Algorithmic Influences on I/O Access Patterns and Parallel File System Performance *

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

C.E. Elford A.J. Lavery A.A. Chien
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
{elford,alavery,achien}@cs.uiuc.edu

Abstract

For many scalable parallel applications, the input/output (I/O) barrier rivals or exceeds that of computation and interprocessor communication. Consequently, scalable parallel secondary and tertiary storage systems are necessary to satisfy the resource demands of many national challenge problems. At present, one major challenge facing the designers of such storage systems is the wide range of I/O access patterns and the lack of general purpose file system policies that achieve high performance for variable I/O requirements.

We analyze the I/O behavior of two scientific applications on the Intel Paragon XP/S. Although the two applications solve the same scientific problem and their I/O access patterns are qualitatively similar, their interactions with the file system are decidedly different. Our results show that appropriate tuning of file system policy parameters to I/O demands can significantly increase I/O throughput.

1 Introduction

In many application domains where manipulating large volumes of data is a necessity, scalable input/output (I/O) is critical for performance. Moreover, as computation and communication hardware speeds continue to rapidly increase, I/O represents a growing fraction of application execution time. Scalable parallel secondary and tertiary storage systems are essential to alleviate this performance bottleneck.

Our growing experience with parallel I/O characterization [6] underscores that for reducing application execution time, it is necessary to tune the application I/O requirements via code restructuring to the idiosyncrasies of the underlying parallel file system. Alternatively, file system policies that adjust data distribution across storage devices and adapt their data management policies to accommodate the application I/O access patterns offer a viable alternative to code restructuring.

The Portable Parallel File System (PPFS) [4] is an experimental software system that provides a flexible application programming interface (API) for controlling data distribution and data management policies. It is implemented as a user level library and provides sufficient flexibility for selecting cache sizes and replacement policies, prefetch and write behind thresholds and limits. In this study we examine the I/O behavior of two scientific applications from the computational chemistry domain. We investigate their I/O resource demands on the Intel Paragon XP/S. We concentrate on I/O request sizes and volume, temporal and spatial access patterns, and distribution of I/O demands across processors. Guided from this characterization, we fine tune PPFS to meet the individual application requirements.

The remainder of the paper is organized as follows. In §2 we present a description of the two applications, the experimentation methodology, and a detailed I/O characterization of the two applications. In §3 we present an overview of the PPFS I/O library, followed in §4 by reduction of application execution time using PPFS. Finally, §5 summarizes our findings.

2 Application I/O Characterization

We examine the I/O behavior of two chemistry codes that perform *ab initio* electronic structure calculations using the Hartree-Fock algorithm [3].

*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. Andrew Chien is supported in part by an NSF Young Investigator Award CCR-94-57809.

MESSKIT and NWChem are two distinct Fortran implementations of the Hartree-Fock self-consistent field method. Both implementations use as input the basis functions derived from the atoms and the relative geometry of the atomic centers. Then, atomic integrals are calculated over the basis functions, representing the Coulomb interactions between electrons, and an approximate molecular density is produced. A Fock matrix is derived using this density and the atomic integrals. Finally, a self-consistent field (SCF) method is used until the molecular density converges to within an acceptable threshold. From an I/O perspective, both codes have the same logical structure of three phases.

- **Phase One:** Initialization data is read from the various input files. During this phase, the problem definition is loaded, the basis sets are calculated, and written to disk. Only one node (processor) actively participates in I/O during this phase.
- **Phase Two:** Integrals are evaluated and results are written to disk. The work is distributed across all nodes and each node writes a private file storing the locally computed integrals. All nodes actively participate in I/O during this phase.
- **Phase Three:** In this 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.

Although the two applications solve essentially the same problem using the same logical sequence of steps, there are differences between the two codes. At a high level, NWChem operates as a single program. MESSKIT consists of three programs, *psetup*, *pargos*, and *pscf*, that correspond to the three logical phases. Although the qualitative behavior of the two applications is the same, their quantitative spatial and temporal I/O behavior differs due to the different algorithmic and implementation approaches chosen by their developers [2]:

1) In NWChem, the integrals are written to disk in parallel with the first Fock matrix construction, procedures that are separated in MESSKIT.

2) The two codes use different integral evaluation libraries, with different timings of the integral evaluation phase.

3) 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.

4) The codes use different data formats and produce different volumes of data even for matching inputs.

5) Although both codes are written in Fortran, MESSKIT uses C I/O calls while NWChem uses standard Fortran calls.

Regardless of the algorithmic implementation, the nature of the Hartree-Fock computation is dominated by high I/O requirements because the I/O data volume is a function of the number of required integrals. A basis set of size N generates $O(N^2)$ one electron integrals and $O(N^4)$ two electron integrals, the data volume grows dramatically with matrix size and is substantial even for small matrices.

Our experiments were conducted on the 512-node Intel Paragon XP/S at the Caltech Center for Advanced Computing Research. The machine provides a variety of disk hardware for secondary storage. There are 12 I/O nodes each controlling a 2GB RAID-3 disk array. There are also 64 4GB Seagate disks, each attached to a computation node. The user needs to select the file system hardware where the program's files reside. Across the selected partition, files are striped using the Intel Parallel File System (PFS) default attributes (i.e., files are striped across disks in units of 64 KBytes in a round-robin fashion).

To characterize and compare the I/O access patterns of MESSKIT and NWChem, we used the Pablo performance environment's I/O instrumentation software [5] to capture each code's I/O behavior including the individual request sizes, their timings, and their spatial and temporal patterns. This characterization will drive in §4 the tuning of the Portable Parallel File System (PPFS) policies.

2.1 I/O Characteristics

As a baseline for I/O performance analysis, we considered two representative data sets, one for each code. Although data sets that solve the same problem are necessary to conduct a consistent cross-application comparison, our focus is on the qualitative I/O behavior of the two codes and subsequent identification and elimination of I/O performance bottlenecks.

We executed MESSKIT and NWChem on 64 nodes with their files striped across the 12 RAID-3 disk arrays.[1] For the relatively small data set used in MESSKIT, the execution time of the three phases was 1,288 seconds. A larger data set was used in NWChem

[1] Experiments using the Seagate disks were also performed but are not presented here for the sake of brevity. The faster Seagate disks reduce the time spent in I/O but the application qualitative I/O behavior remains the same.

MESSKIT

Oper.	Oper. Count	I/O Time Percentage	Exec. Time Percentage
open	163	0.66	0.28
read	51,829	67.64	29.11
seek	497	0	0
write	9,163	28.95	12.54
close	160	0	0
flush	8,626	2.75	1.18
All I/O	70,438	100.00	43.11

NWChem

Oper.	Oper. Count	I/O Time Percentage	Exec. Time Percentage
open	92	0.12	0.09
read	465,154	92.53	72.13
seek	2,115	0.01	0.01
write	32,398	6.91	5.39
close	153	0.43	0.33
All I/O	499,912	100.00	77.95

Table 1: Application I/O summaries.

with a total execution time of 5,439 seconds.

Table 1 presents a high-level description of the I/O activity of the two codes. At a first glance, NWChem appears significantly more I/O intensive than MESSKIT: 77.95% of the program execution time is consumed by I/O operations, making NWChem look almost two times more I/O intensive than MESSKIT. This effect is partially an artifact of the different data sets used for the two code executions resulting in larger integral data volume for NWChem, and partially a direct effect of the number of iterations and Fock matrix constructions prompted by the different algorithmic methods used by the two codes to optimize the molecular orbitals. As Table 1 indicates, the number of writes in NWChem is much higher than in MESSKIT, with a total integral volume almost three times larger than that of MESSKIT. Similarly, the self-consistent field calculation phase is distinctively read intensive since each node repeatedly reads the integral files and solves the SCF equations.

Apart from `read` and `write` operations, the remaining I/O operations performed by the code are insignificant in both count and relative cost. The small number of seeks in the code, caused by file rewinds, is a direct effect of the developers' choice to use a private file per node to store the computed integrals. Consequently, since there is a single file for each node, it is accessed with a simple sequential access pattern. In summary, Table 1 indicates that the major I/O bottleneck for both codes is primarily the cost of `read` operations, followed by the modest cost of `write` operations.

2.2 Temporal I/O Behavior

The summary of the application I/O characteristics gives a high level description of the I/O; the temporal spacing of operations and their request sizes also affects performance. Figure 1 illustrates the volume and intensity of reads and writes, the two most expensive operations in MESSKIT. Request sizes as a function of the program execution time are illustrated in Figures 1(a)-1(b). The costs of these operations are plotted in Figures 1(c) and 1(d) respectively. The three I/O phases are clearly distinguished in the figure. The first code phase, where initialization data are read by node 0 and a few initial results are written to disk, is short and its contribution to I/O performance is insignificant.

The second I/O phase, where the integral data are written to disk is shown in the left region of Figures 1(a) and 1(c). Write request sizes are identical and equal to 81,920 bytes. Write times vary considerably (note that operation durations are on a logarithmic scale). This variation is a direct effect of the load on the disks since all nodes participate in I/O during the integral calculation phase. The right region of Figures 1(a) and 1(c) illustrates the moderate write activity of the third code phase. These writes are of variable size, ranging from four bytes to 204,800 bytes and their costs are modest because they are administered only through node 0 and disk load is minimized.

The intense `read` activity of the third code phase is depicted in Figures 1(b) and 1(d). Read requests are bimodal (either equal to 81,920 bytes or equal to 204,800 bytes[2]), and their durations highly variable. Again, we attribute the high variability of `read` operations to the simultaneous accesses by all nodes. For the data set used in this execution, the integral files are read six times.

Figure 2 illustrates the temporal spacing of `read` and `write` operations in NWChem. The temporal spacing is alike in structure with that of MESSKIT (i.e., the three code phases are clearly distinguished). Request sizes are either very small or equal to 65,544 bytes, with the majority of requests being large.

In summary, observing the temporal I/O behavior of the two codes, we conclude that despite of their algorithmic differences, the two codes share many common I/O characteristics and problems. First, they

[2] The large requests of 204,800 bytes are performed by node 0 only; they are requests to an ancillary file.

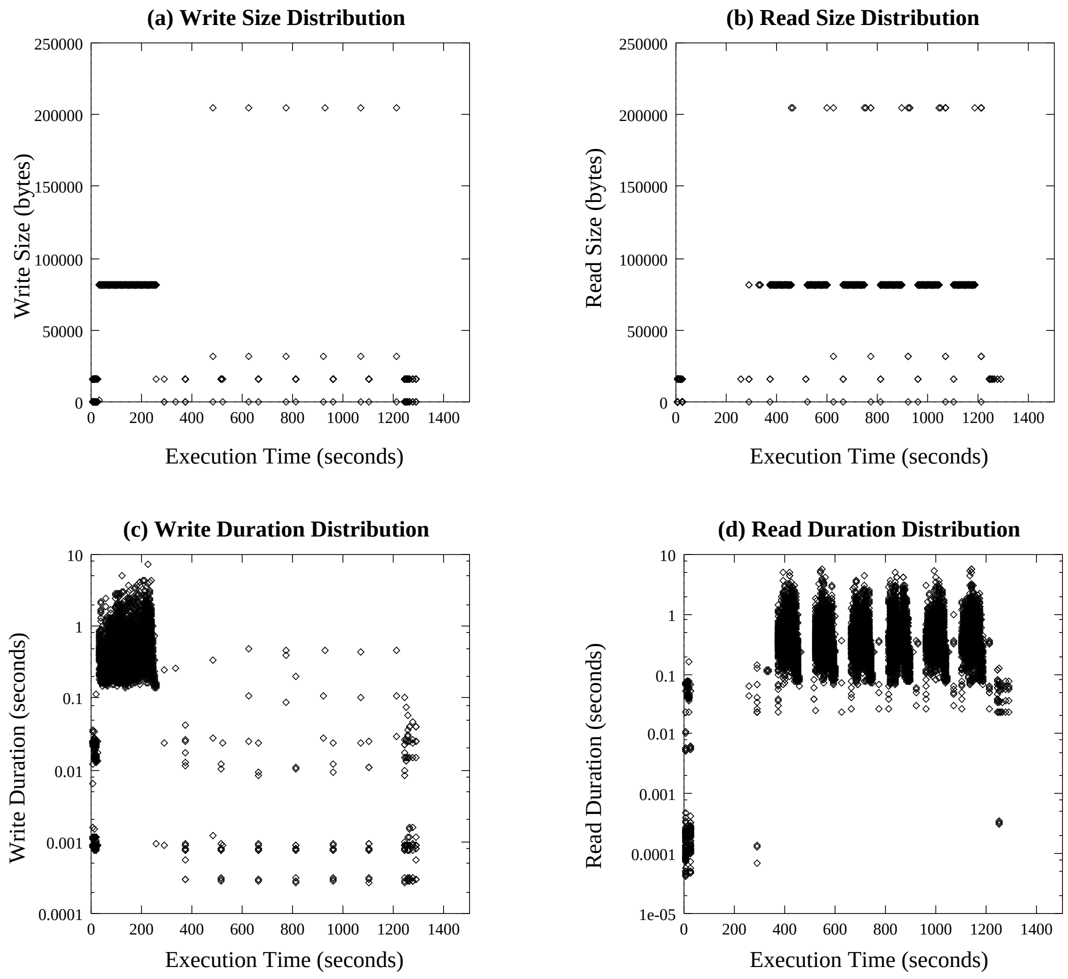

Figure 1: MESSKIT I/O behavior (Intel Paragon XP/S).

both have the same general I/O structure of three distinct phases. Second, not only I/O parallelism remains the same across the two codes but all nodes also use the same, simple access pattern dominated by sequential `read` and `write` operations to private files. The majority of requests are large and greater than one disk stripe size (i.e., 64 KB). Third, both applications exhibit the same I/O bottlenecks: read operations contribute a grave percentage to I/O times followed by write operations. Reducing primarily the cost of reads and secondarily the cost of writes will drive the various PPFS tunings presented in the following sections.

3 Portable Parallel File System

Various studies have suggested that to improve on I/O performance and reduce application execution times, the application developer should carefully tune application access patterns to match the idiosyncrasies of the parallel file system and the available, default file system policies [6, 1]. This approach reduces the application portability across various platforms but is necessary to improve application I/O transfer rates. An alternative approach is to adjust the file system policies to the application I/O patterns.

The Portable Parallel File System (PPFS) is a user level parallel I/O library that allows rapid exploration of new file system policies across several hardware platforms. Its major advantage is its extensible infrastructure for portable, parallel I/O [4]. PPFS currently runs on the Intel Paragon, the IBM SP/2, the SGI Power Challenge, the Convex Exemplar, workstation and Linux clusters.

PPFS is designed using a client/server model. When an application starts execution, *clients* consisting of user code linked with the PPFS library are spawned. In addition, a *PPFS server* is spawned

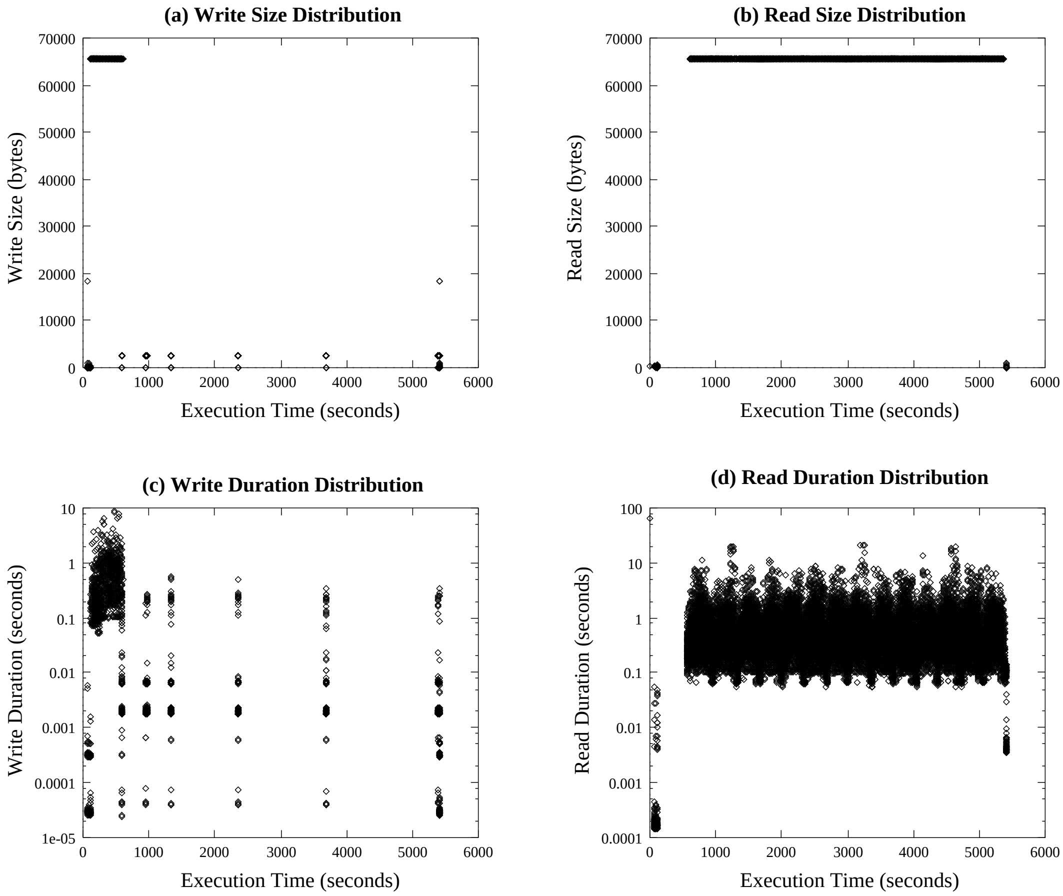

Figure 2: NWChem I/O behavior (Intel Paragon XP/S).

for each logical disk managed by PPFS. A *metadata server* coordinates file opens and closes and maintains data distribution mappings of file data to PPFS servers. The PPFS client library consults the parallel file metadata to decide which PPFS servers should service a given read or write request.

PPFS views parallel files as a sequence of records. File accesses can be of fixed or variable length records, but the application may read or write entire records only. Files can be distributed across the PPFS servers according to various data distribution schemes. PPFS has a flexible application programming interface (API) that requires a modest human effort to control *caching, prefetching,* and *write behind.* Using these controls, the user can adjust the allocated memory for caching file data, cache replacement policies, the amount of prefetched data, and write behind thresholds. However, caution is required while using these controls: they must be well tuned to the application access pattern and hardware characteristics. An inappropriate prefetching technique, (e.g., sequential prefetching for a random access pattern), can degrade performance. Prefetching in a timely fashion is also of great import: prefetching too early may cause useful data to be flushed from cache before ever being referenced, prefetching too late may diminish possible performance improvements.

In this case study, we demonstrate that even for applications with closely resembling I/O requirements as outlined in §2, a good understanding of the application I/O access patterns is required to adjust file system controls. In the following section, we illustrate how we use the characterization information to tune PPFS.

4 PPFS Improvements

As illustrated in Table 1, the most expensive I/O operations in both codes are primarily reads and secondarily writes. Therefore, we concentrate on reducing the cost of these operations by activating the prefetch and write behind PPFS engines. Guided by the code I/O behavior, we first focus on selecting the appropriate prefetch and write behind PPFS parameters. Determining such parameters by exhaustively running the applications across the entire range of possible values is prohibitively expensive. An alternative, cost-effective method is to construct a microbenchmark that reflects application behavior and quickly assesses the interaction of application access patterns with file system policies.

We have constructed a synthetic benchmark that mimics the behavior of the second and third logical phases of MESSKIT and NWChem. We parameterize this synthetic workload by a set of input parameters including the number of active processes, request type (i.e., read or write), request size, and interrequest times. To effectively reproduce the application I/O behavior, we used 64 active processes, and record sizes of 81,920 bytes for MESSKIT and 65,544 bytes for NWChem.

To determine the best write behind parameters, we repeatedly executed the benchmark by varying the number of accumulated records in cache before sending a request to disk. The experiments indicated that for the request sizes used in MESSKIT and NWChem, writes are faster if caching is disabled and each record is immediately written to disk. Had the record size been smaller, caching several records before flushing them to disk would have offered higher performance.

The PPFS prefetch mechanism allows a client node to specify the number of records to keep in cache at any time, trying to always stay A records ahead of the client's most recent data request. The prefetch engine is also instructed to prefetch a unit of U records at a time. Because each I/O operation incurs a transaction overhead and a transfer cost, the prefetch unit should be large enough that transfer cost exceeds transaction overhead and client computation is overlapped with I/O transfer. To parameterize our benchmark, we plotted the time each process spends in computing before issuing the next read request. Figure 3 illustrates the MESSKIT and NWChem read interrequest times for a typical processor for each of the Fock matrix constructions. Recall that in the third phase each node reads its private file and builds the Fock matrix as many times as needed by the algorithm. For the datasets used in the characterization of §2, MESSKIT requires six Fock matrix constructions and NWChem requires fourteen Fock matrix constructions. The comparison of Figures 3(a) and 3(b) is striking. In MESSKIT, the read interrequest times remain tightly clustered by a mean of 40 milliseconds. Interrequest times in NWChem have an average of 50 milliseconds but they vary widely within each of the fourteen Fock matrix constructions.

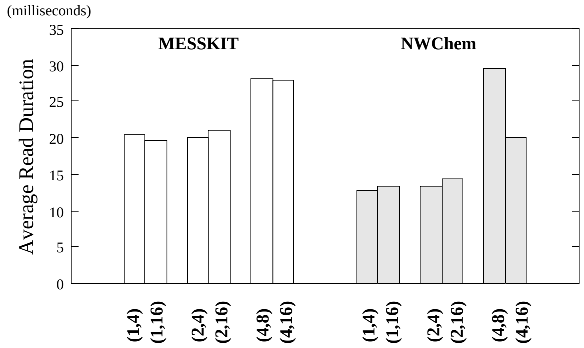

Figure 4: Average read time using the benchmark for various prefetch parameters (U, A).

Guided by Figure 3, we used interrequest times of 40 milliseconds for the benchmark that simulated the MESSKIT behavior, and 50 milliseconds for the benchmark that simulated the NWChem behavior. Figure 4 illustrates a representative sample of the achieved average read times by changing the number of prefetched units U and the total number of records kept in cache A. On the Intel Paragon, requests greater than 64 KB (i.e., the default stripe size) perform well [6]. Because both NWChem and MESSKIT use records larger than 64 KB, a prefetch unit of 1 record is a good performance choice. However, excessively large prefetch units incur additional overhead. When a PPFS prefetch operation returns to a client, data is copied in the client cache by an asynchronous operation handler thread. Due to the high overheads in memory copies in the i860 architecture, substantial delays may occur as the user process attempts to disable the asynchronous thread. Figure 4 shows that when the prefetch unit rises from two records to four records, read operations incur sufficient delays. If the records were smaller or the interaccess times were longer, larger prefetch units would have achieved good performance. Therefore, the request size determines the minimum effective prefetch unit, while the interaccess latency determines the maximum prefetch unit that can effectively be interleaved with computation.

In contrast to the prefetch unit which can be determined by the temporal access pattern and the request size, the total amount of prefetching is bound by the amount of memory available. Both NWChem and MESSKIT allocate a substantial amount of memory for the SCF method. Therefore, the amount of memory available to perform prefetching is limited by the application memory requirements and is less than one megabyte.

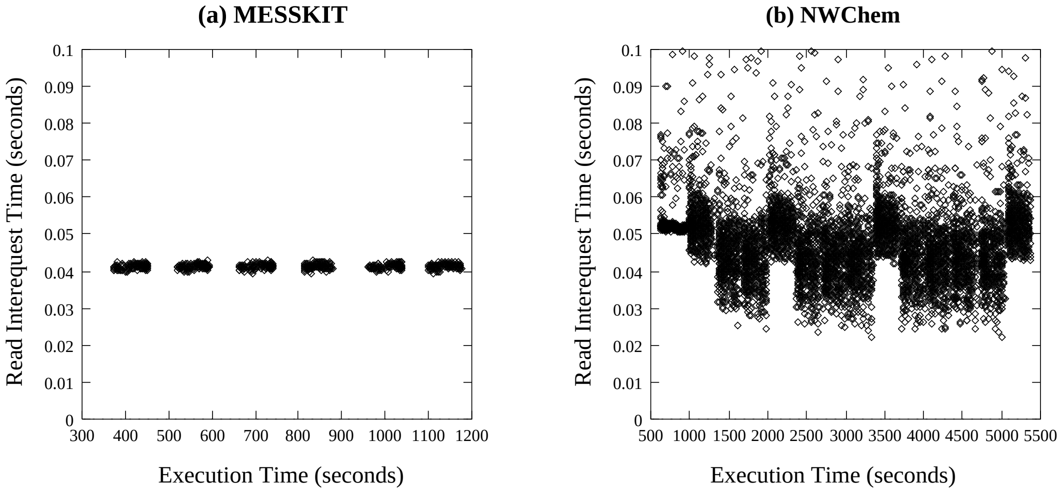

Figure 3: Read interrequest times of MESSKIT and NWChem.

Using the prefetch parameters suggested by Figure 4, we parameterized the PPFS prefetch engine with $U = 1$ and $A = 4$ for MESSKIT, and $U = 1$ and $A = 16$ for NWChem. Dramatic reductions in code execution times with PPFS were obtained. The MESSKIT execution time dropped from 1288.02 seconds to 784.03 seconds. The NWChem execution time dropped from 5439.33 seconds to 2930.32 seconds.

The C I/O library demonstrates better performance than the Fortran library on the Intel Paragon. Since the PPFS API is translated to C calls, to distinguish the PPFS gains in NWChem from the gains of plain C I/O, we translated all Fortran I/O calls to C I/O calls. This simple change reduced the NWChem execution time by almost 2,000 seconds. However, activating the PPFS prefetch engine, the execution time dropped by 600 additional seconds, reaching 2930.32 seconds.

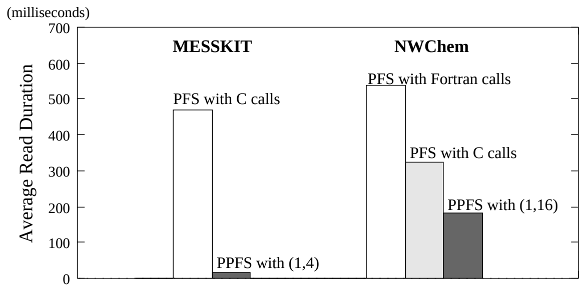

Figure 5: Average read duration in MESSKIT and NWChem.

Figure 5 illustrates the average cost of PPFS reads which substantially reduced the time consumed in the third code phase. MESSKIT's uniform interaccess intervals shown in Figure 3(a), create a controlled, regular load to the PPFS servers. Recall that if an I/O request issued by the client process can not be served by the client's cache, it is immediately forwarded to the PPFS servers. This steady stream of I/O requests to servers results in excellent reduction in the average read duration for MESSKIT. Unfortunately, PPFS read times for NWChem are not as favorable as those for MESSKIT. Unlike MESSKIT, NWChem exhibits extremely high variability in interaccess intervals (see Figure 3(b)). Consequently, the prefetched records in the client's memory are not always enough to satisfy the client's stream of requests. Therefore, the higher number of cache misses places severe loads on the PPFS servers and queuing delays result in higher service times.

The different degree of cache hits in the two codes is also illustrated in Figure 6, where the temporal spacing of read and write operations along with their durations is presented. During MESSKIT's read phase, two classes of performance are achieved and correspond to cache hits and cache misses. Cache hits are serviced in less than five milliseconds.[3] Cache misses are serviced in less than 100 milliseconds, significantly better that the MESSKIT durations depicted in Figure 1(d). Unfortunately, these misses cannot be further eliminated since MESSKIT's memory requirements restrict each client's cache size.

The NWChem read phase is also shown in Fig-

[3]The regular temporal spacing of requests in MESSKIT cause many read requests to occur simultaneously. Because of the scale of Figure 6(a), symbols are plotted one on top of the other giving the illusion that there are less read operations comparing to Figure 1(d).

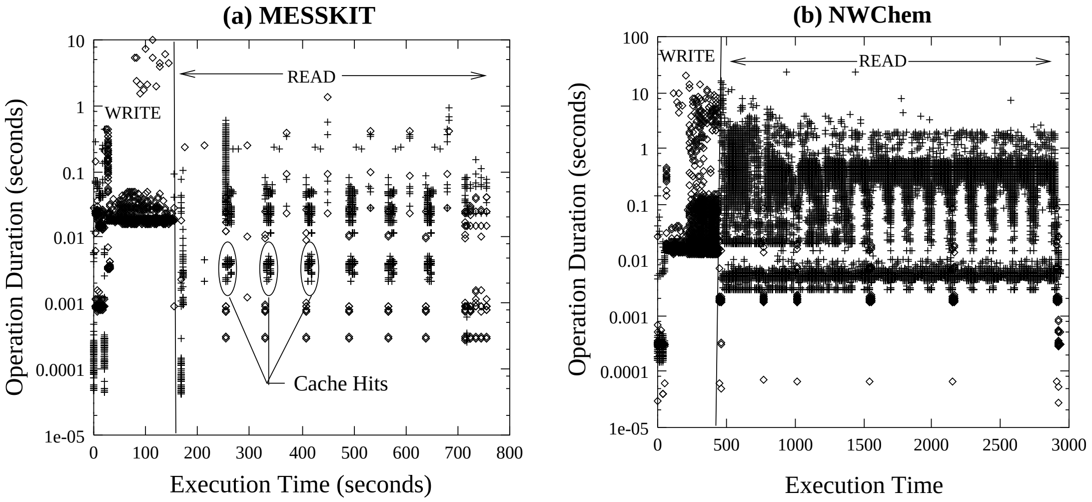

Figure 6: Duration of write and read operations with PPFS. Diamonds represent write operations, crosses represent read operations.

ure 6(b). As in MESSKIT, cache hits are serviced in five to ten milliseconds. The high variability of interaccess intervals causes a larger number of cache misses and greater PPFS server queuing overheads. The resulting cache misses cost is between 15 and 1500 milliseconds. While performance in Figure 6(b) shows substantial improvement in the read durations of Figure 2(d), it also suggests that there are more optimization possibilities. An ideal prefetching policy, would take advantage of the application's varying interaccess intervals to dynamically select the appropriate degree of prefetching for each of the fourteen Fock matrix constructions.

5 Conclusions

We have analyzed the I/O behavior of two scalable parallel implementations of the Hartree-Fock self-consistent field method. Although the qualitative behavior of the I/O access patterns of the two applications is similar, the different algorithmic and implementation approaches affect their quantitative I/O demands. In consequence, the applications' interactions with the file system are distinctly disparate.

Using the I/O characterization results, we finely tuned the data management policies of the Portable Parallel File System (PPFS), an experimental parallel file system that provides a flexible application programming interface (API) for controlling data distribution and data management policies. Our results demonstrate that significant improvements can be achieved if the data management policies are adjusted to accommodate the application I/O requirements.

Acknowledgments

The MESSKIT and NWChem codes were provided by Rick Kendall and David Bernholdt of the Pacific Northwest Laboratory. David Bernholdt's help in understanding the functionality of the two codes was invaluable. 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] Acharya, A., Uysal, M., Bennett, R., Mendelson, A., Beynon, M., Hollingsworth, J. K., Saltz, J., and Sussman, A. Tuning the performance of I/O intensive parallel applications. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems* (Philadelphia, May 1996), pp. 15–27.

[2] Bernholdt, D. Personal communication, August 1996.

[3] 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.

[4] Huber, 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 9th ACM International Conference on Supercomputing* (July 1995), pp. 385–394.

[5] 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.

[6] 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.