Input/Output Characteristics of Scalable Parallel Applications[†]

Phyllis E. Crandall Ruth A. Aydt Andrew A. Chien Daniel A. Reed

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Rapid increases in computing and communication performance are exacerbating the long-standing problem of performance-limited input/output. Indeed, for many otherwise scalable parallel applications, input/output is emerging as a major performance bottleneck. The design of scalable input/output systems depends critically on the input/output requirements and access patterns for this emerging class of large-scale parallel applications. However, hard data on the behavior of such applications is only now becoming available.

In this paper, we describe the input/output requirements of three scalable parallel applications (electron scattering, terrain rendering, and quantum chemistry) on the Intel Paragon XP/S. As part of an ongoing parallel input/output characterization effort, we used instrumented versions of the application codes to capture and analyze input/output volume, request size distributions, and temporal request structure. Because complete traces of individual application input/output requests were captured, in-depth, off-line analyses were possible. In addition, we conducted informal interviews of the application developers to understand the relation between the codes' current and desired input/output structure. The results of our studies show a wide variety of temporal and spatial access patterns, including highly read-intensive and write-intensive phases, extremely large and extremely small request sizes, and both sequential and highly irregular access patterns. We conclude with a discussion of the broad spectrum of access patterns and their profound implications for parallel file caching and prefetching schemes.

1 Introduction

Recent progress building systems whose aggregate computation and communication performance can be economically scaled across a wide range has encouraged application scientists to pursue computational science models that were heretofore considered intractable. Unfortunately, for many scalable parallel applications, the input/output barrier rivals or exceeds

[†]Supported in part by the National Science Foundation under grant NSF ASC 92-12369, by the National Aeronautics and Space Administration under NASA Contracts NGT-51023, NAG-1-613, and USRA 5555-22 and by the Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0039 and DABT63-93-C-0040. Andrew Chien is also supported in part by NSF Young Investigator Award CCR-94-57809.

that for computation. In short, high-performance commodity processors and high-speed networks are necessary but not sufficient to solve many national challenge problems — scalable parallel secondary and tertiary storage systems are needed as well.

Distressingly, input/output and file system research on scalable parallel systems is in its infancy. Moreover, commodity storage technology trends suggest that the disparity between peak processor speeds and disk transfer rates will continue to increase — the commodity disk market favors low cost, low power consumption and high capacity over high data rates. With commodity disks, only disk arrays [2] can provide the requisite peak data transfer rates.

When hundreds of disks and disk arrays are coupled with tertiary storage devices, a multilevel storage management system (e.g., like Unitree), and a broad range of possible parallel file access patterns, the space of potential data management strategies is immense, and identifying optimal or even acceptable operating points becomes problematic. Unfortunately, file system and storage hierarchy designers have little empirical data on parallel input/output access patterns and are often forced to extrapolate from measured access patterns on either traditional vector supercomputers [17, 21, 22] or Unix workstations [20]. Neither of these environments reflects the application usage patterns, diversity of configurations, or economic tradeoffs salient in scalable parallel systems.

The goal of this work is to characterize parallel input/output requirements and access patterns, enabling application developers to achieve a higher fraction of peak input/output performance on existing parallel systems and system software developers to design better parallel file system policies for future generation systems. We analyze the input/output behavior of three parallel applications on the Intel Paragon XP/S: an electron scattering code, a terrain rendering code, and a quantum chemistry code.

These applications represent a snapshot of current input/output practice on scalable parallel machines and reflect the developers' input/output design choices based on perceived and actual limitations of available input/output systems. These initial codes are but a small part of the nascent Scalable Input/Output Initiative's (SIO) code suite [23], and our initial characterization is a first step in a continuing input/output characterization effort.

Our experimental data show that application input/output signatures differ substantially, with a wide variety of temporal and spatial access patterns, including highly read-intensive and write-intensive phases, extremely large and extremely small request sizes, and both sequential and highly irregular access patterns. This data indicates that parallel input/output systems must deliver high performance across broad diversity in application access patterns. Our preliminary experiences with parallel file systems [8, 9] suggests that supporting robust performance requires tuning file system policies to specific access patterns.

The remainder of this paper is organized as follows. In §2–3, we summarize our approach to input/output performance characterization and its relation to the new Scalable I/O Initiative. This is followed in §4 by a brief description of the three application codes and their high-level input/output behavior. In §5–7 we analyze the temporal and spatial input/output patterns of the applications in detail, followed in §8 by discussion of the implications for parallel file system policies. Finally, §9 and §10 describe, respectively, related

work on input/output characterization and a brief summary of our experiences and directions for future research.

2 Background

Though the reasons for input/output in high-performance applications are varied, they can be broadly classified as compulsory, checkpoint, or out-of-core [17]. As the name suggests, compulsory accesses are unavoidable and arise from reading initialization files, generating application output (e.g., scientific data sets or visualizations), or reading input data sets. A high-performance file system can reduce the time needed for these accesses, but they cannot be eliminated by clever cache or memory management schemes.

Checkpoints are necessary because production runs of scientific codes may span hours or even days, the computing resources are typically shared among a large user base, and standard operating practice dictates regular down time for system maintenance. In addition, users often use computation checkpoints as a basis for parametric studies, repeatedly modifying a subset of the checkpoint data values and restarting the computation. The frequency and size of checkpoints is highly application dependent, but a high-performance file system can reduce the cost of checkpointing by exploiting knowledge of checkpoint input/output characteristics.

Finally, out-of-core input/output is a consequence of limited primary memory. Historically, vector supercomputers have, by design, lacked paged virtual memory, and users have managed the limited primary memory by staging data to and from secondary storage. Even on scalable parallel systems with paged virtual memory, many users eschew the convenience of paging for the tight control possible with user-managed overlays and scratch files. Larger primary memories can reduce the number and size of out-of-core scratch files, but not obviate their need — many important problems have data structures far too large for primary memory storage to ever be economically viable.

Within these broad input/output classes, there are wide variations in file access patterns, and such variations have deep performance implications for parallel file systems. Moreover, there are circular dependences between parallel file system efficiency and parallel program access patterns. Parallel file systems are designed based on the system developers' knowledge of extant file access patterns. Often, these patterns are historical artifacts, themselves based on application developers' exploitation of the idiosyncrasies of previous generation file systems. Consequently, it is critical to both quantify current access patterns and understand the reasons for these patterns. Simply put, are the observed access patterns intrinsic to the application algorithms, or are they artifacts of the current software environment? While definitive answers may in general be unobtainable, frank conversations with code developers and analysis of similar applications on different parallel platforms provide a good basis for insight.

Understanding extant parallel file access patterns and developing more effective file system policies is the goal of the Scalable I/O Initiative, a broad-based, multi-agency group

that involves academic researchers, government laboratories, and parallel system vendors.[1] The initiative seeks to develop the technology base needed to support high-performance parallel input/output on future scalable parallel systems. It includes five research working groups: applications, performance characterization (of which the authors are a part), compiler technology, operating systems, and software integration. The three parallel applications described in §4 were obtained from application working group participants.

3 Experimental Methodology

An ideal input/output characterization of an application code includes access patterns and performance data from the application, input/output library, file system, and device drivers. Application file accesses are the logical input/output stimuli; their sizes, temporal spacing, and spatial patterns (e.g., sequential or random) constrain possible library and file system optimizations (e.g., by prefetching or caching). The physical patterns of input/output at the storage devices are the ultimate system response. Minimizing their number and maximizing their efficiency (e.g., by disk arm scheduling and request aggregation) is the final responsibility of the file system and device drivers.

Given performance data from the application and all system levels, one can correlate input/output activities at each level and identify bottlenecks. However, a complete input/output instrumentation of all system levels is a major undertaking that requires in-depth knowledge of operating system structure and access to operating system source. As a prelude to a more detailed instrumentation of system software as part of the Scalable I/O Initiative, we have developed a suite of application input/output software instrumentation and characterization tools. This suite, an extension of the Pablo performance environment [26, 25], brackets invocations of input/output routines with instrumentation software that captures the parameters and duration of each invocation.

3.1 Pablo Input/Output Instrumentation

The Pablo performance environment consists of (a) an extensible performance data metaformat and associated library that separates the structure of performance data records from their semantics, (b) an instrumenting parser capable of generating instrumented SPMD source code, (c) extensible instrumentation libraries that can capture timestamped event traces, counts, or interval times and reduce the captured performance data on the fly, and (d) a group of graphical performance data display and sonification tools, based on the data metaformat and coarse-grain graphical programming, that support rapid prototyping of performance analyses.

To capture and analyze input/output performance data, we have extended the Pablo environment to capture the parameters of application input/output calls on a variety of

[1] See `http://www.ccsf.caltech.edu/SIO/SIO.html` for details. The initial results of the I/O characterization effort is available from `http://www-pablo.cs.uiuc.edu/Projects/IO/io.html`

single processor and parallel systems.[2] To minimize potential input/output perturbations due to performance data extraction, the Pablo instrumentation software supports real-time reduction of input/output performance data in addition to capture of detailed event traces. The former trades computation perturbation for input/output perturbation. Measurements show that the instrumentation overhead is modest for input/output data capture and is largely independent of the choice of real-time data reduction or trace output for post-mortem analysis.

Pablo's real-time input/output performance data reductions include any combination of three summaries: file lifetime, time window, and 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; this window defines the granularity at which data is summarized. File region summaries are the spatial analog of time window summaries; they define a summary over the accesses to a file region. Finally, general input/output statistics computed off-line from event traces provide means, variances, minima, maxima, and distributions of file operation durations and sizes.

3.2 Intel Paragon XP/S

Using the Pablo performance instrumentation software, we measured application input/output performance on the Intel Paragon XP/S [27] at the Caltech Concurrent Supercomputing Facility (CCSF). At the time our experiments were conducted, the system had 512 computation nodes and 16 I/O nodes, each with a RAID-3 disk array composed of 5 1.2GB disks. The software environment consisted of several versions of Intel OSF/1 1.2 with PFS, Intel's parallel file system.

PFS stripes files across the I/O nodes in units of 64 KB, with standard RAID-3 striping on each disk array. In addition to file striping, PFS supports six parallel file access modes:

- `M_UNIX`: each node has an independent file pointer,
- `M_LOG`: all nodes share a file pointer, node accesses are first come first serve, and input/output operations are variable length,
- `M_SYNC`: all nodes share a file pointer and accesses are in node number order,
- `M_RECORD`: each node has an independent file pointer, access is first come first serve and input/output operations are fixed length,
- `M_GLOBAL`: all nodes share a file pointer, perform the same operations and access the same data, and
- `M_ASYNC`: each node has an independent file pointer, access is unrestricted and variable size, and operation atomicity is not preserved.

We will return to these modes in §5–7 when discussing their use in application codes.

[2]This software is available at `http://www-pablo.cs.uiuc.edu`.

4 Application Code Suite

As we noted earlier, one of the primary goals of the new national Scalable I/O Initiative is analyzing the input/output patterns present in a large suite of scientific and engineering codes. These span a broad range of disciplines, including biology, chemistry, earth sciences, engineering, graphics, and physics [23]. Despite large differences in their underlying algorithms, the codes share two features. First, each code runs on one or more scalable parallel systems, permitting cross-machine comparisons of input/output performance. Second, all codes have both high input/output and computational requirements. In short, they typify large-scale scientific and engineering computing.

We have selected three codes from this suite as an initial focus of our input/output characterization effort. In the following subsections we briefly describe the algorithms underlying the three applications, the code structure, and its input/output organization. In §5–7, we examine the input/output patterns in greater detail and discuss their implications for file system design.

4.1 Electron Scattering (ESCAT)

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 [30]. The scattering probabilities are obtained by solving linear systems whose terms include a Green's function which has no analytic form and is evaluated by numerical quadrature. Generation of the quadrature data is compute-intensive, and the size of the data set is highly variable depending on the nature of the problem. The quadrature is formulated to be energy independent so it can be used to solve the scattering problem at many energies.

ESCAT is a parallel implementation of the Schwinger multichannel method written in a combination of C, FORTRAN, and assembly language. From an input/output perspective, there are four distinct execution phases. First, a compulsory read loads the problem definition and some initial matrices. Next, all nodes participate in the calculation and storage of the requisite quadrature data set, with each node processing a different set of integrals. This phase is compute-intensive and is composed of a series of compute/write cycles with the write steps synchronized among the nodes. Memory limitations and the desire to checkpoint the quadrature data set for reuse in later executions prompt the writes during this phase. The third phase involves calculations that depend on the collision energy. In it, energy-dependent data structures are generated and combined with the reloaded quadrature data set to form the system of linear equations. In the last phase, the linear system matrices are written to disk for later solution on another machine.

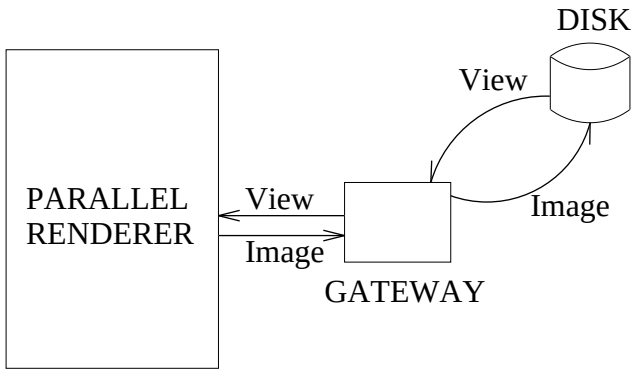

Figure 1: Rendering algorithm organization

4.2 Terrain Rendering (RENDER)

NASA's deep space imaging and Earth observation satellites have obtained multi-spectral data of Mars and Venus, as well as earth. Combining satellite imagery with terrain elevation data can produce intuitive, three-dimensional perspective views of the planetary surfaces. By generating these views in real-time, it is possible to conduct a "virtual flyby" of the planetary surface where scientists can interactively examine false color terrains from a variety of positions and orientations, supporting rapid exploration of large data sets. A parallel ray-identification algorithm [15] distributes terrain data among processing nodes, decomposing via the natural data parallelism, and exploiting positional derivatives to vary rendering resolution. Together these techniques achieve several frames per second on gigabyte data sets, approaching the ten frames per second needed for real-time animation.

The RENDER code is a hybrid control and data parallel implementation of the ray identification rendering algorithm; Figure 1 shows its high-level structure. A single gateway node manages a group of parallel rendering processes and begins by reading the initial data set. The initial read is followed by a read-render-write cycle for each of the subsequent view perspectives (frames). In this loop, the gateway inputs view perspective requests, directs rendering tasks to produce the view, collects rendered views from the group of rendering tasks, and outputs frames to either secondary storage or a HiPPi frame buffer. Thus, RENDER's input/output activity consists of a compulsory read of the initial data set, a series of reads of view coordinates, and corresponding writes of the rendered frames.

4.3 Hartree Fock (HTF)

Ab initio chemistry calculations are the key to a detailed understanding of bond strengths and reaction energies for chemical species. Moreover, they allow chemists to study reaction pathways that would be too hazardous or too expensive to explore experimentally. This version of the Hartree Fock algorithm calculates the non-relativistic interactions among atomic nuclei, electrons in the presence of other electrons, and electrons interacting with nuclei. Basis sets derived from the atoms and the relative geometry of the atomic centers are the

initial inputs. Atomic integrals are calculated over these basis functions and are used to approximate molecular density. This density and the previously calculated integrals are used to compute the interactions and to form a Fock matrix. A self consistent field (SCF) method is used until the molecular density converges to within an acceptable threshold.

The Hartree Fock implementation we studied consists of three distinct programs totaling roughly 25K lines of Fortran. The three programs operate as a logical pipeline, with the second and third accepting file input from the previous one. The first program, *psetup*, reads the initial input, performs any transformations needed by the later computational phases, and writes its result to disk. The next program, *pargos*, calculates and writes to disk the one and two-electron integrals. The final program, *pscf*, reads the integral files multiple times (they are too large to retain in memory) and solves the SCF equations. In subsequent sections, we refer to these three programs as initialization, integral calculation, and self-consistent field calculations.

With these brief descriptions of the electron scattering, parallel rendering, and Hartree Fock codes, in §5–7 we examine the detailed patterns of input/output present in each and discuss the implications for file systems design.

5 Electron Scattering Input/Output Behavior

To accurately assess the input/output patterns of the electron scattering code, we used a data set large enough to capture typical behavior but small enough to permit parametric studies of different code versions. On 128 nodes with this data set, the ESCAT code executed for roughly one and three quarter hours. Production data sets generate similar behavior, but with ten to twenty hour executions on 512 processors.

Succinctly, the dominant input/output behavior in the current version of the ESCAT code is small writes, and most of the time is spent computing. During initialization, a single node uses the `M_UNIX` mode to read the initialization data and broadcast it to the other nodes. In the major execution phase, each node repeatedly seeks and then writes quadrature data to intermediate staging files. Near the end of execution, the nodes reload the previously written data, with each node rereading the same quadrature data that it wrote. As we shall see in §5.2, this software organization is largely due to the constraints system performance places on the application developers — not only would they prefer a different program organization, the problem they wish to solve requires dramatically greater input/output performance.

5.1 Experimental Data

Tables 1–2 are a high-level summary of the input/output behavior of the ESCAT code. During the roughly 6,000 seconds of execution, the total volume of input/output data is only 60M bytes, or 10K bytes/second. Read operations represent 56 percent of the input/output volume, but only two percent of the operations and 0.2 percent of the input/output time. As Figure 2 and Table 2 show, read sizes are bimodal, with roughly equal numbers of small

Operation	Operation Count	Volume (Bytes)	Node Time (Seconds)	Percentage I/O Time
All I/O	26,418	60,983,136	38,788.95	100.00
Read	560	34,226,048	81.19	0.21
Write	13,330	26,757,088	16,268.50	41.94
Seek	12,034	-	20,884.11	53.84
Open	262	-	1179.06	3.04
Close	262	-	376.06	0.97

Table 1: Number, size, and duration of I/O operations (ESCAT)

Operation	Operation Size			
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	297	3	260	0
Write	13,330	0	0	0

Table 2: Read/write sizes (ESCAT)

and large read requests. The overheads for writes and seeks dominate other input/output operations, representing almost 96 percent of the total input/output time.

As we noted earlier, the ESCAT code has four distinct read/write phases. These phases are clearly visible in Figures 2–3, which show timelines of the ESCAT reads and writes. In the first phase, the initial data is read from three files by node zero and broadcast to the remaining nodes. During the second phase, all nodes repeatedly compute, synchronize, and then write 2 K bytes of quadrature data to two intermediate staging files – one file for each of the two possible collision outcomes contained in our test data. To simplify reloading of the data in the next phase, each node seeks to a calculated offset dependent on the node number, iteration, and PFS stripe size before writing the data. Intel's `M_UNIX` file mode is used for these writes.

In the third input/output phase, the previously written quadrature files are read by all the nodes using the Intel `M_RECORD` mode.[3] Finally, in the last input/output phase, data is sent to node zero by all other nodes and written to three output files.

As can be seen in Figure 2, read operations occur only in the first and third phases. The first spike in Figure 2 is the initial, compulsory data input; the phase three read operations at the far right of the figure are the staging of the previously computed quadrature data. Figure 3 shows the initial input phase in greater detail, capturing the variety of access sizes and temporal irregularity of the requests.

[3] Recall that in this mode each node has a separate file pointer, but the nodes must read fixed size records in first-come-first-serve order.

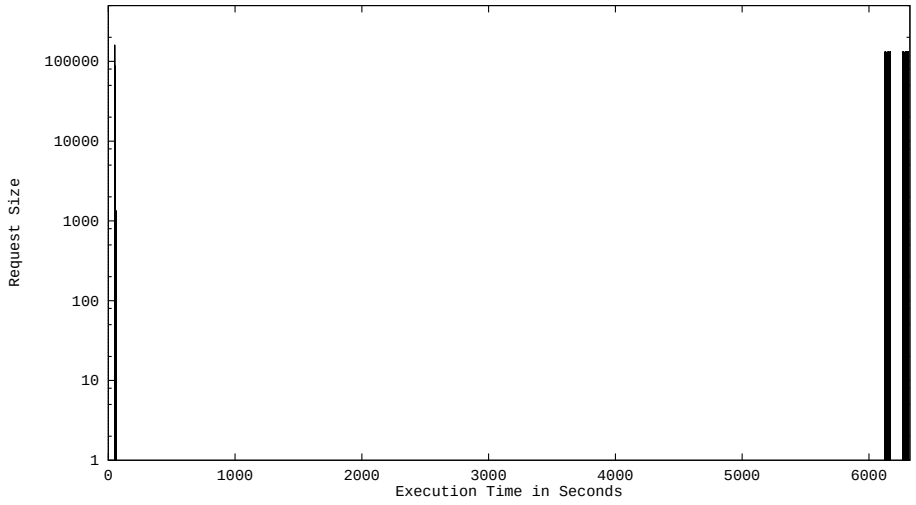

Figure 2: Read operation timeline (ESCAT)

The tight clustering of the quadrature data writes by all the processors is evident in Figure 4. The temporal spacing of the groups decreases as the quadrature calculation phase proceeds, ranging from roughly 160 seconds near the beginning of the phase to half that near the end. Table 1 shows that seek overhead is a major contributor to the temporal dispersion of each group.

Finally, Figure 5 shows when each of the input and output files was accessed during the ESCAT execution. In the figure, diamonds denote reads, and crosses denote writes. Three files with the identifiers 9, 10, and 11 contain the initial input data, two other files with identifiers 7 and 8 are used for staging the quadrature data, and the final output is written to the files with identifiers 3, 4, and 5.

5.2 Discussion

The implementation and input/output behavior of the ESCAT electron scattering code highlight the disparity between the ideal and current practice. At first glance, it appears that the input/output is inefficient, but seemingly of little import because the code is heavily computation bound. Indeed, for current data sets, this is true. However, the complexity of the quadrature data volume grows as $\mathcal{O}(N^3)$, where N is the number of electron scattering outcomes. Conversations with McKoy *et al* reveal that for current problems, with $N \approx 10$, computation dominates. This reflects their pragmatic need to attack only solvable problems, Their research goal is $N \approx 50$, or two orders of magnitude more data. In short, research practice and the behavior of this code would change dramatically were higher performance input/output possible.

Not only is the current problem size constrained by input/output limitations, the ESCAT code's input/output behavior is constrained as well. Although the initial data is needed by

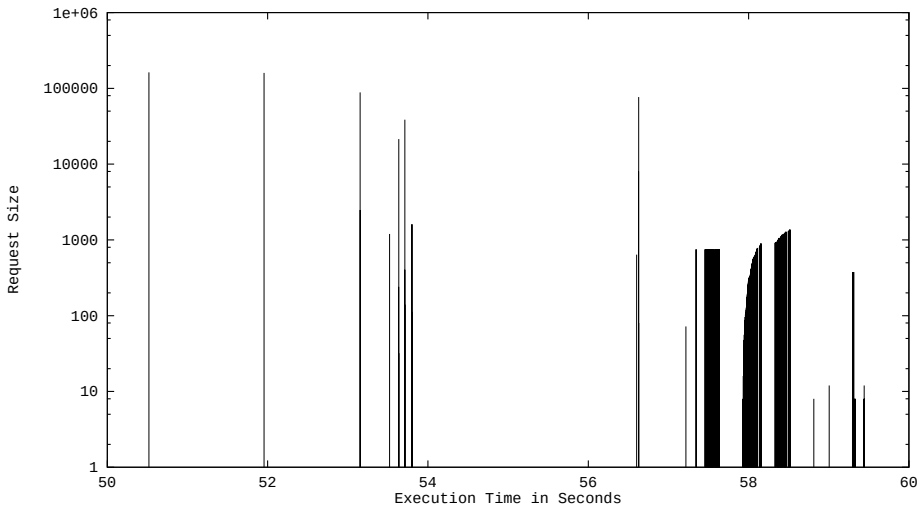

Figure 3: Read operation detail (ESCAT)

all processors, the application developers discovered by experimentation that it was more efficient for a single node to read the initial data and then use the communication network to broadcast the data than it was for each node to independently read the initialization data. More efficient support for parallel reading of complete files would simplify the application, eliminating the need to write data distribution code.

The decision to use the Intel `M_UNIX` mode to write the quadrature data, rather than the `M_RECORD` mode, was driven by the desire to reduce the time needed to read the data. On writes, the `M_RECORD` mode generates a sequence of fixed size records that appear to have been written in node order (i.e., for N nodes, the file consists of groups of N records, with each group written in node order). However, in ESCAT, the data written by a given node is later read by that same node, requiring it to be contiguous if the node is to read it efficiently with a single large access. To efficiently support accesses of this type, either a richer set of file modes is needed, or the application must be redesigned.

Finally, because the input/output in this application is dominated by small writes, read prefetching would benefit little. In contrast, write request aggregation and write behind could dramatically reduce the output cost. To quantify these effects, we ported the ESCAT code to PPFS, our portable parallel file system [8], and configured the file system with write behind and global request aggregation policies. This combination of policies effectively eliminated the behavior seen in Figure 4. In our experience, this type of optimization (i.e., choosing file policies based on access pattern knowledge), is the key to maximizing input/output performance.

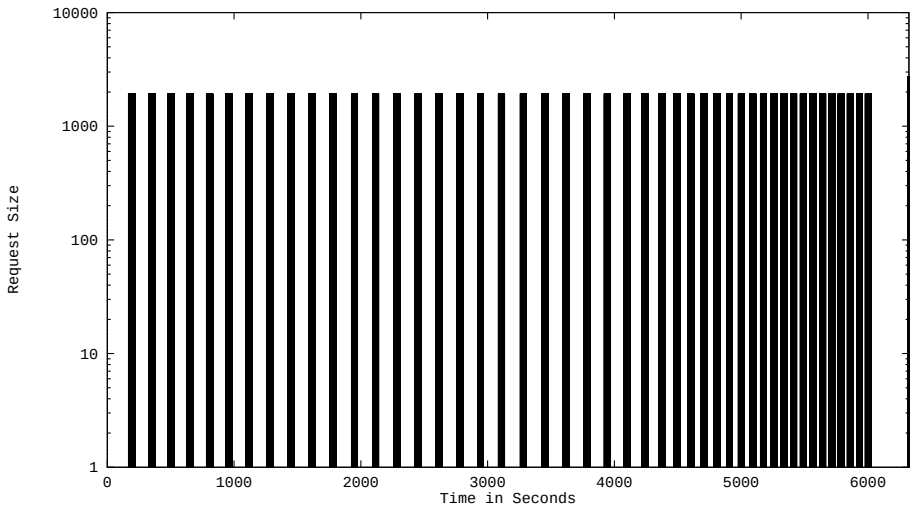

Figure 4: Write operation timeline (ESCAT)

6 Terrain Rendering Input/Output Behavior

To assess the input/output behavior of the terrain rendering code, we used a full production data set (Mars flyby data from the Viking mission), but abbreviated the run by limiting the number of frames rendered. Beyond this point in the computation, the RENDER code performs periodic output of frames, of fixed size and at nearly fixed time intervals. In addition, in actual production use, all of this output would be directed to a HiPPi frame buffer, not the file system. On 128 nodes, the production data set required roughly eight minutes to initialize and output one hundred views (frames). Full production runs consist of 5000 or more frames and execute for approximately thirty minutes. These production runs generate identical initial input/output requirements, extending only the reading of views to render and output views.

Overall, the dominant file input/output requirement of the current version of RENDER is the initial read of a gigabyte data set, followed by compute-limited rendering phases. During initialization, a single node uses the `M_UNIX` file mode to read the entire data set and broadcast it to the rest of the nodes. During the major computation phase (rendering), view coordinates are retrieved from a control file (small reads), and the rendering of each view produces a single large write of the rendered image to the HiPPi frame buffer. As we shall see in §6.2, the input/output structure is the product of restrictions on the file system input/output modes; the developers would like to exploit file system features, but they cannot.

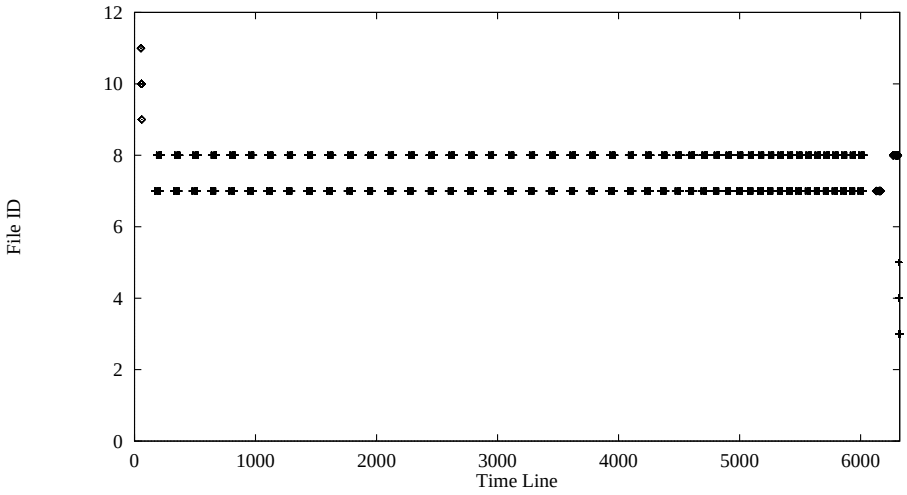

Figure 5: File access timeline (ESCAT) (Crosses denote writes; diamonds denote reads)

6.1 Experimental Data

Tables 3 and 4 contain a high-level summary of the input/output behavior of RENDER. Over the 470 seconds of execution, the total volume of input/output is nearly one gigabyte, dominated by the read of the initial data set.

Read operations dominate the input/output, accounting for 89 percent of the input/output volume and 64 percent of the input/output requests. The read traffic is dominated by asynchronous reads both in volume and in the number of requests. However, the read requests take only a small fraction of the input/output time – reads and asynchronous reads account for 0.17 and 4.6 percent of the input/output time respectively. For asynchronous reads, the measured time is only the cost of issuing the read, some of the input time may be overlapped — the part not overlapped appears as `iowait` time. In this code, the `iowait` time is the major fraction of the I/O time. Read sizes are again bimodal, clustered below 4K bytes and above 256K bytes. Large reads dominate both in number of requests and in volume. The writes account for nearly 20 percent of the input/output time, though accounting for only a small fraction of the input/output volume.

The RENDER code has two basic phases, initialization and rendering. These phases are not only distinguished computationally, they have dramatically different input/output patterns as shown in Figures 6 and 7, timelines of RENDER read and write requests. In the first phase, the terrain data is read in from four files by the gateway node using `M_UNIX` mode. These requests are extremely large, clustered at 3 megabytes and 1.5 megabytes. The input data is then broadcast to the other 128 nodes which form the renderer, each selecting an appropriate subset of the data. The initial phase ends around 210 seconds into the run, and the rendering phase begins. The gateway node reads views from a control file, directing the renderer to produce the requisite image. The data for each view (640x512 24-bit color image,

Operation	Operation Count	Volume (Bytes)	Time (Seconds)	Percentage I/O Time
All I/O	1504	979,162,982	164.75	100.00
Read	121	8457	0.17	.10%
AsynchRead	436	880,849,125	4.60	2.79
I/O Wait	436	-	88.44	53.68
Write	300	98,305,400	31.76	19.28
Seek	4	0	.13	0.08
Open	106	-	32.78	19.90
Close	101	-	6.87	4.17

Table 3: Number, size, and duration of I/O operations (RENDER)

Operation	Operation Size			
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	121	0	0	436
Write	200	0	0	100

Table 4: The sizes of reads and writes in RENDER.

approximately one megabyte) is collected by the gateway and output in a single request. In our runs, this data is written to disk using `M_UNIX` mode, but in a production run, this data would be sent directly to a HiPPi frame buffer.

Figure 6 clearly shows the large read request sizes generated by the initialization phase. The first set of read requests are 3 megabytes, then the size decreases to 1.5 megabytes. At 210 seconds, there is a pronounced transition to the render phase, and the only read requests are small requests to read the view coordinates. Figure 7 shows the write behavior of RENDER, and also reflects the phase structure of the code. There is no write traffic in the initialization phase, and in the rendering phase, write requests consist exclusively of the writes of the one megabyte color images.

Figure 8 shows the file activity for RENDER, and also clearly reflects the two phase structure of the RENDER code. The critical read initialization phase accesses primarily four files (the data set). The control file (views) is accessed in both phases, but heavily in the rendering phase. The output files are each accessed only once (written in their entirety) accounting for the staircase structure.

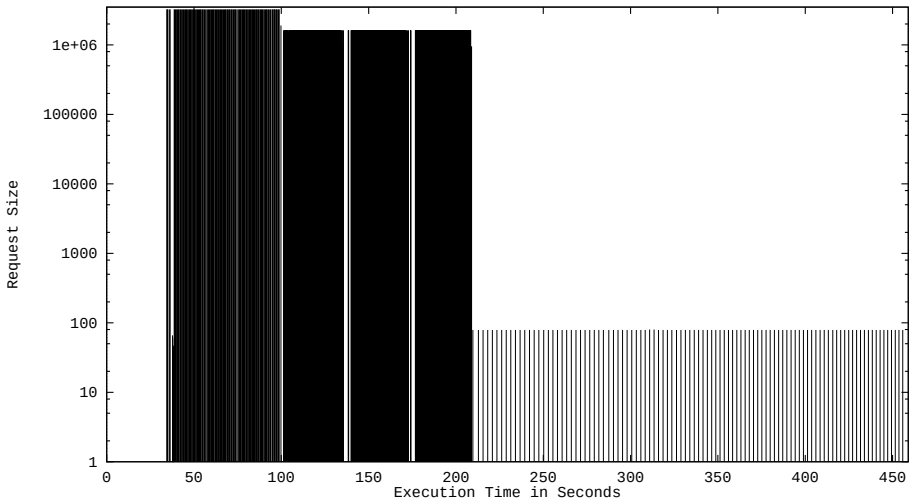

Figure 6: Read operation timeline (RENDER)

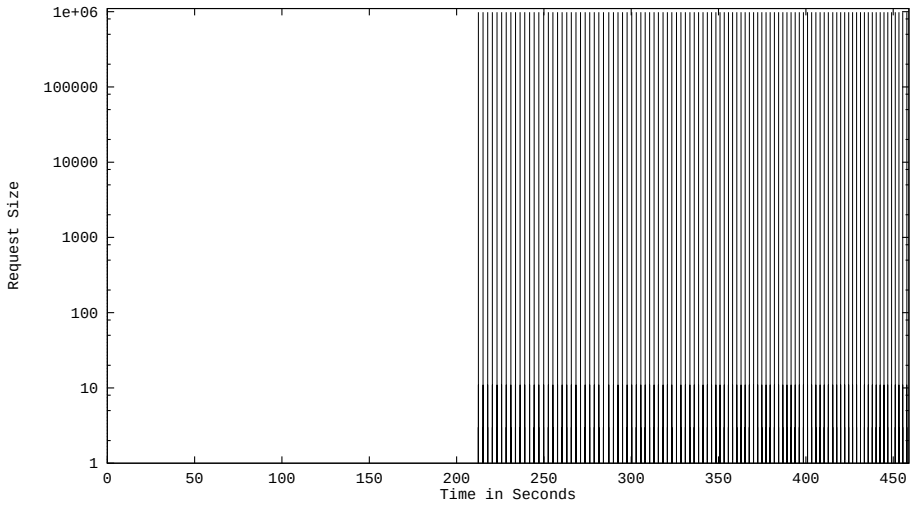

Figure 7: Write operation timeline (RENDER)

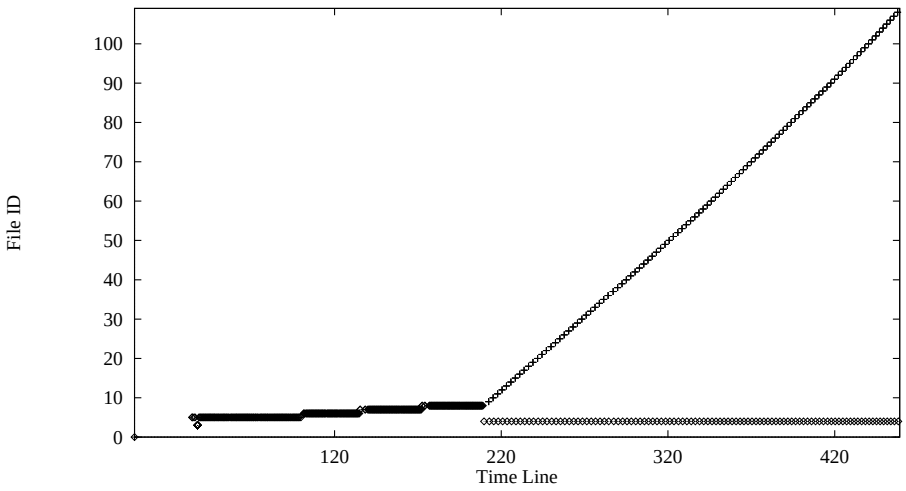

Figure 8: File access timeline (RENDER) Cross marks indicate reads and diamonds indicate writes.

6.2 Discussion

The RENDER code illustrates how scalable parallel systems can enable new classes of applications which in turn engender new challenges for system designers. Increased computational power and memory capacity enable interactive visualization of multi-gigabyte data sets, which introduces the complexity of real-time requirements and on-line output, streaming to a frame buffer. The RENDER input/output pattern fits a classic scientific computing input/output stereotype: large initial read, followed by the writing of output results.

The RENDER code and data set described are matched to the capabilities of currently available systems; however, such scientific visualization applications present much larger computational and input/output challenges. Because RENDER is used for visualizing NASA sensor data, the resolution of input data sets (and their size) is limited by sensor resolution and available data base sizes. Examples include LANDSAT, Mars (Viking), and Venus (Magellan). These data sets currently range from 100's of megabytes to 100's of gigabytes, but with increases in sensor resolution and deployment of systems such as the Earth Observation System, much larger data bases (terabytes) are becoming available. Larger data bases increase the size of the input burst for initialization, and terabyte data bases may require the adoption of out-of-core algorithms. Current images are output with a resolution of 640x512 with 24-bit color; with higher resolution data bases and higher output resolutions (3000x2000), corresponding increases in the computation and output are required. Finally, the current system requires several seconds per frame, but higher frame rates (ten or as high as thirty) are desirable. More directly, higher input/output performance is required for larger data sets and higher resolution output with this code.

The RENDER code performs only sequential file access, and all the input/output is me-

diated by the gateway node. The gateway node reads the data set, then broadcasts it. Although the basic data distribution is block-cyclic, and therefore apparently well-matched to the PFS `M_RECORD` mode, this mode requires all of the nodes to participate. The code developers eschewed the use of `M_RECORD` because RENDER uses nodes asymmetrically (gateway and renderer); not all nodes need to participate. Hence, use of `M_RECORD` mode would require some additional code restructuring and data shuffling regardless.

Using sequential input/output, the code explicitly prefetches initial file data by using asynchronous reads and initiates large read requests, but only achieves a read throughput of approximately 9.5 megabytes/second. Parallel access using the `M_UNIX` mode was empirically determined not to improve code performance. RENDER would benefit from efficient parallel file access modes that allow node subsets to participate without requiring shared file pointers. Another approach, eschewed by the developers, is the use of separate input files for each node. Though this might improve input performance, it incurs additional preprocessing steps, expensive for large data bases, and binds the data base representation to the machine configuration. Efficient parallel access modes that give the effective performance of this change without requiring logical reorganization across files in the application are desirable. Finally, while occasionally single frames or collections of frames might be written to disk, the typical use for RENDER is to write the output data to a HiPPi frame buffer. This presents another dimension of *streaming* input/output which has not yet received much attention in the scalable systems community.

7 Hartree Fock Input/Output Behavior

As we noted in §4.3, the Hartree Fock (HTF) application is composed of three codes: an initialization (*psetup*), an integral calculation (*pargos*), and a self-consistent field calculation phase (*pscf*). For our rather small input data set of 16 atoms, the respective execution times of the three program components were 127, 1173, and 1008 seconds on 128 nodes of the Intel Paragon XP/S, with slightly less than 20 percent of that time consumed by input/output operations. As with the ESCAT code, we shall see that this is not the desired ratio, merely what is currently feasible.

The first code in the HTF application reads a small initial data file and transforms it for use by the later phases. The second, integral calculation phase creates the files of integrals that are consumed by the third, self-consistent field calculation phase. The Intel `M_UNIX` file mode is used exclusively in all three codes.

7.1 Experimental Data

Tables 5–6 summarize the input/output behavior of the three HTF application components. During the initialization phase, the reads and writes are small and occur as initial data is read and transformations are written for use by the ensuing phases. In the integral calculation phase, the number of integrals determines the input/output data volume — quadrature data is written for each integral. Because a Fock matrix of size N generates

Operation	Operation Count	Volume (Bytes)	Node Time (Seconds)	Percentage I/O Time
HTF Initialization				
All I/O	832	7,267,422	55.23	100.00
Read	371	3,522,497	15.34	27.77
Write	452	3,744,872	5.50	9.96
Seek	2	53	0.43	0.78
Open	4	-	31.49	57.02
Close	3	-	2.47	4.47
HTF Integral Calculation				
All I/O	17,854	698,992,502	6,398.03	100.00
Read	145	34,393	0.47	0.00
Write	8,535	698,958,109	1996.4	31.20
Seek	130	0	0.14	0.00
Open	130	-	4056.60	63.40
Close	129	-	11.43	0.18
Lsize	128	-	15.27	0.24
Forflush	8,657	-	317.72	4.98
HTF Self-Consistent Field Calculation				
All I/O	52832	4,205,483,650	32,800.99	100.00
Read	51499	4,201,634,304	32,263.20	98.36
Write	207	3,849,268	5.88	0.02
Seek	813	3,495,198,798	1.67	0.00
Open	157	-	518.74	1.58
Close	156	-	11.50	0.04

Table 5: Number, size, and duration of I/O operations (HTF)

$\mathcal{O}(N^2)$ one electron and $\mathcal{O}(N^4)$ two electron integrals [6], the data volume grows dramatically with matrix size and is substantial even for small matrices. This phase is, therefore, write intensive, which shows clearly in Table 6 and Figure 12. The final, self-consistent field calculation phase is quite read intensive, with each node repeatedly reading the integral files.

Despite the substantial input/output, the maximum request size is rather small, only four times the Intel PFS striping factor of 64K bytes. Moreover, the number of requests smaller than 4K bytes is non-trivial. In short, the request size distribution of Table 6 is bimodal, though skewed toward larger requests, the opposite of the size distribution shown in Table 2 for the ESCAT code

Figures 9–13 and 10–14 show, respectively, read and write request sizes as a function of time. The write intensities of the integral calculation phase and read intensity of the self-consistent field computation phase are striking on this time scale. With this data set, each of the nodes writes roughly 5M bytes of data during the integral calculation. Figures 15–17

Operation	Operation Size			
HTF Initialization				
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	151	220	0	0
Write	218	234	0	0
HTF Integral Calculation				
Read	143	2	0	0
Write	2	1	8,532	0
HTF Self-Consistent Field Calculation				
Read	165	109	51225	0
Write	43	158	6	0

Table 6: Read/write sizes (HTF)

show that each node writes the integral data to a separate file; the nodes then read this data during the final calculation phase.

7.2 Discussion

Quite clearly, the HTF code has substantial input/output requirements even for what is, by current computational chemistry standards, a rather small problem of 16 atoms. In general, though, the input/output pattern in this code is quite regular, with little but sequential accesses except in the final calculation phase.

In conversations with the code developers, we discovered that this is the version of the code they would like to use for larger, more interesting problems. By precomputing the integrals and reusing the quadrature data as needed, the computation requirements can be reduced dramatically. Unfortunately, because the input/output requirements grow as the number of two electron integrals (i.e., as $\mathcal{O}(N^4)$), this is not feasible with current input/output systems. Instead, the integrals are recomputed as needed, substantially increasing the computation requirements but reducing the input/output costs and, with current input/output software, the total execution time.

For integral input/output to be preferable to recomputation, reading an integral from secondary storage must take less than the roughly 500 floating point operations needed for integral calculation. For current systems, this requires a sustained input/output rate of approximately 5–10 Mbytes/second per node. With current and projected disk technology, this implies a system with a disk or disk array directly attached to each processor. Moreover, as processor speeds increase, the input/output rate must increase commensurately, else recomputation becomes the preferred alternative. Simply put, this application requires high storage capacity and high throughput for simple access patterns.

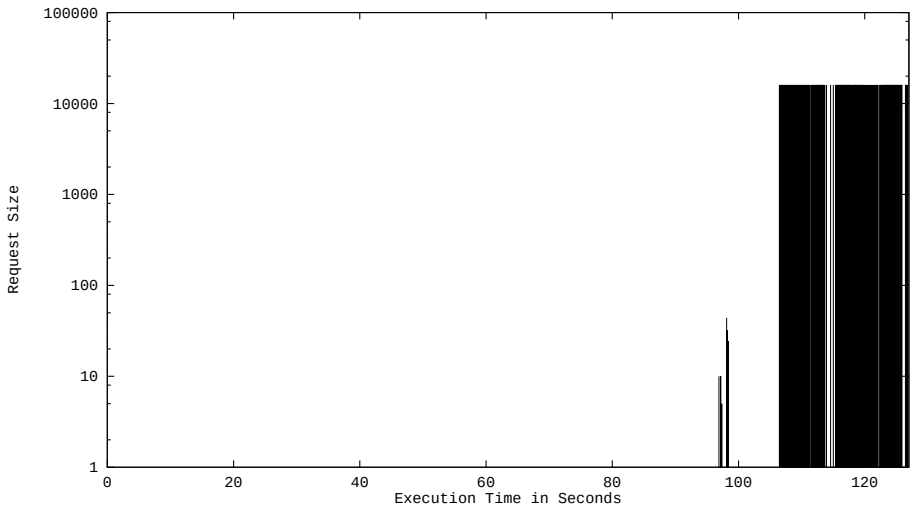

Figure 9: Read operation timeline (HTF initialization)

8 Parallel File System Implications

The most significant observation from our study is that the input/output requirements of scientific codes (electron scattering, terrain rendering and quantum chemistry) greatly exceed the capabilities of existing scalable systems. Scientific applications have input/output patterns and requirements more complex than simple stereotypes, and these requirements are extremely challenging. The mismatch between desired and currently available input/output performance has two important consequences for application scientists: it complicates application code structure, and it reduces the scope of experiments computationally feasible. For parallel systems vendors and file system designers, it is clear that improvements in scalable parallel input/output capabilities can enable or even catalyze advances in science and scientific computing.

All three applications that we studied exhibited a wide variety of read/write mixes and request sizes, with the latter ranging from a few bytes to several megabytes. In short, to provide robust performance, parallel file systems must efficiently support a variety of request size and read/write mixes. However, the performance characteristics of current input/output systems favor large requests because high bandwidths are achieved through parallelism. Consequently, achieving good input/output performance for applications that make small requests admits two basic possibilities: programmers can manually aggregate requests or file systems (and user level libraries) can transform request streams via caching or prefetching, serving as impedance matchers between the application access patterns and disk performance characteristics. The latter approach is promising, and demonstrations of the effectiveness such approaches are appearing [8].

Even for our set of only three application codes, no simple characterization of input/output request sizes or access patterns is viable. Further, studies show that the detailed spatial and

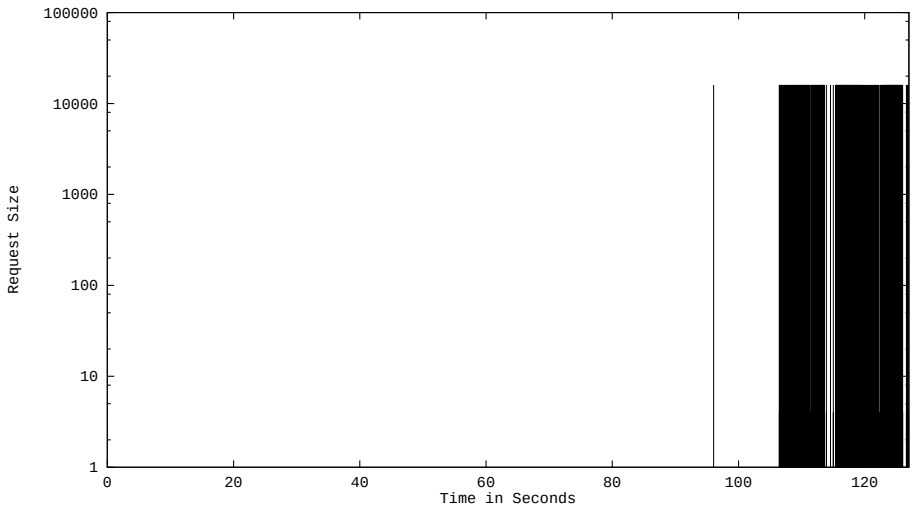

Figure 10: Write operation timeline (HTF initialization)

temporal characteristics of the input/output critically affect input/output performance. We believe this indicates that the simple synthetic kernels often used to evaluate new file system ideas may not be good predictors of potential performance on full-scale applications. The impact of file system changes on real applications or application mixes depends on much more complex application structure, suggesting that the development of larger application skeletons and workload mixes are an essential part of developing high performance input/output systems.

One characteristic of all three applications is that data files were generally read or written in their entirety, in many cases by a single node. This reflects the current inclination of application programmers to control the mapping of files to disks (for performance) and express input/output in ways that are independent of parallel file system features (for portability). For example, in several cases, input/output is done by a single node sequentially, followed by data broadcast through the interconnection network. Such I/O patterns could be expressed as collective operations [1, 5, 11] to allow the filesystem to optimize performance. In general, these observations point out the importance of developing standard parallel file system API's; not only to provide functional portability, but also to provide performance portability. Being able to run the code on several platforms is not enough, performance optimizations on one platform must be portable to others. This is an important guiding principle for file system implementors, but a difficult challenge in the face of the wealth of possible input/output configurations and the richness of their performance space. In short, parallel file system interfaces need to become portable and robust if applications are to adopt and exploit them.

Another common characteristic of the codes is that most of the data written eventually was propagated to secondary storage. This characteristic has been observed previously on supercomputing systems, and differs markedly from Unix file systems where statistics

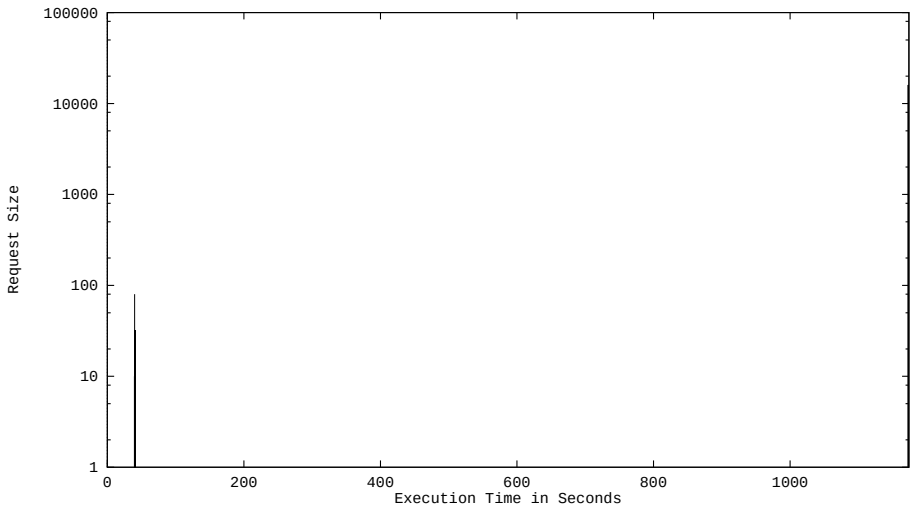

Figure 11: Read operation timeline (HTF integral calculation)

generally record many small short-lived temporary files. If all output data survives to disk, the objective of write caching in the file system must be to increase the achieved bandwidth of the physical input/output system, not to reduce the input/output volume to disk. For example, aggregation of small request to transfer sizes efficient for disks (or internally parallel RAID's) is critical to achieving a large fraction of peak performance. Aggregation is feasible; as an example, the ESCAT code employs multiple writers into disjoint locations in a shared file. Individually, these requests would utilize a disk poorly, however they can be combined, significantly increasing disk efficiency [8]. This experience suggests that in some cases, two level buffering at compute nodes and input/output nodes can be beneficial.

Characterization studies are by their nature inductive, covering only a small sample of the possibilities and attempting to extract more general patterns. The three applications we have studied represent but a few samples from a large space of parallel applications. We believe they are indicative, though they are by no means an exhaustive description of the parallel input/output requirements or behavior exhibited by scalable parallel applications. In addition, it is always difficult to determine how much a code has been influenced by the available technology. Doubtless our characterization results are conditioned by both the parallel input/output hardware, system software, and even machine configurations available. In an attempt to access these dependences, we are currently broadening our input/output characterization studies to applications on other hardware platforms.

9 Related Work

The recent, widespread availability of scalable parallel systems has stimulated development of input/output intensive parallel applications and highlighted the critical need for under-

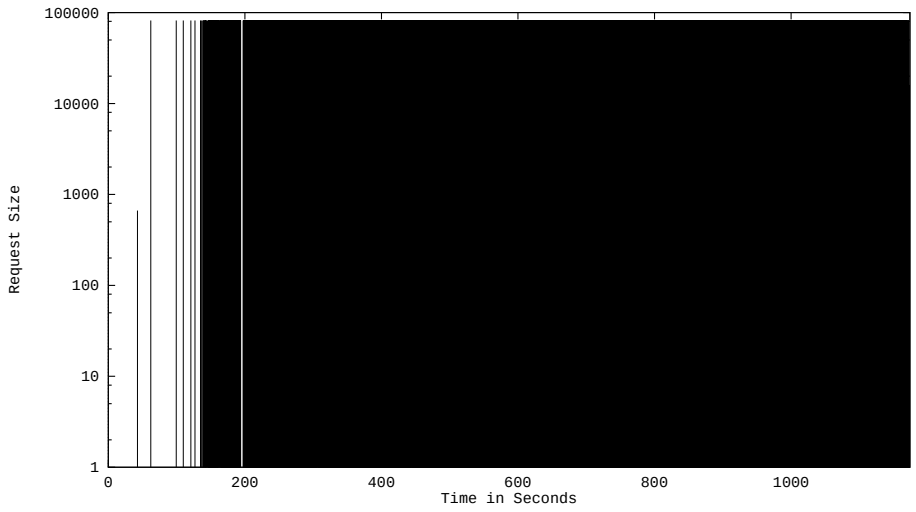

Figure 12: Write operation timeline (HTF integral calculation)

standing parallel file access patterns. Though our understanding of input/output parallelism is still in its infancy, there is a long history of file access characterization for mainframes and vector supercomputers.

Much of the early work considered whole file access characteristics, including file sizes, lifetimes, and reuse intervals. Notable examples of this work include Lawrie and Randell's study [14] of automatic file migration algorithms for the CDC Cyber 175, Stritter's analysis [29] of file lifetime distributions, Smith's study [28] of file access behavior on IBM mainframes, and Reed and Jensen's study [10] of accesses to NCSA's file archive.

More recently Miller and Katz [17] captured detailed traces of application file accesses from a suite of Cray applications from the National Center for Atmospheric Research (NCAR). They observed that many access patterns were sequential and cyclic and that input/output operations could be classified as compulsory, checkpoint, and data staging. Pasquale and Polyzos [21, 22] considered the static and dynamic file access characteristics of production vector workloads at the San Diego Supercomputer Center (SDSC) and concluded that most input/output intensive applications had regular behavior.

The work by Kotz *et al* [12, 24] is closest in spirit to our characterization effort. Using instrumentation in the input/output libraries of the Intel iPSC/860 and the Thinking Machines CM-5, they captured traces of individual file operations and analyzed the data to extract access patterns. The concluded that file access parallelism was important, file caching was important for certain access patterns, and that contrary to intuition, small requests are quite common in large scientific codes.

Our work differs from all of these in considering traces for individual application programs at the level of individual file accesses, coupled with an analysis of the application source code and conversations with application developers to understand the reasons for the access

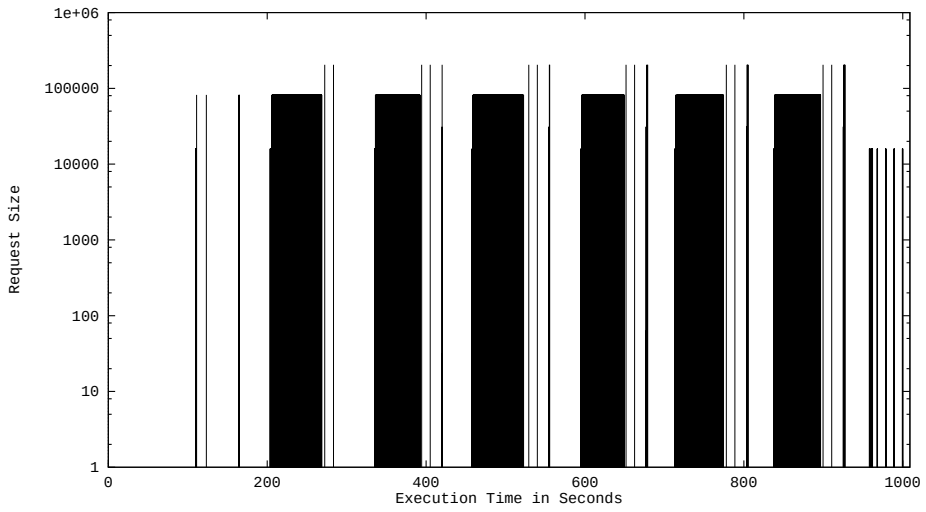

Figure 13: Read operation timeline (HTF self-consistent field calculation)

behavior. Given the limitations of current parallel file systems, extrinsic knowledge is critical to understanding if certain access patterns are inherent or file system artifacts.

Several research groups are now developing parallel file systems and access policies that can exploit an application developer's knowledge of access patterns. PIOUS [18] is a portable input/output system designed for use with PVM. PASSION [3] supports out-of-core algorithms in a user-level library, but focuses on a high-level array oriented interface. PPFS [8] provides user control of file cache sizes and policies, as well as data placement. Similarly, IBM's Vesta parallel file system [4] allows applications to define logical partitions, data distributions, and some access information. In addition to research efforts, several vendors have developed parallel file systems, including the Thinking Machines CM-5 Scalable Parallel File System [16, 13], the Intel Concurrent File System [7] for the iPSC/2 and iPSC/860 [19], the Intel Parallel File System for the Pargon XP/S [27], and PIOFS for the IBM SP-2.

10 Conclusions and Future Work

Inefficient and immature input/output subsystems have emerged as a major performance bottleneck on scalable parallel systems. Unfortunately, file system and storage hierarchy designers have little empirical data on parallel input/output access patterns and have been forced to base designs on extrapolations from the access patterns seen on traditional vector supercomputers. In this paper, we have outlined our methodology for input/output data capture and reported the results of an initial application of this methodology to the first three scientific codes from a much larger application suite.

Our preliminary characterization experiences suggest that there are large variations in spatial and temporal access patterns and in the distribution of request sizes. As others have

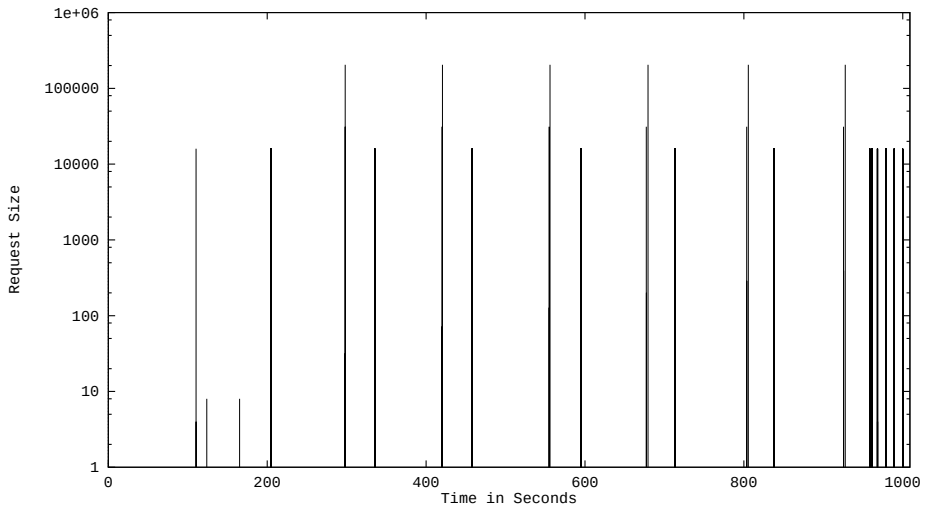

Figure 14: Write operation timeline (HTF self-consistent field calculation)

noted, the majority of the request patterns are sequential. Cyclic behavior, with repeated patterns of file open, access, and close, occur often, but the temporal spacing between requests across cycles is less regular. Requests tend to be of fixed size, though both extremely small and extremely large requests are common.

Given the large diversity in access patterns and request sizes, we believe that design of parallel file systems that rely on a single, system-imposed file system policy is unlikely to be successful. For example, small sequential requests are well served by a caching and prefetching policy, but large and irregular requests are better served by a policy that directly streams data from storage devices to application code. In short, exploitation of input/output access pattern knowledge in caching and prefetching systems is crucial to obtaining a substantial fraction of peak input/output performance. Inherent in such an adaptive approach is the need to identify access patterns and choose policies based on access pattern characteristics.

As we continue to expand our input/output characterization to a larger suite of codes, we are developing a portable parallel file system [8] that allows users to advertize expected file access patterns and to choose file distribution, caching, and prefetch policies. To lessen the cognitive burden of access specification, we have begun developing general, adaptive prefetching methods that can learn to hide input/output latency by automatically classifying and predicting access patterns.

Acknowledgments

We thank the members of the Scalable I/O Initiative who provided us with the applications examined in this paper. In particular, the electron scattering code (ESCAT) was provided by Vincent McKoy and Carl Winstead at Caltech, and the terrain rendering code (RENDER)

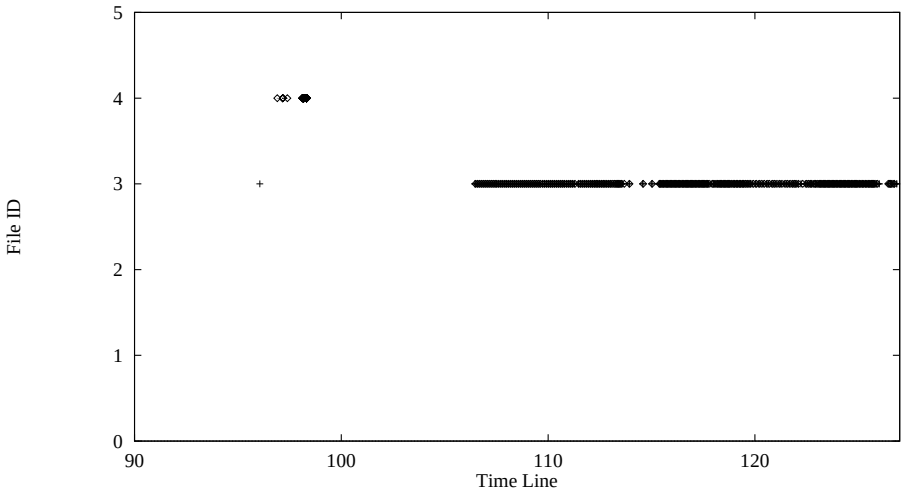

Figure 15: File access timeline (HTF initialization)

was provided by Peggy Li from the Center for Space Microelectronic Technology at the NASA Jet Propulsion Laboratory. The HTF application was provided by Rick Kendall of the Molecular Science Software (MSS) Group at the Molecular Science Research Center (MSRC) of the Pacific Northwest Laboratory (PNL). All data presented here were obtained from code executions on the Intel Paragon XP/S at the CalTech Concurrent Supercomputing Facility.

References

[1] Bordawekar, R., del Rosario, J. M., and Choudhary, A. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93* (1993), pp. 452–461.

[2] Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys 26* (June 1994), 145–185.

[3] Choudary, A., Bordawekar, R., Harry, M., Krishnaiyer, R., Ponnusamy, R., Singh, T., and Thakur, R. PASSION: Parallel And Scalable Software for Input-Output. Tech. rep., Department of Electrical and Computer Engineering, Syracuse University, Nov. 1994.

[4] Corbett, P. F., Baylor, S. J., and Feitelson, D. G. Overview of the Vesta Parallel File System. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems* (1993), pp. 1–16.

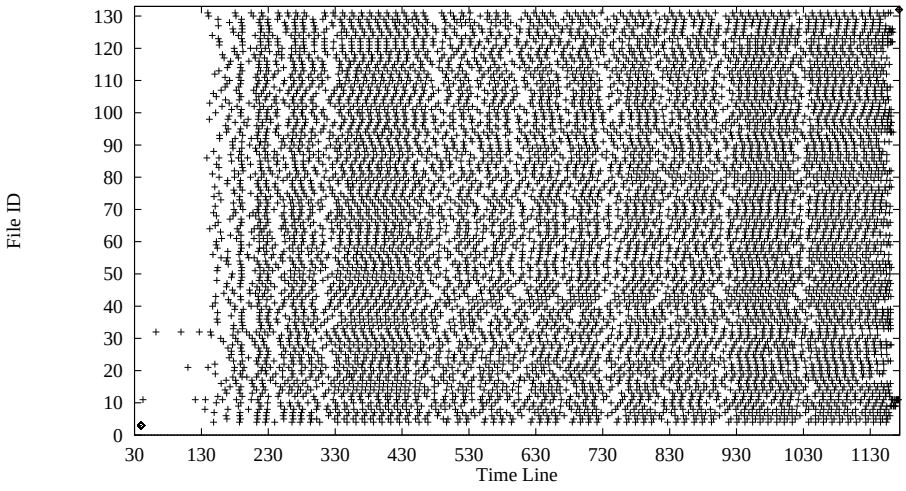

Figure 16: File access timeline (HTF integral calculation)

[5] CORBETT, P. F., FEITELSON, D. G., PROST, J.-P., AND BAYLOR, S. J. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93* (1993), pp. 472–481.

[6] FOSTER, I. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, Reading, MA, 1995.

[7] FRENCH, J. C. Characterizing the Balance of Parallel I/O Systems. In *Sixth Annual Distributed Memory Computer Conference* (1991), pp. 724–727.

[8] HUBER, JR., J. V., ELFORD, C. L., REED, D. A., CHIEN, A. A., AND BLUMENTHAL, D. S. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the International Conference on Supercomputing* (July 1995).

[9] JENSEN, D. W. *Disk I/O In High-Performance Computing Systems.* PhD thesis, Univ. Illinois, Urbana-Champagne, 1993.

[10] JENSEN, D. W., AND REED, D. A. File Archive Activity in a Supercomputing Environment. In *Proceedings of the 1993 ACM International Conference on Supercomputing* (July 1993).

[11] KOTZ, D. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation* (June 1994).

[12] KOTZ, D., AND NIEUWEJAAR, N. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94* (November 1994).

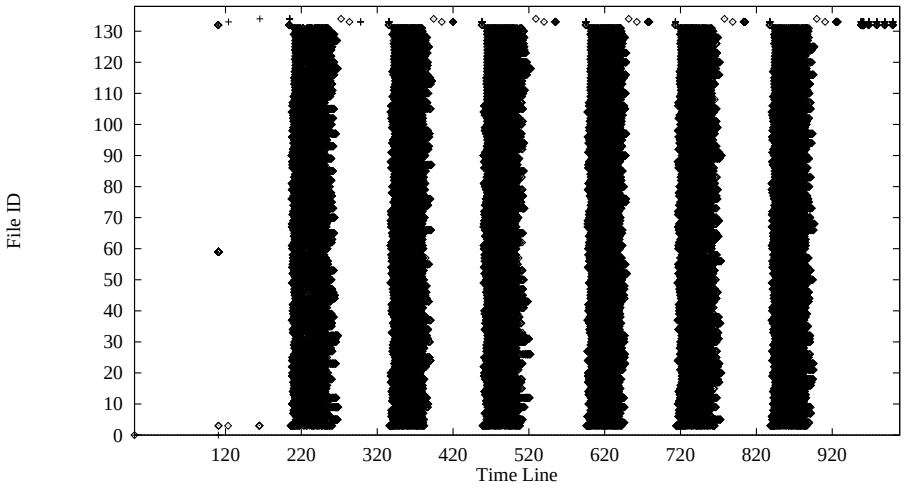

Figure 17: File access timeline (HTF self-consistent field calculation)

[13] KWAN, T. T., AND REED, D. A. Performance of the CM-5 Scalable File System. In *Proceedings of the 1994 ACM International Conference on Supercomputing* (July 1994).

[14] LAWRIE, D. H., RANDAL, J. M., AND BARTON, R. R. Experiments with Automatic File Migration. *IEEE Computer* (July 1982), 45–55.

[15] LI, P., CURKENDALL, D., DUQUETTE, W., AND HENRY, H. Remote interactive visualization and analysis (riva) using parallel supercomputers. In *Proceedings of the 1995 Parallel Rendering Symposium* (October 1995).

[16] 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 *Proceedings of the 1993 Summer USENIX Conference* (1993), pp. 291–305.

[17] MILLER, E. L., AND KATZ, R. H. Input/Output Behavior of Supercomputing Applications. In *Proceedings of Supercomputing '91* (November 1991), pp. 567–576.

[18] MOYER, S. A., AND SUNDARAM, V. S. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *1994 Scalable High Performance Computing Conference* (May 1994), pp. 71–78.

[19] NITZBERG, B. Performance of the iPSC/860 Concurrent File System. Tech. Rep. RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[20] OUSTERHOUT, J., *et al.* A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth Symposium on Operating System Principles* (Dec. 1985).

[21] Pasquale, B. K., and Polyzos, G. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of Supercomputing '93* (Novermber 1993), pp. 388–397.

[22] Pasquale, B. K., and Polyzos, G. Dynamic I/O Characterization of I/O Intensive Scientific Applications. In *Proceedings of Supercomputing '94* (Novermber 1994), pp. 660–669.

[23] Poole, J. T. Preliminary Survey of I/O Intensive Applications. California Institute of Technology, Available at `http://www.ccsf.caltech.edu/SIO/SIO.html`, 1994.

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

[25] Reed, D. A. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (May 1994), pp. 25–51.

[26] 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 *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum, Ed. IEEE Computer Society, 1993, pp. 104–113.

[27] Rudolf Berrendorf and Heribert C. Burg and Ulrich Detert and Rüdiger Esser and Michael Gerndt and Renate Knecht. Intel Paragon XP/S – Architecture, Software Environment, and Performance. Tech. Rep. KFA-ZAM-IB-9409, Forschungszentrum Jülich GmbH, 1994.

[28] Smith, A. J. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering SE-7*, 4 (July 1981), 403–417.

[29] Stritter, T. R. *File Migration.* PhD thesis, Stanford University, Department of Computer Science, Jan. 1977.

[30] Winstead, C., and McKoy, V. Studies of Electron-Molecule Collisions on Massively Parallel Computers. In *Modern Electronic Structure Theory*, D. R. Yarkony, Ed., vol. 2. World Scientific, 1994.