I/O Requirements of Scientific Applications: An Evolutionary View*

Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, Daniel A. Reed
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
email:{esmirni,aydt,achien,reed}@cs.uiuc.edu

Abstract

The modest I/O configurations and file system limitations of many current high-performance systems preclude solution of problems with large I/O needs. I/O hardware and file system parallelism is the key to achieve high performance. We analyze the I/O behavior of several versions of two scientific applications on the Intel Paragon XP/S. The versions involve incremental application code enhancements across multiple releases of the operating system. Studying the evolution of I/O access patterns underscores the interplay between application access patterns and file system features.

Our results show that both small and large request sizes are common, that at present application developers must manually aggregate small requests to obtain high disk transfer rates, that concurrent file accesses are frequent, and that appropriate matching of the application access pattern and the file system access mode can significantly increase application I/O performance. Based on these results, we describe a set of file system design principles.

1. Introduction

For grand challenge scientific applications, where manipulating large volumes of data is a necessity, scalable I/O performance is a critical requirement. Unfortunately, rapid increases in computation and communication hardware performance have long outstripped the performance improvements of secondary and tertiary storage devices. This increasing disparity between the hardware performance of individual processors and storage devices dictates use of hardware and software I/O parallelism.

During the past decade, a wide variety of parallel I/O systems have been proposed and built [2, 8, 11]. All these systems exploit parallel I/O devices (partitioning data across disk arrays [1] for parallelism) and data management techniques (prefetching and write-behind) in an attempt to deliver high I/O performance. However, parallel file system experience has shown that performance is critically sensitive to the distribution of data across storage devices and that no single file management policy yields good performance for all application access patterns. As a result, many parallel I/O systems have high peak performance, but their much lower delivered performance continues to constrain the domain of I/O intensive applications.

Although there have been extensive studies of uniprocessor file access patterns, much less is known about the I/O requirements of scalable scientific applications. In particular, there is no clear consensus on the complexity and diversity of file access patterns and resource demands of parallel scientific applications. As a consequence, parallel file system designers have often been forced to extrapolate from measurements on uniprocessors or vector supercomputers. Neither of these environments is characteristic of parallel application I/O needs on scalable parallel systems.

Clearly, understanding the interactions between application I/O request patterns and the hardware and software of parallel I/O systems is a precursor to designing more effective I/O management policies. As part of the Scalable I/O Initiative (SIO) [14], our twin goals are to collect detailed performance data on the I/O characteristics and access patterns of a variety of scalable parallel scientific applications and to use this information to design and evaluate parallel I/O file system management policies.

In this paper, we report an analysis of the evolving I/O behavior and performance of two SIO codes, an electron scattering application [18] and a computational fluid dynamics application [4]. Over eighteen months, we studied the evolving resource demands of both applications on the Intel Paragon XP/S and captured the interplay between application resource demands and system responses. By tracking the evolution of each application's I/O demands as the code was tuned, we isolated and identified file access patterns that were intrinsic to the applications and not artifacts of the developers' performance optimizations with respect to the file system features.

The remainder of the paper is organized as follows. In

*This work was supported in part by the Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0029 and DABT63-93-C-0040, by the National Science Foundation under grant NSF ASC 92-12369, and by the Aeronautics and Space Administration under NASA Contracts NGT-51023 and USRA 5555-22. Andrew Chien is supported in part by an NSF Young Investigator Award CCR-94-57809.

§2, we outline related work in parallel I/O characterization. This is followed in §3 by a brief description of our experimental methodology using the Pablo performance analysis environment and a summary of parallel file system characteristics. In §4–§5, we describe the evolution of two I/O intensive codes, followed in §6 by a comparison of their I/O behaviors. Finally, §7 summarizes our findings and outlines future work.

2. Related Work

Several I/O characterization efforts for scientific applications have concentrated on the I/O behavior of vector supercomputers [9, 12, 13]. Miller and Katz [9] measured a workload of mostly computational fluid dynamics applications on a Cray Y-MP and characterized the I/O behavior as highly regular, cyclical, and bursty. They first proposed a high-level I/O classification based on compulsory, checkpoint, and data staging I/O operations.

Pasquale and Polyzos [12] used clustering and regression analysis to examine I/O demands and their relationship to the elapsed computation time of the Cray Y-MP workload at the San Diego Supercomputing Center (SDSC). They also isolated and analyzed two I/O intensive applications on a Cray 90 at SDSC [13]. By focusing on both physical resource usage and functional application composition, Pasquale and Polyzos concluded that supercomputer I/O behavior was recurrent and predictable, based on the iterative nature of many scientific applications.

In contrast to the results on vector systems, our more recent analysis on the Intel Paragon XP/S [3] indicated that there are large variations in the temporal and spatial access patterns of scientific applications. Some cyclic behavior was noted, but the applications' I/O patterns were more irregular, with both extremely small and extremely large requests. Our work complements this earlier study by isolating the *reasons* that application developers opt to use parallel file systems in particular ways.

Kotz and Nieuwejaar [7] examined application file access characteristics on the Intel iPSC/860's Concurrent File System at NASA's Ames Research Center. In a related study, Parakayastha *et al* [15] investigated file access patterns on a CM-5 under the Scalable File System (SFS) at the National Center of Supercomputing Applications. The results of both studies showed that the majority of the file accesses were small, even though the vast majority of data was transferred via a few large requests. Subsequent comparison of the file access characteristics on both systems [10] indicated that users "tune" their applications to the underlying parallel system's idiosyncrasies in an effort to maximize performance. Our work builds on these earlier studies by examining the interplay between file system features and application developer decisions.

3. Experimental Approach

In contrast to previous I/O characterization studies that considered a single version of a code, we have tracked the evolving I/O behavior of two applications over an eighteen month period. The two applications are both part of the Scalable I/O application suite, they are in active use by their developers, and they are evolving rapidly based on continuing changes to the Intel Parallel File System (PFS).

To capture traces of application I/O requests and parallel file system responses, we used an extended version of the Pablo performance analysis environment [17]. All data was captured on the 512-node Intel Paragon XP/S at the Caltech Center of Advanced Computing Research. Below, we describe the I/O analysis mechanisms and the salient characteristics of the Intel parallel file system.

3.1. Pablo Software Infrastructure

Building on the lessons of several previous performance analysis toolkits, Pablo [16, 17] is a portable performance environment that supports both performance data capture and analysis. The Pablo instrumentation software captures dynamic performance data via instrumented source code that is linked with a data capture library. During program execution, the instrumentation code generates performance data that can either be directly recorded by the data capture library or processed by one or more data analysis extensions prior to recording. After program execution completes, the data can be analyzed with a toolkit of data transformation modules and a graphical programming model that allows users to interactively connect and configure a data analysis graph.

Detailed I/O event traces include the time, duration, size, and other parameters for each I/O operation. Statistical summaries can take one of three forms: file lifetime, time window, or file region. File lifetime summaries include the number and total duration of file reads, writes, seeks, opens, and closes, as well as the number of bytes accessed for each file, and the total time each file was open. Time window summaries contain similar data, but allow one to specify a window of time for summarization. Finally, file region summaries are the spatial analog of time window summaries; they define a summary over the accesses to a file region. Collectively, the raw event traces and the statistical summaries provide a detailed view of application I/O request patterns and file system responses.

3.2. Intel Paragon XP/S Configuration

Interpreting the results of any empirical study is critically dependent on understanding the experimental environment. All our experiments were conducted on the Caltech 512-node Intel Paragon XP/S, organized as a 16x32 mesh. For all experiments, sixteen I/O nodes were used, each with a 4.8GB RAID-3 disk array, and files were striped across the disk arrays in units of 64K bytes, the PFS default.

PFS offers several file access modes, each designed to support a specific class of application I/O patterns. The default I/O mode is `M_UNIX`, which provides standard UNIX file sharing semantics when more than one process access the same file. In this mode, each process has a unique file pointer and can access information anywhere in the file. There are no restrictions on the range of possible file request sizes. Because request atomicity is preserved, the performance of mode `M_UNIX` can be poor when multiple processes access the same file. In that case, it is often more efficient to use the `M_RECORD` or `M_ASYNC` modes.

With the `M_RECORD` mode, there is a unique file pointer for every process, all nodes must access fixed size records, and concurrent operations occur in node order. In this way, each process can operate on separate file areas in a parallel and highly structured fashion.

In the `M_ASYNC` mode, every process has its own file pointer and, in contrast to `M_RECORD`, variable length accesses are possible. Atomicity is not preserved and processes need not read from or write to the same file concurrently.

The `M_GLOBAL` mode is often used for initial compulsory reads, where all processes must access the same data. With this mode, a shared file pointer is maintained, all processes must access the same data in a synchronized fashion, and data are read/written only once. Intuitively, only one of a group of identical I/O operation occurs, and the data is shared among all the processes.

With the `M_SYNC` mode, all processes share a single file pointer and I/O requests occur in node order. All I/O requests are synchronized, but the request size can vary from node to node.

Finally, with the `M_LOG` mode all processes share a common file pointer, I/O requests are processed first-come-first-serve, they are not synchronized, and the request size can vary across processes. As the name suggests, this mode is normally used to access `stdin, stdout`, and `stderr` files.

4. Schwinger Multichannel Electron Scattering

The study of low-energy electron-molecule collisions is of interest in many contexts, including aerospace applications, atmospheric studies, and the processing of materials using low-temperature plasmas (e.g., semiconductor fabrication). The Schwinger Multichannel (SMC) method is an adaptation of Schwinger's variational principle for the scattering amplitude that makes it suitable for calculating low-energy electron-molecule collisions [18].

The scattering probabilities are obtained by solving linear systems whose terms must be evaluated by numerical quadrature. Generation of the quadrature data is computationally intensive, and the total quadrature data volume is highly dependent on the nature of the problem. The quadrature data is formulated in an energy independent way, making it possible to solve the scattering problem at many energies without quadrature data recalculation. Because the quadrature data is too voluminous to fit in the individual processor memory, an out of core solution is required.

ESCAT is a parallel implementation of the Schwinger Multichannel method written in C, FORTRAN, and assembly language [18]. From an I/O perspective, the code has four distinct execution phases:

- **Phase One**: Initialization data is read from three input files (compulsory I/O). During this phase, the problem definition and some initial matrices are loaded.
- **Phase Two**: Quadrature data is written to disk (data staging). In this phase, all nodes participate in the calculation and storage of the requisite quadrature data set. This phase is composed of a series of compute/write cycles, with the write steps synchronized among the nodes. The number of data files written corresponds to the number of collision channels.
- **Phase Three**: Quadrature data is read from disk (data staging). This phase involves calculations that depend on the collision energy. Energy dependent data structures are generated and combined with the reloaded quadrature data.
- **Phase Four**: Results of the calculations are written to disk (compulsory I/O). The number of output files corresponds to the number of collision channels.

As noted in §3, we used the Pablo performance environment's I/O instrumentation software to capture the ESCAT code's I/O behavior, including the individual request sizes, their timings, and their temporal and spatial patterns. Based on our analysis of this data, we offered several suggestions to the code's developers, many of which were subsequently implemented. Furthermore, the Intel PFS designers modified the Paragon XP/S file system to better meet the application I/O needs and improve I/O performance. Below, we analyze the evolution of ESCAT application I/O requirements for a specific test problem.

4.1. I/O Requirements

As a baseline for I/O performance analysis, we considered a representative, though modest problem, taken from a study of electronic excitation of ethylene to its first triplet state. For this problem, two collision channels are of interest: the elastic-scattering channel and the inelastic triplet-excitation channel.

Using the ethylene data set, we measured the performance and behavior of six distinct versions of the ESCAT code on 128 processors during a period of eighteen months[1].

[1] Although we have collected and analyzed instrumentation data from two much larger problems, the electronic excitation of carbon monoxide and the elastic scattering cross section for boron trichloride, the focus of this paper is the smaller ethylene data set.

	Version A		**Version B**		**Version C**	
	OSF 1.2, Pablo Beta		OSF 1.2, Pablo 4.0		OSF 1.3, Pablo 4.0	
	I/O Activity	I/O Mode	I/O Activity	I/O Mode	I/O Activity	I/O Mode
Phase One	All Nodes	`M_UNIX`	Node zero	`M_UNIX`	Node zero	`M_UNIX`
Phase Two	Node zero	`M_UNIX`	All Nodes	`M_UNIX`	All Nodes	`M_ASYNC`
Phase Three	Node zero	`M_UNIX`	All Nodes	`M_RECORD`	All Nodes	`M_RECORD`
Phase Four	Node zero	`M_UNIX`	Node zero	`M_UNIX`	Node zero	`M_UNIX`

Table 1: Node activity and file access modes (ESCAT)

During this interval, we considered the effects of operating system changes, new application code versions, and software instrumentation updates. Thus, we have a detailed record of the performance pitfalls that triggered application and operating system evolution.

Figure 1 illustrates the progressive reduction in total execution time for ESCAT with the ethylene data set. Overall, total execution time was reduced by 20 percent from the first ESCAT version (version **A** in Figure 1) to the final version (version **C** in Figure 1). Although this reduction may seem modest, the ESCAT code's total execution time is not dominated by I/O for this small problem. For larger problems, the optimizations directly translate to order of magnitude performance improvements.

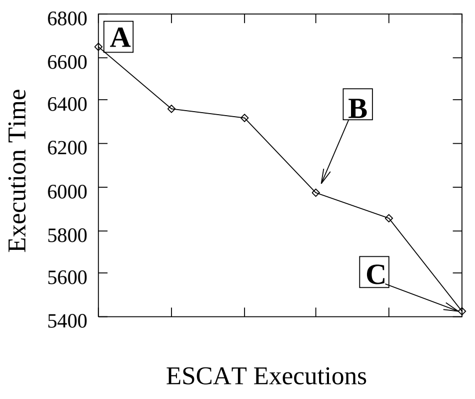


Figure 1: Execution time for six ESCAT code progressions

For brevity's sake, our analysis will concentrate on code versions **A**, **B**, and **C**. Table 1 summarizes the major differences among the three versions: the number of nodes that actively participate in I/O during each of the four application phases and the PFS I/O mode used in each phase. Similarly, Table 2 shows the fraction of time attributable to each I/O operation type.

The I/O activity for version **A** reflects the influence of the Concurrent File System (CFS) on the Intel Touchstone Delta, where the code was first developed. As Table 1 shows, in version **A** all nodes concurrently open and read the initialization files in phase one. In phase two, all nodes repeatedly compute, synchronize, and compose the quadrature data. Node zero then collects the quadrature data and writes it to intermediate staging files on disk. In phases three and four, node zero reads and writes all data.

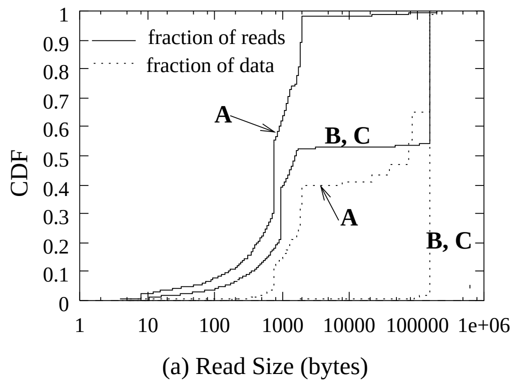
Table 2 shows that for version **A**, most of the I/O time is spent opening and reading files. Because the default `M_UNIX` mode is used, all reads during phase one are serialized. This inefficiency makes reads a high fraction of the total I/O time. However, because all writes occur through node zero, there is no contention, and the time spent on writes and seeks is negligible.

	$\frac{Operation\ Time}{Total\ I/O\ Time} \times 100$		
Operation	**A**	**B**	**C**
`open`	53.68	0.00	0.03
`gopen`	–	4.05	21.65
`read`	42.64	0.24	1.53
`seek`	1.01	63.21	1.75
`write`	1.27	28.75	55.63
`iomode`	–	2.94	16.06
`close`	1.39	0.81	3.34

Table 2: Aggregate I/O performance summaries (ESCAT)

In version **B**, a significant reduction in read time was achieved via code restructuring. In this version, node zero reads the input data and broadcasts it to the remainder of the nodes; see Table 1. In phases two and three, all nodes participate in writing/reading the quadrature data to/from disk. To eliminate the time spent opening the same file concurrently by multiple nodes, a global open operation (`gopen`) is used as an alternative to the more expensive `open` operation. In phase two, to simplify reloading the quadrature data, each node seeks to a calculated offset dependent on the node number, iteration, and the Paragon PFS stripe size before writing any data. Intel's `M_UNIX` file mode is used for these writes, and seek time dominates the total I/O time. The use of `M_RECORD` mode to reload the quadrature data in phase three further contributes to the reduction in read time from version **A**;see Table 2.

As Table 1 shows, the code structure of version **C** is similar to that for version **B**, the only difference is the file access mode of phase two. Due in part to our recommendation,



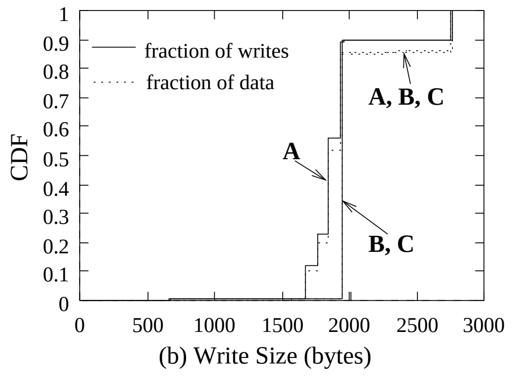


Figure 2: CDF of read/write request sizes and data transfers for ESCAT

Intel introduced the more efficient M_ASYNC mode in the OSF/1 1.3 operating system release. This file access mode dramatically reduces seek and write times. Recall that with M_ASYNC every process has its own file pointer and operation atomicity is left to the discretion of the programmer. Consequently, the system overhead for preserving operation atomicity is avoided.

4.2. I/O Request Sizes

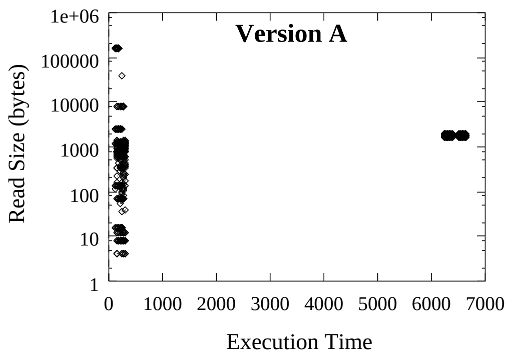
Apart from changes in the code structure and file access modes, the I/O request sizes across the three ESCAT versions also differ. Figure 2 shows the cumulative distribution function (CDF) of the percentage of reads and writes versus the request size for the three code versions, as well as the fraction of data transferred by each request size. Because there are no changes in request sizes from version **B** to version **C**, the CDFs for these two versions are shown in the same plot.

The number of small reads (i.e., those less than 2K bytes) declines dramatically from version **A** to versions **B** and **C**. In version **A**, 97 percent of all read operations are small, though they transfer only 40 percent of the data. In the two following versions, only 50 percent of reads are small, and large 128KB reads (two PFS file stripes) transfer 98 percent of the data read.

The large read sizes are a direct effect of the programmer's choice to use M_RECORD to access the quadrature data in phase three; to guarantee good performance when using M_RECORD, the request size must be a multiple of the stripe size. Changes in the CDFs for writes from version **A** to versions **B** and **C** are not dramatic; all write requests are small, and most are less 2K bytes.

4.3. Temporal I/O Behavior

Spatial access patterns are but a part of the story; the temporal spacing of request sizes, operations, and use of file system access modes also affects performance. Figure 3 depicts read request sizes as a function of program execution time for versions **A** and **C**.[2] In both cases, read activity occurs only near the beginning and the end of the ESCAT execution. First, initialization files are read. Because only node zero reads data in version **C**, there is a significant reduction in the density and span of initial read time between versions **A** and **C**. In the final read phase, where the quadrature data is transferred from disk, the read request sizes change significantly. In version **C**, all nodes read the quadrature with 128KB increments (twice the PFS stripe size), whereas in version **A**, node zero reads the data in small chunks (less than 2K bytes) and broadcasts them to the other nodes.



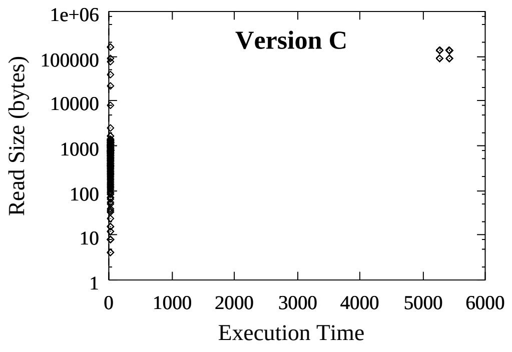


Figure 3: File read sizes for versions **A** and **C** of ESCAT

The differences in the application temporal behavior

[2]The data for version **B** is not shown, because it is very similar to that for version **C**.

across the three versions are the effect of different file access modes, number of concurrent accesses, and corresponding request sizes. In version **A**, all nodes concurrently access the initialization files using the `M_UNIX` mode, serializing all requests. Because only node zero accesses the initialization files in version **C**, the duration and and total number of initial reads declines precipitously.

Figure 4 shows the write request sizes for versions **A** and **C**; version **B** is similar to version **C**. As with reads, the write structure of both versions has two distinct phases. First, the quadrature data is generated and transferred to disk. In version **A**, node zero coordinates these writes with four different request sizes. In version **C**, all write requests are of the same size, and the individual nodes write the data directly using the `M_ASYNC` mode. Finally, in all three versions, node zero writes the results of computation to the disk.

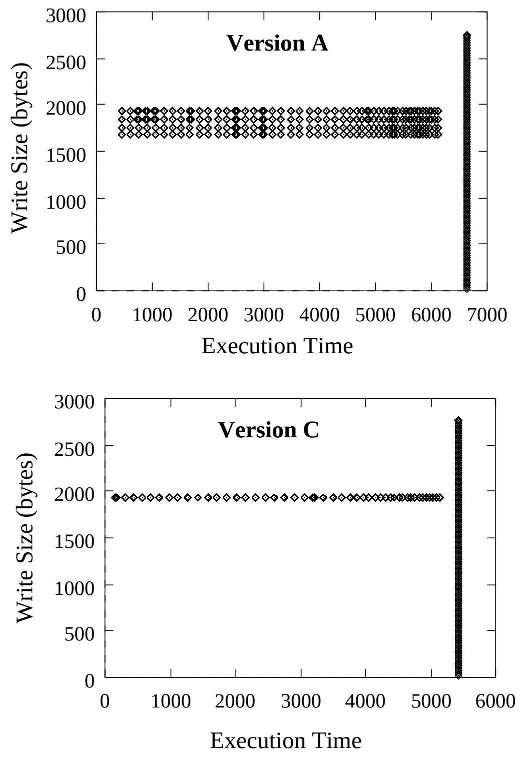


Figure 4: File write sizes for versions **A** and **C** of ESCAT

Figure 5 depicts the seek durations for versions **B** and **C**. Notice the difference in the order of magnitude on the y-axis for the plots of Figure 5. The advantage of `M_ASYNC` versus `M_UNIX` is further illustrated in this figure; seek times are almost eliminated. Recall that `M_UNIX` provides the standard UNIX file sharing semantics by serializing requests when more than one process access the same file. In this case, each node has a unique file pointer and seeks prior to reading or writing data in the file. The use of `M_ASYNC`

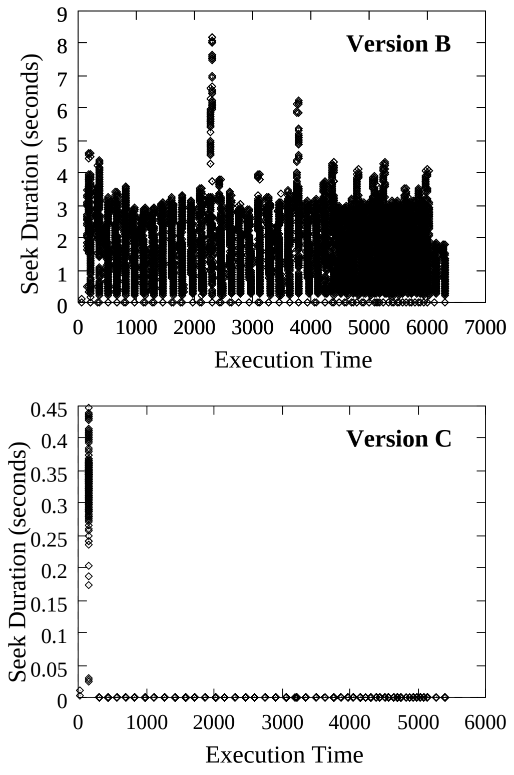


Figure 5: Seek operation durations for versions **B** and **C** of ESCAT

eliminates the need for serialization and clearly improves performance.

4.4. Discussion

The I/O behavior of the ESCAT evolution highlights the importance of application tuning to achieve high I/O performance. The most critical I/O demands are placed on the scratch files that contain the intermediate quadrature data. These demands are proportional to the number of total collision outcomes.

Table 3 shows that for the ethylene case, where just two collision outcomes are of interest, only 0.73 percent of the total execution time is spent on I/O for the optimized version **C**. At first glance, it appears that I/O is of little import because this code is heavily computation bound. While this is true for the ethylene data set, it is not true for larger data sets. The complexity of the quadrature data volume grows as $O(N^3)$, where N is the number of electron scattering outcomes. Thus, gains from I/O performance optimization are more pronounced as the number of collision outcomes increases.

Using version **C**, the code developers can now attack larger, more complex problems. Table 3 shows the I/O costs for a larger carbon monoxide data set, where 13 collision outcomes are of interest. In this case, the total time spent on I/O is twenty percent of total execution time *after*

	$\frac{Operation\ Time}{Total\ Execution\ Time} \times 100$			
	Ethylene (128 nodes)			Carb. Mon. (256 nodes)
Operation	**A**	**B**	**C**	**C**
`open`	1.60	0.00	0.00	0.00
`gopen`	–	0.19	0.16	7.45
`read`	1.27	0.01	0.01	9.50
`seek`	0.03	2.91	0.01	0.00
`write`	0.04	1.32	0.41	0.03
`iomode`	–	0.14	0.12	–
`close`	0.04	0.04	0.02	2.41
All I/O	2.97	4.60	0.73	19.40

Table 3: Percentage of total execution time by I/O operation type for ESCAT

optimization[3].

In summary, the application developers were able to restructure their code, tuning request sizes to match the PFS file stripe size. In addition, they opted to substitute concurrent reads by all nodes with single node read and broadcast mechanisms, despite of the introduced complication in the code structure. Request aggregation and prefetching by the file system would simplify code structure and eliminate the need for code restructuring to exploit file system characteristics.

5. 3-D Numerical Simulation of the Navier-Stokes Equations

Computational fluid dynamics is an increasingly frequent alternative to experimental study, particularly for understanding high-speed turbulent flow. PRISM is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations written in C [4]. The parallelization is achieved by apportioning slides of the periodic domain to the nodes, with a combination of spectral elements and Fourier modes used to investigate the dynamics and transport properties of turbulent flow.

This code models a geometry where the flow is periodic in at least one direction (e.g., flow past a cylinder, flow in a channel, and flow over a backward-facing step). An initial velocity field is given by the input data, and the solution is integrated forward in time by numerically solving the equations that describe advection and diffusion of momentum in the fluid. From an I/O perspective, there are three distinct execution phases:

- **Phase One**: Three input files are used to initialize the system (compulsory I/O). The first file contains parameters such as the Reynolds number, the number of mesh elements, the coordinates for each element, the number of curved edges, and the boundary conditions for each edge. The second file is a restart file that contains the initial conditions. The third input file is the connectivity file that contains information used to establish the internal boundary system of the spectral element mesh.

- **Phase Two**: The integration function for the Navier-Stokes simulation integrates the status of the fluid forward in time from its current to its new state. History points are written to disk during the integration (checkpointing). A measurement file is written by node zero that includes lift and drag forces, viscous forces, and the kinetic energy of each mode. Flow statistics for velocity, vorticity, and turbulent stresses are written to three separate files. Each of these files contains the mean, variance, skewness, and flatness associated with each field.

- **Phase Three**: During the final or postprocessing phase, results are transformed back to physical space and the field file is written to disk (compulsory I/O).

[3]For detailed I/O analysis of the carbon monoxide data set see http://www-pablo.cs.uiuc.edu/Projects/IO/sioDir/escat/escat.html

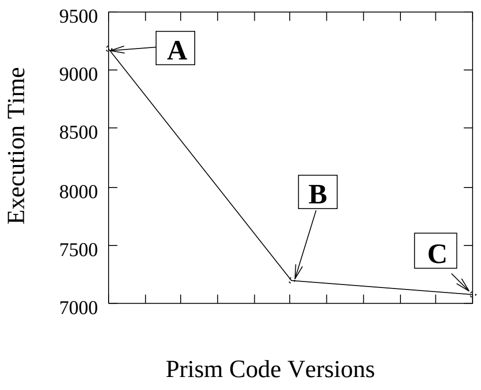


Figure 6: Execution time for three PRISM code versions

As with the ESCAT code, we instrumented PRISM using the Pablo performance environment. Below we describe the code evolution and I/O characterization of PRISM.

5.1. I/O Requirements

We tracked the evolution of the PRISM code using a small test problem, and captured performance data on 64 nodes of the 512 node Intel Paragon XP/S at Caltech. The test problem consisted of 201 mesh elements and a Reynolds number of 1000. The model was simulated for 1250 time steps, with checkpointing every 250 time steps (i.e., a total of five checkpoints). Because our experience with PRISM is somewhat less extensive, we analyzed only three versions of the code, all executed under OSF/1 R1.3.

Table 4 summarizes the I/O modes and activities for the

	Version A OSF 1.3, Pablo 4.0		**Version B** OSF 1.3, Pablo 4.0		**Version C** OSF 1.3, Pablo 4.0	
	I/O Activity	I/O Mode	I/O Activity	I/O Mode	I/O Activity	I/O Mode
Phase One	All Nodes	P: M_UNIX R: M_UNIX C: M_UNIX	All Nodes	P: M_GLOBAL R(h): M_GLOBAL R(b): M_RECORD C: M_GLOBAL	All Nodes	P: M_GLOBAL R: M_ASYNC C: M_GLOBAL
Phase Two	Node Zero	M_UNIX	Node Zero	M_UNIX	Node Zero	M_UNIX
Phase Three	Node Zero	M_UNIX	All Nodes	M_ASYNC	All Nodes	M_ASYNC

Table 4: Node activity and file access modes (PRISM)

three code versions[4], Figure 6 shows the twenty-three percent reduction in total execution time across the versions, and Table 5 shows the effects of the different I/O mode choices on application performance. As with ESCAT, application I/O optimization substantially reduced I/O time.

Comparison of the high level I/O characteristics of versions **A** of ESCAT and PRISM (see Tables 1 and 4) shows marked similarities. Both codes use standard UNIX I/O operations, and all nodes read data in the initial, compulsory I/O phase. For the other two PRISM phases, all I/O is administered though node zero. As Table 5 shows, the majority of the PRISM I/O time for version **A** is spent opening and reading files. In phases two and three, node zero writes the measurement file, the statistics files, and the field file using the M_UNIX mode. Because only one node writes, there is little I/O contention and the seek and write operations are efficient.

After code restructuring, the read and file open times are substantially lower for version **B**. The PRISM developers preferred not to perform all reads through node zero, but instead used the M_GLOBAL access mode that aggregates identical I/O requests, yielding a single disk I/O. In version **B**, the parameter and connectivity files are accessed using the M_GLOBAL mode. Two access modes are used for the restart file; its header is accessed using the M_GLOBAL mode and its body using the M_RECORD mode. With the M_RECORD mode, every node is synchronized to read a specific area in the file. As Table 5 shows, the write time in version **B** increases as a consequence of the concurrent writes by all processors to the field file of phase three.

In both versions **A** and **B**, the open operation is very expensive. As with ESCAT, open times are reduced by using the more efficient gopen operation in version **C**. Because it also sets the file mode, the gopen call eliminates expensive file mode operations.

Although the open time drops sharply in version **C**, the read time increases significantly. In version **C**, to reduce the time required to access the body of the restart file in phase one (the body of the restart file is accessed via a few requests of 155,584 bytes each), the programmer opted to disable any system I/O buffering. By disabling buffering, the read time of the header of the restart file (that is accessed via a few requests of less than 40 bytes each) augmented disproportionately. If I/O buffering were disabled after accessing the header of the restart file, read time would have been as low as in version **B**, and a significant improvement in the total I/O time would have been achieved. We will return to this point below, where we more closely examine the read behavior across the program execution timeline.

[4]In Table 4, the three input files in phase one are denoted by the first letter of the file name.

	$\frac{Operation\ Time}{Total\ I/O\ Time} \times 100$		
Operation	**A**	**B**	**C**
open	75.43	57.36	3.36
gopen	–	–	3.42
read	16.24	9.47	83.92
seek	3.87	1.22	0.40
write	1.83	9.91	6.51
iomode	–	17.75	–
flush	–	–	0.06
close	2.63	4.50	2.32

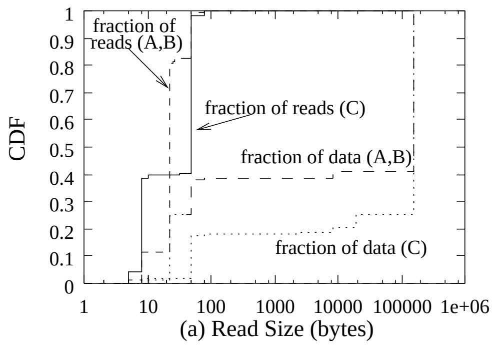
Table 5: Aggregate I/O performance summaries (PRISM)

5.2. I/O Request Sizes

Figure 7 shows the cumulative distribution function for reads and writes for the three PRISM versions. There is no significant variation in the access sizes across the three versions. In version **C**, the connectivity file is read as binary rather than text data, reducing the number of small reads. In general, there are a large number of small (less than 40 bytes) read and write requests, although a few large requests (greater 150KB) constitute the majority of I/O data volume.

5.3. Temporal I/O Behavior

Figure 8 shows that there is significant variation in the temporal distribution of read requests during execution of the three code versions. First, from version **A** to version **B**, the total read time decreases by 125 seconds. This decrease is an immediate effect of using the M_GLOBAL and M_RECORD



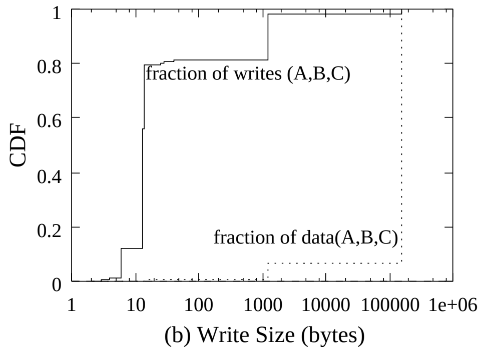
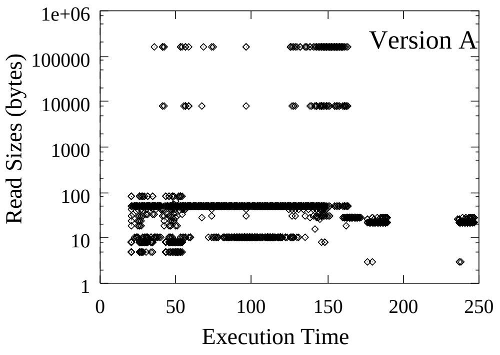
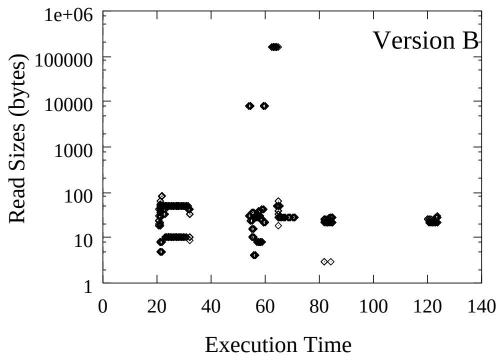


Figure 7: CDF of read and write request sizes and data transfers for PRISM





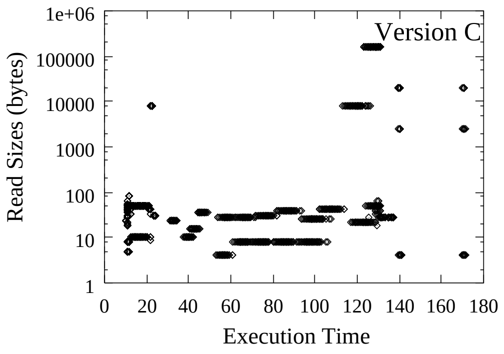


Figure 8: File read sizes for for three versions of PRISM

modes instead if the M_UNIX mode, to access the three initialization files.

In version **A**, the developers used standard UNIX I/O for reads, resulting in serialization of the I/O operations seen in Figure 8. An indirect effect of using the M_GLOBAL and M_ RECORD modes is node synchronization and compact distribution of read requests across the execution timeline in version **B**. Moving from version **B** to version **C**, PRISM's execution timeline becomes longer due to the disabling of system I/O buffering when accessing the restart file.

Because most writes occur through node zero, the temporal write patterns and sizes differ little across the three PRISM versions. Figure 9 shows the temporal write patterns for version **C**, where the five checkpoints are clearly visible.

5.4. Discussion

For the test problem we tracked, the PRISM code is not as I/O intensive as ESCAT. However, as with ESCAT, the I/O scalability of the PRISM code is crucial to solving larger problems. Moreover, analysis of the PRISM code versions highlights the need to efficiently support a range of request sizes. Although current parallel systems favor large requests because high bandwidths are achieved through parallelism, low latency access for small requests is also essential.

As illustrated in version **C** of PRISM, a few small reads can dominate overall I/O time. Robust I/O operations that employ caching or prefetching are an attractive and less confusing alternative to manual request aggregation. A file system that dynamically tunes its policy to match the requirements of the application access patterns and disk performance characteristics is a promising alternative [6].

6. Application Comparisons

Although the ESCAT and PRISM codes differ dramatically in their algorithmic approaches, they share many common I/O characteristics and problems. First, small code changes (e.g. a few I/O calls) can produce large changes in I/O performance. This is both heartening and dismaying. Wholesale code changes are not necessary to improve I/O performance, but selecting inappropriate I/O routines can severely affect achieved performance.

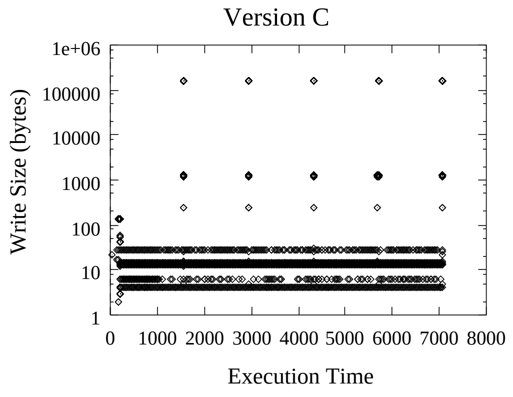


Figure 9: File write sizes for version **C** of PRISM

Second, tuning application I/O request sizes to match the underlying parallel file system is necessary to achieve acceptable parallel I/O transfer rates. However, understanding when and how to aggregate requests is possible only if one understands the performance implications and implementation of the parallel I/O system. In short, optimizations are closely tied to the idiosyncrasies of the parallel I/O system.

Third, both applications exhibit the same general I/O structure of three distinct phases. In the first phase, initialization data is read from a small number of files (compulsory I/O). During the second phase, all nodes compute and synchronize, and most data output occurs. PRISM uses checkpointing, producing periodic bursts of I/O activity, and ESCAT employs data staging for its out-of-core computations. In the last phase, both codes write final results.

In each of the three phases, I/O activity can be classified across three dimensions: I/O request size, I/O parallelism, and I/O access modes. To clearly establish similarities in the evolution of both codes, we compare the initial access patterns of both applications and the optimized patterns of the final application versions as influenced by the idiosyncrasies of PFS.

6.1. Initial Access Patterns

In the initial version of both codes, at least 98 percent of all reads were small (i.e., less than 1K bytes), although the vast majority of data is read via a small number of large requests (i.e., greater than 128K bytes). In both codes, small writes predominate. This distribution of request sizes reflects the application codes' "natural" I/O needs (i.e., patterns that are natural and intuitive to the application developer based on the code structure and algorithm). Moreover, only standard UNIX I/O calls were used. Conversations with the application developers revealed that they opted for the easiest and most natural implementation for their I/O [19, 5].

Both codes relied on a single node to coordinate parallel read and write operations by all nodes. This partitioning of I/O tasks across nodes is partially an artifact of the codes' previous platforms and partially a consequence of a restricted set of I/O modes. Because there was no equivalent to the `M_GLOBAL` mode on the Intel Delta, where ESCAT was developed, or the Intel iPSC/860, where PRISM was developed, concurrent reads using the `M_UNIX` were the easiest and most natural choice [19]. Likewise, both codes would have benefited from concurrent, synchronized writes by all nodes, but in the first release of PFS no such I/O mode was available, and both developers opted to to perform all writes through node zero [19, 5].

6.2. Optimized Access Patterns

Because there were clear research benefits, both code developers worked assiduously to optimize their application I/O behavior and to exploit Intel PFS features. Most changes optimized the read request sizes.

For ESCAT, the number of small requests, those less than 1K bytes, dropped substantially. After optimization, about 45 percent of reads were large (i.e., 128K bytes) and equal to twice the PFS stripe size, effectively exploiting the data striping of the PFS file system. Moreover, the majority of data (98 percent) is now transferred via these reads. Similarly, the PRISM developer reduced the fraction of small reads, by using a binary data format.

To minimize the time consumed by reading initialization data, both application developers changed the file access scheme, albeit in different ways. In ESCAT, node zero now reads the initialization files and broadcasts the data to the other nodes. In PRISM, the same solution is realized implicitly by opening the files using the Intel `M_GLOBAL` and `M_RECORD` modes.

Writes were optimized in ESCAT by using the `M_ASYNC` mode, eliminating costly file access serialization; the application is then responsible for ensuring that no two nodes concurrently write to the same file region. PRISM uses `M_UNIX` for the writes because they are still administered through node zero.

Overall, the analysis of the three code versions for both ESCAT and PRISM revealed many strengths and weaknesses of PFS. PFS achieves high transfer rates for large request sizes that are multiples of the file stripe size. However, the performance for small requests is quite low[5]. PFS provides collective and non-collective operations that cover a large spectrum of application needs. In addition, PFS offers a wide variety of synchronous and asynchronous operation modes. However, in some cases, significant performance degradation is caused by unexpected behavior of a file access mode, or perhaps inappropriate use of that mode.

7. Conclusions and Future Work

Characterization studies are by their nature inductive, covering only a small sample of the possibilities. The two

[5]To redress these limitations, Intel is developing optimizations to support small accesses.

applications we have studied represent but a few cases from a large space of parallel applications. We believe they are indicative, though they are by no means an exhaustive description of the parallel I/O requirements or behavior exhibited by scalable applications. In this work, we tried to isolate the influence of available technology on the application code. Doubtless, our performance results are conditioned by I/O hardware, system software, and machine configurations.

We conclude that there are many opportunities for improving the performance of current parallel file systems. Parallel I/O standardization is an important step that will greatly contribute to code scalability and portability across different architectures. The need for asynchronous and collective operations is imperative. Request aggregation, prefetching, and write behind are possible approaches. Recognition and parallelization of I/O operations by the compiler will enable the system to reorder and/or cluster I/O requests and will provide substantial leverage to the application developer by reducing the unavoidable burden of code tuning to exploit file system idiosyncrasies.

We are currently broadening our characterization studies. A study of a larger set of applications is underway. Additionally, we plan to examine the effects of different machine configurations (e.g., number of I/O nodes) and different architectures on I/O performance. From these characterizations, a comprehensive set of parallel file system I/O benchmarks will be derived.

Acknowledgments

Phyllis Crandall contributed to the early evaluation of I/O behavior of ESCAT and PRISM, and Chris Elford's analysis of code behavior was invaluable. The electron scattering code (ESCAT) was provided by Carl Winstead and Vincent McKoy at Caltech. The PRISM code was obtained from Ron Henderson and Dan Meiron at Caltech. All data presented here were obtained from code executions on the Intel Paragon XP/S at the Caltech Center of Advanced Computing Research.

References

[1] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A., "RAID: High-performance, Reliable Secondary Storage", *ACM Computing Surveys*, Vol. 26(2), pp. 145-185, June 1994.

[2] Corbett, P., Feitelson, D. G., Prost, J.-P., and Baylor, S. J., "Parallel Access to Files in the Vesta File System", in Proc. *Supercomputing '93*, pp. 472-481, November 1993.

[3] Crandall, P. E., Aydt, R. A., Chien A. A., and Reed, D. A., "Input/Output Characterization of Scalable Parallel Applications", in Proc. *Supercomputing '95*, December 1995.

[4] Henderson, R. D., and Karniadakis G. E., "Unstructured Spectral Element Methods for Simulation of Turbulent Flows", *J. Comput. Phys.*, Vol. 122(2), pp. 191-217, 1995.

[5] Henderson, R. D., Private Communication, January 1996.

[6] Huber, J., Elford, C. L., Reed, D. A., Chien, A., and Blumenthal, D. S., "PPFS: A High Performance Portable Parallel File System", in Proc. *9th ACM International Conference on Supercomputing*, pp. 385-394, July 1995.

[7] Kotz, D. and Nieuwejaar, N., "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload", in Proc. *Supercomputing '94*, pp. 640-649, November 1994.

[8] LoVerso, S. J., Isman, M., Nanopoulos, A., Nesheim, W., Milne, E. D., and Wheeler, R., "*sfs*: A Parallel File System for the CM-5", in Proc. *1993 Summer USENIX Conference*, pp. 291-305, 1993.

[9] Miller, E. L., and Katz, R. H., "Input/Output Behavior of Supercomputing Applications", in Proc. *Supercomputing '91*, pp. 388-397, November 1991.

[10] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C. S., and Best, M., " File-Access Characteristics of Parallel Scientific Workloads", PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995.

[11] Nitzberg, B., "Performance of the iPSC/860 Concurrent File System", RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[12] Pasquale, B. K., amd Polyzos, G., "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload", in Proc. *Supercomputing '93*, pp. 388-397, November 1993.

[13] Pasquale, B. K., amd Polyzos, G., "Dynamic I/O Characterization of I/O Intensive Scientific Applications", in Proc. *Supercomputing '94*, pp. 660-669, November 1994.

[14] Poole, J. T., "Preliminary Survey of I/O Intensive Applications", California Institute of Technology, http://www.ccsd.caltech.edu/SIO/SIO.html, 1994.

[15] Purakayastha, A., Ellis, C. S., Kotz, D., Nieuwejaar, N., and Best, M., "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor", in Proc. 9th International Parallel Processing Symposium, pp. 165-172, April 1995.

[16] Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B. W., and Tavera, L. F., "Scalable Performance Analysis: The Pablo Performance Analysis Environment", in Proc. *Scalable Parallel Libraries Conference*, A. Skjellum (ed.), IEEE Computer Society, pp. 104-113, 1993.

[17] Reed, D. A., Elford C.L., Madhyastha T., Scullin W.H., Aydt, R. A., and Smirni E., "I/O, Performance Analysis, and Performance Data Immersion", in Proc. *MASCOTS'96*, pp. 5-16, 1996.

[18] Winstead, C., Pritchard, H., and McKoy, V., "Parallel Computation of Electron-Molecule Collisions", *IEEE Computational Science & Engineering*, pp. 34-42, 1995.

[19] Winstead, C., Private Communication, January 1996.