I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon[*]

Rajeev Thakur *Ewing Lusk* *William Gropp*

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439

`{thakur, lusk, gropp} @mcs.anl.gov`

Preprint MCS–P534–0895
Revised October 1995

Abstract

Many large-scale applications on parallel machines are bottlenecked by the I/O performance rather than the CPU or communication performance of the system. To improve the I/O performance, it is first necessary for system designers to understand the I/O requirements of various applications. This paper presents the results of a study of the I/O characteristics and performance of a real, I/O-intensive, portable, parallel application in astrophysics, on two different parallel machines—the IBM SP and the Intel Paragon. We instrumented the source code to record all I/O activity, and analyzed the resulting trace files. Our results show that, for this application, the I/O consists of fairly large writes, and writing data to files is faster on the Paragon, whereas opening and closing files are faster on the SP. We also discuss how the I/O performance of this application could be improved; particularly, we believe that this application would benefit from using collective I/O.

[*]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

1 Introduction

Parallel computers are being increasingly used for applications requiring a large amount of computational power. Many such applications also deal with large quantities of data, and hence have significant I/O requirements. The Applications Working Group of the Scalable I/O Initiative has compiled a list of several data-intensive applications and their I/O requirements [1]. The I/O performance of parallel computers has always been orders of magnitude lower than the CPU and communication performance because improvements in the I/O subsystem have not kept pace with improvements in the CPU and communication subsystems. As a result, I/O is often the bottleneck for large-scale applications on parallel computers.

Improving the I/O performance of parallel computers is critical to realizing their full potential in solving problems which were previously considered intractable. This requires coordinated improvements at all levels—I/O hardware, parallel file systems, operating systems, application program interfaces, runtime libraries, compilers, and languages. However, it is first necessary to understand the I/O characteristics of applications, so that we know what the common I/O requirements are, and what improvements can be made to better support these requirements.

This paper presents the results of a study of the I/O characteristics of a real, large-scale, I/O-intensive, portable, parallel application in astrophysics [21], on two different parallel computers—the IBM SP and the Intel Paragon. We instrumented the source code using the Pablo performance analysis environment [30] and collected trace files on both systems. The traces were analyzed and visualized using Upshot [16]. The objectives of this study were to understand the I/O behavior of the application on each individual machine and to compare the I/O performance of the two machines for the same application. Note that we instrumented the existing parallel code for the application. We did not parallelize the sequential version, nor did we modify any of the I/O already present in the code.

In the next section, we describe some of the work done previously in the area of I/O characterization. Section 3 gives an overview of the astrophysics application used in this study. Details about how the code was instrumented and analyzed, and the various experiments performed, are provided in Section 4. The results of our study on the IBM SP and the Intel Paragon are presented in Sections 5 and 6 respectively. In Section 7, we discuss how the I/O performance of this application could be improved, and draw overall conclusions in Section 8.

2 Related Work

The file access characteristics of applications on uniprocessor and vector machines have been studied quite extensively. Ousterhout et al. [24] and Floyd and Ellis [12] studied file access patterns on

Unix workstations. Baker et al. [4] analyzed the user-level file access patterns and caching behavior of the Sprite distributed file system. Smith [32] studied the file access patterns on IBM mainframes at the Stanford Linear Accelerator Center. Ramakrishnan et al. [28] performed a study of the I/O access patterns in a commercial computing environment on a VAX/VMS system. Jensen and Reed [17] studied the patterns of access to the file archive of the National Center for Supercomputing Applications. Miller and Katz [22] traced certain I/O-intensive applications on the Cray Y-MP at NASA Ames Research Center. Pasquale and Polyzos [25, 26] studied the file access characteristics of scientific applications on the Cray Y-MP and Cray C90 at San Diego Supercomputer Center.

Only recently have there been any efforts to characterize the file access patterns on parallel machines. Kotz and Nieuwejaar [19] present the results of a three-week tracing study in which all file-related activity on the Intel iPSC/860 at NASA Ames Research Center was recorded. They did not instrument individual applications; instead, they instrumented the Concurrent File System (CFS) library routines called by all applications, and studied the file-related activity of all applications together. A similar study on the CM-5 at the National Center for Supercomputing Applications was performed by Purakayastha et al. [27]. The results of these two studies are compared and contrasted in [23]. They found that file sizes were large, request sizes were fairly small, data was accessed in sequence but with strides, and I/O was dominated by writes.

Crandall et al. [8] instrumented and analyzed the I/O characteristics of three parallel applications on the Intel Paragon at Caltech. They found a wide variety of access patterns, including both read-intensive and write-intensive phases, large as well as small request sizes, and both sequential and irregular access patterns. Baylor and Wu [5] performed a study of the I/O characteristics of four parallel applications on an IBM SP using the Vesta parallel file system. They found I/O request rates on the order of hundreds of requests per second, mainly small request sizes, and strong temporal and spatial locality. Reddy and Banerjee [29] studied the I/O activity of five applications from the Perfect benchmark suite on an Alliant multiprocessor, and found only sequential accesses. Del Rosario and Choudhary [11] provide an informal summary of the I/O requirements of several grand-challenge applications. Cypher et al. [9] discuss the I/O requirements of eight parallel scientific applications.

Our work differs from previous efforts in that we have performed a comparative study of the I/O characteristics and performance of one portable parallel application on two different parallel machines. To the best of our knowledge, there has been no other study of the I/O characteristics of any application on different machines.

3 Application Overview

The application we benchmarked is an astrophysics code developed at the University of Chicago as part of a NASA/ESS-funded grand-challenge project on convective turbulence and mixing in astrophysics [21]. This application focuses on the study of the highly turbulent convective layers of late-type stars, like the sun, in which turbulent mixing plays a fundamental role in the redistribution of many physical ingredients of the star, such as angular momentum, chemical contaminants, and magnetic fields. The code solves the equations of compressible hydrodynamics for a gas in which the thermal conductivity changes as a function of temperature. The computational algorithm consists mainly of two independent components—a finite difference higher-order Godunov method for compressible hydrodynamics, and a Crank-Nicholson-based multigrid scheme. The two algorithms are combined by using a time-splitting technique. Further details about the application are given in [21, 6, 7].

The code has been implemented in Fortran 77 and C. The Chameleon library is used for both message passing and I/O [15]. Chameleon provides a uniform interface to whatever native message-passing library is available on a given machine, and also provides a portable high-level interface for I/O. As a result, the code is directly portable across a wide range of parallel machines.

The version of the code we instrumented solves a two-dimensional problem, with the size of the domain specified as an input parameter. The main data structures in the code are two-dimensional arrays of double-precision floating-point numbers. The application performs a significant amount of I/O and is bottlenecked by its I/O requirements [20]. The authors would like to solve three-dimensional problems, but are constrained by the I/O performance [20]. To reduce the amount of data transfer, I/O is performed in single-precision though computation is done in double-precision.

All arrays fit in main memory, so the application is essentially in-core. The solution algorithm is iterative: a typical run may require 20,000 iterations and take as long as 24 hours even on a parallel machine. Hence, the code has been provided with a checkpoint/restart facility. All relevant arrays are periodically written to files so that in the event of a system crash, the program can be restarted from the previous checkpoint. Certain arrays are also written to files periodically for the purpose of visualizing the progress of the computation in the form of a movie. Some arrays are also written periodically for data analysis. In summary, distributed arrays are written to files for three reasons—restart, movie, and data analysis. The number of times each type of write is to be performed can be specified as input parameters to the program. All I/O that takes place is for writes; there are no reads except when the program is restarted from a checkpoint. In this case, distributed arrays are read from previously stored files.

All I/O is performed using the Chameleon I/O library, which provides a portable set of routines for high-performance I/O that hide the details of the actual implementation from the user [13]. The user can select at run-time whether a file is stored as a simple Unix file (for compatibility with other tools) or as a parallel file readable only with the Chameleon I/O library routines (for maximum performance). In the astrophysics application, all data is stored in simple Unix files for portability reasons. The Chameleon I/O library uses a portable, high-level application program interface (API). For example, an entire distributed array can be written to a file with a call to a single routine called `PIWriteDistributedArray`. This routine is used extensively in the astrophysics application. For improved I/O performance, the Chameleon I/O library performs caching of data. All data to be written is first stored in a cache, and only when the cache gets full is data actually written to disks. This results in larger granularity data transfer between main memory and disks, and reduces the effect of high I/O latency. The user can either specify the cache size, or the library will use a default value.

4 Details of Experiments

We describe how the application code was instrumented and analyzed, and the various experiments that were performed.

4.1 Instrumentation and Analysis

We instrumented the source code using Pablo [30], a comprehensive performance analysis environment developed at the University of Illinois at Urbana-Champaign. The Pablo environment includes instrumentation libraries that can record timestamped event traces, a performance data meta-format called Self-Defining Data Format (SDDF) [2] and associated library, a parser for generating instrumented SPMD source code, and tools for analyzing and graphically displaying the traces.

Pablo also includes a special set of tools for capturing and analyzing I/O performance data [3]. The I/O extension to the Pablo trace library provides the user with a set of high-level routines that generate trace events corresponding to the I/O operations taking place in a program. A set of machine-independent I/O trace routines and events are provided for all platforms supported by the I/O trace extension. In addition, machine-dependent I/O trace routines and events are provided for some platforms such as the Intel Paragon. Both C and Fortran programs can be instrumented for generating traces of I/O events.

When an instrumented program is run, each processor creates its own trace file. Pablo provides a tool called `SDDFmerge` to merge the separate trace files into a single file. Other Pablo tools, such as `IOStats` and `IOtotalsByPE`, can be used to analyze the trace file, and provide detailed information about the I/O operations recorded.

We visualized and analyzed the trace files using Upshot [16], a tool for studying parallel program behavior developed at Argonne National Laboratory. Upshot takes as input a *logfile*, which is basically a list of the events of interest in a parallel program, in the order in which they occurred during the execution of the program. Upshot provides a view of the logfile, with events aligned on the parallel time lines of individual processors. Different events can be assigned different colors. Upshot displays all events in the logfile against time, and it is possible to zoom-in to any particular time location for a more detailed view. Upshot thus provides a global view of the trace data across all processors, which can often reveal interesting patterns and peculiarities in the performance of the code.

Upshot expects the logfiles to be in a specific format described in [16]. The traces created by Pablo are in a different format called Self-Defining Data Format (SDDF) [2]. Hence, we had to first convert Pablo trace files to the format required by Upshot. This conversion was trivial to do using Perl.

4.2 Machine Specifications

The instrumented code was run on the IBM SP at Argonne National Laboratory and the Intel Paragon at Caltech, which are the two testbeds for the Scalable I/O Initiative.

4.2.1 IBM SP

The IBM SP at Argonne National Laboratory has 128 compute nodes. Eight of these nodes also perform the function of I/O servers for the parallel I/O system. Each of these eight nodes is an RS/6000 Model 970 with 256 Mbytes of main memory; each of the other 120 nodes is an RS/6000 Model 370 with 128 Mbytes of main memory. Each node is provided with a 1 Gbyte local disk, of which 400 Mbytes are available to users, the rest is for paging and other operating system use. The operating system on each node is IBM AIX 3.2.5. The nodes are interconnected by a high-performance omega switch with approximately 63 μsec latency and 35 Mbytes/sec bandwidth [14]. The peak performance of each node is 125 Mflops/sec [14]. The parallel I/O system on the SP consists of four RAID-5 disk arrays, each of 50 GBytes. In addition to the regular Unix (AIX) file system on each node, the SP supports two other file systems—NSL Unitree and IBM's parallel file system PIOFS. Two of the four RAIDs are used for Unitree and the other two for PIOFS. There are four ways in which a program can access files on the SP:

1. From the user's home directory via NFS. This is expected to be the slowest.
2. From the local disk on each processor (`/tmp` directory). This results in independent parallel I/O, but data cannot be stored in one common file.

3. From Unitree. Unitree is a hierarchical file system that allows the user to access data from disk as well as tape. A single hierarchy consists of one disk server and one tape server. The Argonne SP has 8 hierarchies (corresponding to the 8 I/O servers). A Unitree file is stored on a single hierarchy and is not striped across different hierarchies. But different files can be stored on different hierarchies, and thus may be accessed in parallel.

4. From PIOFS. PIOFS is a parallel file system in which a file is striped across the I/O servers, allowing different portions of the file to be accessed concurrently.

4.2.2 Intel Paragon

The Intel Paragon at Caltech has 512 compute nodes, each of which is an Intel i860/XP microprocessor. Each compute node has a peak performance of 75 Mflops/sec and is provided with 32 Mbytes of main memory. The nodes are connected to each other by a two-dimensional mesh interconnection network. The parallel I/O system on the Paragon consists of 21 I/O nodes, each connected to a 4.8 Gbyte RAID-3 disk array. The I/O nodes are dedicated nodes and do not run any compute jobs. The I/O nodes also have 32 Mbytes of main memory. The Paragon runs the OSF/1 operating system. Intel's Parallel File System (PFS) provides parallel access to files. PFS stripes files across the I/O nodes, the default stripe factor being 64 Kbytes. Files can also be accessed from the user's home directory via NFS, or from the OSF/1 Unix File System (UFS), but these accesses are expected to be slow.

4.3 Experiments

Since we were interested mainly in studying the I/O characteristics of the application, we instrumented only the portion of the code that performs I/O. One complication that arises is the fact that the application does not perform I/O by directly using the routines provided by the file system. All I/O is done via calls to the Chameleon I/O library, and Pablo does not directly support instrumentation of code written using Chameleon. However, Pablo does provide generic routines to trace any kind of event. We used these generic routines to trace the calls to the Chameleon I/O library. This tracing provided us with information about the time taken for the Chameleon I/O routines to complete, but no detailed information about the actual I/O activity going on in the program.

To understand the I/O characteristics better, we instrumented the Chameleon I/O library code itself, so that I/O events occurring within the library were recorded. The astrophysics application uses the `PIO_AS_SEQUENTIAL` mode of Chameleon I/O in which all data is written to simple Unix files. The way this is implemented in Chameleon I/O is that all processors send data to one master processor using interprocessor communication, and the master processor writes the entire file. As a result, all I/O is done by one processor. When this mode is instrumented for I/O, only the I/O activity of the master processor is recorded. The other processors are shown to perform no I/O, although in fact they do perform communication for this I/O event. Hence, we also instrumented the communication portion of this I/O activity, to record the work being done by other processors.

In summary, we performed three different "levels" of instrumentation of this application:

- *app-level*: Instrumentation was done across calls to the Chameleon I/O library, using the generic Pablo routines `StartTimeEvent` and `EndTimeEvent`.
- *libio-level*: The Chameleon I/O library itself was instrumented to record all I/O activity.
- *libio-comm-level*: In the Chameleon I/O library, the I/O activity of the master processor, as well as the communication performed by other processors to send their data to the master processor, were instrumented.

Note that the application code was run on both machines without any change. We did not perform any machine-specific optimizations. The application uses the Unitree file system on the SP because the Chameleon I/O library does not currently support PIOFS. On the Paragon, the application uses PFS.

4.4 Parameter Settings

The astrophysics application is iterative, and the number of iterations can be specified as an input parameter. The I/O is done every few iterations, which are also specified as parameters. For the results given in this paper, the parameters were chosen as follows:

- problem size = 512×256
- total number of iterations = 20
- movie arrays written every 5 iterations
- horizontal averages written every 5 iterations
- restart arrays written every 10 iterations
- arrays for data analysis written every 10 iterations

These parameters reflect the relative frequency of the different types of output required in this application: horizontal averages and movie arrays are written more frequently than restart and data arrays.

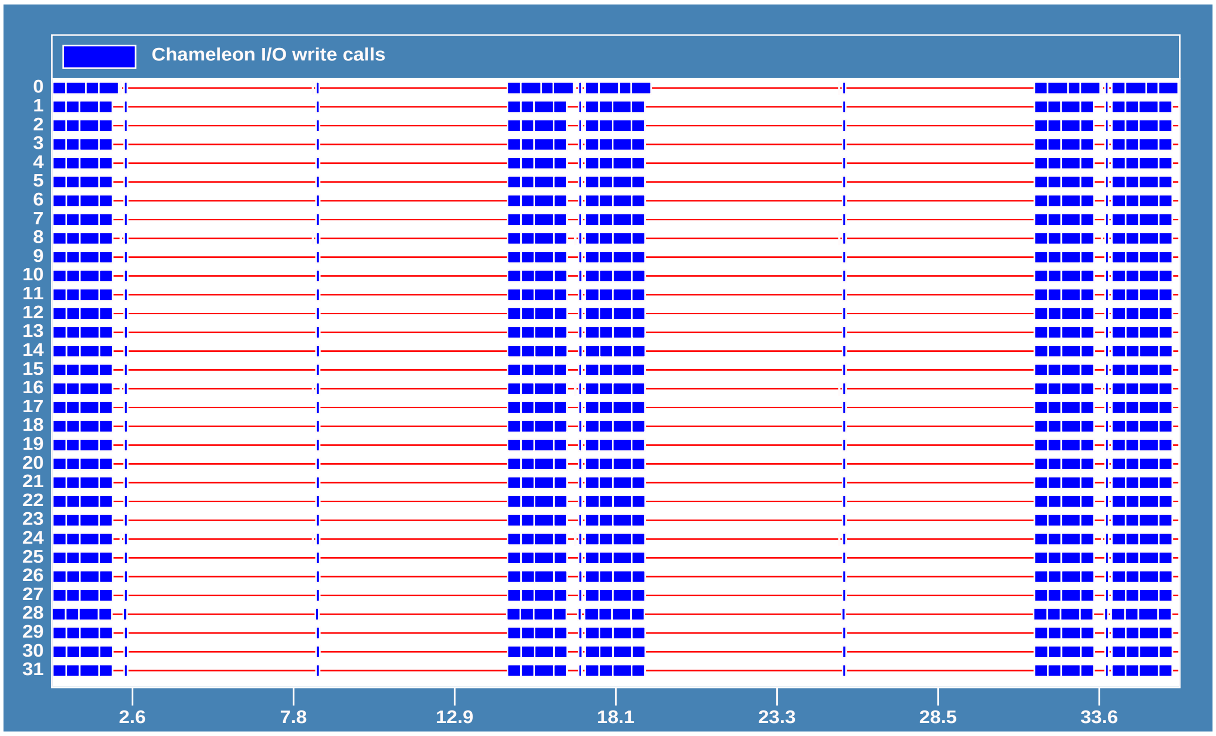

Figure 1: Upshot view with *app-level* instrumentation on the SP (32 processors)

We also ran the code for larger numbers of iterations, but we found that the results are more or less similar. The same observations as in the first 20 iterations are repeated for the remaining iterations. Hence, in this paper, we report only the I/O activity in the first 20 iterations.

5 I/O Characteristics on the SP

We ran the code and collected traces for all three levels of instrumentation on the SP. We selected two cases for the number of processors—32 and 128 (all available processors). Let us first consider the results for the 32-processor case.

Figure 1 shows the Upshot view of the logfile with *app-level* instrumentation on 32 processors. Upshot always shows time in seconds on the x-axis, and processors on the y-axis. The bands in the figure correspond to calls to Chameleon I/O library routines. The narrow bands every 5 iterations correspond to the writing of horizontal averages and movie arrays. The larger bands every 10 iterations are due to the writing of arrays for data analysis and restart. Table 1 summarizes the I/O activity. We observe that calls to Chameleon I/O routines take an average of 26.61% of the total program time. The total amount of data transferred is not available at this level of instrumentation. Figure 2 shows a zoomed-in view of a portion of Figure 1. The bands in the figures are very regular because the Chameleon I/O routines are synchronous and blocking.

Table 1: I/O operations with *app-level* instrumentation on the SP (32 processors).

Total program time = 43.59 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Chameleon I/O routines	1816	11.60	26.61

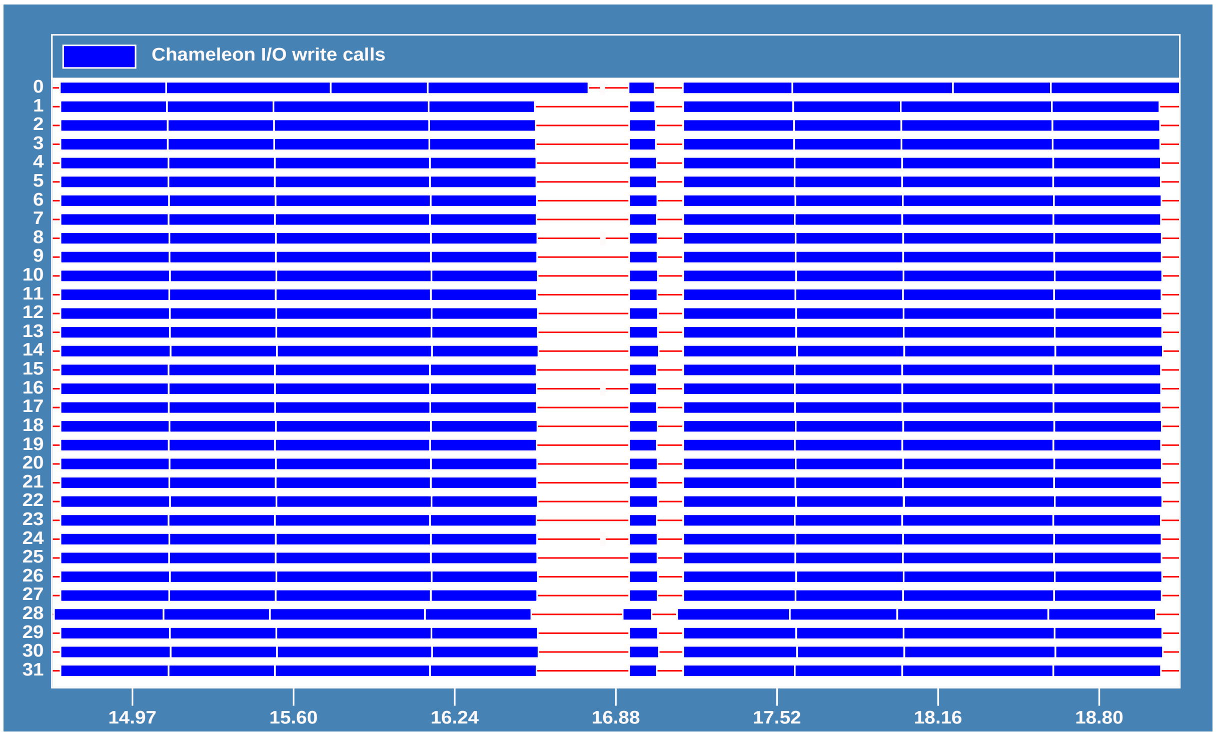

Figure 2: A zoomed-in view of a portion of Figure 1

Figure 3 shows the Upshot view with *libio-level* instrumentation on 32 processors. In this case, all I/O activity within the Chameleon I/O library routines was instrumented. Since the application uses the `PIO_AS_SEQUENTIAL` mode of Chameleon I/O, all I/O is performed only by processor 0. Other processors only send data to processor 0 using interprocessor communication, and do not perform I/O directly. Hence, in Figure 3, no I/O activity is seen for processors 1–31.

Figure 4 shows a zoomed-in Upshot view with *libio-comm-level* instrumentation on 32 processors. In this case, both I/O and communication occurring within the Chameleon I/O library are instrumented. Note that the application also performs communication explicitly to fetch off-processor data, but that was not recorded. The I/O activity is summarized in Table 2. It can be observed that 8.89% of program time is spent on pure I/O activity (open, close, write), whereas

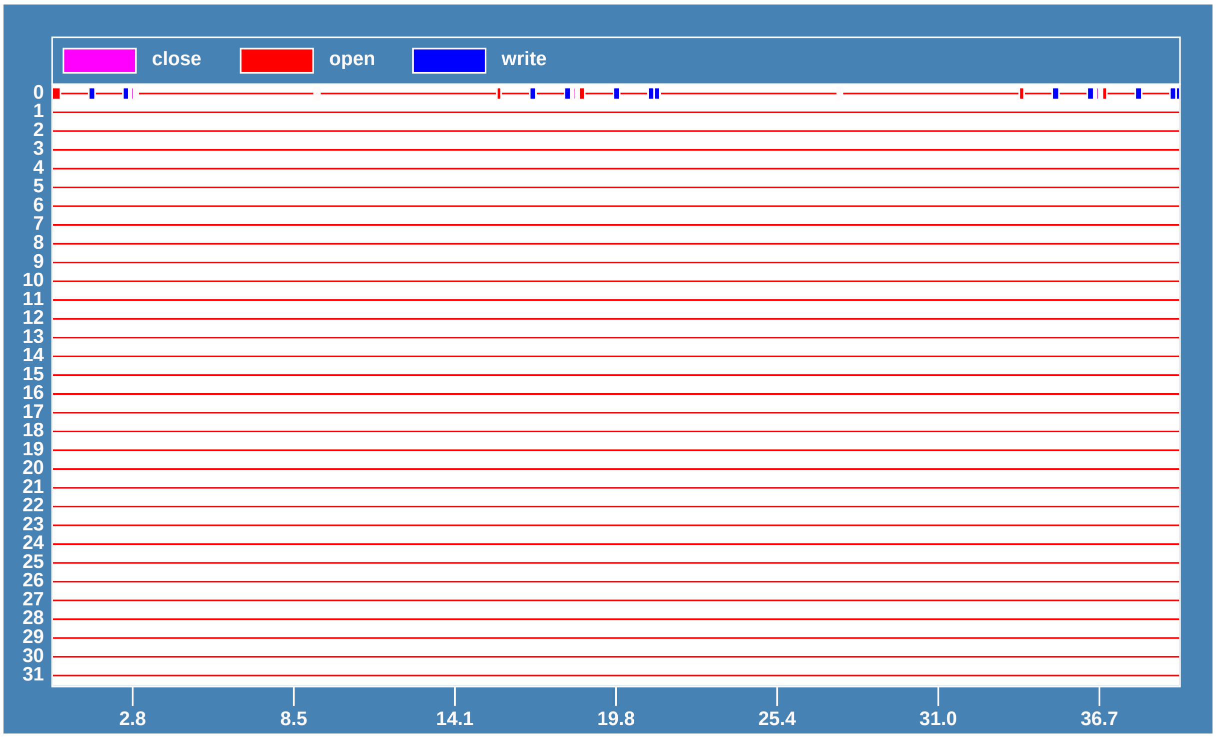

Figure 3: Upshot view with *libio-level* instrumentation on the SP (32 processors)

communication for I/O takes 21.21% of program time. This communication time includes the idle time other processors spend waiting while processor 0 performs I/O. Hence, the percentage of program time spent on communication for I/O is slightly exaggerated in Table 2. Even so, a significant amount of time is spent on communication. This is because the communication involves processors 1–31 simultaneously sending data to processor 0, resulting in a bottleneck. The communication is performed in units of a certain buffer size predefined within the Chameleon I/O library. As a result, in Figure 4, we see several bands corresponding to communication. Comparing Tables 1 and 2, we observe a difference in the total program time as well as in the percentage of program time spent on I/O activities. This difference can be attributed to the fact that the two experiments were run at different times and we did not have exclusive access to the system. Also, the instrumentation overheads, though small, are different in the two cases.

Table 3 gives details about the write operations in the code. There were a total of 27 write operations, all performed by processor 0. A total of 10.67 Mbytes were written during 20 iterations of the program. The writes were of fairly large size, up to 1 Mbyte; there were only two small writes of 24 bytes. The large writes are mainly due to the fact that the Chameleon I/O library performs caching, and the write cache size was set at 1 Mbyte. The aggregate throughput of all write operations was 4.063 Mbytes/sec.

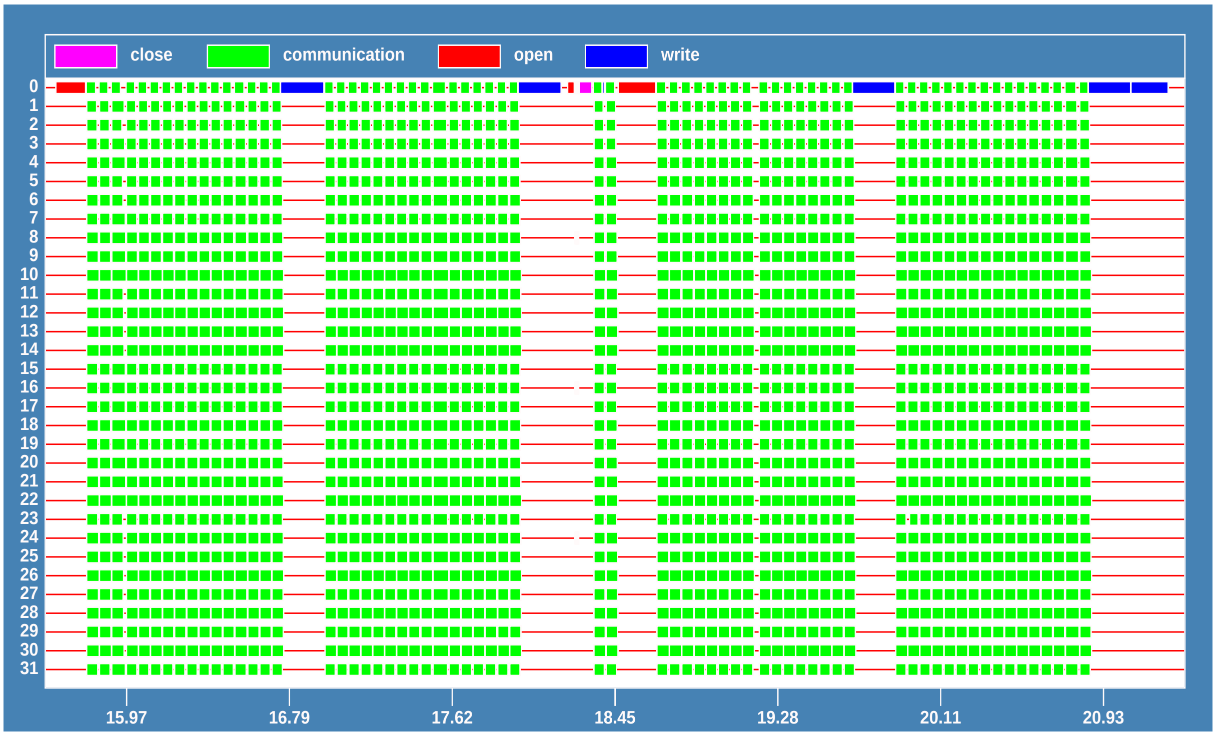

Figure 4: Zoomed-in Upshot view with *libio-comm-level* instrumentation on the SP (32 processors)

Table 2: I/O operations with *libio-comm-level* instrumentation on the SP (32 processors). Open, close, and write take place only on processor 0. The communication time includes the idle time other processors spend waiting while processor 0 performs I/O.

Total program time = 44.97 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Open	11	1.121	2.493
Close	10	0.249	0.554
Write	27	2.627	5.842
Communication (for I/O)	5640	9.538	21.21

Table 3: Details of write operations with *libio-comm-level* instrumentation on the SP (32 processors)

Total count	Size distribution (bytes)				Total data written (Mbytes)	Total time (sec)	Throughput (Mbytes/sec)
	24	10K	64K	1M			
27	2	5	10	10	10.67	2.627	4.063

Table 4: I/O operations with *app-level* instrumentation on the SP (128 processors).

Total program time = 59.35 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Chameleon I/O routines	6704	20.88	35.18

Table 5: I/O operations with *libio-comm-level* instrumentation on the SP (128 processors). Open, close, and write take place only on processor 0. The communication time includes the idle time other processors spend waiting while processor 0 performs I/O.

Total program time = 63.95 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Open	11	0.816	1.276
Close	10	0.267	0.417
Write	27	3.635	5.684
Communication (for I/O)	22160	16.34	25.55

We also ran all three cases on all 128 processors on the SP. We do not show the Upshot views of these because the general patterns are similar to the corresponding 32-processor cases. The results are summarized in Tables 4 and 5. The total program time is higher than in the 32-processor case. The average time spent on Chameleon I/O routines is also higher, which in turn means that a larger amount time was required for communication within the Chameleon I/O routines.

The distribution of time spent on different operations is clearer with *libio-comm-level* instrumentation (Table 5). The time taken by I/O operations is slightly different from the 32-processor case (Table 2), though in theory it should be identical because only processor 0 performs I/O. The difference is because the two cases were run at different times and we did not have exclusive access to the system. The communication time, however, is significantly higher in the 128-processor case because there are 127 processors simultaneously sending data to processor 0. Table 6 shows that the distribution of write sizes is the same as in the 32-processor case, but the aggregate write throughput obtained is lower.

Table 6: Details of write operations with *libio-comm-level* instrumentation on the SP (128 processors)

Total count	Size distribution (bytes) 24	10K	64K	1M	Total data written (Mbytes)	Total time (sec)	Throughput (Mbytes/sec)
27	2	5	10	10	10.67	3.635	2.935

6 I/O Characteristics on the Paragon

We repeated all the above experiments on the Paragon to obtain a comparison of the performance on the two machines. We selected the same number of processors—32 and 128. Let us first consider the results for the 32-processor case.

Figure 5 shows the Upshot view of the logfile with *app-level* instrumentation on 32 processors on the Paragon. The general pattern is similar to that for the SP in Figure 1, but we find that the total time taken on the Paragon in higher. A zoomed-in Upshot view for a portion of the trace is given in Figure 6. This pattern is also similar to that for the SP in Figure 2, but we observe that corresponding events on different processors are better synchronized on the SP, whereas they are slightly skewed on the Paragon (this is even more evident in Figure 7). Since Pablo uses the global clock to record timestamps, the skew may indicate that the global clock on the Paragon is not as well synchronized as that on the SP. Table 7 summarizes the I/O activity. A comparison of Tables 1 and 7 reveals that the total program time on the SP is about 55% of that on the Paragon. Calls to Chameleon I/O library routines also take less time on the SP.

Table 7: I/O operations with *app-level* instrumentation on the Paragon (32 processors).

Total program time = 79.26 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Chameleon I/O routines	1816	14.25	17.98

Since *libio-level* is a subset of *libio-comm-level* instrumentation, we only show results for *libio-comm-level* instrumentation on the Paragon. Figure 7 shows a zoomed-in Upshot view with *libio-comm-level* instrumentation on 32 processors. Unlike on the SP, we observe a distinct skew in the occurrence of corresponding events on different processors. The I/O activity is summarized in Table 8. A comparison of Tables 2 and 8 shows that the time for opening files on the Paragon is 2.87 times that on the SP. The time for closing files is only slightly higher on the Paragon.

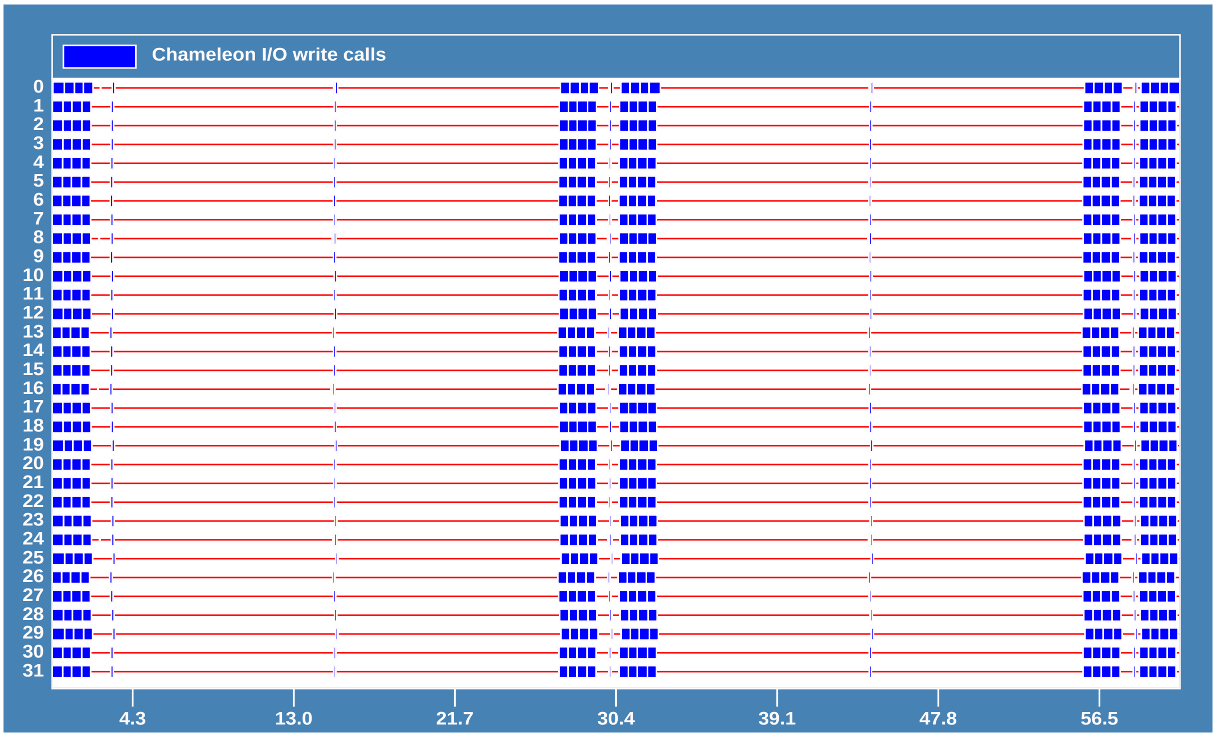

Figure 5: Upshot view with *app-level* instrumentation on the Paragon (32 processors)

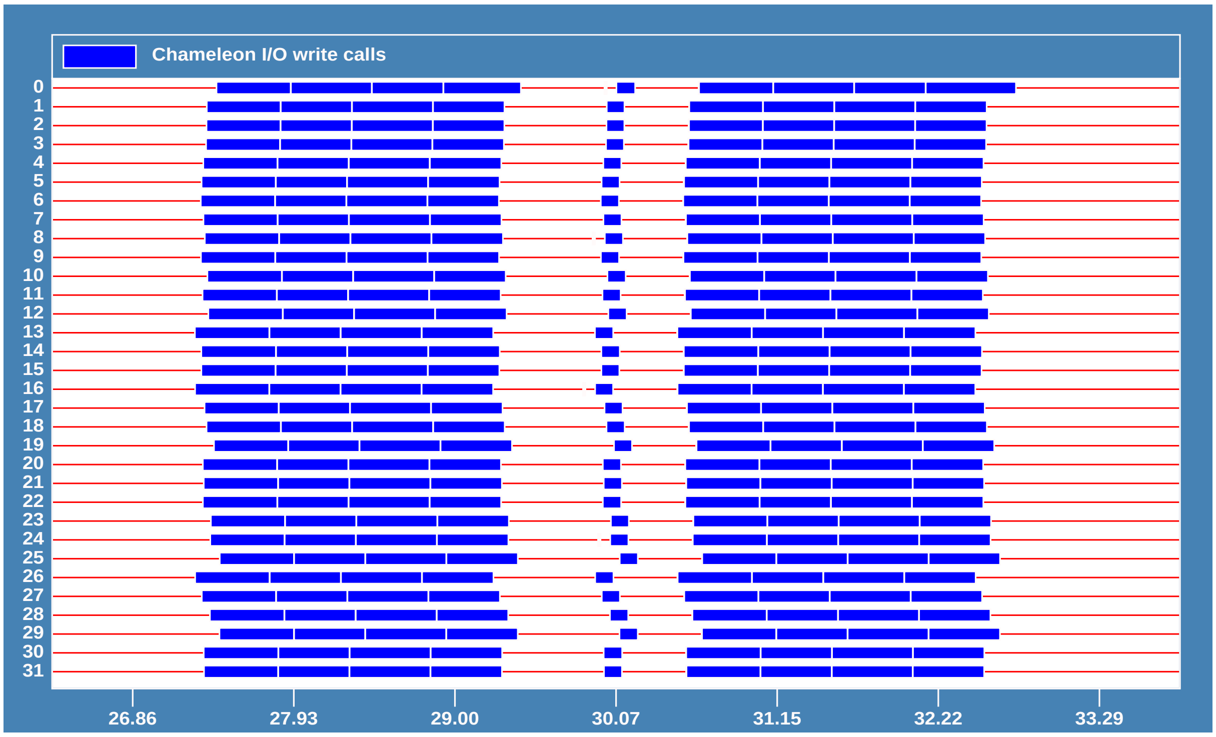

Figure 6: A zoomed-in view of a portion of Figure 5

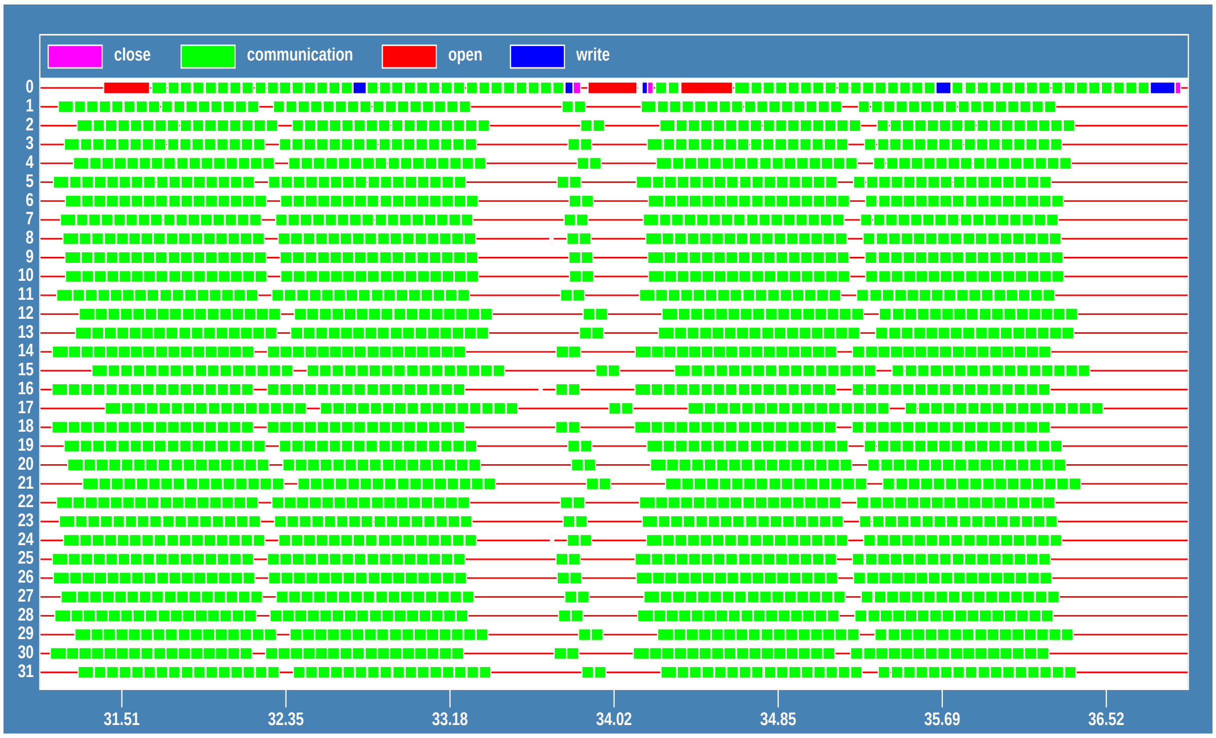

Figure 7: Zoomed-in Upshot view with *libio-comm-level* instrumentation on the Paragon (32 processors)

However, the time for writing data on the Paragon is 37.42% of that on the SP. The time spent on communication for I/O is higher on the Paragon. In summary, writing data is faster on the Paragon, whereas opening files, closing files, and communication are faster on the SP. The sizes of different write operations are the same as on the SP (Table 3) because the code run on both machines was identical.

The difference in write bandwidths on the SP and Paragon may be due to the fact that the application uses Intel's Parallel File System (PFS) on the Paragon, and the Unitree file system on the SP. PFS stripes files across multiple disks, which enables data be written to different disks in parallel. Unitree, on the other hand, is a hierarchical file system. It allows users to directly access files from tapes, but it stores each file on a single disk-tape hierarchy. Hence, there is no parallelism of access within a file. IBM's parallel file system PIOFS could not be used on the SP because the Chameleon I/O library does not currently support PIOFS.

We also took traces on 128 processors on the Paragon for all three levels of instrumentation. The I/O activity is described in Tables 9 and 10. The total program time as well as the time for Chameleon I/O routines is higher than in the 32-processor case. The time for I/O operations was expected to be the same, but turned out to be slightly different because the two cases were run at

Table 8: I/O operations with *libio-comm-level* instrumentation on the Paragon (32 processors). Open, close, and write take place only on processor 0. The communication time includes the idle time spent by other processors waiting while processor 0 performs I/O.

Total program time = 84.59 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Open	11	3.216	3.801
Close	10	0.298	0.352
Write	27	0.983	1.162
Communication (for I/O)	5640	10.34	12.22

Aggregate write throughput = 10.85 Mbytes/sec.

Table 9: I/O operations with *app-level* instrumentation on the Paragon (128 processors).

Total program time = 106.4 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Chameleon I/O routines	6704	21.96	20.65

different times. The time spent on communication for I/O is considerably higher on 128 processors. These results are consistent with corresponding results on the SP.

A comparison of Tables 4 and 9 reveals that, with 128 processors, the total program time on the SP is 55.78% of that on the Paragon. Calls to Chameleon I/O library routines also take less time on the SP. From Tables 5 and 10 we observe that the time for opening files on the Paragon is 5.77 times that on the SP. The time for closing files is only slightly more on the Paragon. However, writing data on the SP takes 3.05 times that on the Paragon. Communication for I/O is faster on the SP. Thus, the relative performance on the two machines for the 128-processor case is similar to that for the 32-processor case.

7 Improving the I/O Performance

The above results show that the I/O performance of this application is limited by the use of the `PIO_AS_SEQUENTIAL` mode of Chameleon I/O. This mode does not make good use of the available I/O bandwidth since all I/O is performed only by processor 0. It also results in a communication

Table 10: I/O operations with *libio-comm-level* instrumentation on the Paragon (128 processors). Open, close, and write take place only on processor 0. The communication time includes the idle time other processors spend waiting while processor 0 performs I/O.

Total program time = 114.2 sec.

Operation	Total count (all procs.)	Average time per proc. (sec)	Percentage of program time
Open	11	4.707	4.122
Close	10	0.298	0.261
Write	27	1.191	1.043
Communication (for I/O)	22160	16.87	14.77

Aggregate write throughput = 8.959 Mbytes/sec.

bottleneck when all processors simultaneously send data to processor 0.

The I/O performance of the application could be improved by using the `PIO_AS_PARALLEL` mode of Chameleon I/O. In this mode, each processor independently writes its local array to a separate file. However, the authors of the application prefer the `PIO_AS_SEQUENTIAL` mode because it writes a distributed array into a single Unix-compatible file. This makes it easier to postprocess the output data using sequential Unix tools and also to restart the program on a different number of processors. We did not run the instrumented code using the `PIO_AS_PARALLEL` mode because the authors of the application never run it in that mode [20].

Clearly, the I/O performance could be improved by having several processors write to a common file in parallel. This would make better use of the available I/O bandwidth and also eliminate the communication bottleneck. The I/O access pattern of this application suggests that it would benefit from using collective I/O, in the form of either two-phase I/O [10], disk-directed I/O [18], or server-directed collective I/O [31]. In this application, arrays are distributed in a block fashion in two dimensions, resulting in each processor getting a sub-block of the array. The sub-block from each processor needs to written to the file containing the entire array. It has been shown in [10, 18, 31] that this can be done efficiently using collective I/O.

To study the benefits of using collective I/O for this application, we have implemented the Chameleon I/O routines `PIWriteDistributedArray` and `PIReadDistributedArray` using two-phase I/O. We are currently in the process of characterizing the performance of the application with these new routines. Results from these experiments will be included in the final version of this paper.

8 Conclusions

Our tracing study provides some interesting results. We find that all I/O in this astrophysics application is for writes only, except when restarting from a checkpoint. Most of the writes are fairly large, up to 1 Mbyte each, mainly because of caching performed by the Chameleon I/O library. The total program time on the SP is about 55% of that on the Paragon. Writing data to files takes less time on the Paragon, but opening and closing files take less time on the SP. The I/O access pattern of the application suggests that the I/O performance could be improved by using collective I/O. We are currently studying the performance benefits of using two-phase I/O for this application.

We note that some of the results in this paper may be specific to this particular application and the Chameleon I/O library. To reach a more definite conclusion, we plan to study the I/O characteristics of several other parallel applications. We also plan to instrument and characterize the three-dimensional version of this application being developed [20], which is much more I/O-intensive than the present two-dimensional version.

Acknowledgments

We thank Andrea Malagoli for giving us the source code of the application and helping us understand it. We also thank Ruth Aydt for helping us understand how to use Pablo.

References

[1] Applications Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. Scalable I/O Initiative Working Paper Number 1. On World-Wide Web at http://www.ccsf.caltech.edu/SIO/SIO_apps.ps, 1994.

[2] R. Aydt. The Pablo Self-Defining Data Format. Technical report, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1992.

[3] R. Aydt. A User's Guide to Pablo I/O Instrumentation. Technical report, Dept. of Computer Science, University of Illinois at Urbana-Champaign, December 1994.

[4] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[5] S. Baylor and C. Wu. Parallel I/O Workload Characteristics Using Vesta. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[6] T. Bogdan, F. Cattaneo, and A. Malagoli. On the Generation of Sound by Turbulent Convection: I. A Numerical Experiment. *The Astrophysical Journal*, 407:316–329, 1993.

[7] F. Cattaneo, N. Brummel, J. Toomre, A. Malagoli, and N. Hurlburt. Turbulent Compressible Convection. *The Astrophysical Journal*, 370:282–294, 1991.

[8] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[9] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[10] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Runtime Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[11] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, pages 59–68, March 1994.

[12] R. Floyd and C. Ellis. Directory Reference Patterns in Hierarchical File Systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[13] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, November 1993.

[14] W. Gropp and E. Lusk. User's Guide for the ANL IBM SPx. On World-Wide Web at ftp://info.mcs.anl.gov/pub/ibm_sp1/guide-r2.ps.Z, March 1995.

[15] W. Gropp and B. Smith. Chameleon Parallel Programming Tools User's Manual. Technical Report ANL–93/23, Mathematics and Computer Science Division, Argonne National Laboratory, March 1993.

[16] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL–91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[17] D. Jensen and D. Reed. File Archive Activity in a Supercomputing Environment. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[18] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Technical Report PCS–TR94–226, Dept. of Computer Science, Dartmouth College.

[19] D. Kotz and N. Nieuwejaar. File-System Workload on a Scientific Multiprocessor. *IEEE Parallel and Distributed Technology*, pages 51–60, Spring 1995.

[20] A. Malagoli. Personal communication, 1995.

[21] A. Malagoli, A. Dubey, F. Cattaneo, and D. Levine. A Portable and Efficient Parallel Algorithm for Astrophysical Fluid Dynamics. In *Proceedings of Parallel CFD '95*, June 1995.

[22] E. Miller and R. Katz. Input/Output Behavior of Supercomputer Applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

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

[24] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[25] B. Pasquale and G. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of Supercomputing '93*, pages 388–397, November 1993.

[26] B. Pasquale and G. Polyzos. Dynamic I/O Characterization of I/O Intensive Scientific Applications. In *Proceedings of Supercomputing '94*, pages 660–669, November 1994.

[27] A. Purakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[28] K. Ramakrishnan, P. Biswas, and R. Karedla. Analysis of File I/O Traces in Commercial Computing Environments. In *Proceedings of ACM SIGMETRICS and PERFORMANCE*, pages 78–90, 1992.

[29] A. L. N. Reddy and P. Banerjee. A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[30] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113, October 1993.

[31] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. To appear.

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