Title: "Workload Characterization of Input/Output Intensive Parallel Applications"

Corresponding author: Evgenia Smirni

email: esmirni@cs.wm.edu
tel: (757) 221-3580
fax: (757) 221-1717

College of William and Mary
Department of Computer Science
P.O. Box 8795
Williamsburg, VA 23187-8795
USA

Workload Characterization of Input/Output Intensive Parallel Applications *

Evgenia Smirni
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187
esmirni@cs.wm.edu

Daniel A. Reed
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
reed@cs.uiuc.edu

Abstract

Because both processor and interprocessor communication hardware is evolving rapidly with only moderate improvements to file system performance in parallel systems, it is becoming increasingly difficult to provide sufficient input/output (I/O) performance to parallel applications. I/O hardware and file system parallelism are the key to bridging this performance gap. Prerequisite to the development of efficient parallel file system is detailed characterization of the I/O demands of parallel applications.

In the paper, we present a comparative study of parallel I/O access patterns, commonly found in I/O intensive scientific applications. The Pablo performance analysis tool and its I/O extensions is a valuable resource in capturing and analyzing the I/O access attributes and their interactions with extant parallel I/O systems. This analysis is instrumental in guiding the development of new application programming interfaces (APIs) for parallel file systems and effective file system policies that 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. To improve performance, advances of I/O hardware and file system parallelism are of principal importance.

In the last few years, a wide variety of parallel I/O systems have been proposed and built[11, 12, 4, 5, 20, 14, 17]. All these systems exploit parallel I/O devices (i.e., partitioning data across

*This work was supported in part by the Advanced Research Projects Agency under ARPA contracts DABT63-94-C0049 (SIO Initiative), DAVT63-91-C-0029 and DABT63-93-C-0040, by the National Science Foundation under grant NSF ASC 92-12369, and by the Aeronautics and Space Administration under NASA Contracts NGT-51023 and USRA 5555-22.

disks for parallelism) and data management techniques (i.e., prefetching and write-behind) in an attempt to deliver high I/O performance. However, the lack of characterization data of parallel I/O access patterns have often forced the designers of such systems to extrapolate from measurements from uniprocessor or vector supercomputer applications. In consequence, the high peak performance of parallel file systems cannot be effectively delivered because file system policies are not well tuned to satisfy the I/O requirements of parallel applications.

Understanding the interactions between application I/O request patterns and the hardware and software of parallel I/O systems is necessary for the design of more effective I/O management policies. The primary objectives of the Scalable I/O initiative (SIO) [22] 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 management policies and parallel file system application programming interfaces (APIs).

In this paper we present the I/O behavior of five representative scientific applications from the SIO code suite. Using the Pablo performance analysis tool and its I/O extensions, we captured and analyzed the access patterns of the SIO applications and their interaction with the Intel's parallel file system (PFS) [11]. This analysis proved instrumental in guiding the development of new file system application programming interfaces (APIs) that drive 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 [16, 21], 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 [7], the Intel iPSC/860 [13], and the CM-5 [23], showed greater irregularity in I/O access patterns, with the majority of file request being small but with the greatest data volume transferred by a few large requests. Subsequent studies [19, 27] indicated that users attempt to adjust the application I/O access patterns to match the characteristics of the underlying parallel file system to maximize performance.

To accommodate the variability of parallel I/O, a growing number of parallel platform vendors provide extended sets of I/O APIs that can capture subsets of access patterns that have been observed in the literature. Furthermore, many experimental I/O libraries focus on specific classes of I/O patterns and provide flexible data distribution and data management policies. For example, Galley [18], Panda [26], PASSION [2] and Jovian [1] support external multi-dimensional arrays, can restructure disk requests, and provide collective I/O operations that effectively utilize the available disk bandwidth. The desire for intuitive APIs with high expressive power has led to a host of domain specific I/O libraries that support specific problem domains and access pattern classes. Such libraries have emerged for computational chem-

istry [8] and for out-of-core linear algebra computations [29]. 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.

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 [27]. 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 Experimentation Infrastructure

To sketch a global view of the variability and complexity of the observed I/O behavior in the SIO application suite, we selected five representative applications. To capture and analyze the I/O access patterns of these applications, we used the Pablo performance environment and its I/O extensions. Below, we present the Pablo environment and its I/O analysis mechanisms followed by a brief description of the selected applications.

3.1 The Pablo Performance Environment

Pablo [24] is a portable performance environment that supports performance data capture and analysis. The instrumentation software captures dynamic performance data via instrumented source code that is linked with a data capture library. During program execution, the instrumentation code generates trace 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 at real-time.

The Pablo environment can be effectively used in both explicit message passing codes and data parallel HPF programs. The graphical instrumentation interface allows users to interactively specify either local or global instrumentation. Local instrumentation is activated by bracketing each call of interest with trace library calls. When global instrumentation is activated, all application calls are automatically substituted with trace library calls. The parser then produces instrumented source code that can be compiled and linked with the data capture library.

Via the Pablo I/O extensions, it is possible to capture traces of detailed I/O events during the application's lifetime. 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 extraction tools provide a wealth of trace statistical information: 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 provide comprehensive information over accesses to a file region.

Finally, using a set of data extraction tools, it is possible to visualize I/O activity as a function of the application elapsed time using either workstation graphics or immersive

virtual environments [25]. Collectively, the event traces, the statistical summaries, and the visualization toolkit provide a powerful set of I/O analysis and display options.

3.2 The SIO Applications

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 comprises the access pattern attributes commonly found in parallel codes.[1] A high level description of the selected applications follows.

MESSKIT and NWChem The MESSKIT and the NWChem codes [10] are two distinct Fortran implementations 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.

From an I/O perspective, both codes have three execution phases. After an initial I/O phase where the relative geometry of the atomic centers is read, the basis sets are computed and written to disk, and atomic integrals are computed over these basis sets. Because the integral volume grows dramatically with the basis sets, an out-of-core solution is used as the atomic integrals are too voluminous to fit in each processors memory. Each processor writes the integral it computes to its own private file. In the final phase, all processors repeatedly read the integral data from disk, construct the Fock matrix, and use the SCF method until the molecular density converges to within an acceptable threshold. At the end of computation, final results are written to disk.

Although the two applications solve essentially the same problem using the same logical sequence of steps, there are differences between the two codes. In NWChem, the integrals are written to disk in parallel with the first Fock matrix construction, procedures that are separated in MESSKIT. Additionally, the two codes use different integral evaluation libraries, with different timings of the integral evaluation phase. NWChem uses a preconditioned conjugate gradient approach to optimize the molecular orbitals, while MESSKIT uses a procedure known as direct inversion of the iterative subspace. This results in a different number of iterations and different number of Fock matrix constructions even if both codes execute the same input. The codes use different data formats and produce different volumes of data even for matching inputs.

QCRD This quantum chemical reaction dynamics (QCRD) application[30] is used to understand the nature of elementary chemical reactions. The code is written in C and uses the method of symmetrical 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 nodes execute the same code on different data portions of the global matrices with data elements equally distributed among processes.

[1] For detailed analysis of the SIO applications access patterns see http://www-pablo.cs.uiuc.edu/Projects/IO.

Because of the selected data distribution scheme and the prohibitively large global matrices to be stored in memory, an out-of-core solution is required. Due to the iterative nature of the numerical algorithms used by QCRD, there is intensive I/O activity on disks. I/O demands on disk have a repetitive, cyclic pattern.

PRISM The PRISM code is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations written in C [9] and models high speed turbulent flow that is periodic in one direction. Slides of the periodic domain are proportionally distributed among processors and a combination of spectral elements and Fourier modes are 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.

From an I/O perspective, there are three distinct execution phases. After initialization data are read from disk, the Navier-Stokes simulation integrates the status of the fluid forward in time and history points are periodically written to disk during integration. During the final or post processing phase, after the end of the integration, final results are written to disk.

ESCAT The ESCAT code is a parallel implementation of the Schwinger Multichannel method written in C, FORTRAN, and assembly language [3]. 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. The scattering probabilities are obtained by solving linear systems whose terms must be evaluated by numerical quadrature. Generation of the quadrature data is computationally intensive, and the total quadrature data volume is highly dependent on the nature of the problem. The quadrature data is formulated in an energy independent way, making it possible to solve the scattering problem at many energies without quadrature data recalculation. Because the quadrature data is too voluminous to fit in the individual processor memory, an out of core solution is required.

From an I/O perspective there are four distinct execution phases. After initialization data is read from the disk, all nodes participate in the calculation and storage of the requisite quadrature data set that is energy independent. Then, the quadrature data is read from disk and energy dependent data structures are generated and combined with the reloaded quadrature data. Finally, the results of calculation are written to disk.

In the following sections we will present the I/O requirements of the selected applications, concentrating on request sizes, their timings, and their temporal and spatial patterns.

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 executed in dedicated mode, using representative application data sets. The NWChem and MESSKIT applications used the 12 RAID-3 disk arrays, while the QCRD, PRISM, and ESCAT used the 16 Seagate disks.[2] The MESSKIT, NWChem, QCRD, PRISM, and ESCAT execution times were equal to 1,788 seconds, 5,439 seconds, 16,356 seconds, 7,394 seconds and 5,803 seconds respectively.

4.1 I/O Overview

Table 1 summarizes the I/O activity of the codes. MESSKIT, NWChem, and QCRD appear significantly more intensive than PRISM and ESCAT, with a large percentage of the application execution times is consumed in I/O. This is despite the fact that the total count of I/O operations in PRISM is much larger than any other application. Intuitively, most of the PRISM read operations are small and are efficiently satisfied from local I/O buffers.

MESSKIT and NWChem are dominated by read operations. During 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, both MESSKIT and NWChem codes are heavily read limited. Overall, the ratios of operation costs are analogous to operation counts. The remaining of operations contribute insignificant amounts to the I/O time.

A different behavior is observed for the QCRD code: seeks are the most expensive operations and their cost dominates the execution time consumed by I/O. 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. We demonstrate in §5 that forced serialization of seek requests due to the selected algorithmic implementation is the reason for such behavior.

ESCAT uses a data decomposition and storage scheme for the quadrature data similar to QCRD's. However, the code achieves to minimize the average seek cost by avoiding the forced serialization of seek requests. In §5 we will return to this issue.

The behavior of the PRISM code is also worth a more precise analysis. Although the code appears heavily read bound when looking at operation counts, read operations have a limited cost. We will show in the following sections that this is a result of read request sizes and their access pattern.

4.2 I/O Request Sizes

Apart from operation counts, the distribution of their sizes across the five codes is also important from a workload characterization perspective. 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

[2]As we shall see, the older, slower RAID-3 disk arrays give the illusion that MESSKIT and NWChem are more I/O intensive than QCRD, PRISM, or ESCAT. 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.

(a) MESSKIT

Operation	Operation Count	Percentage Count	I/O Time (seconds)	I/O Time Percentage	Exec. Time Percentage
open	163	0.23	160.33	0.77	0.13
read	51,829	73.58	18,289.03	87.49	15.35
seek	497	0.71	1.52	0.00	0.00
write	9,163	13.00	2,085.21	9.97	1.72
close	160	0.23	9.07	0.04	0.01
flush	8,626	12.25	361.5	1.73	0.03
All I/O	70,438	100.00	20,906.66	100.00	17.24

(b) NWChem

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	499,912	100.00	271,365.03	100.00	77.95

(c) QCRD

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	510,540	100.00	257,540.29	100.00	31.77

(d) PRISM

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	1,624.15	100.00	0.34

(e) ESCAT

Operation	Operation Count	Percentage Count	I/O Time (seconds)	I/O Time Percentage	Exec. Time Percentage
open/gopen	262	0.97	357.64	13.84	0.05
read	815	3.03	56.77	2.20	0.01
seek	12,034	44.69	283.98	10.99	0.04
write	13,300	49.39	1,123.48	43.48	0.15
iomode	256	0.95	584.52	22.62	0.08
close	262	0.97	177.26	6.86	0.02
All I/O	26,929	100.00	2,583.65	100.00	0.35

Table 1: Basic I/O summary table

write behind. Conversely, large I/O requests require different approaches like direct streaming to/from storage devices.

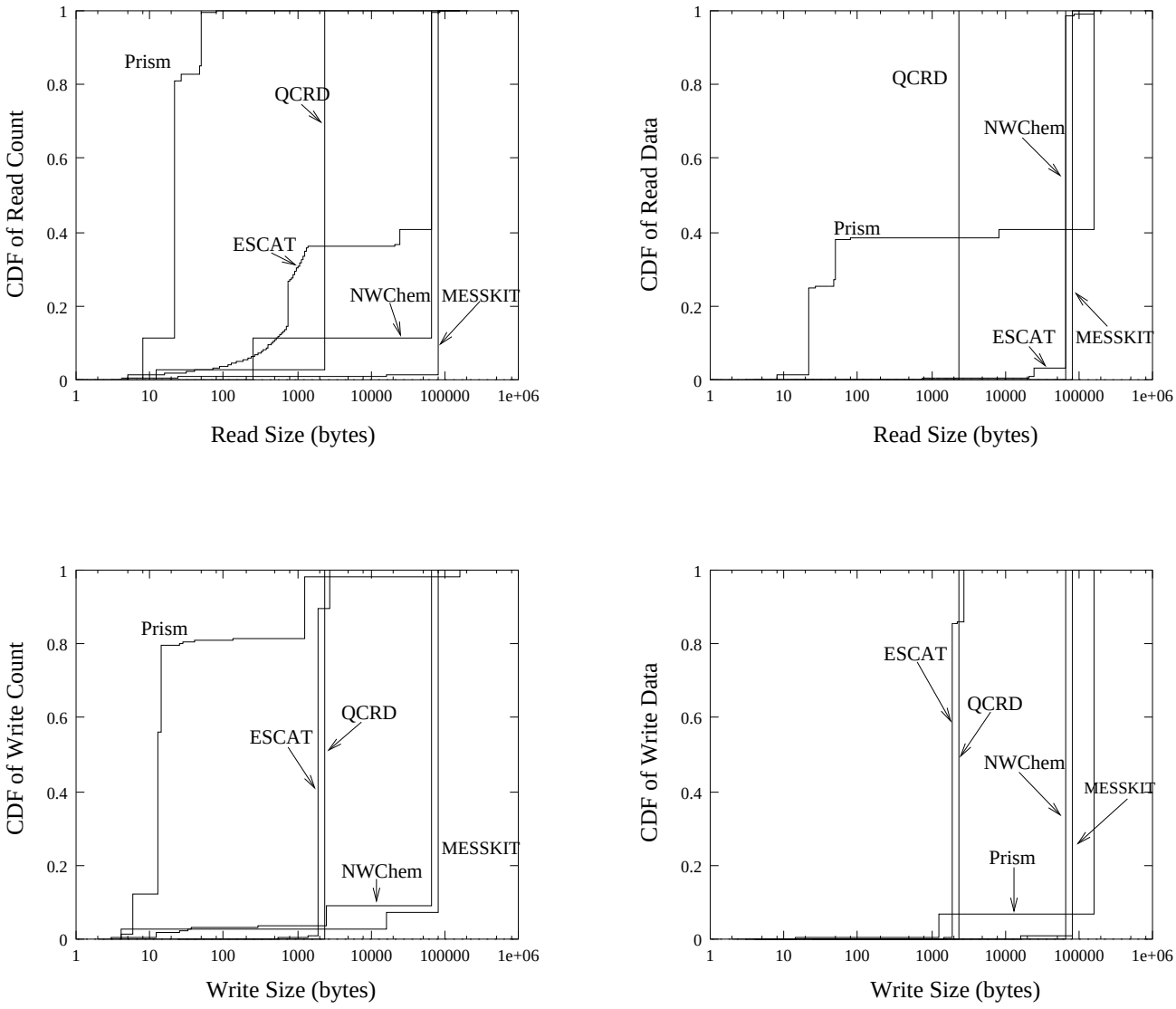

Figure 1: CDF of read/write request sizes and data transfers.

Figure 1 shows the cumulative distribution function (CDF) of the percentage of reads and writes versus the request size for the five codes, as well as the fraction of data transferred by each request size. PRISM and ESCAT show a distinctive variation in the application request sizes. For PRISM, about 80 percent of read and write requests are small (less than 40 bytes), though a few large requests (greater 150KB) constitute the majority of I/O data volume. For ESCAT, about 40 percent of read requests are small (less than 1000 bytes) although the majority of data volume is transfered by a few read requests of 80,000 bytes. The small request sizes are excellent candidates for data aggregation.

There is much less variability in the read and write request sizes of QCRD, MESSKIT, and NWChem. For QCRD, 99 percent of requests are equal to 2,400 bytes and they transfer almost all the data volume. Similarly to QCRD, where the majority of requests are of the same size, about 90 percent of reads and writes in NWChem are equal to 64 KB. The smaller request sizes in NWChem transfer insignificant portions of the application data. Similarly to

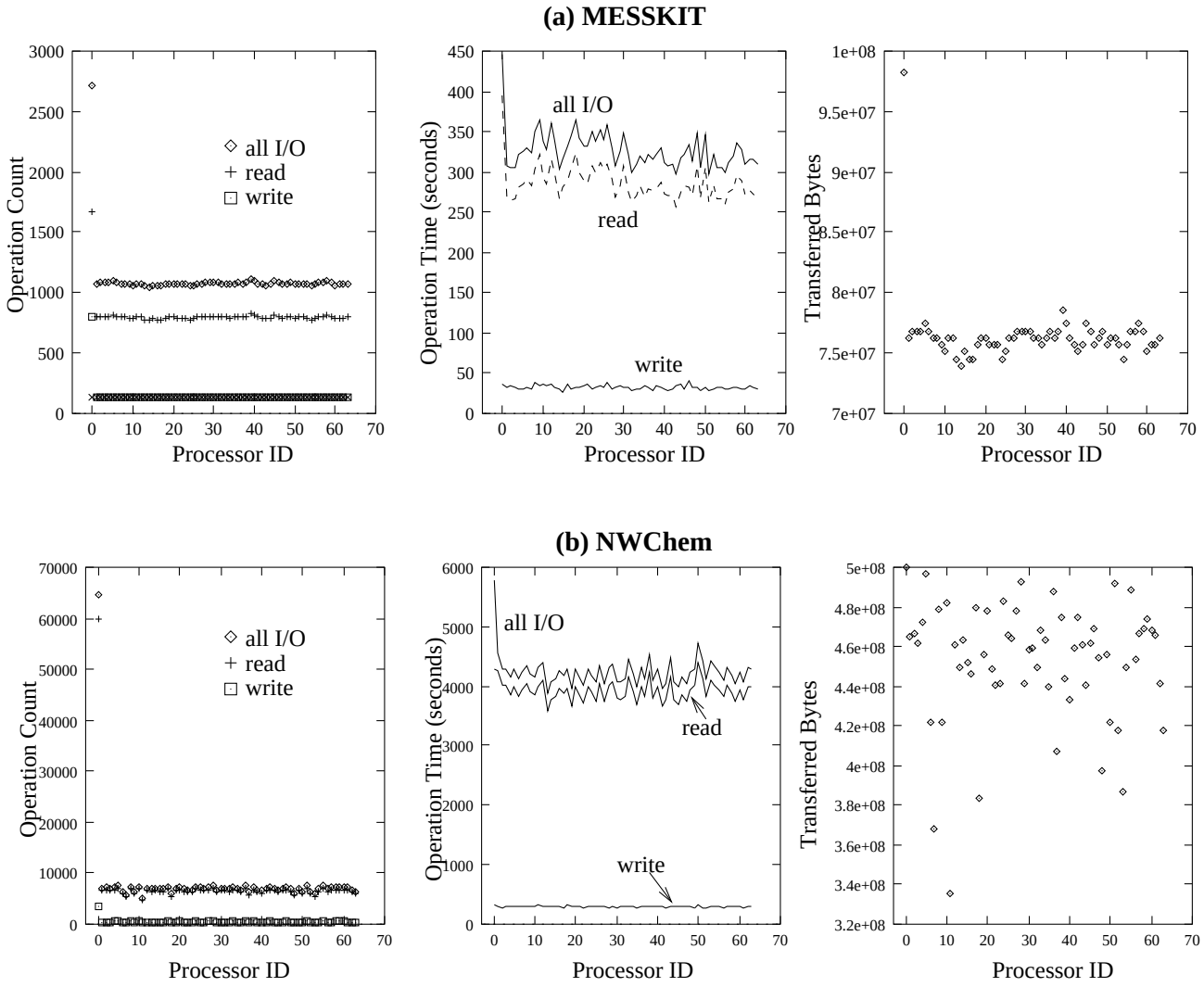

Figure 2: I/O times per processor for the two Hartree-Fock applications, MESSKIT and NWChem.

NWChem, more than 90 percent of read and write requests are equal to 84,000 bytes and transfer almost 99 percent of data.

In general, we observe that even from our small, inductive sample of five applications, there is a great variability in request sizes, ranging from a few bytes to several megabytes. Parallel I/O APIs that deliver high performance for a wide variety of request sizes are necessary. In §6, we will return to this issue.

4.3 Processor I/O Behavior

Finally, to complete the high level view of I/O characteristics of the selected codes, we turn to variations across processors. Although all five applications use the SPMD programming model, operation counts, operation durations, and transferred volumes differ from processor to processor.

Figures 2 and 3 show that despite the big variation in request counts across the five codes, typically logical processor (node) 0 executes more operations and consistently transfers the

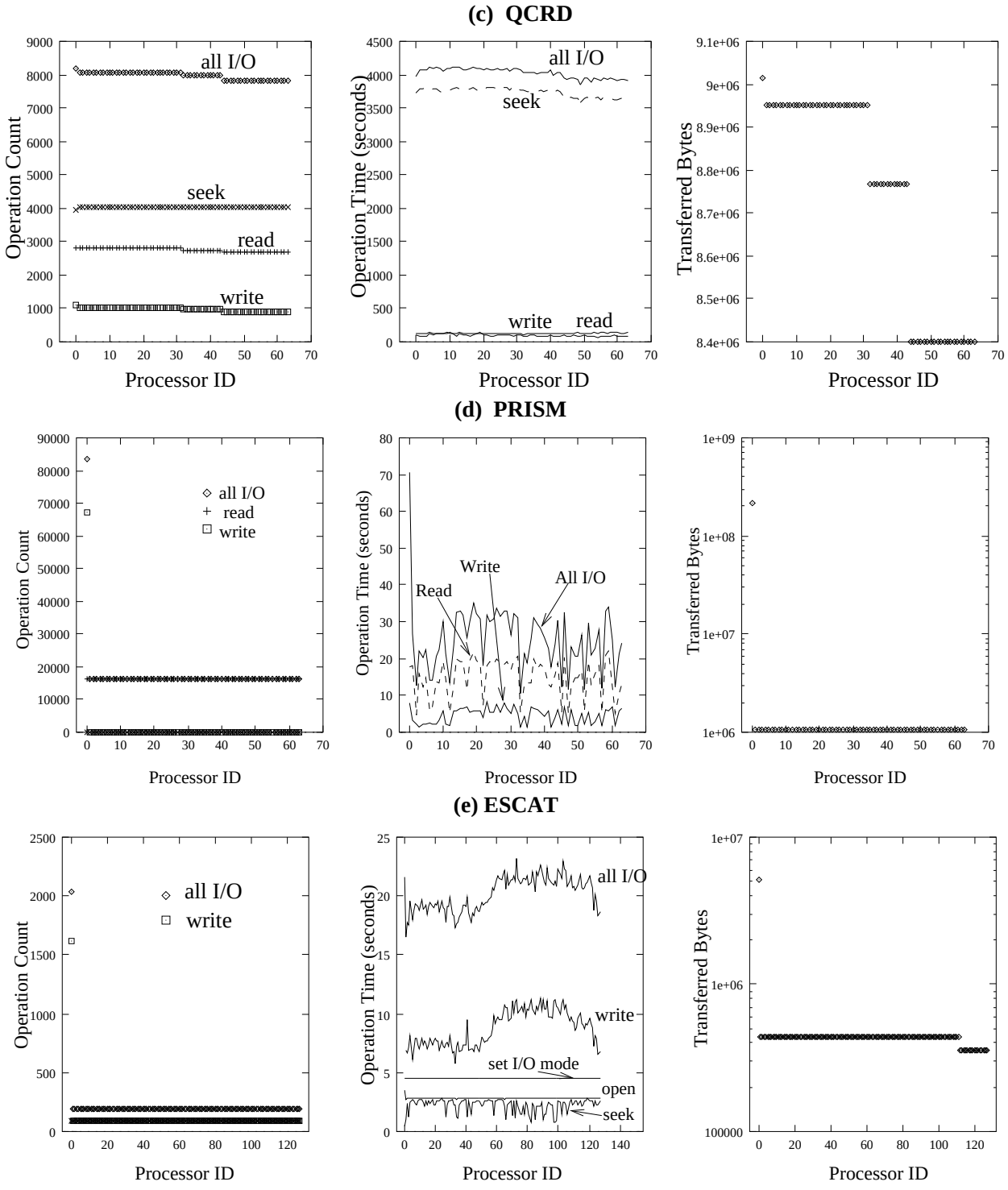

Figure 3: I/O times per processor for QCRD, PRISM, and ESCAT.

largest data volume. This is a side effect of the developers choice to use node 0 as coordinator of disk data transfers to disk. We will demonstrate in §5 that frequently, initialization data are read by node 0 and then broadcasted to the remaining nodes. In the same fashion, node 0 often collects results from all nodes and funnels them to disk.

Figures 2 and 3 also illustrate that because of the data partitioning approach to realize parallelism, there are distinct subsets of nodes that execute the same number of operations and transfer that same data volume. Nonetheless, even among the same node subsets, there is significant variation in the operation execution times. In the following section we will illustrate how queuing delays contribute to operation costs, advocating the need of collective I/O operations on subsets of processors.

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 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.

Since §4 suggested that typically node 0's I/O activity is more intense than that of the rest of the nodes, we illustrate the application temporal behavior of node 0 and of node 60, which is representative of the remaining nodes. 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. Because space precludes a complete description of the applications' access patterns, we will concentrate on the temporal spacing and mutual interaction of the three basic operations: seek, read, and write.

5.1 Sequential Access Pattern

Figure 4 illustrates the operation durations of nodes 0 and 60 of MESSKIT and NWChem as a function of the program execution time. To increase legibility, we plot only the first 2500 seconds of NWChem's execution. Limiting the execution time restricts the large number of data points and attenuates the otherwise severe overplotting that obscures the location and distribution of less frequent I/O operations. The remaining I/O activity not plotted in the graph does not convey additional information about the access pattern because of the iterative nature of the code.

In both codes, the three execution phases are clearly distinguished in the figure. First, node 0 loads the problem definition from the various input files, the basis files are calculated, and the results are written to disk.

In the second phase all nodes participate in I/O: they evaluate the integrals and write them to disk. The work is distributed across all nodes and each node accesses its *private* file where the locally computed integrals are stored. Because of the choice to use a private file per node, the code uses a sequential access pattern to write the results. Note that there is a single seek[3] operation (that corresponds to the file rewind) for each node, followed by intense read activity.

[3]This seek operation is more clearly illustrated in Figures 4(b) and 4(d).

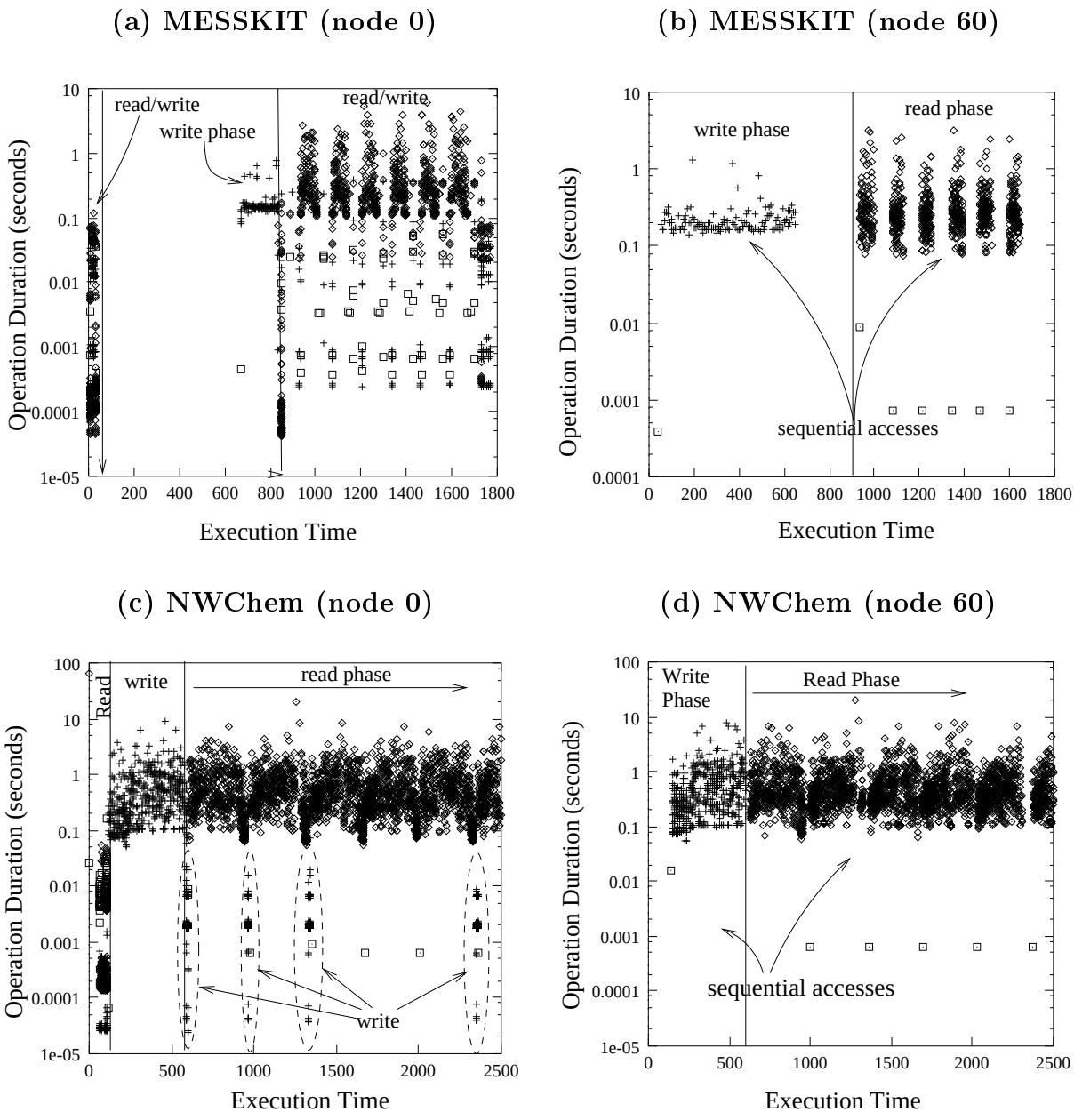

Figure 4: Operation durations for MESSKIT and NWChem. Boxes represent seeks, diamonds represent reads, and crosses represent writes.

Finally, in the third code phase all nodes concurrently open their private files where the integrals are stored, and they repeatedly read integral data, construct the Fock matrix, compute, synchronize, and solve the SCF equations. Node 0 periodically collects the results and writes them to disk as indicated by the short bursty write activity in Figures 4(a) and 4(b). Again, the read access pattern is sequential, with a single seek preceding bursts of intense read activity on the file.

5.2 Interleaved Access Pattern

The different parallelization choices of the QCRD and ESCAT codes are clearly illustrated in Figure 5. Instead of using a private file per node to store the data segment of the global matrix

apportioned to each node, the developers opted to use one file per global matrix. Consequently, an interleaved access pattern was used by each node to access its portion of the matrices, with one seek operation preceding every read or write operation. The interleaved access pattern used by QCRD is clearly distinguished in Figure 5(a) that illustrates a snapshot of the I/O activity towards the end of code execution.[4]

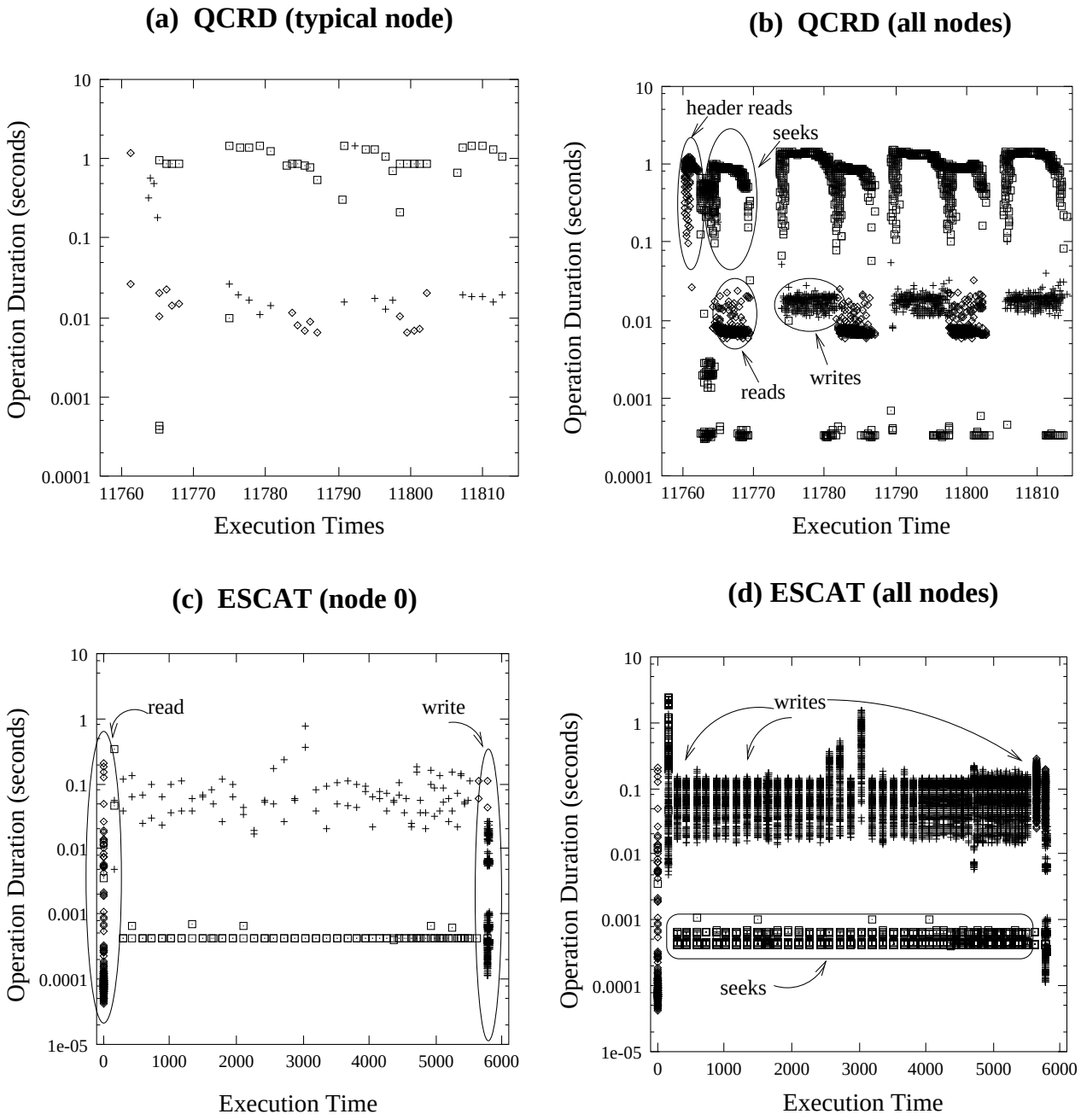

Figure 5: Operation durations for QCRD and ESCAT. Boxes represent seeks, diamonds represent reads, and crosses represent writes.

To give a flavor of the file system responses using the interleaved access pattern, we depict the duration of operations of all nodes in Figure 5(b) during the same time period. Because

[4]Because there are no significant differences in the I/O volume transfered by node 0 comparing to the rest of nodes, we only illustrate the activity of a typical node.

of standard UNIX file sharing semantics, seek requests are serialized when more than one processes access the same file, and the average seek cost increases commensurately with the number of concurrent accesses.

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.

The designers of ESCAT took advantage of the available `M_ASYNC` file system mode to access the files using an interleaved access pattern and managed to decrease the average seek time by two orders of magnitude (see Figures 5(c) and 5(d)). `M_ASYNC` allows 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 6 illustrates the duration and spacing of I/O operations for the PRISM code. Because of the developers decision to channel all write operations through node 0, the I/O activity of the remaining nodes is confined within the first 200 seconds of code execution.

Figure 6(a) depicts the I/O activity of node 0 for the first 1,700 seconds of application execution time. Again, for visualization purposes, we restricted the plotting area to only a part of the execution time. The application phases are clearly visible in the figure: in the first 200 seconds all nodes access the initialization files and in the subsequent, write intensive phase, the simulation is carried out and results are written to disk using a sequential access pattern.

The mixed sequential and interleaved access patterns used to access the initialization files are clearly demonstrated in Figure 6(b). During the read phase three files are accessed: the first file contains the problem parameters and is accessed with a sequential pattern by all nodes. The second file is a restart file that contains the initial conditions for the simulation, and each node first reads the file header sequentially and then accesses its own portion in an interleaved fashion. Finally, the third input file is accessed sequentially by all nodes.

6 Discussion

Characterization studies are by their nature inductive, covering a subset of the possibilities. Although the five applications codes of this study differ dramatically in their algorithmic approaches, they are subject to several general observations about their common 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

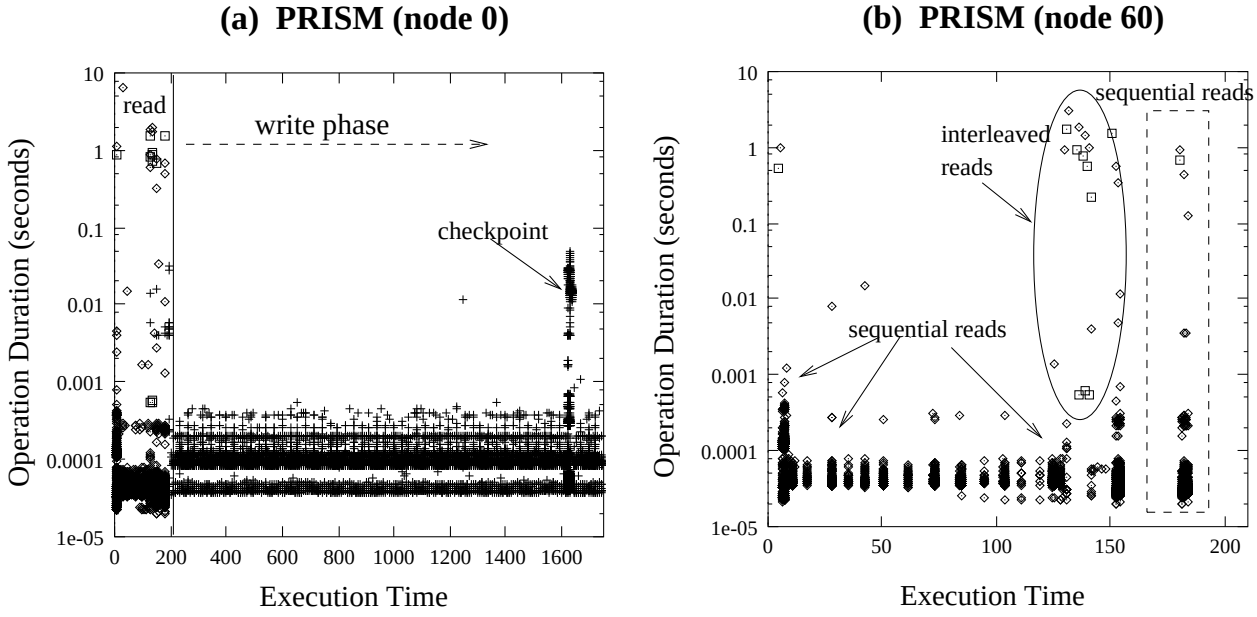

Figure 6: Operation durations for PRISM. Boxes represent seeks, diamonds represent reads, and crosses represent writes.

before initiating a transfer or disabling all caching and bypass 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. Often, the file header is accessed via a number of sequential operations, while the main file body is accessed through a strided pattern since data are equally distributed among nodes. Due to the high concurrency of such operations, efficient file pointer consistency protocols are necessary to deliver high performance.

- Request parallelism ranges from no parallelism (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 opted by application developers when their codes experience performance bottlenecks with concurrent accesses. However, such solutions require unnecessary synchronizations and ineffective use of the parallel file system. 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 [6] is one of the goals of the SIO effort. Our I/O characterization study has been instrumental in the design and implementation of MPI-IO [28] that is adopted by the SIO community as the standard parallel I/O interface.

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 [15], 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

We have presented a comparative study of parallel I/O access patterns, commonly found in I/O intensive scientific applications. Even with the restricted application subset that we investigated here, we demonstrated that there is much variation in temporal and spatial 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.

Acknowledgments

The MESSKIT and NWChem codes was provided by Rick Kendall David Bernholdt of the Pacific Northwest Laboratory. The PRISM code was obtained from Ron Henderson and Dan Meiron of Caltech. The QCRD code was provided by Mark Wu and Aaron Kuppermann at Caltech. The ESCAT code was provided by Vincent McKoy and Carl Winstead at Caltech. Phyllis Crandall, Ruth Aydt, and Sharon Burnett contributed to application instrumentation. All data presented here were obtained from code executions on the Intel Paragon XP/S at the Caltech Center for Advanced Computing Research.

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] C., W., H., P., and V., M. Parallel computation of electron-molecule collisions. *IEEE Computational Science and Engineering* (1995), 34–42.

[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.

[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] 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.

[7] Crandall, P., Aydt, R. A., Chien, A. A., and Reed, D. A. Input/Output characterization of scalable parallel applications. In *Proceedings of Supercomputing 1995* (1996), pp. CD–ROM.

[8] Foster, I., and Nieplocha, J. ChemIO: High-performance I/O for computational chemistry applications. WWW http://www.mcs.anl.gov/chemio/, February 1996.

[9] Henderson, R. D., and Karniadakis, G. E. Unstructured spectral element methods or simulation of turbulent flows. *Journal of Computational Physics 122(2)* (1995), 191–217.

[10] High Performance Computational Chemistry Group, Pacific Northwest National Laboratory. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 1.1.* Richland, Washington, 99352, USA, 1995.

[11] Intel Corporation. *Paragon System User's Guide.* Intel SSD, Beaverton, OR, 1995.

[12] King, S. M. *Installing, Managing, and Using the IBM AIX Parallel I/O File System.* Information Development Department, IBM Kingston, New York, 1994.

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

[14] 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.

[15] Madhyastha, T., and Reed, D. A. Intelligent, adaptive file system policy selection. In *Proceedings of Frontiers'96* (1996).

[16] Miller, E. L., and Katz, R. H. Input/Output behavior of supercomputer applications. In *Proceedings of Supercomputing '91* (November 1991), pp. 567–576.

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

[18] Nieuwejaar, N., and Kotz, D. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing* (May 1996). To appear.

[19] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C. S., and Best, M. File-access characteristics of parallel scientific workloads. Tech. Rep. PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. To appear in IEEE TPDS.

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

[21] 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.

[22] Poole, J. T. Scalable I/O Initiative. California Institute of Technology, Available at `http://www.ccsf.caltech.edu/SIO/`, 1996.

[23] 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.

[24] 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.

[25] 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.

[26] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95* (December 1995).

[27] 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.

[28] The MPI-IO Committee. MPI-IO: a parallel file I/O interface for MPI, April 1996. Version 0.5.

[29] 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* (Philadelphia, May 1996), pp. 28–40.

[30] Wu, Y.-S. M., Cuccaro, S. A., Hipes, P. G., and Kuppermann, A. Quantum chemical reaction dynamics on a highly parallel supercomputer. *Theoretica Chimica Acta 79* (1991), 225–239.