Workload Characterization of Input/Output Intensive Parallel Applications *

Evgenia Smirni and Daniel A. Reed

Department of Computer Science
University of Illinois
Urbana, Illinois 61801, USA
{esmirni,reed}@cs.uiuc.edu

Abstract. The broadening disparity in the performance of input/output (I/O) devices and the performance of processors and communication links on parallel systems is a major obstacle to achieving high performance for a wide range of parallel applications. I/O hardware and file system parallelism are the keys to bridging this performance gap. A prerequisite to the development of efficient parallel file systems is detailed characterization of the I/O demands of parallel applications. In this paper, we present a comparative study of the I/O access patterns commonly found in I/O intensive parallel applications. Using the Pablo performance analysis environment and its I/O extensions we captured application I/O access patterns and analyzed their interactions with current parallel I/O systems. This analysis has proven instrumental in guiding the development of new application programming interfaces (APIs) for parallel file systems and in developing effective file system policies that can adaptively respond to complex application I/O requirements.

1 Introduction

The broadening disparity in the performance of input/output (I/O) devices and the performance of processors and communication links on parallel platforms is the primary obstacle to achieving high performance in parallel application domains that require manipulation of vast amounts of data. Understanding the interactions between application I/O request patterns and parallel I/O system hardware and software is necessary for the design of more effective I/O management policies. The primary objectives of the Scalable I/O initiative (SIO) [11] are (a) to assemble a suite of I/O intensive, national challenge applications, (b) to collect detailed performance data on application characteristics and access patterns, and (c) use this information to design and evaluate parallel file system

* This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), DAVT63-91-C-0029, DABT63-93-C-0040, F30602-96-C-0161, DABT63-96-C-0027, and F30602-96-2-0264, by the National Science Foundation under grant NSF ASC 92-12369, and by the Aeronautics and Space Administration under NASA contracts NGT-51023, USRA 5555-22, and NAG-1-613.

management policies and parallel file system application programming interfaces (APIs).

In this paper, we present the I/O behavior of three representative scientific applications from the SIO code suite. Using the Pablo performance analysis environment and its I/O extensions, we captured and analyzed the access patterns of the SIO applications and their interactions with Intel's parallel file system (PFS) on the Paragon XP/S. This analysis proved instrumental in guiding the development of new file system application programming interfaces (APIs) and adaptive file system policies. We demonstrate why API controls for efficient data distribution, collective I/O, and data caching are necessary to maximize I/O throughput.

The remainder of this paper is organized as follows. In §2, we present related work. This is followed in §3 by an overview of the Pablo performance environment and its I/O extensions, and a high level description of the selected SIO applications. In §4–§5, we describe the applications' detailed I/O access patterns. This is followed in §6 by a discussion of the proposed SIO API and the design issues that are important for parallel I/O performance. Finally, §7 summarizes our observations.

2 Related Work

The first I/O characterization efforts of scientific applications on vector supercomputers concluded that I/O behavior is regular, recurrent, and predictable [8, 10], characteristics that were attributed to the iterative nature of such applications. In contrast to the results on vector systems, recent analyses on the Intel Paragon XP/S [4], the Intel iPSC/860 [6], and the CM-5 [12], showed greater irregularity in I/O access patterns, with the majority of file requests being small but with the greatest data volume generated by a few large requests.

To accommodate the variability of parallel I/O, a growing number of parallel system vendors provide extended I/O APIs that describe, at a higher semantic level, the access patterns that have been observed in practice. Early experience with these APIs has shown major I/O performance improvements — with high-level descriptions of I/O request patterns, file system policies can more intelligently prefetch and cache I/O data.

The desire for intuitive APIs with high expressive power has led to a host of experimental I/O libraries [9, 2, 1] that support specific problem domains and access pattern classes. For example, such domain-specific libraries have emerged for computational chemistry [5] and out-of-core linear algebra computations [16].

Despite the demonstrated performance rewards from use of more expressive APIs, several studies have shown that users frequently opt to continue using UNIX I/O primitives on parallel systems. The rationale for this lies in the desire to maximize code portability across diverse parallel platforms and to minimize software restructuring [15]. Simply put, many scientific application developers are unwilling to sacrifice portability for performance. Only when a standard parallel I/O API is widely deployed, these developers will restructure their codes.

3 Experimental Infrastructure

To illustrate the variability and complexity of the observed I/O behavior in the SIO application suite, we selected three representative applications. Below, we present the Pablo performance environment and its I/O analysis mechanisms, followed by a brief description of the selected applications.

3.1 Pablo Performance Environment

Pablo [13] is a portable performance environment that supports 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 instrumented code generates dynamic performance data that can be directly recorded by the data capture library for off-line analysis, processed by one or more data analysis extensions prior to recording, or directly processed and displayed in real time [14].

Via the Pablo I/O extensions, one can capture detailed traces of I/O events during application program execution. These I/O event traces include the time, duration, size, file name, the processor identifier that initiated the I/O call, and other parameters particular to each I/O operation.

The Pablo environment's off-line analysis tools provide a wide variety of I/O statistical characterizations, including file lifetime summaries, time window summaries, and file region summaries. File lifetime summaries include the number and total duration of file related operations (e.g., reads, writes, seeks, opens, closes) as well as the number of bytes accessed for each file, and the total time each file had been open. Time window summaries contain similar data, but for a specified window of time. File region summaries describe I/O accesses to each range of file byte offsets.

Finally, using Pablo's dynamic graphics displays, one can visualize I/O activity using either workstation graphics or immersive virtual environments. Collectively, the event traces, the statistical summaries, and the visualization toolkit provide a powerful set of I/O analysis and display options.

3.2 SIO Application Suite

Space precludes a complete description of the I/O behavior of all SIO applications. In this study, we selected a representative subset from the SIO application suite that highlights the access patterns commonly found in parallel codes.[2] A high level description of the selected applications follows.

QCRD This quantum chemical reaction dynamics (QCRD) application is used to understand the nature of elementary chemical reactions. The code is written in C and uses symmetric hyperspherical coordinates and local hyperspherical surface functions to solve the Schrödinger equation for the cross-sections of the scattering of an atom by a diatomic molecule. Parallelism is achieved by data decomposition (i.e., all processors execute the same code

[2] For a detailed analysis of the SIO application access patterns see `http://www-pablo.cs.uiuc.edu/Projects/IO`

on different data portions of the global matrices, with data elements equally distributed among processors).

PRISM The PRISM code is a parallel implementation of a three-dimensional numerical simulation of the Navier-Stokes equations written in C and models high speed turbulent flow that is periodic in one direction. Slices of the periodic domain are proportionally distributed among processors, and a combination of spectral elements and Fourier modes is used to investigate the dynamics and transport properties of turbulent flow. The input data provide an initial velocity field, and the solution is integrated forward in time from the fluid's current state to its new state by numerically solving the equations that describe advection and diffusion of momentum in the fluid.

NWChem The NWChem code is a Fortran implementation of the Hartree-Fock self-consistent field (SCF) method that calculates the electron density around a molecule by considering each electron in the molecule in the collective field of the others. The calculation iterates until the field felt by each electron is consistent with that of the other electrons. This "semi-empirical" computational chemistry method predicts molecular structures, reaction energetics, and other chemical properties of interest.

Below, we describe the I/O attributes of the three application codes, with emphasis on request sizes, temporal and spatial patterns, and observed performance on the Intel Paragon XP/S.

4 I/O Requirements

Our experiments were conducted on the Caltech 512 node Intel Paragon XP/S at the Caltech Center for Advanced Computing Research. The system, which is configured for high-performance I/O research, supports multiple I/O hardware configurations. The experiments presented in this paper were conducted using two of the possible I/O configurations: (a) 12 I/O nodes, each controlling a relatively slow 2 GB RAID-3 disk array and (b) 64 4 GB Seagate disks, each attached to a computation node. For all experiments, files were striped across the disks in units of 64 KB, the default configuration for the Intel Paragon XP/S Parallel File System (PFS).

All experiments were conducted on 64 computation nodes in dedicated mode, using representative application data sets. The NWChem application used the 12 RAID-3 disk arrays, whereas the QCRD and PRISM codes used the 64 Seagate disks.[3] The NWChem, QCRD, and PRISM execution times were 5,439 seconds, 16,356 seconds, and 7,394 seconds, respectively.

[3] As we shall see, the older, slower RAID-3 disk arrays give the illusion that NWChem is more I/O intensive than QCRD or PRISM. Via other experiments using different hardware configurations, we have verified that the fraction of application execution time devoted to I/O is sensitive to hardware configuration but that the application I/O behavior is qualitatively unchanged.

4.1 I/O Overview

Table 1 summarizes the I/O activity for the three codes. Based on the fraction of execution time spend performing I/O operations, NWChem and QCRD are significantly more I/O intensive than PRISM. This is despite the fact that PRISM's total I/O operation count is twice that of NWChem and QCRD. Intuitively, most of the PRISM read operations are small and are efficiently satisfied from local I/O buffers.

Looking more closely at individual codes shows that NWChem is dominated by read operations. During NWChem's execution, all processors repeatedly read integrals from secondary storage to construct the Fock matrix and then solve the SCF equations. Because this integral database is computed and written to storage once, but then reread many times, the NWChem code is heavily read limited.

In contrast to the NWChem code, the QCRD code's behavior is dominated by file seeks, both in number and cost. The large number of seeks is a direct effect of the code's data decomposition and storage scheme — each processor must repeatedly seek to its designated portion of the global matrices before each access. In §5 we will return to this issue, showing how much higher application I/O performance could be achieved with minimal code change.

4.2 I/O Request Sizes

The distribution of I/O request sizes is a key determinant of possible file system optimizations. For example, the overhead for small reads can decrease by aggressive prefetching, and small writes are best served by conservative write behind. Conversely, large I/O requests require different approaches like direct streaming to/from storage devices.

Figure 1 shows the cumulative distribution function (CDF) of the percentage of reads and writes versus the request size for the three SIO codes, as well as the fraction of data transferred by each size request. For PRISM, about eighty percent of read and write requests are small (less than 40 bytes), though a few large requests (greater than 150 KB) constitute the majority of I/O data volume. These small requests are excellent candidates for data aggregation.

QCRD shows much less variability in read and write request sizes. For this code, 99 percent of requests are of a single size (2400 bytes) and represent almost all the data volume. Like QCRD, about 90 percent of NWChem's operations are of a single size, in this case 64 KB.

Although these three SIO applications are each dominated by a few request sizes, the range of sizes across the applications is quite large, ranging from a few bytes to several megabytes. In short, parallel I/O file systems must efficiently support a variety of request sizes, both small and large, with policies appropriate to each. In §6, we will return to this issue.

5 Temporal and Spatial Access Patterns

Like request sizes, the temporal and spatial attributes of requests have profound implications for file system policies. Most workstation file systems are

Operation	Operation Count	Percentage Count	I/O Time (seconds)	I/O Time Percentage	Exec. Time Percentage
open	92	0.02	317.25	0.12	0.09
read	465,154	93.05	251,105.84	92.53	72.13
seek	2,115	0.42	36.87	0.01	0.01
write	32,398	6.48	18,746.65	6.91	5.39
close	153	0.03	1,158.43	0.43	0.33
All I/O	499,910	100.00	271,365.03	100.00	77.95

(a) NWChem (12 RAIDs)

Operation	Operation Count	Percentage Count	I/O Time (seconds)	I/O Time Percentage	Exec. Time Percentage
open	6,592	1.29	3,618.32	1.4	0.45
read	176,228	34.52	8,027.68	3.12	0.99
seek	258,648	50.66	238,770.16	92.73	29.5
write	61,904	12.13	6,148.96	2.4	0.75
close	7,168	1.4	975.17	0.35	0.08
All I/O	510,670	100.00	257,493.07	100.00	31.77

(b) QCRD

Operation	Operation Count	Percentage Count	I/O Time (seconds)	I/O Time Percentage	Exec. Time Percentage
open	415	0.04	203.12	12.51	0.04
read	1,037,568	93.70	950.11	58.50	0.20
seek	704	0.06	293.17	18.05	0.06
write	68,273	6.17	42.93	2.64	0.01
close	414	0.04	134.82	8.30	0.03
All I/O	1,107,374	100.00	1,624.16	100.00	0.34

(c) PRISM

Table 1. Basic I/O summary table

optimized for sequential file access. Burstiness and non-sequentiality necessitate new and different file system approaches (e.g., aggressive prefetch or write behind or skewed data storage formats). Below, we consider the implications of these temporal and spatial patterns based on the SIO code suite.

Because in the NWChem and the PRISM codes logical processor (node) zero executes more operations and consistently transfers more data than the the other processors, we focus on two representative processors, zero and sixty. In the QCRD code there is no such behavior – all nodes execute the same operations and we demonstrate the activity of a representative node only.

5.1 Sequential Access Patterns

Figure 2 illustrates the operation durations of nodes zero and sixty of NWChem as a function of the program execution time.[4] The three NWChem code phases

[4] To increase legibility, we plot only the first 2500 seconds of NWChem's execution.

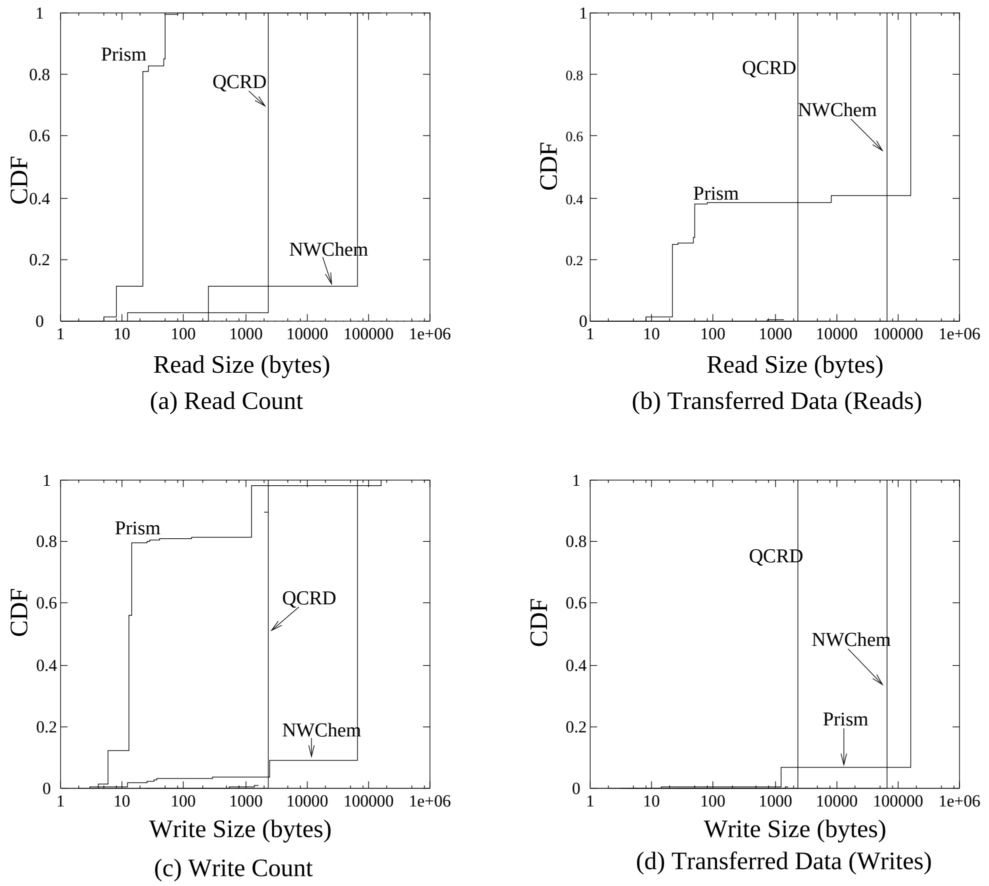
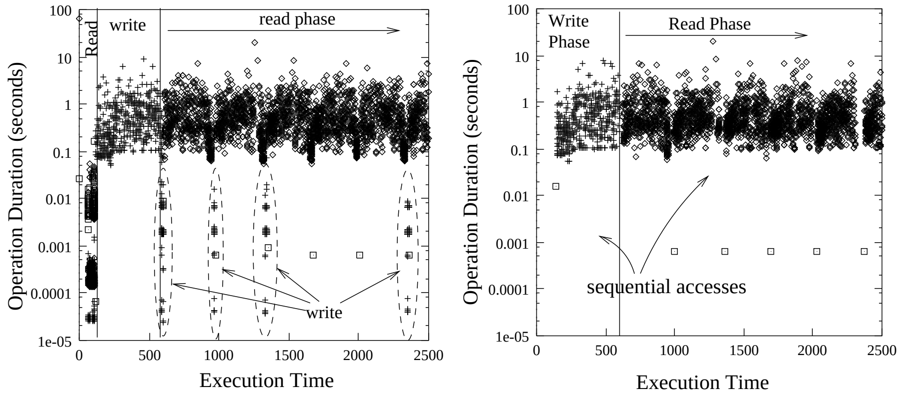


Fig. 1. CDF of read/write request sizes and data transfers.

are clearly distinguished in the figure. First, node zero loads the problem definition from the various input files, the basis sets are calculated, and the results are written to the disks.

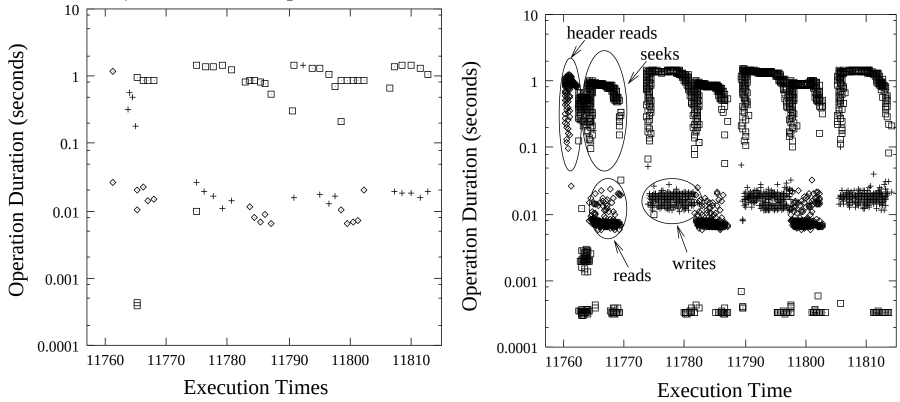
In the second phase all nodes participate in I/O: they evaluate the integrals and write them to secondary storage. The work is distributed across all nodes, with each node accessing a *private* file that stores the locally computed integrals. Because NWChem's designers opted to store the integral data such that each node could read independently, the quadrature data write pattern is predominately sequential. Each processor first seeks to rewind the file, then sequentially writes the quadrature data.

In the third phase, all nodes concurrently open their private integral files, repeatedly read integral data, construct the Fock matrix, compute, synchronize, and solve the SCF equations. Node zero periodically collects the results and writes them to disk; these are the short, bursty writes in Figure 2(a). Again, the read access pattern is sequential, with a single seek preceding bursts of intense



(a) NWChem (node 0) (b) NWChem (node 60)

Fig. 2. Operation durations for NWChem. Rectangles represent seeks, diamonds represent reads, and crosses represent writes.



(a) QCRD (typical node) (b) QCRD (all nodes)

Fig. 3. Operation durations for QCRD. Rectangles represent seeks, diamonds represent reads, and crosses represent writes.

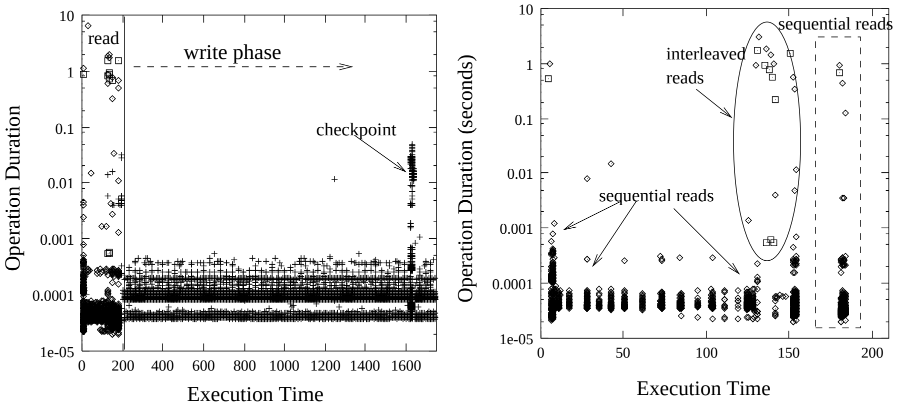
read activity on the file.

5.2 Interleaved Access Patterns

Rather than using private files to store the portion of the global matrix apportioned to each node, QCRD's developers opted to use a single file for the matrix. As Figure 3 shows, this leads to a strikingly different I/O access pattern.

Because all nodes in QCRD must contend for access to the single, shared file, each read or write operation is preceded by a seek to position the file pointer. The pattern is clearly visible in Figure 3(a), a snapshot of one node's activity near the end of its execution.

As a comparison, Figure 3(b) shows the combined effects of I/O operations by all nodes during the same time period. The Intel Paragon XP/S OSF/1 file



(a) PRISM (node 0) (b) PRISM (node 60)

Fig. 4. Operation durations for PRISM. Rectangles represent seeks, diamonds represent reads, and crosses represent writes.

system serializes file seeks when more than one node accesses the same file. Thus, seek times during parallel execution approach one second.

The design choice to use a single file and the semantics of UNIX I/O operations combine to yield extraordinarily poor I/O performance for the QCRD code. Despite the large number of seeks, the dominant QCRD access patterns are highly regular, and a more powerful I/O API would have allowed the developers to express this regular, though non-sequential pattern. In turn, such an expression would have allowed the file system to relax the file pointer consistency protocol, yielding much higher performance. We will return to this relation between API and performance in §6.

5.3 Mixed Access Patterns

Figure 4 illustrates the duration and spacing of I/O operations for the PRISM code. Figure 4(a) depicts the I/O activity of node zero for the first 1,700 seconds of application execution time. During the first 200 seconds, all nodes access the initialization files. During the subsequent, write intensive phase, results are written to disk and these writes are administered through node zero only.

The mixed sequential and interleaved access patterns used to access the initialization files are clearly demonstrated in Figure 4(b). During the read phase, three files are accessed. The first file and the third files are accessed sequentially by all nodes. The second file is a restart file that contains the initial conditions for the simulation; each node first reads the file header sequentially and then accesses its own portion in an interleaved fashion. The code demonstrates a case commonly found in the SIO applications: file headers that contain metadata information are accessed sequentially by all nodes while the rest of the file body is accessed in an interleaved fashion.

6 Discussion

Characterization studies are by their nature inductive, covering a subset of the possibilities. Although the three SIO applications in this study differ dramatically in their algorithmic approaches, they share certain common features and are amenable to some general observations about I/O characteristics.

- Parallel I/O is bursty, with computation intervals of little or no I/O activity interleaved with time periods of intense, concurrent I/O operations. In such situations, I/O can be effectively overlapped with computation via caching, prefetching, and write-behind mechanisms.
- The size of requests can differ dramatically, even within the same application, ranging from a few bytes to several kilobytes each. Different optimization mechanisms are required for such transfers (i.e., aggregating small requests into large data chunks in cache before initiating a transfer or disabling all caching and bypassing the cache for large request sizes). Understanding when and how to aggregate requests is essential for high performance.
- Access patterns range from simple sequential to interleaved. Furthermore, there are combinations of sequential and interleaved accesses even within the same file. Due to the high concurrency of such operations, efficient file pointer consistency protocols are necessary to deliver high performance. Alternatively, expressive APIs are needed to describe internal file structure and expected access patterns, enabling the file system to relax the consistency protocol.
- Request parallelism ranges from little or none (i.e., all I/O is administered through a single node) to full parallelism (i.e., all nodes concurrently send independent requests to disks). Channeling all requests through a single node is often an alternative solution chosen by application developers to avoid inefficient parallel file systems. Collective I/O is a desirable alternative for high-performance concurrent accesses.
- Files can be either shared across processors or can be simply private per processor. Different file system policy optimization approaches should be used if files are shared or private.

6.1 Current I/O APIs

One of the most striking observations of our characterization study was that despite the existence of several parallel I/O APIs and clear performance advantages, most computational scientists eschew their use. Discussions with these researchers revealed that the restricted, often inefficient, but highly portable UNIX API was preferable to vendor provided, extended I/O interfaces.

Comparisons of vendor proprietary I/O interfaces (e.g., IBM's PIOFS and Intel's PFS) show that they have few shared features, mainly due to different design philosophies. Therefore, the design and implementation of a standard parallel I/O API that is expressive, compact, intuitively appealing, and at the same time offers high performance [3] is one of the goals of the SIO effort.

6.2 Emerging I/O APIs

Even within our small application sample, the diversity of I/O request sizes and patterns suggests that achieving high performance is unlikely with a single file system policy. Instead, one needs a file system API via which users can "inform" the file system of expected access patterns. Using such hints or an automatic access pattern classification scheme [7], an adaptive file system could then choose those file policies and policy parameters best matched to the access pattern.

For example, via user controls and hints one might advise the file system that the file access pattern is read only, write only, mixed read/write sequential, or strided. For small, read only requests, the file system would aggressively prefetch large blocks of data, satisfying future requests for previously cached data. Similarly, for small strided writes, the file system might first cache the data then write large blocks using a log-structured file model.

Similarly, with knowledge of request concurrency and real-time measurements of I/O device performance, a flexible file system might dynamically choose the number of disks across which data is striped and the size of the stripes. When there are few competing requests, striping data in large blocks across all disks reduces write time by trading inter-request parallelism for reduced intra-request latency.

Finally, with knowledge of the evolution of access patterns, a file system might choose a file striping that optimizes globally across multiple patterns. Thus, a system could balance the cost of efficiently writing a file against the cost of repeatedly reading the file using an access pattern not well matched to the storage format.

All these examples share a common theme — they presume higher level knowledge of current and future I/O access patterns. Rather than composing complex access patterns from read, write, and seek "atoms," with expressive I/O APIs users can describe temporal and spatial access patterns directly. Alternatively, file systems must glean this information automatically (e.g., by classifying access patterns using trained neural networks or hidden Markov models). In either case, only with such knowledge can flexible file systems maximize performance by matching policies to access patterns

7 Conclusions

Even with the restricted application subset that we investigated here, we demonstrated that there is much variation in temporal and spatial I/O patterns across applications, with both very small and very large request sizes, sequential and interleaved accesses, shared and non-shared files, and access time scales ranging from microseconds to minutes. We conclude that there are many opportunities for improving the performance of parallel file system and parallel I/O standardization is an important step towards this process. The analysis of the parallel I/O workload characteristics of the SIO suite has contributed towards this effort. From this analysis, various design issues related to parallel I/O application programming interfaces have emerged. An API that is compact, expressive, and provides the access pattern information to the file system, can exploit alternative data management policies that better match these patterns. Controls for efficient data distribution, collective I/O, and data caching are necessary to provide high I/O throughput.

References

1. BENNETT, R., BRYANT, K., SUSSMAN, A., DAS, R., AND SALTZ, J. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (October 1994), IEEE Computer Society Press, pp. 10–20.
2. BORDAWEKAR, R., THAKUR, R., AND CHOUDHARY, A. Efficient compilation of out-of-core data parallel programs. Tech. Rep. SCCS-622, NPAC, April 1994.
3. CORBETT, P. F., PROST, J.-P., DEMETRIOU, C., GIBSON, G., RIEDEL, E., ZELENKA, J., CHEN, Y., FELTEN, E., LI, K., HARTMAN, J., PETERSON, L., BERSHAD, B., WOLMAN, A., AND AYDT, R. Proposal for a common parallel file system programming interface version 1.0, September 1996.
4. CRANDALL, P., AYDT, R. A., CHIEN, A. A., AND REED, D. A. Input/Output characterization of scalable parallel applications. In *Supercomputing 1995* (1996).
5. FOSTER, I., AND NIEPLOCHA, J. ChemIO: High-performance I/O for computational chemistry applications. http://www.mcs.anl.gov/chemio/, February 1996.
6. KOTZ, D., AND NIEUWEJAAR, N. Dynamic file-access characteristics of a production parallel scientific workload. In *Supercomputing '94* (November 1994).
7. MADHYASTHA, T., AND REED, D. A. Intelligent, adaptive file system policy selection. In *Proceedings of Frontiers'96* (1996).
8. MILLER, E. L., AND KATZ, R. H. Input/Output behavior of supercomputer applications. In *Supercomputing '91* (November 1991), pp. 567–576.
9. NIEUWEJAAR, N., AND KOTZ, D. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing* (May 1996).
10. PASQUALE, B. K., AND POLYZOS, G. C. Dynamic I/O characterization of I/O intensive scientific applications. In *Proceedings of Supercomputing '94* (November 1994), pp. 660–669.
11. POOLE, J. T. Scalable I/O Initiative. California Institute of Technology, Available at `http://www.ccsf.caltech.edu/SIO/`, 1996.
12. 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), pp. 165–172.
13. 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.
14. 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 *Proceedings of MASCOTS '96* (Feb. 1996), pp. 1–12.
15. SMIRNI, E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. I/O requirements of scientific applications: An evolutionary view. In *High Performance Distributed Computing* (1996), pp. 49–59.
16. TOLEDO, S., AND GUSTAVSON, F. G. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems* (May 1996), pp. 28–40.