An Experimental Evaluation of I/O Optimizations on Different Applications

Meenakshi A. Kandaswamy, Mahmut Kandemir, *Member*, *IEEE*, Alok Choudhary, *Senior Member*, *IEEE*, and David Bernholdt

Abstract—Many large scale applications have significant I/O requirements as well as computational and memory requirements. Unfortunately, the limited number of I/O nodes provided in a typical configuration of the modern message-passing distributed-memory architectures such as Intel Paragon and IBM SP-2 limits the I/O performance of these applications severely. In this paper, we examine some software optimization techniques and evaluate their effects in five different I/O-intensive codes from both small and large application domains. Our goals in this study are twofold. First, we want to understand the behavior of large-scale data-intensive applications and the impact of I/O subsystems on their performance and vice versa. Second, and more importantly, we strive to determine the solutions for improving the applications' performance by a mix of software techniques. Our results reveal that different applications can benefit from different optimizations. For example, we found that some applications benefit from file layout optimizations whereas others take advantage of collective I/O. A combination of architectural and software solutions is normally needed to obtain good I/O performance. For example, we show that with a limited number of I/O resources, it is possible to obtain good performance by using appropriate software optimizations. We also show that beyond a certain level, imbalance in the architecture results in performance degradation even when using optimized software, thereby indicating the necessity of an increase in I/O resources.

Index Terms—I/O optimizations, parallel architectures, I/O intensive applications, disk layout, collective I/O.

1 Introduction

Large scale parallel scientific applications in general tend to be computationally-intensive as well as data-intensive. The advances in I/O systems, both in hardware and[1] software, lag behind those in processors and interconnection networks, resulting in poor performance for I/O-intensive applications [13]. In this paper, we investigate the I/O performance of five different I/O-intensive applications. Our experiments confirm that, for all of these applications, poor I/O performance limits the overall performance of the application. This impact is sometimes so severe that when a certain number of compute nodes (processors) is reached, the execution time starts to increase. Although, such a situation can sometimes occur with computationally-intensive applications as well, in I/O-intensive applications such poor scalability occurs with even very few processors such as six or eight. We believe that two main obstacles to high performance of I/O-intensive applications are

1. limited number of I/O resources (e.g., I/O nodes, disks), and
2. unoptimized I/O software.

The first problem severely affects the I/O scalability; beyond a certain point, increasing the number of compute nodes (processors) has a negative impact on the I/O performance and consequently the execution time increases. The second problem, on the other hand, is more deeply-rooted and operating system (OS) writers, compiler writers, and even application writers can be blamed for that. An additional difficulty associated with this second problem is that the unoptimized I/O software of one domain can affect all other domains that depend on it. For example, insufficient compiler analysis for out-of-core computations (i.e., computations that heavily work on large disk-resident arrays) will eventually prevent application level software from taking full advantage of the underlying I/O subsystem.

A typical high-performance parallel computer consists of compute nodes and I/O nodes (which have disks and/or disk arrays attached to them). A combination of architectural and software solutions is normally needed to obtain good I/O performance. Since continuously increasing the I/O hardware is prohibitively expensive, we eventually need to design smart software optimizations to improve the I/O performance of scientific codes.

In this paper, we evaluate five real I/O-intensive applications to have a better insight into the I/O problem.

1. EDITOR'S NOTE: This paper originally appeared in *TPDS*, Vol. 13, No. 7. Unfortunately, due to extraordinary circumstances, errors were introduced into the figure captions of the paper. We are reprinting the paper in its entirety here.

• *M. Kandaswamy is with the Enterprise Server Group, Intel Corporation, Hillsboro, OR 97124-6497. E-mail: meeena.a.kandaswamy@intel.com.*
• *M. Kandemir is with the Microsystems Design Group, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: kandmir@cse.psu.edu.*
• *A. Choudhary is with the Department of Electrical and Computer Engineering, Northwestern University, Technological Institute, 2145 Sheridan Road, Evanston, IL 60208-3118. E-mail: choudhar@ece.nwu.edu.*
• *D. Bernholdt is with the Northeast Parallel Architecture Center, Syracuse University, 111 College Place, Syracuse, NY 13244. E-mail: bernhold@npac.syr.edu.*

Manuscript received 16 Oct. 1998; revised 12 Aug. 2000; accepted 12 Dec. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 108073.

TABLE 1
Applications in Our Experimental Suite and Their Important Characteristics

Application	Source	Lines	Brief Description	Platform	Type of I/O Performed
SCF 1.1	PNL	16,500	self consistent field computation	Paragon	writes integrals to disk, and reads them
SCF 3.0	PNL	19,000	self consistent field computation	Paragon	writes integrals to disk, and reads them
FFT	Authors	500	2D out-of-core FFT	Paragon	reads and writes two matrices
BTIO	NASA Ames	6,713	simulates the I/O of a flow solver	SP-2	periodic writes of arrays
AST	Univ. of Chicago	17,000	simulates gravitational collapses	Paragon	writes arrays for check-pointing and visualization

On one side, we investigate I/O performance of these applications on message-passing parallel machines; and on the other side, we study the impacts of different optimizations, which have been developed by previous research, on these applications. Although, we believe that these optimizations can be automated in a compilation framework or can be structured as a runtime layer, in this paper we apply them from within the applications in an attempt to make our conclusions more accessible to application programmers. Our objectives in this study are manifold:

1. To understand I/O behavior of real scientific codes,
2. To evaluate different I/O optimizations available on different applications and try to see which optimizations are appropriate for a given application,
3. To see at what point the software optimizations become ineffective and to try to see whether increasing hardware resources sparingly will solve the problem, and finally,
4. To reach a general guideline (rule of thumb) that can be applied by the end user (or can be automated) which aims at improving I/O performance of the applications.

We show that with a limited number of I/O resources, it is possible to obtain good performance by using appropriate software optimizations including layout transformations, collective I/O, and I/O prefetching. We also show that beyond a certain level, imbalance in the architecture results in performance degradation even when using optimized software. In the course of this work, we also acknowledge the importance of an I/O characterization tool, Pablo [27], which provides invaluable information to the user regarding I/O access patterns of the program. Finally, we focus on two applications and report some performance numbers on a Linux-based cluster environment.

The remainder of this paper is organized as follows: In Section 2, we introduce the applications in our experimental suite. In Section 3, we describe the I/O optimizations that we have chosen to evaluate. In Section 4, we present the experimental data obtained from original (unoptimized) as well as optimized programs, discuss the individual I/O optimizations and explain how they improve the I/O as well as the overall performance of the applications. In Section 5, we discuss (using our experimental results and experience) a systematic approach for optimizing I/O performance of a given application. In Section 6, we discuss the related work and, finally, in Section 7, we present conclusions and future research avenues along this direction.

2 Overview of Applications

In this study, we focus on *five* different I/O-intensive parallel applications written in Fortran by using message-passing constructs, ranging from 500 lines of code to 19,000 lines (up to 538,000 lines if accompanying libraries are included). The important characteristics of these applications are given in Table 1. `SCF 1.1` and `SCF 3.0` are from the computational chemistry domain and are very large applications [25]. `AST` is an astrophysics code from Univ. of Chicago, `BTIO` is a disk-based version of a flow-solver program from NAS benchmarks [14], and `FFT` is a 2D out-of-core FFT program written by the authors.

We have chosen `SCF 1.1` and `SCF 3.0`, as they represent I/O requirements of typical large computational chemistry applications. The `AST` code, on the other hand, performs check-pointing, an I/O-intensive activity that would be beneficial to explore. These three codes are very large programs. In order to strike a balance, we have also included one small-sized and one medium-sized program. We have implemented the `FFT` code ourselves, paying special attention not to optimize it aggressively in an attempt to leave room for measuring the impact of the I/O optimizations on it. Our medium-sized code, `BTIO`, has several versions; one of them is written in MPI-IO [9], an emerging parallel I/O standard. Appendix A contains a brief explanation of these applications; more details can be found in the PhD thesis of the first author [16].

3 Overview of I/O Optimizations

In this section, we briefly discuss the I/O optimizations used in this work. Most of these optimizations can be applied from within the application or can be automated in a compilation framework or an optimization tool. In this paper, we focus on collective I/O, prefetching, file layout, efficient I/O interface, and balanced I/O optimizations. While there are many other I/O optimizations proposed by previous research (e.g., file caching [15]), our applications get the most benefit from these five optimizations, so we focus only on them.

3.1 Collective I/O

Collective I/O is a term used generally to define a technique where a processor can perform I/O on behalf of other processors. This technique is especially useful in the cases where the logical layout of data is different from its physical layout on disks [12], [31].

Suppose, for example, an application divides a large disk-resident column-major array of size $N \times N$ across four processors in such a way that each processor has a square portion of $N/2 \times N/2$ to work on. If, in a read operation, all the processors try to access their own portions independently, every processor may need to issue $N/2$ I/O requests, each of which for a vector of size $N/2$. Instead of doing it this way, we can let each processor access a contiguous portion of the array of size $N \times N/4$. Then, in order to have the correct elements on correct processors, we need a communication phase where processors exchange data. Note that, by performing I/O on behalf of other processors, we optimize the disk accesses at the expense of some extra communication, which is generally all-to-all type.

Del Rosario et al. [12] are among the first ones who discuss such a technique where each processor reads the portion of the disk data that is least costly for it; and then the processors use the available interconnection network to exchange the parts of the data so that each processor gets what it needs. Although, this approach slightly increases the communication time of the program, it generally minimizes the number of I/O calls which in turn reduces the execution time significantly as I/O, in general, is much more costly than communication on modern message-passing parallel architectures. In this approach, which they call *two-phase I/O*, the processors cooperate in accessing the data on disks. The aim is to combine several I/O requests into fewer larger granularity requests and reorder requests such that the file will be accessed in a close-sequential fashion. Additionally, the total I/O workload can be partitioned among the processors dynamically [31]. Collective I/O has successfully been implemented in the PASSION library for out-of-core parallel applications [8].

It should be noted that collective I/O is on its way to becoming an important I/O optimization and has found its way to MPI-IO [9], an emerging I/O programming standard.

3.2 Prefetching

It has been shown that prefetching can be very useful for I/O-intensive applications whose access patterns can be predicted, that is, the I/O access patterns exhibit regularity [23]. In such cases, before starting to work on the $n\text{th}$ data block we can issue a prefetch request for the $(n+1)\text{th}$ data block. In this way the computation of the $n\text{th}$ block can overlap with the I/O of the $(n+1)\text{th}$ block. In fact, an experienced programmer or an optimizing compiler can choose to prefetch more than one block at a time depending on the relative costs of computation and I/O of a block.

Notice that this optimization is highly amenable to compiler analysis. Since the compiler has a global view of access patterns of a given application, it can schedule prefetch requests ahead of time to utilize the disk bandwidth as effectively as possible. Mowry et al. [24] automated the prefetching optimization within a compilation framework whereas Arunachalam et al. [1] used prefetching in the system software level.

We should emphasize that as compared with collective I/O, which is a *latency eliminating* technique, the prefetching optimization is a latency *hiding technique* and is useful only if there is additional disk bandwidth to utilize. Most implementations of the prefetch optimization use the asynchronous I/O calls of the underlying parallel file system.

3.3 File Layout Optimization

This is another high-level I/O optimization technique which can be automated in a compiler or in an optimization tool. It is especially suitable in I/O-intensive applications where the bulk of the execution time is spent in a couple of loop nests.

Traditionally, file layouts of multidimensional arrays are fixed at a specific form and correspond to memory layout adopted by the language used. However, in some (e.g., out-of-core) applications fixing the file layout to the same form for all arrays can hurt the performance severely. Suppose, for example, that we would like to take the transpose of one disk-resident array into another. Due to the characteristic of the transpose operation, the first row of the source array should be transferred to the first column of the target array and so forth. It is easy to see that in this example, instead of, lets say, fixing layouts of both arrays to column-major (or row-major), it might be beneficial to set one of the layouts to row-major and the other to column-major. In this way, accesses to both the arrays will span to consecutive locations in file. Kandemir et al. [21] showed that such optimizations can be automated with a little additional help from the current optimizing compiler technology.

Notice that although collective I/O is meaningful for only multiprocessor cases, prefetching and file layout optimizations can be used in both multiprocessor and uniprocessor cases.

3.4 Efficient I/O Interface

Ideally an I/O interface should contain sufficient information such that the underlying software level should be able to optimize the I/O operation requested. A simple form of this is to convert a Fortran interface to a C interface where appropriate. Although, it looks simple at first glance, Smirni et al. [30] report that in an implementation of the Hartree-Fock code, they obtained impressive speedups just by changing the interface to the file system from Fortran to C.

More recently, Thakur et al. [33] observed the importance of using efficient interfaces provided by the MPI-IO library [9], instead of more intuitive interfaces that one might prefer. For example, suppose that a number of processors will access disjoint but interleaved parts of a shared array. Such an access pattern can be coded in a number of ways. One way is to set up a loop in which every iteration identifies a portion of accesses of each processor. Although, considering all iterations such a loop will cover all accesses by all processors, by looking at just one iteration it is difficult to say what the global access pattern looks like. Or, alternatively, we can use a single MPI-IO call which contains the entire access pattern of all the processors. Then, the underlying implementation can take advantage of that and use, for example, collective I/O instead of naive I/O accesses.

3.5 Balanced I/O

This is a more specialized optimization technique which can be used by an experienced programmer, who is highly familiar with the application in question.

The performance of many I/O-intensive applications can depend on an interplay between factors such as available in-core node memory, program access pattern, and available I/O resources. Smart management of memory can lead to huge savings in the time spent in I/O subsystem. Unfortunately, this management is, in general, highly application dependent and in some cases, even within the same application, it is input dependent. Some sophisticated applications such as the `SCF 3.0` code considered in this paper give the user the ability to define what percentages of available memory will be used for what purposes. Using such a fine-grain control over memory, which is an important resource in I/O-intensive computations, the end user can make several experiments and can determine the best memory allocation and partitioning for a given program and input set.

We should stress that these I/O optimizations are by no means exhaustive. There are many other I/O optimization techniques, some of which are similar to those explained above and some of which are quite different (see for example [29], [2], [3], [4], [17], [34], [16] and the references therein). Since it is not possible to evaluate all the optimizations in this work, we have selected the five representative optimizations mentioned. Also, in some cases, experimenting with those optimizations on a specific program can give a rough idea of how the other optimizations will perform on the same program. As a case in point, in principle disk-directed I/O proposed by Kotz [22] and server-directed I/O proposed by Seamons et al. [29] are similar to two-phase I/O [12]. Therefore, it is reasonable to think that they will be useful for the types of applications for which the two-phase I/O is useful.

4 Evaluation of I/O Optimizations

In this section, we present experimental results on our program suite and explain how some of the I/O optimization techniques improve the I/O performance of the applications.

4.1 Methodology

Our experimental methodology is as follows: For each application, we applied (where possible) several software optimizations, but due to lack of space we present only the results of the most effective optimizations. Based on the platform on which we ran the application, we also changed the number of I/O nodes to observe its impact on the application's I/O behavior. In the small Paragon machine, we used two and four I/O node configurations, the only available partitions. In the large Paragon, we used 12, 16, and 64 I/O node partitions. In the SP-2, on the other hand, the number of I/O nodes is fixed at four, which is the only available partition.

4.2 Platforms

In this section, we summarize some of the salient characteristics of our platforms, Intel Paragon and IBM SP-2, emphasizing their I/O capabilities.

Intel Paragon. The small Intel Paragon that we used for the `FFT` experiments consists of 56 compute nodes and four I/O nodes. The compute nodes are arranged in a two-dimensional mesh comprised of 14 rows and four columns. The compute nodes use the Intel i860 XP microprocessor and have 32 MBytes of memory each. The i860 has a peak performance of 75 MFlops, yielding a system peak speed of 4.2 GFlops. The total memory capacity of the compute partition is around 1.8 GBytes. For the other experiments (except `BTIO`), we used a larger Paragon machine with 512 compute nodes and that has service (I/O) partitions of sizes 12, 16, and 64. The compute node topology is mesh and the processor characteristics are the same as the small Paragon mentioned above. In the experiments, we use 12, 16, and 64 node I/O partitions. In both machines, the parallel file system, PFS [28], stripes the user files across the available I/O nodes in a round-robin fashion. The default stripe unit size, which is 64 KB in the PFS, is used in all of our experiments, except `SCF 1.1` where we make experiments with different stripe unit sizes.

IBM SP-2. For the `BTIO` application, we used an SP-2 with 80 nodes. All nodes that are used in the experiments were RS/6000 Model 390 nodes with 256 MB memory. The parallel file system, PIOFS [11], distributes the files across multiple I/O nodes. Only four out of the five I/O nodes are available for the user files and each such node has four 9 GB SSA disks attached to it. The fifth node is the directory server. The striping unit (called BSU in the PIOFS terminology) is 32 KB. The IBM SP-2 that we used in our experiments was running version 1.2 of the PIOFS software and there were four I/O nodes or servers that were part of the I/O subsystem. A PIOFS file is stored by striping data across the various servers in a round-robin fashion.

4.3 SCF 1.1: Effect of Efficient Interface and Prefetching

We investigated the performance of `SCF 1.1` for small as well as large numbers of compute nodes separately. Our analysis for small numbers of compute nodes is more detailed and uses the Pablo traces, whereas in large numbers of nodes, we were more interested in the scalability-related issues. Also, in large numbers of nodes, the huge disk space requirements and time constraints prevented us from using Pablo. An important observation about this application is that the programmers reasonably optimized its I/O-related parts. They first packed the data to be written onto disk in larger chunks and then wrote the packed chunk in a single I/O call. While this effort renders further I/O optimizations difficult; it makes the applications' I/O pattern amenable to prefetching. In these experiments, we evaluated three versions of the application:

1. The original version [25] that uses Unix I/O with a Fortran interface (to the underlying *parallel file system*) as provided by the Pacific Northwest Lab (PNL),
2. an optimized version that uses the PASSION [8] library's read/write calls, and
3. an optimized version that uses PASSION prefetch calls instead of read calls.

Essentially, the second version improves the interface to the file system (by using the PASSION's optimized I/O techniques) and the third version applies prefetching optimization.

As mentioned earlier, the I/O activity of the `SCF 1.1` code comprises two main phases:

TABLE 2
I/O Summary of the Original Version for `SMALL`: 4 Compute Nodes

Operation	Operation Count	I/O Time (Seconds)	I/O Volume (Bytes)	Percentage of I/O time	Percentage of Execution time
Open	19	3.13		0.20	0.08
Read	14,521	1,489.07	909,301,536	93.76	39.28
Seek	1,018	17.0		1.07	0.45
Write	2,442	78.01	57,477,540	4.91	2.06
Flush	50	0.44		0.03	0.01
Close	14	0.52		0.03	0.01
All I/O	18,064	1,588.17	966,779,076	100.0	41.9

TABLE 3
Read and Write Size Distribution of the Original Version for `SMALL`: 4 Compute Nodes

Operation	Size $<$ 4K	4K $\leq$ Size $<$ 64K	64K $\leq$ Size $<$ 256K	256K $\leq$ Size
Read	646	3	13,872	0
Write	1,572	3	867	0

1. *Write phase* (performed only once); integrals are calculated and packed in a memory buffer and written to disk after the buffer is full.
2. *Read phase* (performed in every iteration); integral values are read from disk to a memory buffer to compute Fock matrix; integral file is resident on disks, striped across various I/O nodes and accessed through the parallel file system.

The other small I/O accesses are to the input file and a runtime database file used for check-pointing some values, but these constitute an insignificant part of the total I/O activity.

We consider three representative inputs which we call `SMALL` ($N = 108$), `MEDIUM` ($N = 140$), and `LARGE` ($N = 285$). Problems as large as 2,500 basis functions as input can be successfully executed, but they require a lot of processor-hours, perhaps even processor-days. Due to the lack of resources of such magnitude available to us, we had to scale down our problem sizes. Nevertheless, we believe that these inputs capture all the I/O issues faced by larger input sizes and, hence, can be considered as representative.

In the following, we will present some I/O summary tables (obtained using Pablo [27]) for the various experiments, these will capture the various I/O operation counts, time required, volume of data read or written, and percentages of I/O and execution times for different operations. Note that this includes the I/O activity performed by *all* the compute nodes executing the application. Along with these, we give information about the distribution of the request sizes issued from the application. All of the above information is useful in understanding the dynamics of the I/O activity and the actual I/O bandwidth experienced by the application.

In Table 2, we summarize the results of the `SMALL` input with the various I/O measurements. First, we find reads to be dominant in operation count, I/O time, and volume of data, because the reads are repeated over several iterations of the HF calculations. 93.76 percent of the I/O time consists of read operations, which is 39.28 percent of the total execution time. This is followed by writes, which are done to write the integrals to files in the write phase and they contribute to 4.91 percent of the total I/O time or 2.06 percent of the total execution time. All other operations such as open, close, seek, and flush take less than 2 percent of the total I/O time. We clearly see that, for this input, I/O plays a major role in the HF code. In Table 3, we present the distribution of the read and write request sizes. The small reads and writes in the range of ($<$ 4K) correspond to the initial reads to the input file and writes to the runtime database file for check-pointing some calculations. The larger writes and reads (64K $\leq$ Size $<$ 256K) correspond to the integral file writes and reads. We also made experiments with the `MEDIUM` and `LARGE` inputs. The results are similar to those given for the `SMALL` input and indicate that the I/O time constitutes 62.34 percent (54.06 percent) of the total execution time for the `MEDIUM` (`LARGE`) input (on four processors). We have also found that the majority of the I/O requests are of size 64K - 256K.

In Fig. 1, we present the durations of the read and write operations performed in `SMALL` over the entire execution time of the program. We can clearly identify the write phase of integral values (performed just once), followed by the read phase (performed many times), and the small reads at the very beginning, where the small input files are read and some scarce small writes to the runtime database file sprinkled about. We found that the average duration of read operations is 0.1 second and that of write operations is 0.03 second. Although, the actual values naturally differ, the I/O behavior of the original code with the `MEDIUM` and `LARGE` inputs is very similar to that with the `SMALL` input.

Table 4 gives the I/O summary of the second version of HF (with the PASSION read/write calls) for the `SMALL` input size. The I/O time now constitutes only 27 percent as opposed to the 41.9 percent of the original case (refer to Table 2). Out of this 14.9 percent reduction, almost 14 percent is due to the reduction in the read times. Note that the number and order of I/O calls remain the same; that is, all the benefit is coming from efficient interface

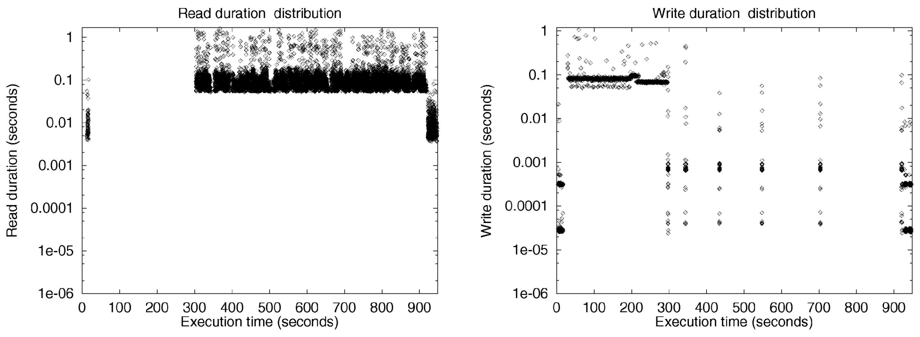


Fig. 1. Read and write operation durations of the original version for SMALL: 4 compute nodes.

TABLE 4
I/O Summary of the PASSION Version for SMALL: 4 Compute Nodes

Operation	Operation Count	I/O Time (Seconds)	I/O Volume (Bytes)	Percentage of I/O time	Percentage of Execution time
Open	19	0.67		0.08	0.02
Read	14,521	732.49	909,301,536	93.23	25.17
Seek	15,693	14.16		1.8	0.49
Write	2,446	37.39	57,477,556	4.81	1.3
Flush	50	0.17		0.02	0.01
Close	14	0.44		0.06	0.02
All I/O	32,743	785.72	966,779,092	100.0	27.0

TABLE 5
I/O Summary of the Prefetch Version for SMALL: 4 Compute Nodes

Operation	Operation Count	I/O Time (Seconds)	I/O Volume (Bytes)	Percentage of I/O time	Percentage of Execution time
Open	19	0.64		0.67	0.02
Read	649	3.17	75,168	3.32	0.12
Async Read	13,936	35.07	913,421,184	36.83	1.36
Seek	15,757	13.20		13.87	0.51
Write	2,446	38.58	57,477,556	40.52	1.5
Flush	50	0.16		0.17	0.01
Close	14	4.39		4.61	0.17
All I/O	32,871	95.20	970,973,908	100	3.69

between PASSION and the PFS. The PASSION library is highly optimized for the underlying parallel file system and uses smart buffering strategies to optimize the I/O accesses as much as possible. In this program, the reduction in I/O time equals the reduction in execution time. Since the PASSION library has no knowledge of where the file pointer is from a previous I/O call, a fresh seek must be performed for every call. This increase in the number of seeks contributes only to a small fraction of the I/O time.

Table 5 presents the I/O summary of the third version of HF (with PASSION prefetch calls) for SMALL. The I/O times here measure only the time spent on issuing the calls. They neither take into account the waiting time nor the copy time. While in the current iteration of the program, we use the PASSION prefetch routines to prefetch the next group of integrals required for the successive iteration. It should be noted that PASSION prefetch is implemented using the underlying file system's asynchronous I/O support and its success depends largely on how efficiently it has been implemented. A majority of the reads are asynchronous reads, that correspond to the prefetch operations. The total read time contributed to 93.23 percent of the I/O time in the PASSION case and now contributes to a much smaller fraction (36.83 percent) of the total I/O time. The reduction is so significant that the time has become less than that taken by the write operations, which contributed to only 4.81 percent of the total I/O time in the case of the PASSION version (see Table 4). We see that the I/O time is reduced very significantly (from 785.72 seconds to 95.20 seconds) in the entire program. The I/O time only contributes to 3.69 percent of the execution time, as opposed to 27 percent in the PASSION version. However, there is only an 11 percent reduction in the total execution time, which shows that the reduction in I/O time is not equal to the reduction in execution time. This is due to the fact that there is insufficient overlap between the computation time and the time to complete a read operation.

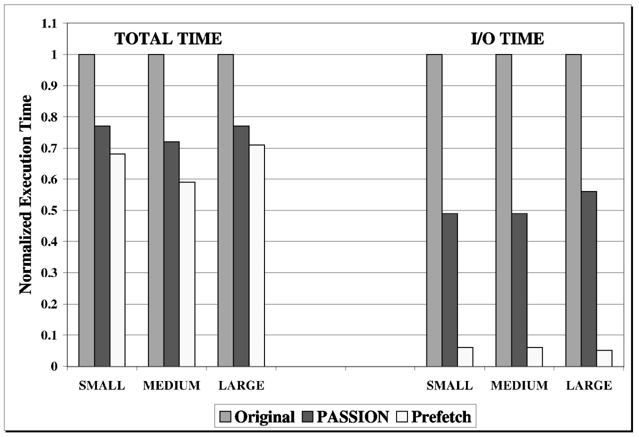


Fig. 2. Performance summary of the PASSION and prefetch versions for SMALL, MEDIUM, and LARGE inputs.

We also note that prefetching did not produce the results we expected and we argue that this is due to:

1. The PASSION library uses the file system's non-blocking or asynchronous reads for prefetching and because of this, it has to translate a single request to a logically contiguous chunk of data access into multiple requests to physically contiguous chunks of data accesses. This bookkeeping contributes to a big percentage of the prefetching overhead. Posting of individual requests also adds to the overhead, as each request needs to obtain a token to be entered in the queue of asynchronous requests to a given file.
2. Copying data from the prefetch buffer to the application buffer also contributes to a nonnegligible percentage of the prefetch overhead.

Fig. 2 summarizes the execution times of the Original version, the PASSION version and the Prefetch version for our inputs. We see that there is a significant reduction in the execution time due to the change of the file system interface. We find a moderate reduction in the execution time going from the PASSION version to the Prefetch version. PASSION produces a 23 percent, 28 percent, and 23 percent reduction in total time for SMALL, MEDIUM and LARGE, respectively; and 51 percent, 51 percent, and 44 percent reduction in I/O time for SMALL, MEDIUM and LARGE, respectively. Prefetch produces a 32 percent, 43 percent, and 29 percent reduction in execution times for SMALL, MEDIUM, and LARGE, respectively; and 94 percent, 94 percent, and 95 percent reduction in I/O time for SMALL, MEDIUM, and LARGE, respectively.

The impact of the different factors for the small number of compute nodes are summarized in Fig. 3. This figure presents a summary of an incremental evaluation of the I/O optimizations that we performed for small processor sizes. As detailed in Table 6, we represent each optimization combination in Fig. 3 with a five-tuple of (V,P,M,Su,Sf), where V is the version used (O—original version with Fortran I/O calls, P—optimized version with PASSION read/write calls, F—optimized version with PASSION prefetch calls); P is the number of processors (compute nodes); M is the memory available to the application (in MB); Su is the stripe unit size (in KB); and Sf is the stripe factor (number of I/O nodes in this case). Notice that the tuple (O, 4, 64, 64, 12) corresponds to our default configuration. For the measurement of the I/O times in the prefetching versions, we take into account the I/O, wait and copy times also. The important point to note is that the effect of the optimizations is quite similar in all three input sizes. It is easy to see that the factors that can be modified from within the software are much more effective than the system-related factors like number of compute nodes and number of I/O nodes within this experimental domain. In general, we can conclude that when the number of compute nodes is small, the application-related factors have a high impact on the execution and I/O times of the application.

The results for the larger processor case are presented in Figs. 4 and 5 (using the LARGE input). We notice from Fig. 4b) that up to 64 processors, the software optimizations are more effective in the overall performance, however, beyond 64 processors, the lack of I/O resources dominates, and the unoptimized version with 64 I/O nodes starts to outperform the optimized versions with 16 I/O nodes. From Fig. 5, we infer that as the number of compute nodes used increases so does the contention at the I/O nodes. We observe that the increase in I/O nodes (Sf) translates into reduced I/O contention and results in improved total execution times, especially when we use larger number of compute nodes.

4.4 SCF 3.0: Effect of Balanced I/O

We evaluated the I/O and overall performance of the SCF 3.0 from both hardware and software points of view. As in SCF 1.1, we have found efficient interface and prefetching quite useful. Since we discussed those issues

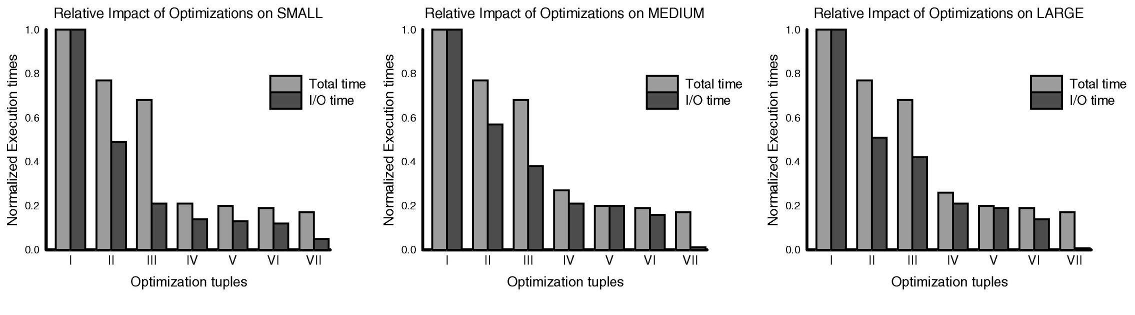


Fig. 3. Impact of different parameters on different input sizes for SCF 1.1 on the Intel Paragon. The number of basis functions (N—problem size) for SMALL, MEDIUM and LARGE are 108, 140, and 285, respectively. The configuration tuples are given in Table 6.

TABLE 6
The Configuration (Optimization) Tuples Used in the Experiments

Tuple #	Version (V)	# of Processors (P)	Memory Size (MB) (M)	Stripe Unit (KB) (Su)	Stripe Factor (Sf)
I	Original (O)	4	64	64	12
I I	PASSION (P)	4	64	64	12
I I I	Prefetch (F)	4	64	64	12
I V	Prefetch (F)	32	64	64	12
V	Prefetch (F)	32	256	64	12
VI	Prefetch (F)	32	256	128	12
VI I	Prefetch (F)	32	256	128	16

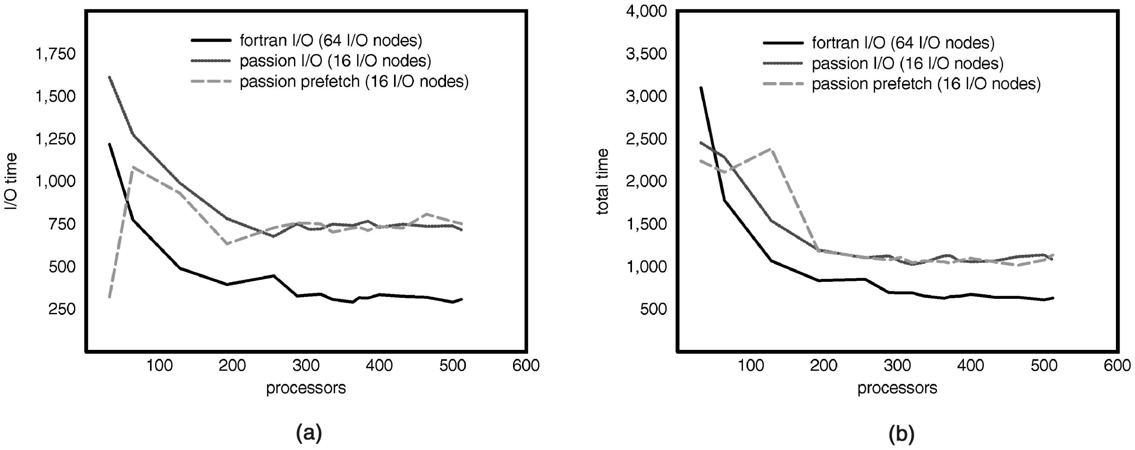


Fig. 4. Performance summary for `SCF 1.1` for `LARGE` input on the Intel Paragon. (Up to 64 compute nodes optimized versions perform well; beyond 64 compute nodes the unoptimized version with larger number of I/O nodes performs better.)

with `SCF 1.1`, due to lack of space, we do not elaborate on them in this section. Instead, we focus on another optimization technique, which we call *balanced I/O*, that is based on the efficient use of available memory. In contrast to `SCF 1.1`, the application programmers of the `SCF 3.0` give the user the opportunity of balancing I/O versus computation. That is, the user can specify what percentage of the integrals are to be cached on disk and what percentage are to be recomputed when necessary. We found that the ratio can make a critical difference in the overall performance of the application. The problem, however, is that the best ratio is dependent on the input size.

Fig. 6 shows the overall performance of the application (times in seconds) on 16 and 64 I/O nodes for different processor sizes and percentages of cached integrals. The first observation is that the number of I/O nodes is not very effective on the overall performance. The other two factors, however, namely the number of compute nodes and percentage of cached integrals do make a difference as shown in the figures. This is in contrast to `SCF 1.1` where the number of I/O nodes is a critical factor. One reason for this is that, in `SCF 3.0`, I/O is not as dominant as in `SCF 1.1`. Another point to note is that changing the number of compute nodes makes a big difference, especially with the low percentages of cached data. This can be expected as in low percentages of cached data, there is really little I/O activity going on within the application.

The capability of changing the percentage of the cached integrals presents the user with some opportunities. In order to improve the performance, either the number of processors or the percentage of the cached integrals can be increased. The choice depends on the availability of extra disk space versus additional number of compute nodes. For the platform on which these experiments were conducted (the Intel Paragon), we found that increasing the percentage of integrals stored on the disk gave better performance. For example, when the percentage of integrals cached is around 90 percent or so, (for the 64 I/O nodes case), we found that increasing the number of processors from 32 to 256 did not give any observable performance gain in the execution time. But if the disk space is limited and can only partially fit the

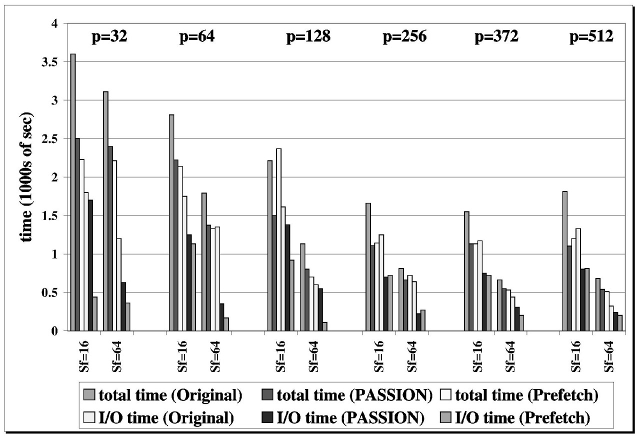


Fig. 5. Effect of increasing the number of I/O nodes on `SCF 1.1` on the Intel Paragon.

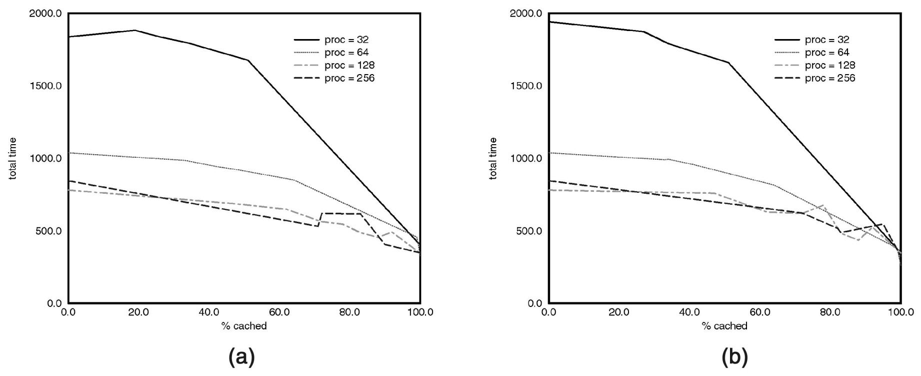


Fig. 6. Performance of `SCF 3.0` for different percentages of cached integrals for `MEDIUM` input on the Intel Paragon (a) with 16 I/O nodes and (b) with 64 I/O nodes. (Note that for the full recompute version (0 percent cached), increasing the number of processors is very effective whereas for the full disk version (100 percent cached) increasing the number of processors does not make a significant difference. All times are in seconds.)

integrals, then using larger number of processors to reduce the load of recomputation per processor is beneficial in decreasing the total execution times. From the results, we can conclude that for `SCF 3.0` on the Intel Paragon, the amount of disk space available for caching is more important followed by the number of compute nodes. Only in the event of less disk space, would increasing the number of processors be desirable.

At this point, we would like to explain why the other optimizations do not perform well for both the `SCF` codes. First, we cannot take advantage of the file layout optimizations, because these optimizations mainly work for *multi-dimensional* arrays [21], whereas the main data arrays used in the `SCF` codes are *one-dimensional* as the implementors linearized the multidimensional arrays in order to take better advantage of software level buffering. Unfortunately, we cannot employ collective I/O optimization either, because collective I/O is, in general, useful for the applications where each node's data are interleaved in file with the other nodes' data. In the `SCF` codes, this is not the case; so, the nodes can access their portion of the data by issuing minimum number of I/O calls, obviating the need for collective I/O.

4.5 FFT: Effect of Layout Optimization

Fig. 7a shows the I/O times (in seconds) for three different versions of the `FFT` code on the Intel Paragon: two versions of the original program with 2 and 4 I/O nodes, and an optimized version on 2 I/O nodes. The results show that the I/O performance of the unoptimized program is very poor. In the original unoptimized (2 I/O node case) version, the I/O time actually increases when we use more than 4 compute nodes. When we increase the number of I/O nodes to 4, the increase in the I/O time happens after 8 compute nodes. (Recall that on the small Paragon machine where these experiments are performed we can use maximum 4 I/O nodes and 16 compute nodes). We note that this trend in the I/O time almost identically reflects the total execution time (see Fig. 7b). The reason for this is that the I/O time for this application constitutes 90 percent-95 percent of the execution time and therefore is the dominant factor in the overall behavior of the application.

The most costly operation in the 2D out-of-core FFT is a 2D out-of-core local transpose performed by each processor using two disk-resident arrays. In the original program, file layout for these two arrays is *column-major*. The transpose is performed by reading a rectangular chunk from one of the files (size of which depends on the available local node memory), transposing it in the local memory, and writing it in the other file. Since both the files are column-major,

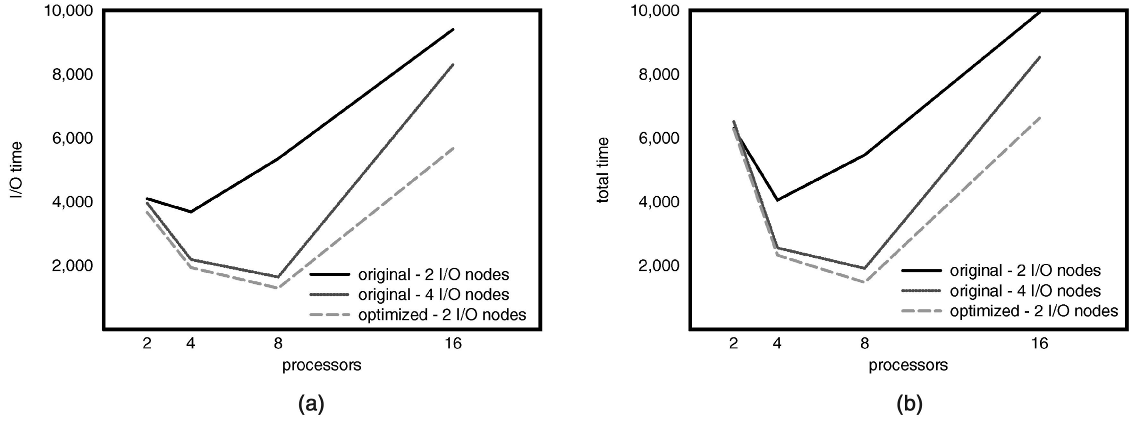


Fig. 7. Performance of the `FFT` code on Intel Paragon. (Total I/O amount is 1.5 GB and all reported times are in seconds.)

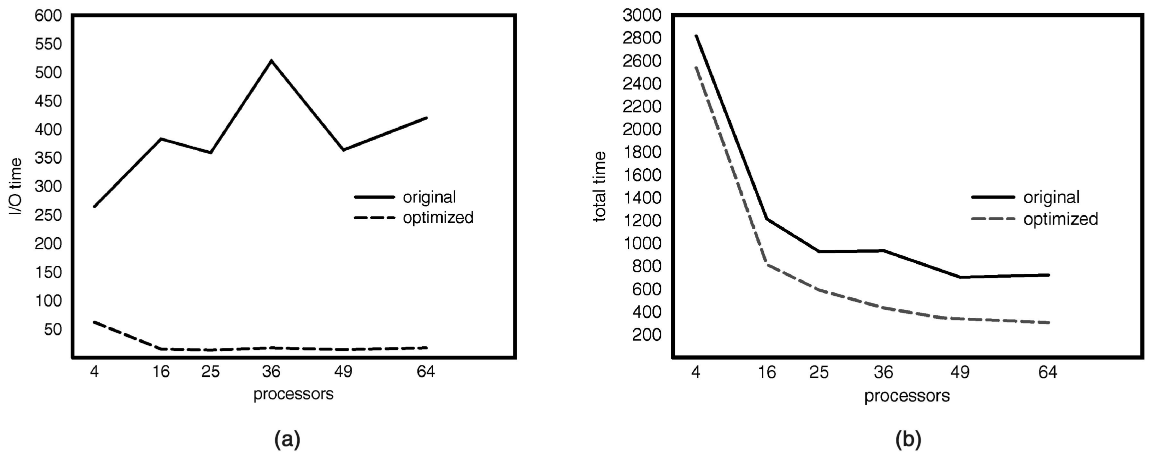


Fig. 8. Performance of BTIO on IBM SP-2 for Class A input. (a) I/O time and (b) Total time. (Total I/O amount is 408.9 MB and all times are in seconds.)

optimizing the block accesses for one array has a negative impact on I/O performance of the other array (due to the transpose operation), resulting in poor I/O performance observed in Fig. 7. On the other hand, if we store one of the arrays in *row-major* order, the I/O performance of both the arrays improves. This is evident from Fig. 7 where the optimized version of the program on two I/O nodes outperforms the unoptimized program on four I/O nodes for all processor sizes. For this example, within this experimental domain, we can conclude that the layout optimizations are effective and the optimized version outperforms the unoptimized version which uses a greater number of I/O nodes.

An important point about those types of layout optimizations is that they can be detected by parallelizing compilers using suitable linear algebraic techniques. For example, Kandemir et al. [21] shows how the data layout optimizations can be automated within a parallelizing compiler targeting out-of-core computations. The main idea is to choose the appropriate file layouts for disk-resident arrays referenced in an I/O-intensive program. To achieve this goal, an optimizing compiler employs a suitable analysis to detect the access pattern of the individual loop nests in the program at compile-time, then depending on the collected information, it decides which file layout to choose for each disk-resident array.

There is a potential problem, however, with the file layout optimizations explained above. If a program contains a number of consecutive loop nests, the different loop nests accessing the same array may require different optimal file layouts for the array. Fortunately, in most of the cases where these conflicts occur, we can apply loop (iteration space) transformations to mitigate the negative impact of the new layout in different loop nests. We refer the interested reader to [19] and [20] for an in-depth discussion of the file layout transformations.

We now discuss why the other optimizations were not effective for this application. First, collective I/O was not required as each processor could read its portions of both arrays in an efficient manner without the involvement of the other processors. Since we have used the PASSION library in implementing the application, the interface to the file system was able to capture the high-level view of the application's access pattern.

Someone might think that prefetching might be useful for such an application. But, due to the amount of the computation and to the amount of the associated I/O in a single iteration of the program, we found that the prefetching was not very useful. However, since these costs largely depend on our specific implementation, we believe that in some other (more efficient) implementations prefetching can prove useful for such an access pattern.

4.6 BTIO: Effect of Collective I/O

As mentioned earlier, the base version of this application uses MPI-IO as the interface and contains a lot of seek operations. The I/O performance of BTIO for the input Class A is shown in Fig. 8a and the overall performance is shown in Fig. 8b, respectively, (as usual all the reported times are in seconds). In contrast to FFT, the BTIO code is not as I/O dominant. From Fig. 8a, it is easy to see that the I/O time in the unoptimized program changes drastically with the increasing number of processors. This, in turn, causes a hump in the execution time when 36 processors are used. The main problem with the I/O performance of this application is that each node performs its I/O independently from the others. For example, if a node needs 12 chunks of data, it will issue 12 separate I/O calls, one for each of the chunks. While this approach simplifies the programming, it incurs a substantial overhead, as the number of I/O calls is the dominant factor in the I/O time. This behavior was observed with other classes of inputs as well.

The reference [8] discusses a technique called *two-phase I/O* (a form of collective I/O) which means that each processor reads the portion of the disk data that is least costly for it; and then the processors use the available interconnection network to exchange the parts of the data so that each processor gets what it needs. Although, this approach increases the communication time of the program, it generally minimizes the number of I/O calls which in turn reduces the execution time significantly. In two-phase I/O, the processors cooperate in accessing the data on disks. The aim is to combine several I/O requests into fewer larger granularity requests

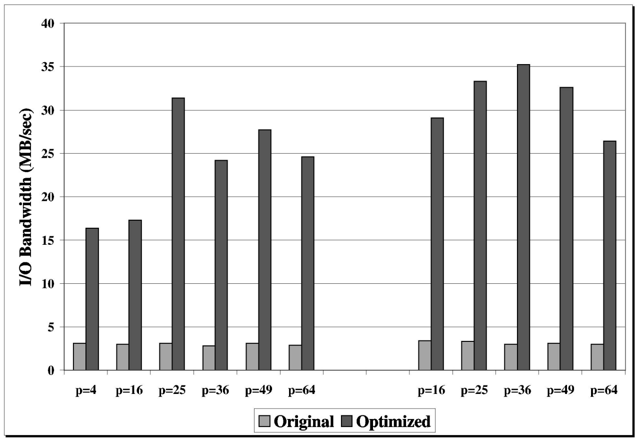


Fig. 9. I/O bandwidths of the original and optimized versions of `BTIO` on the IBM SP-2 for Class A and Class B inputs.

and reorder requests such that the file will be accessed in a close-sequential fashion.

The optimized version of `BTIO` uses the two-phase I/O. The solution vector is completely described by using MPI data types. Fig. 8a shows that the I/O time is reduced significantly in the optimized version and it does not behave unpredictably with the increasing number of compute nodes. The reason is that in the unoptimized program, increasing the number of compute nodes will decrease the volume of data processed by a processor; but, in general, does not change the number of I/O calls per processor. Consequently, the total number of I/O calls in the program increases substantially. On the other hand, in the optimized program, the increase in the number of I/O calls is equal to the increase in the number of processors as each processor issues a single I/O request from the application. The impact of the two-phase I/O in the overall performance is shown in Fig. 8b. As an example, with 36 and 64 processors, there is 46 percent and 49 percent reduction in the overall execution time, respectively. A similar trend is observed in the other input classes as well.

We also measured the I/O bandwidth of the original and the optimized versions. The results are given in Fig. 9. The I/O bandwidth of the original program is between 2.8 MB/sec and 3.4 MB/sec while the I/O bandwidth of the optimized version is between 16.4 MB/sec and 35.2 MB/sec. In summary, `BTIO` is an example of a class of I/O intensive programs in which the I/O performance can be improved by collective I/O optimization, but since the I/O does not constitute a large bulk of the execution time, the impact of the optimizations on the overall performance is limited.

We now briefly discuss the impacts of other optimizations on the performance of this application. Since the application is written using MPI-IO, there was not a severe interface problem. Moreover, in this application, we have found that for the best I/O performance all the disk-resident arrays should have the same file layout whether it is row-major or column-major. We believe that the programmers paid enough attention to successively order the computations in costly loop nests to avoid any file layout transformation. As with the previous application, we did not observe any marked difference on the I/O performance when we applied prefetching.

TABLE 7
Execution Times (in Seconds) for `AST`

Number of Processors	Unoptimized		Optimized	
	16 I/O Nodes	64 I/O Nodes	16 I/O Nodes	64 I/O Nodes
16	2,557	2,546	428	399
32	1,203	1,199	100	97
64	638	628	76	69
128	385	369	86	77

(The I/O amount is 2.2 GBytes.)

4.7 AST: Effect of Collective I/O

The I/O characteristics of this application are very similar to those of the previous application. We have included this application in our suite, however, as it is much larger than the `BTIO` code.

The results for this application (in terms of execution times in seconds) with a reasonably large input array size of 2K $\times$ 2K elements is presented in Table 7 for 16 and 64 I/O nodes on the Intel Paragon. As mentioned earlier, the astrophysics application performs I/O for data analysis, check-pointing, and visualization purposes. At every dump point, data for the three purposes are written by the various processors onto a shared file. To be specific, the snapshots of the input array are written to disk at fixed dump points for check-pointing and data analysis. Data are also processed and written out for the purposes of visualization. We compare two different implementations of the code:

1. *Unoptimized version:* I/O done using the Chameleon library and
2. *Optimized version:* I/O done using a runtime system library performing two-phase I/O.

We see a significant performance improvement in the overall execution time in the optimized case due to huge reduction in the I/O time. The Chameleon library makes I/O in smaller noncontiguous chunks and also has a bottleneck of all I/O performed by a single node and this adds to the I/O time. The two-phase I/O approach, on the other hand, eliminates small I/O requests by performing large chunks of sequential I/O. Therefore, in this application we see that this factor is more important (within our experimental domain) than increasing the number of I/O nodes as shown in Table 7.

To evaluate the performance of collective I/O further, we made experiments with the Spec/Nasa7 (from SpecFP92) using 16 nodes SP2. For this purpose, the procedures in the Spec/Nasa7 have been modified such that they read and write disk-resident multidimensional arrays. Fig. 10 gives the execution times in seconds for both unoptimized (left bars) and optimized (right bar) version for different set of experiments (called setups in the figure). These results show that collective I/O improves the performance significantly.

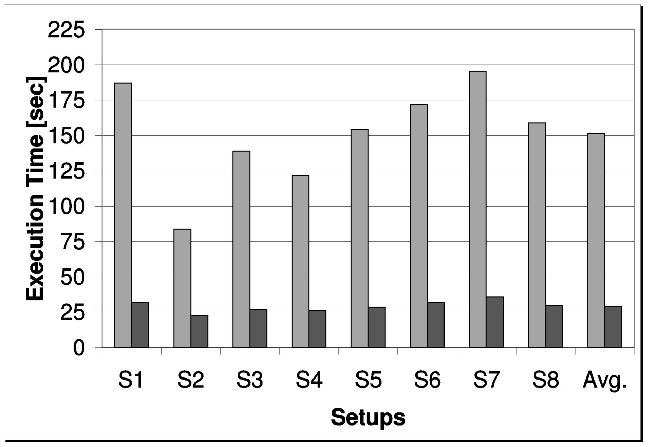


Fig. 10. The execution times for the unoptimized (left bars) and optimized (right bars) on the IBM SP-2.

4.8 Results on Clusters

Recent trends in computer architecture show that networks of workstations (also referred to as clusters) are emerging as a cost-effective solution for high performance. Clusters are expected to gradually phase out the conventional parallel machines/supercomputers. In this section, we focus on two applications, `FFT` and `BTIO`, and evaluate the impact of our optimizations on an 8-node Pentium/Linux based cluster environment. Each node on this cluster has a 750 MHz Intel Coppermine microprocessor and equipped with a 20GB Maxtor hard disk drive and a 32bit PCI 10/100Mbps 3-Com 3c59x network interface card. All the nodes are connected through a Linksys Etherfast 10/100Mbps 16 port hub.

Fig. 11 gives the percentage I/O time improvements brought by our optimizations over the original versions of the applications. From these results, we can observe two things. First, our optimizations are effective even in a modern cluster-based environment (in addition to a supercomputer environment). In fact, we see that the improvements in the Linux cluster are higher (in most cases) than the improvements in the supercomputer environment when the same number of processors are used. Second, the effectiveness of our optimizations increase with the increased number of processors. This is because the original codes perform extremely poor when the number of processors is increased. Note that this last result is very significant as the current trend in high performance computing is to employ larger and large clusters. In general, we expect our I/O optimizations to be effective in a cluster environment as well.

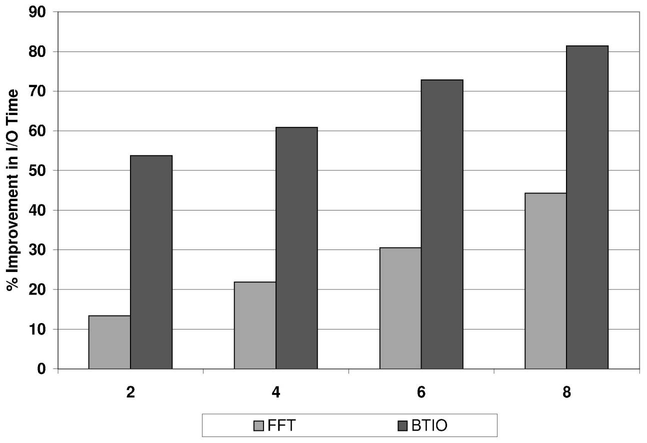


Fig. 11. Percentage improvements in I/O time for the `FFT` and `BTIO` applications on a Linux cluster. x-axis corresponds to the number of nodes.

TABLE 8
Applications and Effective Optimization Techniques

Application	Optimization Techniques				
	Collective I/O	File Layout	Efficient Interface	Prefetching	Balanced I/O
SCF 1.1			√	√	
SCF 3.0			√	√	√
FFT		√			
BTIO	√				
AST	√				

A tick-mark (√) is entered in the table on the effective optimization.

5 Systematic Approach

In this section, we discuss our results from a general perspective and try to reach some guidelines which might be beneficial for application programmers as well as library and compiler writers. First, from our experiments (whose results are summarized in this paper), we observe the following: the I/O intensive applications in our experimental suite (in their original form) deliver very poor performance mostly due to the I/O bottleneck. This bottleneck originates from both hardware and software perspectives. From the hardware perspective, the limited number of I/O nodes on the Intel Paragon and the IBM SP-2 limits the performance of such applications. The problem becomes so severe that beyond a number of compute nodes, the execution times actually increase. As an example, while the users of `SCF 1.1`, for small number of compute nodes, use the version of the code which makes I/O instead of the version which recomputes the integrals; for large number of compute nodes, they tend to use the recompute version, as the I/O version performs very poorly. Although, an obvious solution to this problem is to increase the number of I/O nodes, we cannot indefinitely increase the I/O resources and at some point we need to resort to efficient I/O optimizations from within the software.

From the software point of view, the I/O software is not easy to use and is not easily portable. For example, both PFS and PIOFS have different I/O modes which make the programming for I/O very difficult for the user. Unfortunately, the compiler techniques for I/O are not robust enough to attack the problem either. Throughout the years, several I/O optimization techniques have been developed, but they have been either tested on specific applications or specific machines. In this paper, we applied several I/O optimization techniques found in the literature to the programs in our experimental suite.

We summarize our results in Table 8. An entry (x, y) in this table is marked with $\surd$ if we observe a *significant impact* of optimization y on the I/O performance of application x.

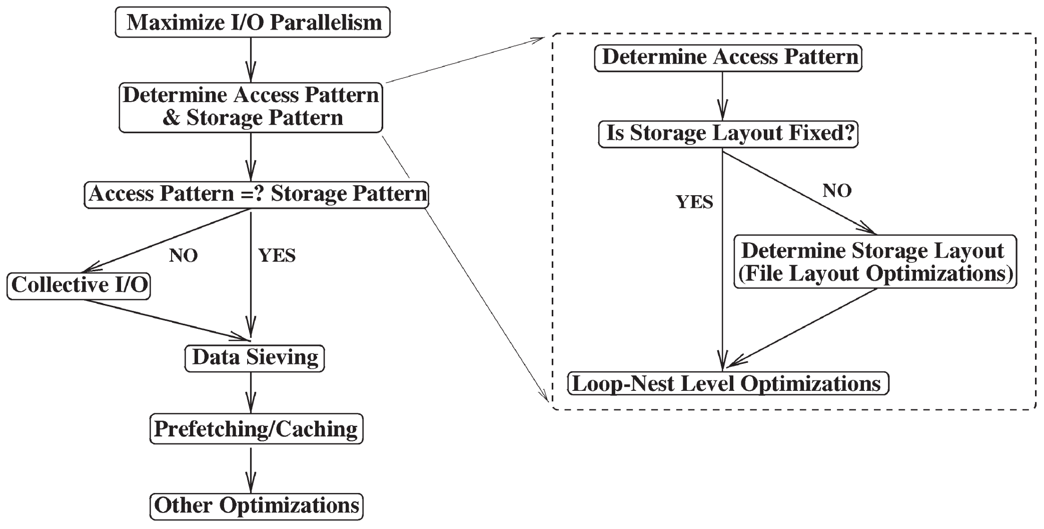

Maximize I/O Parallelism
Determine Access Pattern & Storage Pattern
Access Pattern =? Storage Pattern
NO
YES
Collective I/O
Data Sieving
Prefetching/Caching
Other Optimizations
Determine Access Pattern
Is Storage Layout Fixed?
YES
NO
Determine Storage Layout (File Layout Optimizations)
Loop-Nest Level Optimizations

Fig. 12. A systematic approach for optimizing I/O.

An obvious fact is that different I/O-intensive applications are amenable to different types of I/O optimizations.

We now discuss what types of applications can benefit from what types of optimizations. We can broadly divide the I/O optimizations into two classes:

1. Optimizations that target access pattern of a number of processors and
2. Optimizations that target access pattern of a single node.

The class 1) optimizations include collective I/O. These optimizations will be most useful in applications in which each processor accesses noncontiguous parts of a global data structure. In such cases, for example, collective I/O can modify the global access pattern to enable more efficient (contiguous) disk accesses at the expense of some extra communication. In cases where

$$\begin{aligned} &Unoptimized\ I/O > Optimized\ I/O \\ &+ Extra\ Communication \end{aligned}$$

holds, this optimization will be useful. The class 2) optimizations include, for example, prefetching and caching. If each processor will be accessing consecutive parts of a disk-resident structure, we can consider prefetching provided that there is sufficient extra disk bandwidth. In that case, each processor can overlap computation with I/O, leading to high performance. Similarly, in cases where I/O calls issued from a processor exhibit temporal locality, we can consider caching. Of course, orthogonal to all these optimizations, the users of the I/O-intensive applications must try to take advantage of different interfaces to the next level software and application specific optimizations.

Given all these, an important question from the software point of view is how to select a proper subset of optimizations given an application. Although, for the I/O-intensive programs, it is hard to generalize the optimization process, from our experiments in this paper and from our previous experience on I/O software, we can infer a systematic approach that can be adopted by application programmers. The overall optimization scheme for a given data set (array) is sketched in Fig. 12. Normally this process should be repeated for each data set. However, some steps can be performed for multiple data sets simultaneously (e.g., loop transformations).

Our approach starts with optimizing I/O parallelism. This in general involves modifying the access pattern of loop nests to enhance data parallelism. Depending on the application at hand, an optimizing compiler [20] can also be used to maximize the data access parallelism. Following this step, the global access and storage patterns are optimized. We consider two different scenarios. If the data set has already been created on disk, we need to compare the access pattern and the storage pattern. If these two patterns are the same, that means each processor can access its portion of the file in question independently and we do not need to perform collective I/O. Otherwise, collective I/O (e.g., two-phase I/O or a variant of it) should be performed in order to maximize the number of consecutive elements accessed by each processor in a single I/O call. This step completes the global I/O access pattern optimization. If, on the other hand, the data set has not been created yet, an important task is to determine a suitable storage pattern so that the future I/O accesses to the said array will have low I/O latency. This goal can be achieved if the potential future accesses to the data set are considered first. In other words, we can select a suitable storage pattern by looking at possible use (i.e., access pattern) of the data set. Note that depending on the application under consideration, these future access patterns can be as close as those of the first nest that would use the data set, or as far as those of other procedures in the same application or even of code from other applications that share the data set with the current application. This step corresponds to layout optimizations. Once the storage pattern has been determined, we can check each access pattern with the storage pattern and perform collective I/O if necessary.

After an improved global access pattern has been obtained, then we can try to enhance the I/O performance of individual nodes; this step is called local optimization. In this step, we can use optimizations such as prefetching/caching and data sieving. Data sieving is an optimization where a processor reads a superset of the required data if doing so reduces the total number of I/O calls needed [8]. Finally, additional optimizations ([3]) that interact with I/O such as communication optimizations can be applied.

Returning to our example applications, in `BTIO` and `AST`, this systematic approach can detect that collective I/O is required. Similarly, in the `FFT` code, it successfully detects that assigning different file layouts to different disk-resident arrays would be beneficial. For our computational chemistry codes, on the other hand, since the global access pattern is very well optimized by the application programmers, the approach focuses on local optimizations and employs I/O prefetching as explained in the paper.

Finally, when we have an improved view of the I/O access pattern for all the nodes, we should consider different interfaces (where available) to convey as much information as possible to the underlying software about what the application is intended to do. We hope that in the future libraries and runtime systems will be able to take better advantage of semantic information provided to them through sophisticated interfaces.

6 Related Work

In this section, we summarize the related software work in optimizing performance of the I/O-intensive codes. Throughout the years, researchers have handled the I/O problem in several levels including but not limited to applications, file systems and operating systems, runtime libraries, languages, and compilers. In the following, we discuss the work most relevant to our study.

6.1 Application I/O

In addition to requiring significant amounts of processing power, parallel applications also require high-capacity I/O subsystems that can sustain high bandwidths. In [13], Del Rosario and Choudhary present the various problems and prospects in achieving high-performance I/O. They discuss the I/O requirements of *Grand Challenge* applications—a list of scientific and technical applications that range from 500 MBytes to 500 GBytes and can reach the range of Terabytes with the advent of Teraflops machines. Three I/O-intensive applications from the scalable I/O initiative's application suite are studied by Crandall et al. [10] using Pablo [27], a performance analysis tool. Their recommendations to parallel file systems are to support various access patterns, request sizes and orderings efficiently through optimizations such as prefetching and caching. They found that the applications exhibited a wide variety of temporal and spatial access patterns. Smirni et al. [30] compare two different applications implementing the Hartree-Fock (HF) method and use a high-level I/O library called the PPFS [15] and determine appropriate caching and prefetching policies. They discuss two different algorithms for the SCF method, namely, MESSKIT and NWChem. They point out the importance of the C interface to the I/O subsystem as one reason that the MESSKIT version performs smaller duration I/O accesses than NWChem, which uses the Fortran interface. Thakur et al. [32] use an I/O-intensive, three-dimensional parallel application code to evaluate the I/O performances of the IBM SP-2 and the Intel Paragon. They experiment with different input sizes and for large inputs they find the IBM SP-2 to be faster for read operations and Paragon for writes. In comparison with those studies, we evaluated a larger set of applications and a larger set of I/O optimization techniques and tried to reach some guidelines that can be useful to application programmers as well as library and compiler writers.

6.2 Parallel File Systems and Runtime Libraries

Commercial file systems such as PFS [28] and PIOFS [11] support various access patterns in defined modes and hints that the user can specify at the time of using a parallel file. The PASSION (Parallel and Scalable Support for Input/Output) library [8] performs collective I/O using a two-phase method. SOLAR [34] is another out-of-core library that supports dense matrix computations that provide out-of-core functionality similar to the in-core BLAS and LAPACK for shared-memory machines and the in-core ScaLAPACK for distributed-memory machines. The PANDA library [29] uses *server directed I/O*, which implements a form of collective I/O similar to the two-phase I/O except that it is a server-side implementation. PPFS [15] is a portable user-level parallel I/O library that uses a client/server model. PPFS clients consisting of the user application linked with the PPFS library are spawned across the processors and servers are spawned on each logical disk managed by PPFS. In addition to portability, the MPI-IO standard [9] provides optimizations in a high-level interface. PVFS [6] is a parallel file system for Linux clusters that presents three different APIs and accommodates frequently used UNIX shell-commands. Its optimizations for noncontiguous data are perhaps less powerful than MPI-IO's optimizations.

6.3 Compiler Optimizations

Current work in the compilation of out-of-core computations uses techniques based on explicit file I/O. In [5], Brezany et al. design directives that can be used in a high-level language to give hints to the compiler and runtime system about the intended use of disk resident data.

From a compiler perspective, an I/O-intensive program can be optimized in three ways: 1) Computation transformations [26], [3], 2) data (file layout) transformations [19], and 3) unified transformations [21], [20]. The techniques based on computation transformations attempt to choreograph I/O, given the high-level compiler directives mentioned above. The computation transformations used by the compiler for handling disk resident arrays can be roughly divided into two categories: 1) approaches based on tiling 2) approaches based on loop permutations. We have found in our study that file layout transformations can be very useful for some codes and hinted at where unified transformations might be necessary.

7 Conclusions

In this paper, we have studied the impact of several I/O optimization techniques found in the literature on the I/O performance of five I/O-intensive applications. We have first analyzed the I/O behavior of the original version of each application and then attempted to apply the optimizations in our suite. We have found that although different applications benefit most from different optimization(s), a rough guideline can be adopted to select a reasonable sequence for applying optimizations. Our experience, albeit limited, suggests first optimizing global I/O pattern to force processors to make as many accesses to consecutive locations as possible and then improving the individual access patterns further using techniques such as file layout transformations and prefetching.

We intend to extend this work in several ways. First, we would like to increase the size of our application suite and optimization techniques. Second, we plan to investigate more the interactions between different optimizations and third, we want to design and implement an optimization tool which, given I/O characteristics of an application in a specific well-defined format, can output a suitable order for optimizations.

Appendix

Explanation of Applications

SCF 1.1: The Hartree-Fock (HF) method obtains the energy and wave function of a molecular system by iterating over two basic steps until self-consistency (SCF) is obtained. At the heart of the Hartree-Fock method is the construction of the Fock matrix F according to the formula,

$$F_{pq} = h_{pq} + \sum_{r=1}^{N} \sum_{s=1}^{N} D_{rs}[(pq|rs) - 12(pr|qs)],$$

where N is the dimension of the basis set, h's are one-electron Hamiltonian matrix elements, D is the one-particle density matrix, and $(pq|rs)$ are two-electron integrals. The iterative procedure starts with an initial guess for the density matrix, D, which is then used to construct a Fock matrix, which in turn is used to improve the density matrix. The one- and two-electron integrals, h and $(pq|rs)$ depend on the positions of the atoms in the molecules and on the specific choice of basis set. Their values do not change during the iterative procedure. The two-electron integrals in particular are quite numerous, formally $O(N^4)$. In addition, evaluating each integral is a nontrivial computation involving 300 to 500 floating-point operations, on average. Compared to the integral evaluation and Fock matrix construction, other phases of the HF calculation are relatively inexpensive.

While the initial implementations of the HF method preferred *recomputing* the integrals whenever needed, recent improvements in high-performance I/O subsystems and in supporting software have made it possible to reconsider the decision to focus on recomputing methods and the majority of quantum mechanical methods in NWChem now include some kind of *disk-based* algorithm. In the HF algorithm, the two-electron integrals constitute the largest volume of data and a sizable computational expense. In a disk-based implementation, the integrals are computed on the first iteration and written to disk, then read from disk rather than being recomputed for each subsequent iteration. In NWChem, each compute node writes a private file of the integrals it evaluated during the first construction of the Fock matrix.

SCF 3.0: The `SCF 3.0` parallel computational chemistry package encompasses a broad range of functionality, including the self-consistent field (SCF) module. In `SCF 1.1`, calculations could be either *direct*, meaning that integrals are recomputed for every iteration of the SCF algorithm, or *disk-based*, meaning that integrals are evaluated once and written to disk during the first iteration, then read from disk on every subsequent iteration (as explained above). The *semidirect* integral calculation approach adopted by the `SCF 3.0` implementors is a compromise between these two extremes, where limits may be specified on the size of disk files and any integrals which are not stored on disk are recomputed. This is the fundamental difference between `SCF 1.1` and `SCF 3.0` and will be studied in detail later in the paper. Some attempt is also made to arrange the integral evaluation from most to least expensive, so that those integrals which must be recomputed on every iteration are generally less expensive than those kept on disk. Taken together, the changes between `SCF 1.1` and `SCF 3.0` mean that SCF calculations with the two can, in some cases, have marked differences in performance due to changes in both computational and I/O components of the code. Consequently, we consider `SCF 1.1` and `SCF 3.0` as two separate applications.

FFT: The Fast Fourier Transform (FFT) is widely used in many areas such as digital signal processing, partial differential equation solutions and various other scientific and engineering problems. We implemented a 2D out-of-core FFT on the Intel Paragon. The 2D out-of-core FFT consists of three steps: 1) 1D out-of-core FFT, 2) 2D out-of-core transpose, and 3) 1D out-of-core FFT. The 1D FFT steps consist of reading data from a two-dimensional out-of-core array and applying 1D FFT on each of the columns. In order to perform 1D out-of-core FFTs, the data on disk is strip-mined into memories of compute nodes. This step is highly parallel, limited in general only by the size of the available memory, individual processor speeds, and the load on the I/O subsystem. After this step, the processed columns are written to file. In the transpose step, the out-of-core array is staged into memory, transposed and written to a file. This step is very expensive in terms of both I/O and communication.

BTIO: This application simulates the I/O required by a pseudo-time-stepping flow solver. It is a disk-based version of a program from the NAS parallel benchmark suite [14]. The main operation in the code is periodic writes performed by all processors to a multidimensional array stored in a disk-resident file. Basically after every k iterations, a three-dimensional solution vector of size N^3 is written to a shared disk file, where N is the input size. Note that periodic write operations are used by such applications for check-pointing and/or offline visualization and analysis of data. An important characteristic of this application is the large number of small-sized write operations, which can be very costly on a typical architecture. No file read operation is performed; that is, this code represents the class of write and seek dominant I/O-intensive applications.

AST: The astrophysics application [32] performs an analysis of highly turbulent convective layers of late-type stars such as the sun. The application simulates the gravitational collapse of self-gravitating gaseous clouds due to the Jeans instability process. This is the fundamental mechanism through which intergalactic gases condense to form stars. It uses the piecewise parabolic method to solve the compressible Euler equations and a multigrid elliptic solver to compute the gravitational potential. The application uses several distributed arrays and processes them and writes them on to the disk to one common shared file.

Acknowledgments

This work was suported in part by the US National Science Foundation Grant CCR-0097998 and by the Department of Energy ASCI Adademic Strategic Alliance Program (ASAP) Level 2 under subcontract W-7405-ENG-48 from Lawerence Livermore National Laboratories.

References

[1] M. Arunachalam, A. Choudhary, and B. Rullman, "A Prefetching Prototype for the Parallel File System on the Paragon," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems, ACM Sigmetrics '95/Performance '95,* May 1995.

[2] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz, "Jovian: A Framework for Optimizing Parallel I/O," *Proc. 1994 Scalable Parallel Libraries Conf.,* 1994.

[3] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny, "A Model and Compilation Strategy for Out-of-Core Data-Parallel Programs," *Proc. Fifth ACM Symp. Principles and Practice of Parallel Programming,* July 1995.

[4] R. Bordawekar, A. Choudhary, and J. Ramanujam, "Automatic Optimization of Communication in Out-of-Core Stencil Codes," *Proc. 10th ACM Int'l Conf. Supercomputing,* pp. 366-373, May 1996.

[5] P. Brezany, T. Mueck, and E. Schikuta, "Language, Compiler and Parallel Database Support for I/O Intensive Applications," *Proc. High Performance Computing and Networking Conf.,* 1995.

[6] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," *Preprint ANL/MCS-P804-0400,* submitted to the 2000 Extreme Linux Workshop April 2000.

[7] Y. Chen, J. Plank, and K. Li, "CLIP: A Check-Pointing Tool for Message-Passing Parallel Programs," *Proc. Supercomputing '97,* 1997.

[8] A. Choudhary, R. Bordawekar, S. More, K. Sivaram, and R. Thakur, "The PASSION Runtime Library for the Intel Paragon," *Proc. Intel Supercomputer User's Group Conf.,* June 1995.

[9] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI–IO Parallel I/O Interface," *Proc. Third Workshop I/O in Parallel and Distributed Systems,* Apr. 1995.

[10] P. Crandall, R. Aydt, A. Chien, and D. Reed, "Input/Output Characteristics of Scalable Parallel Applications," *Proc. Supercomputing '95,* 1995.

[11] P. Corbett, D. Feitelson, J. Prost, G. Almasi, S. Baylor, A. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. Morgan, and A. Zlotek, "Parallel File Systems for the IBM SP Computers," *IBM Systems J.,* vol. 34, no. 2, pp. 222-248, Jan. 1995.

[12] J. Del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O via A Two-Phase Run-Time Access Strategy," *Proc. 1993 IPPS Workshop Input/Output in Parallel Computer Systems,* Apr. 1993.

[13] J.D. Rosario and A. Choudhary, "High Performance I/O for Parallel Computers: Problems and Prospects," *Computer,* pp 59-68, Mar. 1994.

[14] S. Fineberg, "Implementing the NHT-1 Application I/O Benchmark," *Proc. Int'l Parallel Processing Symp. (IPPS '93) Workshop Input/Output in Parallel Computer Systems,* pp. 37-55, 1993, Also published in *Computer Architecture News,* vol. 21, no. 5, pp. 23-30, Dec. 1993.

[15] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal, "PPFS: A High Performance Portable Parallel File System," *Proc. Int'l Conf. Supercomputing,* July 1995.

[16] M. Kandaswamy, "Design and Evaluation of Optimizations in I/O-Intensive Applications," PhD Thesis, EECS Dept., Syracuse Univ., Syracuse New York, May 1998.

[17] M. Kandaswamy, M. Kandemir, A. Choudhary, and D. Bernholdt, "Optimization and Evaluation of Hartree-Fock Application's I/O with PASSION," *Proc. SC '97 Conf., (formerly known as Supercomputing),* Nov. 1997.

[18] M. Kandemir, "A Collective I/O Scheme Based on Compiler Analysis," *Proc. Fifth Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers,* May 2000.

[19] M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar, "Compilation Techniques for Out-of-Core Parallel Computations," *Parallel Computing,* vol. 24, nos. 3-4, pp. 597-628, June 1998.

[20] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy, "A Unified Compiler Algorithm for Optimizing Locality, Parallelism and Communication in Out-of-Core Computations, *Proc. Workshop I/O in Parallel and Distributed Systems (IOPADS '97),* pp. 79-92, Nov. 1997.

[21] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving the Performance of Out-of-Core Computations," *Proc. 1997 Int'l Conf. Parallel Processing,* pp. 128-136, Aug. 1997.

[22] D. Kotz, "Expanding the Potential for Disk-Directed I/O," *Proc. 1995 IEEE Symp. Parallel and Distributed Processing,* pp. 490-495, Oct. 1995.

[23] D. Kotz and C. Ellis, "Practical Prefetching Techniques for Multiprocessor File Systems," *J. Distributed and Parallel Databases,* vol. 1, no. 1, pp. 33-51, Jan. 1993.

[24] T. Mowry, A. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," *Proc. Second Symp. Operating Systems Design and Implementations (OSDI'96),* Oct. 1996.

[25] "NWChem, A Computational Chemistry Package for Parallel Computers, Version 1.1,"High Performance Computational Chemistry Group, Pacific Northwest Laboratory (PNL), 1995.

[26] M. Paleczny, K. Kennedy, and C. Koelbel, "Compiler Support for Out-of-Core Arrays on Parallel Machines," CRPC Technical Report 94509-S, Rice Univ., Houston Tex., Dec. 1994.

[27] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera, "Scalable Performance Analysis: the Pablo Performance Analysis Environment," *Proc. Scalable Parallel Libraries Conf.,* pp. 104-113, 1993.

[28] B. Rullman Paragon Parallel File System, External Product Specification, Intel Supercomputer Systems Division. 1996.

[29] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," *Proc. Supercomputing '95,* Dec. 1995.

[30] E. Smirni, C. Elford, A. Laevry, D. Reed, and A. Chien, "Algorithmic Influences on I/O Access Patterns and Parallel File System Performance," Technical Report, Pablo Group, Univ. of Illinois at Urbana-Champaign, 1996.

[31] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming,* vol. 5, no. 4, pp. 301-317, Winter 1996.

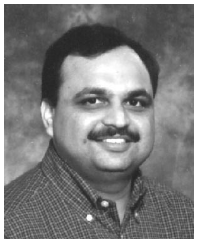
[32] R. Thakur, W. Gropp, and E. Lusk, "An Experimental Evaluation of the Parallel I/O Systems of the IBM, SP, and Intel Paragon Using a Production Application, *Proc. Third Int'l Conf. Austrian Center for Parallel Computation (ACPC),* pp. 24-35, Sept. 1996.

[33] R. Thakur, W. Gropp, and E. Lusk, "A Case for Using MPI's Derived Data Types to Improve I/O Performance, Preprint, ANL/MCS-P717-0598, Math. and Computer Science Division, Argonne Nat'l Laboratory, May 1998.

[34] S. Toledo and F. Gustavson, "The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations, *Proc. Fourth Ann. Workshop I/O in Parallel and Distributed Systems,* May 1996.

Meenakshi A. Kandaswamy received the BE (Honors) degree in computer science and engineering from Regional Engineering College, Trichy, India in 1989. She received the MS and PhD degrees from the Department of Electrical Engineering and Computer Science at Syracuse University in 1998 and 1995, respectively. She currently works in the Enterprise Architecture Labs, Intel Corporation as a senior software engineer. Her research interests include high-performance I/O, parallel applications and benchmarks, multiprocessor file systems, performance modeling, and simulation.

Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD degree from Syracuse University, Syracuse, New York in electrical engineering and computer science, in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.



Alok Choudhary received the BE (Hons.) degree from Birla Institute of Technology and Science, Pilani, India in 1982, the MS degree from the University of Massachusetts, Amherst, in 1986 and the PhD degree from the University of Illinois, Urbana-Champaign, in electrical and computer engineering. He is a professor of electrical and computer engineering at Northwestern University. He received the US National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award. His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing and scientific computing. He is a senior member of the IEEE and a member of the ACM. He also serves in the High-Performance Fortran Forum, a forum of Academia, Industry and Government Labs working on standardizing programming languages for portable programming on parallel computers.

David Bernholdt received the PhD in chemistry from University of Florida in 1993. He is currently a research assistant professor in the Department of Chemistry at Syracuse University and a research scientist at the Northeast Parallel Architectures Center. He is also affiliated with the Pacific Northwest National Laboratory. His main research interests are methods for large-scale correlated electronic structure calculations, efficient use of MPPs in quantum chemistry, and general problems in computational science. He is a member of American Chemical Society, American Physical Society, and Association for Computing Machinery.