is a general discussion of Stream* modes, followed by detailed descriptions of their implementation. Throughout this section, we illustrate design tradeoffs by presenting results from C* programs compiled and run using Stream*. We finish by describing how Stream* interfaces with external programs and how the C* programmer can, without knowing implementation details, ensure high performance modes are used.

2 Related Work

The first C* file system was built by Thinking Machines. Its interface, similar to that for VP-oriented CM-Fortran on both the CM-200 and CM-5, includes limited functionality in which all VPs transfer data to or from a single stream [23, 24]. Virtual processor streams for C* were proposed by Hatcher [9], whose results with a general implementation are reported in [1]. Moore *et al* [18] point out the shortcomings of the single-stream approach to VP files and suggest the use of high-performance modes for parallel streams. Here we extend this work to include automatic mode detection, design details and tradeoffs, and optimizations.

Most proposed parallel file systems for MIMD machines give the programmer one view of a file, namely as a single stream of bytes. Some systems, including Vesta [3], PIOUS [19] and others [12] support "multifiles", in which a parallel file is broken into multiple subfiles or segments, typically one per physical processor. The CM-200 supports *parallel files*, in which each physical processor accesses its own subfile [23]. The notion of parallel VP streams is a large-scale generalization of this idea, which can simplify I/O programming significantly. Unfortunately, one cannot simply implement VP streams as segments or subfiles on top of these systems. PIOUS would require the opening of many thousands of files, and array-oriented Vesta assumes all subfiles are accessed in the same manner, so it does not store EOF for each subfile. The parallel files of the CM-200 are too closely tied to the physical machine size to manage streams for a larger virtual

machine. Further, if each VP explicitly accesses its own subfile, many fine-grained file operations are required. Performance suffers significantly when this approach is used.

Modes were introduced with the earliest parallel file systems [20, 25], and some more recent systems use modes [2, 11]. The systems provide a limited number of operations whose synchronization requirements and file access semantics vary depending on the current mode. The implementations of modes on these systems have several drawbacks. They are machine-dependent. They give a single file operation multiple meanings; this leads to poor code readability. Finally, modes complicate the parallel programmer's task with low-level details. In contrast, Stream* modes are machine-independent and affect only the performance of file operations, not their semantics within C*.

3 C*

3.1 Programmer's Model

The C* data-parallel language is targeted for a logical SIMD machine consisting of a front-end sequential processor and a back-end processor array. C* programs contain scalar variables and parallel variables. Scalar variables reside on the front-end processor. Parallel variables are associated with a *shape*, whose left indices define the layout of virtual processors along any number of dimensions. The declarations in Figure 2(a) illustrate several language features. The shape `S` is logically laid out as a 256 × 128 array of VPs. Stored at each VP is a single integer `x` and an array `y` containing 10 doubles. The variables `i`, `sum`, and `scalarFile` are scalars. The pointer `parFile` is a scalar which will point to a parallel file variable when it is allocated upon opening of a file.

```
shape [256][128]S;

int:S     x;
double:S  y[10];
FILE:S   *parFile;
int       i;
double    sum;
FILE     *scalarFile;
```

(a)

```
sum = 0;                          /* Scalar code */
for (i = 0; i < 10; i++) {        /* Also scalar */
  with (S) {                      /* Perform in parallel */
    x += y[i];
    where (x > 100) {             /* Only some VPs will do this */
      x = x % 100;
    }
  }
}
```

(b)

Figure 2: (a) Sample C* declarations. (b) Scalar and parallel C* code.

Sequential, or scalar, code is logically executed by the front-end processor. Parallel code, contained in the `with` statement, is executed by the back-end array of VPs. When not all VPs need to work, some can be *masked off* using the `where` statement. The sample code segment in Figure 2(b) demonstrates these features.

Finally, parallel data can be manipulated using two different types of functions. The first of these can be passed parallel arguments and can return a parallel variable. We call such a function a *parallel function*. The second type of function with which parallel data can be manipulated is an *elemental function*. Elemental functions are an important extension to C* formulated by ANSI X3J11.1 [8]. Elemental functions allow VPs to invoke scalar functions by passing individual elements of parallel variables as arguments. A good example of the need for elemental functions is the `sin` function. This C library function cannot receive a parallel variable as an argument. We need not write our own version of `sin`; a header file simply specifies that `sin` is `CS__elemental`. Users can write their own elemental functions, but some operations must be avoided in elemental functions. In particular, elemental functions cannot have side effects; they can read only local VP data and scalars, and they can write only local VP data.

3.2 C* Implementation on MIMD Machines

Although C* provides the programmer with a simple SIMD view of computation, the University of New Hampshire (UNH) C* compiler, with which this work is integrated, has efficient implementations for a variety of MIMD machines. We briefly describe how key SIMD features are modeled on a MIMD machine. For more details see [10, 15]. Scalar variables, and operations on them, are replicated on all compute nodes.

Scalar I/O operations are intercepted so only one compute node physically performs I/O and broadcasts the results to the others. Parallel data are distributed among the compute nodes of a MIMD computer. Parallel code is implemented through the use of *VP emulation*, in which each physical processor performs the operations of the VPs assigned to it. Parallel functions are implemented using a single function call on each compute node, while elemental functions are called once for each VP being emulated by a compute node.

4 File Modes

Parallel file systems ought to be fast. In a production system, though, general operations must be supported with consistency and robustness. Tension exists between the need for speed and generality. Modes are a reasonable solution: high speed is offered by less flexible modes, and more general operations are supported by slower modes.

4.1 Stream* Modes

In our Stream* virtual processor file implementation, we rely on three file modes. Two modes limit the available operations in exchange for speed, while one mode supports more general operations. The restrictions on operations for the high-performance Stream* modes are shown in Table 1. We say an operation meeting the restrictions for a given mode are *compliant* with that mode. All three modes break the file into small (usually less than 64 bytes), fixed-sized VP blocks, from which VP streams are built. The Stream* modes lay out VP blocks in the file in different ways. The high-performance modes, which require that *all* VPs move the same amount of data during an operation, lay VP blocks out in a structured, array-oriented fashion that is independent of the number of compute nodes. That is, assuming N total VPs, VP i's stream consists of the ith block and every Nth block after that. Although a VP block logically has a small size, data are moved in large, contiguous chunks between the regularly laid out file and compute node memory.

The highest performance mode is No Buffering (NB) mode. In this mode, VP-level file operations are implemented in parallel using a single, collective file system operation. This operation moves VP data directly between the file system and the desired parallel variables on the compute nodes with no intermediate buffering. Figure 3(a) shows how the array-oriented layout of data on compute nodes matches the desired layout in the file. Note that, to maintain the regular file layout with no intermediate buffering, *all* NB-compliant operations must transfer the same amount of data.

Collective Buffering (CB) mode supports more general operations such as transfer of strided data, in which the layout of data in compute node memory does not directly match the structure of the file. In this case, a VP-level file operation is implemented by copying data between fixed-sized VP buffers on the compute nodes and the desired parallel variables. The CB buffers themselves are contiguous, so their layout in memory matches the array-oriented layout of data in the file. Because the restrictions on CB guarantee that all VPs read or write the same amount of data during a given operation, all VP buffers become full (when writing) or empty (when reading) at the same time. An example showing the buffers filling with VP data during a write is shown in Figure 3(b)-(d). As with NB, all the data are moved between compute nodes and the file system in a single, collective large-grained operation that can take advantage of optimizations such as disk-directed I/O [13] and the two phase access strategy [6]. The size of the VP buffers on the compute nodes defines the VP block size for the file; this value is not dependent on file operations, while the VP block size for NB is.

It is important to point out that Stream* does not suffer the problems of a general-purpose file system when buffering VP data on compute nodes. A more general file system must immediately perform writes to a parallel file to ensure that other processors see the update. If compute nodes cache prefetched data for reading, they must ensure the data remain consistent with the physical file contents. With Stream*, however, each VP accesses only its own stream, and only the compute node emulating a VP can update its blocks in the file system. Therefore, each compute node is guaranteed to have the most up-to-date information for the VPs it emulates.

Independent Buffering (IB) mode is the most general, supporting any C file operations at the VP level. Because some VPs may be inactive during a file operation, or VPs may move different amounts of data, the collective, shared-offset, array-oriented implementation of NB and CB cannot be used for IB. Although

Collective Buffering (CB) Mode Restrictions:
1. All VPs are active during every operation
2. All VPs transfer the same number of bytes *in a given operation*
3. No elemental file operations are performed
4. All VPs choose the same file offset *in a given seek operation*

No Buffering (NB) Mode Restrictions:
1. All of the above restriction, plus
2. All *operations* transfer b bytes per VP
3. VP data are contiguous (unstrided) in compute node memory
4. All seek offsets are an integral multiple of b

Table 1: VP operations which allow use of Stream* high-performance modes.

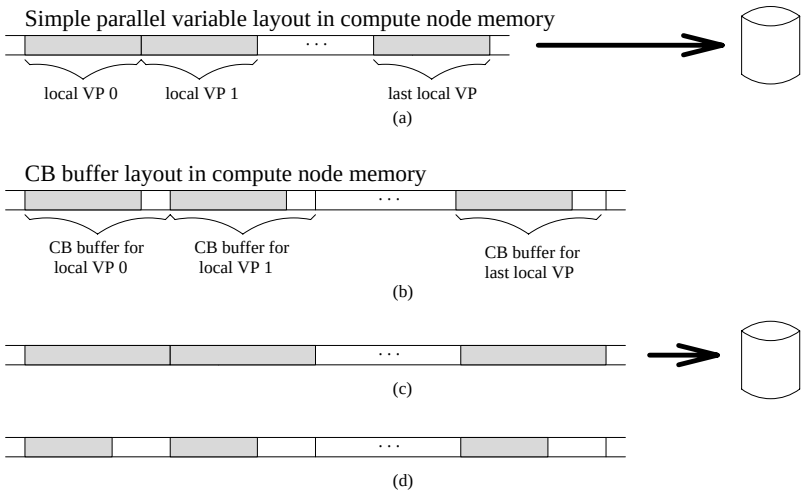

Figure 3: VP-level writes implemented using high-performance modes. (a) In NB, the parallel variable can be moved directly from its location in memory to the file system with a single, collective file system call. Collective Buffering (CB) mode is shown in (b)–(d). In this example, the VPs collectively write two 8-byte values to their own 10-byte CB file buffers. The state of the buffers, each containing 8 bytes after the first write, is shown in (b). The second write takes two steps. The VP buffers are filled with 2 bytes each and written to the file system as a single unit, as shown in (c), then the remaining 6 bytes for each VP is copied into the buffers (d).

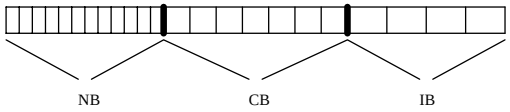

Figure 4: The segmented nature of a Stream* file. The first segment is written using NB mode, and VP blocks of size b_{NB} are regularly laid out in the file. The second segment is written using CB mode. This segment has a structure similar to that of NB; the VP block size b_{CB} may differ from that of NB. The last segment consists of VP blocks of size b_{IB} with an unstructured layout. The IB segment is read with two directory files which describe which blocks make up which VP streams.

the regular layout of data created by NB and CB eases the job of distinguishing individual VP streams, such a layout cannot be efficiently utilized in IB mode. VPs using IB may not fill their buffers at the same time or in any specific order. If, when IB is used, blocks making up VP streams are laid out in regular fashion as in NB and CB, each VP buffer must be written individually using an expensive, fine-grained file operation. An early prototype showed the cost of these operations is prohibitive. Our solution, which combines fine-grained VP file operations into large-grained calls to the file system, writes VP data to the file on a first-come-first-served basis and manages it using a directory. A description of the implementation is in Section 5.

In our first VP file system implementation [18], the user wanting high-performance must specify the desired mode when declaring a parallel file variable. This approach has several drawbacks. First, the programmer must understand implementation details to choose the correct mode. Second, unfamiliar syntax must be added to the language to specify mode hints. Finally, the single mode assigned to a file must be the most general with which it is accessed, even if many of the operations on the file could be performed using less general, higher performance modes. Our current design, which relies on dynamic mode detection, addresses these drawbacks.

4.2 Automatic Mode Detection and File Segments

Stream* dynamic mode detection uses an optimistic scheme that divides a parallel file into three distinct segments. The first segment contains VP data written using NB mode, the second contains data written using CB mode, and the last contains data written using IB mode. We name the segments after the modes in which they were written (NB, CB, and IB). The scheme is optimistic because it assumes the program writing a file will use NB, the highest performance mode available. Recall from Table 1 that NB requires that all operations move the same amount of VP data. The first NB write establishes the VP block size b_{NB} for the NB segment of the file. If a program writes only one data type to the file, as is frequently done in data-parallel applications, the file will consist entirely of an NB segment. When a non-NB-compliant write operation, or an NB-compliant operation relying on a different value of b_{NB} is used, the NB segment of the file is complete. A transition to CB or IB takes place.

The CB segment of the file is written using a VP block size of b_{CB}, whose value is determined by the run-time system, until an operation requiring IB is encountered.[1] The remaining writes, assuming no backwards seeks, are performed using IB to the IB segment of the file, even if subsequent operations are NB or CB-compliant. A scheme could be developed to allow unlimited switching between modes, but the costs of this approach potentially outweigh the benefits. The volume of metadata describing the mode switches and VP activity in every IB segment potentially would be huge. IB's performance relative to the other modes' is good enough that the extra overhead for allowing more file segments is not justified. Finally, files used by most data-parallel applications would remain in NB or CB modes during their lifetimes [5, 7, 14, 16, 17, 21], and Stream*'s optimistic approach matches their needs without becoming overly complex.

5 Implementation

Stream* is implemented as part of the C* run-time library. Machine-independent routines make up a majority of the Stream* implementation, while a few routines make calls to the machine-dependent file system. We

[1] Note that operations allowed in NB mode form a subset of those allowed in CB.

assume the underlying file system manages a parallel file in the traditional manner, as a single stream of bytes. Therefore, Stream* can be implemented on top of PFS [11], Thinking Machines' CMMD I/O [2], and the single stream views of more flexible file systems like PIOUS [19] and Vesta [3]. Knowledge of the underlying implementation of the file system (e.g., programmable I/O nodes, disk arrays, etc.) may be used to optimize the machine-specific portions of the Stream* run-time system, but our discussion is independent of the file system. All of the features of Stream* are implemented on the compute nodes.

For our experiments, we built a simple striped file system using varying numbers of nodes on a Meiko CS-2 multicomputer. Although a parallel file system is available with the CS-2, it is not installed on our machine. Instead, specific nodes are selected as I/O nodes, and they read from and write to their local SCSI disks using Unix file operations. File operations are requested and fulfilled through messages on the CS-2's low-latency, high-bandwidth network. For the experiments shown in subsequent sections, eight compute nodes were used with one to four I/O nodes and a striping unit of 32K bytes.

Associated with each Stream* data file is a second *metafile*, whose name is the data file name with the suffix `.meta`. This file contains information about the file such as b_{NB}, b_{CB}, b_{IB}, mode transition points, distribution information, and shape. Files whose IB segment is not empty also have files with suffixes `.first` and `.dir` to hold directory information. Their function will be described in section 5.2.2. In the remainder of the paper, we refer to the meta files by their suffixes (e.g., the `.meta` file).

5.1 Opening a File

When a parallel file is opened, a parallel `FILE` variable is allocated and a pointer to the parallel variable is returned. A sequential C `FILE` has a corresponding Unix file descriptor, or `fd`, an integer. Our Stream* implementation also associates an `fd` with a `FILE` variable. The `fd` is an index into a structure containing the data needed to manage the parallel file. Our system reserves a fixed number (0-63) of `fd` values for sequential files, and `fd` values higher than that correspond to parallel files. The typical implementation of a `FILE` struct allocates a byte to the `fd` field, so 256 `fd` values are available to represent both sequential and parallel files, although the underlying operating system may not support that many open files simultaneously.

5.2 Writing

Although C provides several ways to write to a stream, our examples below are based on the parallel overloading of the function `fwrite`:

```
int:current fwrite(char:current *buf,int:current size,int:current nitems,FILE:current *fp);
```

Note that the C* keyword `current` matches the current shape, so the `fwrite` function can be used for any shape. Depending on whether or not all VPs are active, and depending on the parameters for an invocation of `fwrite`, this function can comply with any of NB, CB, and IB modes. The mode detection performed at the start of `fwrite` is based on the current mode, file segment, and the characteristics of the parameters. The goal is to use the best mode possible, with IB being selected if NB and CB tests fail. To use **NB or CB** for `fwrite`, the mode detection logic checks the following:

1. NB requires that the current mode be NB. CB allows a current mode of either NB or CB.

2. The file segment to which the write will occur must be compatible with the mode to be used. CB can be used to write to the NB segment of the file using a buffer size of b_{NB} (after a backwards seek, for example), and only in rare instances—the current implementation does not check for these—can NB be used to write to the CB segment of the file.

3. All VPs must be active. Two tests are used to determine VP context. The C* compiler emits code to manage a flag called `CS__everywhere`. The flag is true when VPs are not masked off by a `where` clause. If the flag is false, meaning a `where` clause is in effect, each compute node can directly test whether or not all its VPs are active.

4. `size * nitems` is the same for all VPs.

In addition to the above, the following conditions must hold to use **NB**:

1. `size * nitems` is equal to b_{NB} for all VPs (if this is the first write to the file, b_{NB} is established for subsequent tests).

2. The stride of the parallel data being written is equal to `size * nitems`. That is, the data are contiguous in compute node memory. The UNH C* compiler stores the stride as part of each parallel variable.

Different compute nodes may get different results from the above tests. For example, on exactly one compute node, a VP may have performed a file operation in an elemental function. That compute node will come into the `fwrite` with a current mode of IB. All others will have a current mode of NB. To guarantee that all compute nodes are using the proper mode, a reduction is performed. The processors exchange the calculated mode, number of bytes transferred per VP, and all-active status of VPs with each other. After the reduction, all compute nodes agree on the mode. The reduction operation is cheap relative to file operations, and it can act as the synchronization for a collective file operation, since both NB and CB can take advantage of collective operations.

If NB is the agreed-upon mode, the compute nodes perform an efficient, array-oriented transfer of data directly from the parallel variable to the file system. If CB is chosen, the compute nodes copy from the specified parallel variable to the VP buffers. If the VP buffers become full, an efficient array-oriented transfer, like that for NB, moves data from the contiguous VP buffers to the file system. As shown in Figure 3, the VPs may write more bytes than their buffers can hold. In this case, the buffers are filled to capacity and sent to the file system. This copy-and-write cycle continues until the VP buffers can store the remaining VP data.

5.2.1 Enhancing Interface and Performance

The version of `fwrite` presented above offers general functionality that may be rarely exploited. For instance, the `size` and `nitems` parameters are most often constants, with the `size` often denoted using the `sizeof` operator. With the general `fwrite` prototype, the programmer must cast constants to parallel values:

```
fwrite(&parVar,(int:S)sizeof(double),(int:S)1,parFile);
```

Further, the run-time system must allocate and initialize parallel arguments for the call to `fwrite`. Finally, these arguments must be checked, element-by-element, during mode detection to ensure that all VPs write the same amount of data. To eliminate these frequently unnecessary costs, Stream* provides another overloading of `fwrite` (and, correspondingly, `fread`), in which `size` and `nitems` are scalars:

```
int:current fwrite(char:current *buf,int size,int nitems,FILE:current *parFile);
```

With this version, no parallel arguments must be built, and checking for size consistency among VPs is unnecessary. We expect this to be the normal usage of `fwrite`, so this is the version we use when comparing performance in subsequent experiments.

Because NB is performed using the fastest file operations, it is the standard against which other modes are measured. Figure 5 shows that the differences between NB and CB modes, despite the extra buffering required by CB, is negligible when repeatedly writing a simple parallel `double`. The results shown in Figure 5 are with a CB block size of 8 bytes. A block size of 32 bytes takes more compute node memory and may result in inefficient cache usage, but the larger buffers result in fewer file operations. Our experiments show no statistically significant difference between CB performance with 8 and 32-byte buffers, so the current default b_{CB} is a space-saving 8 bytes. The mode detection cost of Stream*, that is, the bandwidth difference between Stream* with mode detection and without, is also negligible on all I/O node configurations. Statistically, NB bandwidth without mode detection is greater than with mode detection with only 40% confidence.

5.2.2 Writing in IB Mode

If IB is the selected mode, a VP emulation loop performs the write for each active VP by moving data from the parallel variable being written to the VP's IB buffer. VPs may fill their compute-node buffers at different times, which is why they are managed individually, but our goal is to avoid fine-grained writes of individual VP blocks. Figure 6 shows the steps taken when a VP fills its buffer. The contents are moved to a *superblock*, a collection whose size would typically be the size of a striping unit in the underlying file

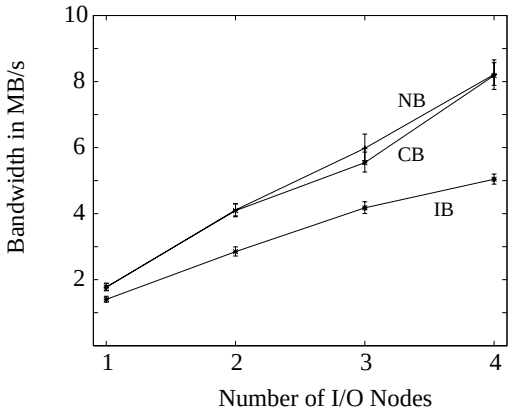

Figure 5: Comparison of NB, CB, and IB modes on the Meiko CS-2. 64K VPs each output 64 `double` values for a total of 32MB. 95% confidence intervals are shown.

	IB Block Size				
	64 bytes	32 bytes		16 bytes	
# I/O Nodes	Bandwidth	Bandwidth	% Relative to 64	Bandwidth	% Relative to 64
1	1.37	1.17	85	0.85	62
2	2.89	2.36	81	1.87	65
3	4.14	3.44	83	2.55	62
4	4.96	4.47	90	3.59	72

Table 2: Comparison of achieved I/O bandwidth when writing in IB mode with different IB block sizes. All bandwidth values are distinct at the 95% confidence level.

system. When a superblock is filled, the data contained in it are sent to the file system as a single chunk. VP directory information is also managed using the notion of superblocks. By combining VP file writes into superblocks, our implementation avoids many fine-grained writes of VP data, which would significantly degrade performance. To support IB superblock writes without processor synchronization (recall that the superblock may be filled from an elemental function, in which no synchronizations may occur), each compute node must know where in the IB segment of the file to write its data. We allocate superblocks in the data file and directory file to compute nodes in round-robin fashion; the pth processor among P total processors writes to the pth superblock, followed by every Pth superblock. This does mean that the structure of the data in the file changes with the number of physical processors, but alternative file layouts relying on virtual processor instead of physical processor configuration require expensive fine-grained file system operations to write VP blocks.

Figure 5 shows that the bandwidth achieved by IB is approximately 60% of NB's bandwidth. Note that the times shown include closing the file in IB, an operation with considerable overhead discussed in Section 5.2.4. With a faster file system, the time for the extra work done on compute nodes limits the achievable bandwidth. This explains the flattening of the IB curve as the number of I/O nodes increases.

Several variables play a part in IB performance. The first of these is IB block size. A large block size requires more compute node memory, but the buffer management overhead becomes a smaller percentage of work done on compute nodes. In the benchmark shown in Figure 5, the block size is 64. Table 2 shows the relative performance with smaller block sizes. Reducing the block size to 16 or 32 has an impact on bandwidth, but the cost of these block sizes is not prohibitive. Our tests with a block size of 8 bytes show that little work is done moving data relative to managing it, and the cost becomes prohibitive at this point. Another variable is the number of data block pointers in a single directory entry. The benchmarks shown were run with only two pointers per directory entry. By increasing that number to six[2], the bandwidth increases on all I/O node configurations by approximately 7%. Increasing the number to fourteen provides negligible additional bandwidth.

[2] With 2 other values in a directory entry, these block pointer counts (2, 6, and 14) ensure a power-of-2 total size in bytes.

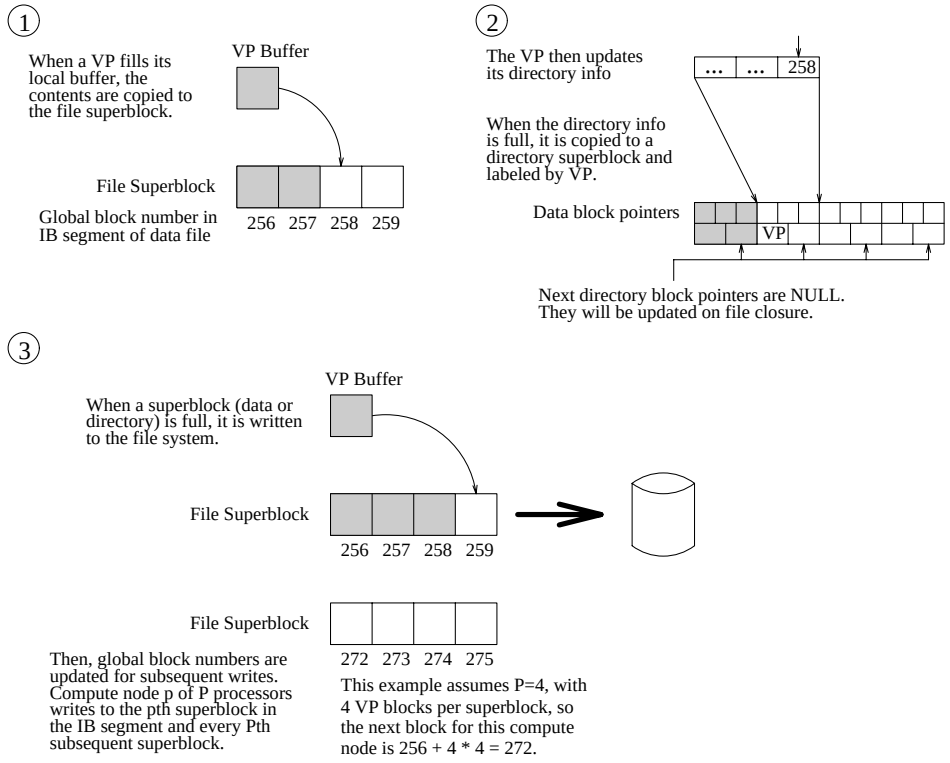

Figure 6: Using superblocks for VP writes in IB mode. Each VP has its own active file buffer. When it is filled, the buffer is copied to the superblock. When the superblock is filled, it is written as a unit to the file system. The VP keeps track of which data blocks its data has been written to, and this directory information is written to the directory file using superblocks.

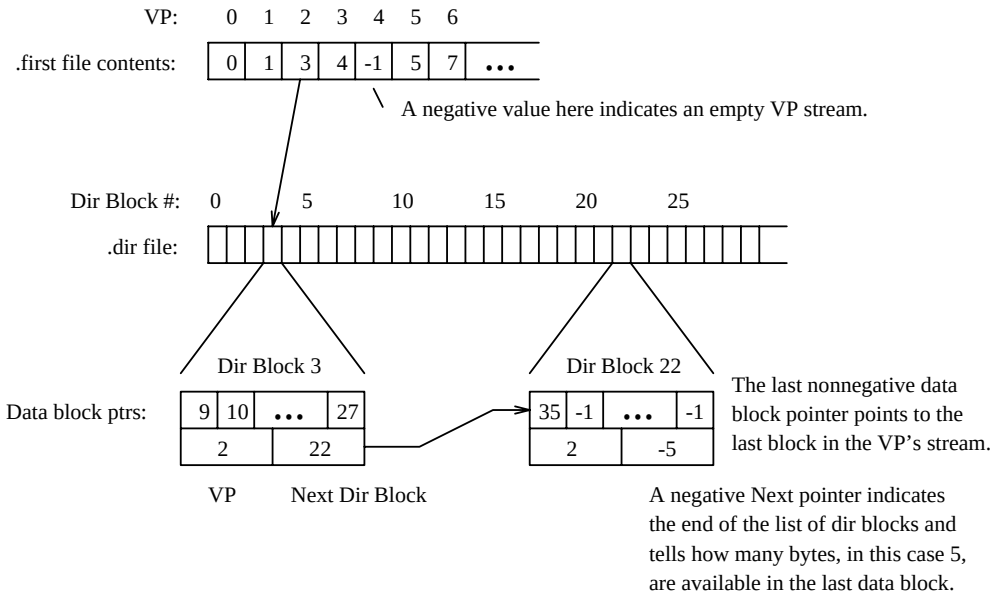

Figure 7: Directory structure for VP data blocks in a single file. The `.first` file contains a pointer to the first directory block for each VP. Each directory block contains pointers to a single VP's data blocks in the IB segment of the data file. The `Next` pointer in a directory block is used to build a linked list containing all of a VP's directory blocks.

5.2.3 Mode Transitions During Writing

A mode transition may occur on all compute nodes in parallel or, when an elemental file operation is performed, on individual nodes. A transition to CB occurs on all compute nodes at once and entails allocation of CB buffers. A transition to IB may require flushing and deallocation of CB buffers, if they are not empty. Then, IB buffers, superblocks, directory superblocks, and VP directories must be allocated. The parallel `FILE` variables, used to manage individual VP streams, are initialized. The IB transition is now complete, and the requested operation is performed, either for an individual VP (in an elemental function), or in a VP emulation loop (in a parallel function like `fwrite`). Note that when a compute node transitions a file to IB due to an elemental file access, the other compute nodes may remain in NB or CB mode. The compute node using IB must participate in the next parallel file operation's reduction to ensure that all other nodes are aware that an IB transition has occurred. No compute nodes can perform CB or NB operations without the other compute nodes participating.

5.2.4 Closing a File

When a file is closed, housekeeping work must be performed. Regardless of the mode(s) used to write a file, the sequential `.meta` file is written by compute node 0. A file closed while in CB must have its VP buffers flushed if they aren't empty. The `total_CB_data` field in the `.meta` file lets the reader know how many bytes of the last CB block are valid.

A file written using IB requires more work on closing. First, the VP buffers are flushed to superblocks. Then, the superblocks are written to the parallel file. Note from Figure 6 that the directory information written so far contains VP values but NULL `next` values. These are updated by reading directory superblocks in reverse order, updating `next` pointers, and writing the data back again. A linked list is formed, with the head of each list stored in the `.first` metafile. Although this directory patching step incurs extra overhead, it is performed using large-grained file operations. An earlier prototype updates the directory in a small-grained fashion as VP blocks are written; this approach reduces effective bandwidth by an order of magnitude. One might consider omitting the patching step, since the VP values in the directory blocks allow the streams to be reconstructed again. However, we chose to eliminate the need for a reading program to search the directory. As parallel applications' file needs become more well-understood, we may find it necessary to

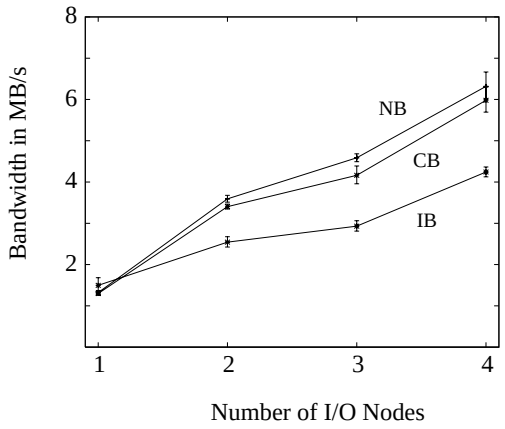

Figure 8: Comparison of NB, CB, and IB modes for reading on the Meiko CS-2. 64K VPs each read 64 `double` values for a total of 32MB. 95% confidence intervals are shown.

change the directory structure entirely, or to perhaps use a doubly-linked list. All the performance figures presented for IB include the extra cost of updating the directory upon closing of the file.

5.3 Reading

A file may be opened for reading using one of two `fopen` overloadings:

```
FILE:current *fopen(char *name, char *type);
FILE:void    *fopen(char *name, char *type, shape *s);
```

The first of these requires that the existing file be of the same shape as `current`. The second, the only Stream* operation whose usage is not analogous to that of traditional C, returns a parallel FILE variable whose shape is defined at run time from the `.meta` file. The `void` shape of the return value specifies that it can match any shape, while the actual shape of the file opened is returned via the `s` parameter.

The mode detection performed for reading is essentially the same as for writing described in section 5.2. NB can be used to read only from the NB segment of the file[3]. CB can be used to read data from the NB and CB segments, because their structure is identical aside from the VP block size. IB can be used to read from any of the file segments.

Reads in NB move data directly from the file system to parallel variables. In CB, data are moved to VP buffers on the compute nodes; VP-level reads then move data from the buffers to parallel variables. As shown in Figure 8, bandwidth achieved by CB scales well with the number of I/O nodes but slightly lags NB bandwidth.

Reading in IB mode was designed to emphasize movement of collections of VP blocks rather than individual blocks between compute nodes and the file system. The design is based on the fact that most programs read a file in the same way it was written. In this case, VP blocks written to the file system in the same superblock will be needed in the reading program at approximately the same time. Therefore, when a VP block (data or directory) is needed from the file system, a superblock containing the desired block and subsequent blocks is read. Like all prefetching schemes, this one may actually hurt performance when a particularly ill-behaved read pattern is used. However, if no prefetching is done, or if prefetches fulfill soon-to-be-emulated VPs' requirements, file accesses are fine-grained and guaranteed to be slow. We feel that prefetching based on VP write patterns is a good heuristic for avoiding fine-grained file accesses.

The reading process for IB mode is detailed in Figure 9. Several variables impact the IB reading performance. As with writing, one of these is VP block size. Table 3 shows the relative performance when several VP block sizes are used. Reducing the block size from 64 to 16 or 32 has an impact on bandwidth, but not as much as for IB writes. The size of a directory entry affects reading performance as it does for writing. That is, moving from two to six data block pointers per directory entry increases performance by about

[3] If the VP block sizes for the NB and CB segments are identical, NB could be used to read the CB segment. We have not implemented this optimization.

# I/O Nodes	IB Block Size				
	64 bytes	32 bytes		16 bytes	
	Bandwidth	Bandwidth	% Relative to 64	Bandwidth	% Relative to 64
1	1.50	1.49	99	1.21	81
2	2.54	2.07	81	2.15	85
3	2.93	2.45	84	2.05	70
4	4.24	3.97	94	3.52	83

Table 3: Comparison of achieved I/O bandwidth when reading in IB mode with different IB block sizes. Bandwidth values are distinct at the 95% confidence level for 3 and 4 I/O nodes.

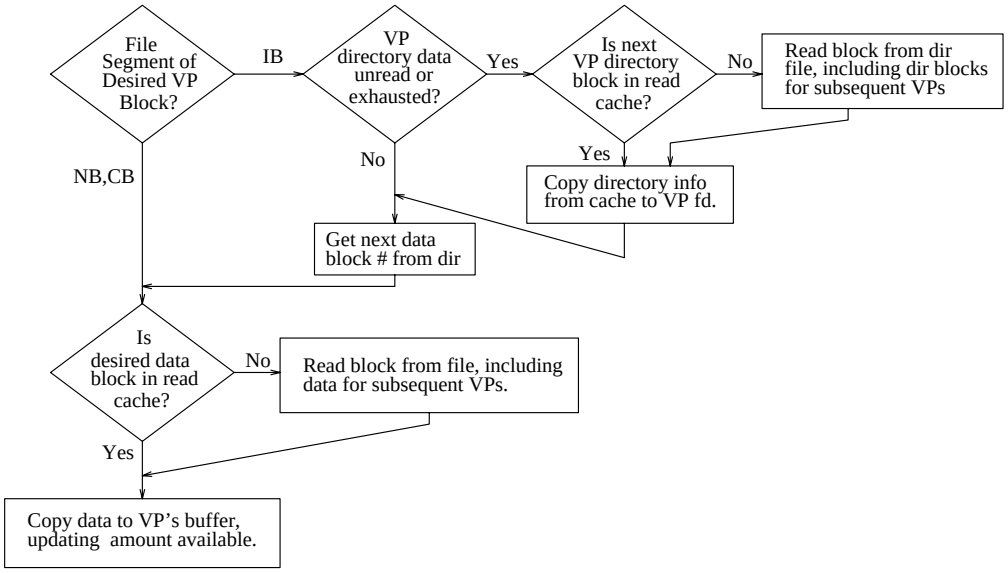

Figure 9: Simple decision flow for reading a single VP block in IB mode. This sequence may be repeated several times to satisfy a user-level request.

6%, but increasing that value to fourteen data block pointers makes negligible additional impact. Finally, the number of read buffers impacts performance. For the results presented in Figure 8, the compute nodes use only two read buffers of 32K bytes each. During the reads, one buffer holds the directory information while another holds data. Because the data are read exactly as written, no more read buffers are needed. In other cases, more read buffers may be needed to cache data for extended periods. Unfortunately, our current implementation uses a simple linear search of buffers to satisfy a given request; the search becomes expensive as the number of buffers increases. We shall update the search to a more efficient scheme; until then, our implementation handicaps reads relying on more than a few read buffers.

5.3.1 Mode Transitions During Reading

Mode transitions can either be forced by the segmented nature of the file (e.g., when the CB segment has been exhausted, a transition to IB must occur), or through the parameters of the requested operation. For example, data in the NB segment of a file may be read using CB if the parallel variable being filled has a stride. A transition to CB requires allocation of CB buffers on the compute nodes. The size of the CB buffers is based on the b_{CB} value stored in the `.meta` file. If the NB segment is being read, and if $b_{NB} < b_{CB}$, b_{NB} is the effective block size until the CB segment of the file is reached.

Transitions to IB occur in two steps. The first step allocates and initializes the read buffers. If the NB or CB segment of the file is being read, directory information is not needed. When the IB segment of the file is reached, the second step of the transition takes place. The `.first` file is read to get the first directory block pointer for each VP. Actual directory blocks are not read until individual VPs perform read operations.

5.4 I/O with Elemental Functions

As shown in [18], a single-stream file model for C* cannot support file output in elemental functions. Here we show that, to maintain a familiar programmer's interface, formatted input and output, critical when debugging at the VP level, must be performed using elemental functions. The functions `fprintf`, `vfprintf`, and `fscanf` receive a variable number of arguments whose type is specified in a format string. The programmer writes, for example:

```
fprintf(parFile,"Variable someVar is %d\n",someVar);
```

and wants the semantics to match `someVar`'s type (scalar or parallel). Unfortunately, the run-time system must rely exclusively on the type specifier in the format string to determine the type of `fprintf`'s arguments, and `%d` dictates a single integer, not a parallel variable. Two obvious solutions present themselves: allow a new parallel type specifier in the format string, or call `fprintf` once per VP so the values passed in are logically scalars. The former solution does not maintain the familiar C programming interface, while the latter, namely implementation of formatted output via elemental functions, does.

The above example shows a file operation implemented as an elemental function. A file operation may also be called from within an elemental function. The Stream* library includes its own versions of the scalar C file operations; these are called from within elemental functions. They check the `fd` value for the `FILE` variable used. If the `fd` represents a scalar file, the original scalar version of the routine (now renamed) is called. Otherwise, the Stream* handler performs the requested operation in IB mode, forcing a transition to IB on that compute node if necessary. Other compute nodes will be informed of the transition during the reduction in the next collective file operation.

5.5 Seeking

The Stream* `fseek` operation, in its most general form, allows each VP to seek to a different location in its stream. Whether reading or writing, typical data-parallel applications will have all VPs seeking to the same position in their streams. When this is the case, and the seek is to the NB or CB segment of the file, NB or CB mode may be used, even if IB was the previous mode. To save space, we simply point out some implementation concerns. When writing to a file, seeks may require flushing of buffers beforehand and read-modify-write sequences afterward. These operations can be supported in all three modes, although updates in IB mode may require that VP blocks be written individually, since each VP block's position is dictated by the earlier writing pattern. Our implementation does not yet support this type of operation, which does not appear frequently in data-parallel applications.

6 Interfacing Stream* to External Programs

Stream* utilizes a unique file format; thus, the files are "internal" [4]. Ideally, Stream* could exchange files with external applications using little or no filtering of files. With NB mode, this is the case. Files consisting of a single NB segment are laid out exactly as an array-oriented program would write them. These files can be used without conversion by external programs. External data files can be treated as files containing only an NB segment. If the first overloading of `fread` in Section 5.3 is used, an external file lacking a `.meta` file simply takes on the current shape and is assumed to consist of a single NB segment. A `.meta` file may have to be built using a simple utility program in some situations (e.g., if a file has a different distribution than the program that will read it). Because most data-parallel applications rely on regular array-oriented I/O, the use of NB as an interface to the outside world should work in most situations [5, 7, 14, 16, 17, 21].

Files with CB and IB segments require explicit conversions. A file containing a CB or IB segment can be converted to one containing a single NB segment using a high-level C* program. The program has each VP reading its existing stream and writing its data, or a fixed value upon reaching EOF on its input, in NB mode. Following three rules guarantees that an output file will consist of only an NB segment, which means it will be written at top speed and will be readable by external applications.

- Use the simple form of `fwrite` with all VPs active.
- Don't output fields of structs or individual parallel array elements.

- Output the same number of bytes per VP during every operation.

Note that these rules are at the language level, so the programmer does not need Stream* implementation knowledge to get high performance.

7 Conclusions

We have shown that the programmer's I/O interface can be seamlessly integrated with C*'s virtual processor programming paradigm using data-parallel streams. Their implementation using machine-independent, automatically detected modes lets the most common file operations found in data-parallel programs run at the top speed supported by the file system. The high-performance modes, because of their array-oriented nature, can also take advantage of file redistribution optimizations developed for languages such as HPF. The general mode supports a wide variety of file operations while achieving bandwidth over half that of the high-performance operations by combining fine-grained virtual processor operations into large-grained file system operations.

Acknowledgements

This research was supported by NSF grants ASC-9208971 and CDA-9216172.

References

[1] S. Batra, P. J. Hatcher, and R. Russell. The design and implementation of data-parallel files. In *Workshop on Modeling and Specification of I/O*, 1995. Publication via `http://www.cs.duke.edu/~dev/msio95`.

[2] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[3] P. F. Corbett and D. G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

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

[5] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[6] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[7] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[8] P. J. Hatcher. Elemental functions. Technical Report X3J11.1/92-076, Numerical C Extensions Group (ANSI X3J11.1), 1992.

[9] P. J. Hatcher. Extending C* for data-parallel I/O. Technical Report TR 94-16, University of New Hampshire Department of Computer Science, 1994.

[10] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, Massachusetts, 1991.

[11] Intel. *Paragon OSF/1 User's Guide*, 1993.

[12] D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[13] D. Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dartmouth College Department of Computer Science, July 1994.

[14] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[15] A. J. Lapadula, K. P. Herold, and P. J. Hatcher. A retargetable C* compiler and run-time library for mesh-connected MIMD computers. Technical Report TR 92-15, University of New Hampshire Department of Computer Science, 1992.

[16] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing '91*, pages 567–576, 1991.

[17] J. A. Moore. Parallel I/O requirements of four oceanography applications. Technical Report 95-80-1, Oregon State University Department of Computer Science, 1995.

[18] J. A. Moore, P. J. Hatcher, and M. J. Quinn. Stream*: Fast, flexible data-parallel I/O. In *Parallel Computing '95*, September 1995.

[19] S. A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[20] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[21] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995.

[22] Thinking Machines Corporation. *C* Programming Guide*, June 1991.

[23] Thinking Machines Corporation. *Connection Machine I/O System Programming Guide*, October 1991.

[24] Thinking Machines Corporation. *CM-5 I/O System Programming Guide*, September 1993.

[25] A. Witkowski, K. Chandrakumar, and G. Macchio. Concurrent I/O system for the hypercube multiprocessor. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1398–1407, 1988.