Performance Implications of Architectural and Software Techniques on I/O-Intensive Applications*

Meenakshi A. Kandaswamy
EECS Dept.
Syracuse University
Syracuse, NY, 13244
meena@ece.nwu.edu

Mahmut Kandemir
EECS Dept.
Syracuse University
Syracuse, NY, 13244
mtk@ece.nwu.edu

Alok Choudhary
ECE Dept.
Northwestern University
Evanston, I L, 60208-3118
choudhar@ece.nwu.edu

David E. Bernholdt
NPAC
Syracuse University
Syracuse, NY, 13244
bernhold@npac.syr.edu

Abstract

Many large scale applications, have significant I/O requirements as well as computational and memory requirements. Unfortunately, limited number of I/O nodes provided by the contemporary 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 architectural scalability and evaluate the effect of them in five I/O intensive applications 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 subsystem 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 architectural and software solutions. Our results reveal that the different applications can benefit from different optimizations. For example, we found that some applications benefit from file layout optimizations whereas some others benefit from 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 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 increase in I/O resources.

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 software, are much behind compared to those in processors and interconnection networks; resulting in poor performance for I/O-intensive applications. 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 sometime so severe that when a certain number of compute nodes (processors) is reached, the execution time increases. Although such a situation can sometime occur with computationally intensive applications as well, the main problem with the programs in our application suite is the limited number of I/O nodes and unoptimized I/O performed by the programs. Therefore, beyond a certain point increasing the number of compute nodes has a negative impact in I/O as well as execution times. 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. For example, we show that with 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 prefetching. We also show that beyond a certain level, imbalance in the architecture results in performance degradation even when using optimized software.

We show that several optimizations are very effective on the I/O performance. For instance, we found that performance of some applications can be substantially improved by changing the file layout of the out-of-core arrays involved. For some other application, we found collective I/O to be very useful. For yet another application, we found prefetching to be effective. While many of these optimizations are not new, we show here that the different applications can benefit from different optimizations.

In this paper we investigate the software optimizations and resource scalability issues in detail. In particular, we address the following questions:

- How much improvement can be obtained by optimizing the I/O software; and what kind of optimization techniques can be used?
- How much improvement can be obtained by increasing I/O resources (e.g. number of I/O nodes) thereby making the architecture more balanced?
- How do the hardware and software improvements compare to each other?
- Do different I/O intensive applications have different improvements when the I/O resources are increased and/or the software is improved?

The rest of this paper is organized as follows. In Section 2 we describe the applications in our experimental suite. In Section 3 we discuss the Intel Paragon and IBM SP-2 machines' salient I/O features. In Section 4 we present the experimental data obtained from

*This work was supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143, NSF ASC-9707074, Sandia National Labs Contract AV-6193, and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Defense Advanced Research Projects Agency(DARPA) administered by US Army at Fort Huachuca. Dr. D.E.Bernholdt was supported by the Alex G. Nason Fellowship at Syracuse University.

original (unoptimized) programs, discuss the individual optimizations and explain how they improve the I/O as well as the overall performance of the applications. In Section 5 we discuss the related work and present the conclusions of the paper.

2 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 computational chemistry domain and are very large applications [8]. AST is an astrophysics code; BTIO is disk-based version of a flow-solver program from NAS benchmarks [4]; and FFT is a 2-D out-of-core FFT program. More details on the applications can be found in [6]. Below, we give a summary of the various applications.

SCF 1.1 The Hartree-Fock 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 and in the process about $\mathcal{N}^4$ integrals must be evaluated, where $\mathcal{N}$ is the dimension of the basis set of the input. The values of these integrals remain constant throughout the iterations and evaluating each integral is a non-trivial computation, involving 300–500 floating point operations, on average. To lend perspective to these figures, it is worth mentioning that the design goal for SCF 1.1's SCF module is calculations of 1,000 atoms with basis sets of 10,000 functions, which would involve as many as 10^{16} integrals. In the HF algorithm, the 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 the write phase, each node writes a private file of the integrals it evaluated during first construction of the Fock matrix. The read phase, on the other hand, consists of several iterations. In each iteration, each processor reads its private file in its entirety. SCF 1.1 application code consists of about 16,500 lines of code and about 225,000 lines when supporting libraries are included.

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 re-computed 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. The semi-direct SCF 3.0 approach is a compromise between the two, where limits may be specified on the size of disk files, and any integrals which are not stored on disk are recomputed. Some attempt is 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. Finally, to help account for the difference in load balance between the evaluation of integrals (on the first iteration) and reading them from disk (subsequent iterations), the sizes of the integral files are balanced (currently to within 10% or 1 MB, whichever is larger) after writing is complete. In addition to change in the SCF module itself, there have also been changes to the I/O part of the application from the 1.1 to 3.0 releases.

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 2-D out-of-core FFT on the Intel Paragon. The 2-D out-of-core FFT consists of three steps: (1) 1-D out-of-core FFT, (2) 2-D out-of-core transpose, and (3) 1-D out-of-core FFT. The 1-D FFT steps consist of reading data from the two-dimensional out-of-core array and applying 1-D FFT on each of the columns. In order to perform 1-D 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 and individual processor speeds. After this, 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. The innermost loop of the transpose routine uses two disk resident files one of them is transposed into the other.

BTIO BTIO application simulates the I/O required by a pseudo-time-stepping flow solver. It is a disk-based version of a program from NAS parallel benchmark suite [4]. The main operation in the code is periodic writes performed by all processors to a multi-dimensional array stored in a file. Note that periodic write operations are used by such applications for check-pointing and/or off-line visualization and analysis of data. The code contains a lot of seek operations which in turn causes the performance to be poor. It represents the class of write dominant I/O intensive applications.

AST The astrophysics application [11] performs a study 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 inter-galactic gases condense to form stars. It uses the piecewise parabolic method to solve the compressible Euler equations and a multi-grid 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. The reasons that this application performs I/O is threefold, namely, check-pointing [1], data analysis, and visualization. All the three cases make the I/O mainly write-intensive, except when there is a restart of the application from previously check-pointed data, it becomes read-intensive. The application uses several data arrays that are processed during the course of the application and are stored on disk in one file in column-major order for data analysis and check-pointing purposes. For visualization purposes several shared files are created by the application. The original application performs I/O accesses in small non-contiguous chunks.

3 Platforms

In this Section we summarize some of the salient characteristics of our platforms, Intel Paragon and IBM SP-2, emphasizing the I/O capabilities. The reason that we use these platforms is the fact that both machines represent a class of distributed-memory message-passing architectures that present the user with scalable I/O architectures. Also the parallel file systems (PFS [9] in Paragon and PIOFS [3] in SP-2) on these machines are versatile and enable the users to code a variety of optimization techniques. The large scale applications in our experimental suite are available to us at these machines. Finally, these architectures are widely used by the scientists, so represent natural targets in our experimental study.

Table 1: Applications in our experimental suite and their important characteristics.

Application	*Source*	*Lines*	*Description*	*Platform*	*Type of I/O*
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	6713	simulates the I/O required by a flow solver	SP-2	periodic writes of arrays
AST	Univ. of Chicago	17000	simulates gravitational collapses of clouds	Paragon	writes arrays for check-pointing

Table 2: I/O Summary of the original version of SCF 1.1 for LARGE input : 4 processors [Total I/O time is 4.4 Hours].

Oper	Oper Count	I/O Time (Sec)	Vol (GB)	% of I/O time	% of exec time
Open	19	1.97		0.00	0.00
Read	566,315	60,284.31	37	95.56	51.66
Seek	994	8.01		0.01	0.01
Write	40,331	2,792.11	2.5	4.43	2.39
Flush	49	0.25		0.00	0.00
Close	14	0.46		0.00	0.00
All I/O	607,722	63,087.11	39.5	100.00	54.06

Table 3: I/O Summary of PASSION version of SCF 1.1 for LARGE input : 4 processors [Total I/O time is 2.5 Hours].

Oper	Oper Count	I/O Time (Sec)	Vol (GB)	% of I/O time	% of exec time
Open	19	0.65		0.00	0.00
Read	566,330	33,805.21	37	95.38	37.73
Seek	604,342	256.56		0.72	0.29
Write	40,336	1,380.79	2.5	3.90	1.54
Flush	49	0.15		0.00	0.00
Close	14	0.37		0.00	0.00
All I/O	1,211,090	35,443.72	39.5	100.00	39.56

Intel Paragon The Paragon that we use for FFT experiments consists of 56 compute nodes, 3 service nodes, and 1 HIPPI node. The compute nodes are arranged in a two-dimensional mesh comprised of 14 rows and 4 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 use a Paragon machine with 512 compute nodes, and 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, stripes the user files across the available I/O nodes in a round-robin fashion. The default stripe unit size, which is 64 KB, is used in all of our experiments, except SCF 1.1 where we make experiments with different stripe unit sizes.

IBM SP-2 For BTIO application, we use an SP-2 with 80 nodes. All nodes that are used in the experiments were RS/6000 Model 390 nodes with at least 256 MB memory. The parallel file system, PIOFS [3], 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) is 32 KB.

4 Optimizations and Performance Comparison

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

4.1 Experimental Methodology

Our experimental methodology is as follows: For each application, we applied several software optimizations but due to lack of space we present only the results of most effective optimizations. Based on the platform on which we ran the application on 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 2 and 4 I/O node subsystems, 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 SCF 1.1: effect of efficient interface and prefetching

We investigated the performance of SCF 1.1 for small as well as large number of compute nodes separately. The results for the small number of compute nodes are summarized in three bar charts given in Figure 1. In fact, that figure presents a summary of an incremental evaluation of the I/O optimizations that we performed for this application for small processor sizes. We consider three representative inputs which we call SMALL, MEDIUM and LARGE. An important observation about this application is that the programmers have reasonably optimized the I/O related parts of the programs. Instead of directly using the access pattern imposed by the application, they first pack the data to be written onto disk in larger chunks and then write 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) original version [8] with Fortran I/O. This version was obtained from PNL; (2) an optimized version which uses PASSION [10] (from Northwestern University) I/O calls; (3) an optimized version which uses PASSION prefetch calls. We represent each optimization combination in Figure 1 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 I/O calls, F - optimized version with

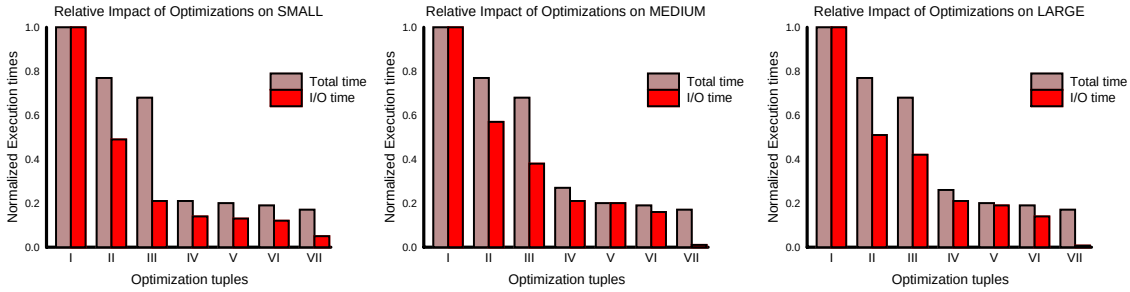

Figure 1: Impact of optimizations on different input sizes for `SCF 1.1` on Intel Paragon. The number of basis functions ($\mathcal{N}$ – problem size) for `SMALL`, `MEDIUM` and `LARGE` are 108, 140 and 285 respectively. The configuration tuples are : I - (O,4,64,64,12); II - (P,4,64,64,12); III - (F,4,64,64,12); IV - (F,32,64,64,12); V - (F,32,256,64,12); VI - (F,32,256,128,12); VII - (F,32,256,128,16). [For small number of processors, the application-related factors are more effective than system-related factors].

(a) (b)

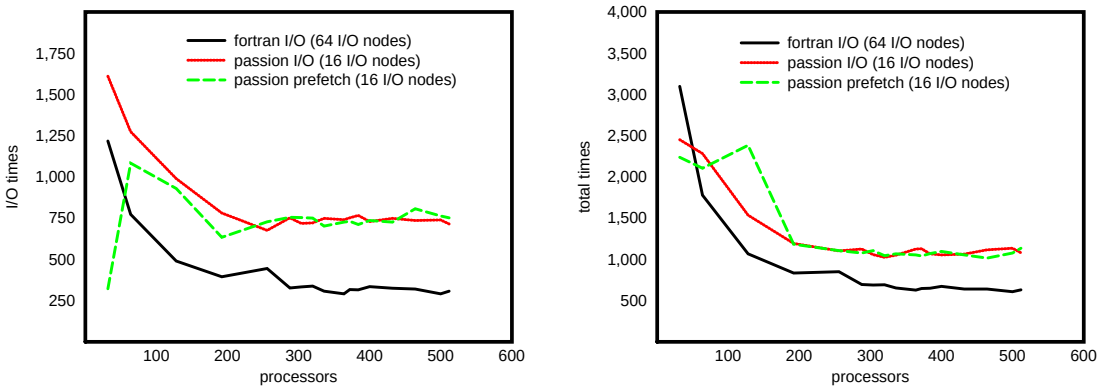

Figure 2: Performance summary for `SCF 1.1` for `LARGE` input on Intel Paragon [Up to 64 compute nodes optimized versions perform well; beyond 64 compute nodes the unoptimized version with larger number of I/O nodes performs better].

PASSION prefetch calls); P is the number of processors; M is the available memory to the application (in KB); Su is the stripe unit size (in KB); and Sf is the stripe factor (number of I/O nodes in this case). The tuple (O,4,64,64,12) corresponds to the 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 results shown in Figure 1 summarize our evaluations. 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 processors used are small in number the application-related factors have a much higher impact on the execution and I/O times of the application than the system-related factors. Tables 2 and 3 show a detailed quantitative breakdown of the various I/O operations performed by the application on four processors for the Original and PASSION version using the Pablo I/O tracing library [2]. We clearly see that the `SCF 1.1` is extremely read intensive and by using a different interface to the file system (Table 3) we obtain better read and write times thereby reducing the total time. We must mention that the tracing library we used obtained results from the application level and we used it mainly in runs using small number of processors. Also for the larger number of processors cases, we were more interested in studying the scalability of the application.

The results for the larger processor case are presented in Figures 2 and 3. We notice from Figure 2(b) 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 outperforms the optimized versions with 16 I/O nodes. From Figure 3, we infer that as the number of processors or compute nodes used increases so does the contention at the I/O nodes. We observe that the increase in I/O nodes translates into reduced I/O contention and results in improved total execution times, especially when we use larger number of compute nodes.

4.3 `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 an efficient interface and prefetching quite useful. Since we discussed those issues with `SCF 1.1`, due to lack of space, here we do not elaborate on them. Instead, we focus on another optimization technique, which we call `balanced I/O` which is made possible by the application programmers. 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 re-computed when necessary. We found that the ratio can make a critical difference in

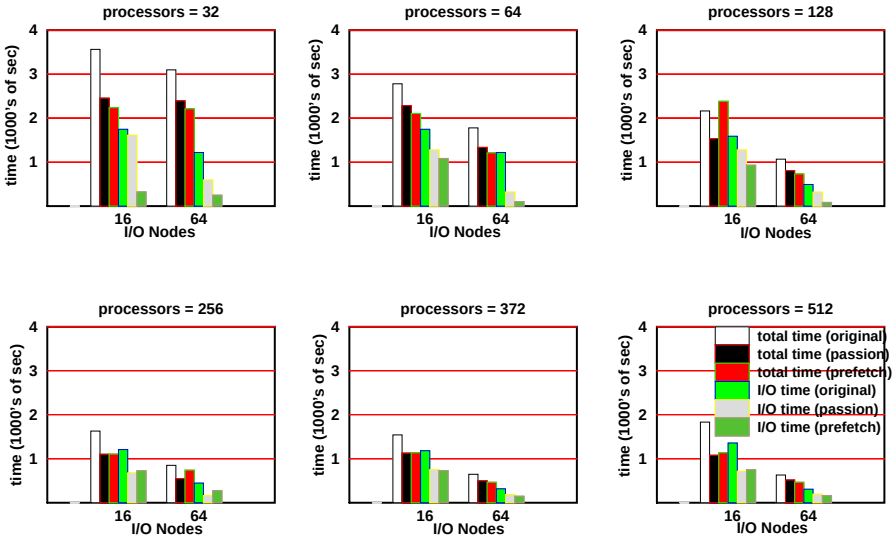

Figure 3: Effect of increasing the number of I/O nodes on `SCF 1.1` on Intel Paragon.

the overall performance of the application. The problem, however, is that the best ratio is dependent on the input size. Therefore, it is almost impossible to make an educated guess beforehand.

Figure 4 shows the overall performance of the application on 16 and 64 I/O nodes respectively 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. The reason 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.

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 (Intel Paragon) on which these experiments were conducted 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% or so, (for the 64 I/O nodes case), we found that even 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 integrals, then using larger number of processors to reduce the load of re-computation per processor is beneficial in decreasing the total execution times. From the results, we can conclude that for `SCF 3.0`, the amount of disk space available for caching is more important followed by the number of processors. Only in the event of the less disk space, increasing the number of processors would be desirable.

4.4 FFT: effect of layout optimization

Figure 5(a) shows the I/O times (in seconds) for three different versions of `FFT` application 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. We note that this trend in the I/O time almost identically reflects on the total execution time (see Figure 5(b)). The reason for this is that the I/O time for this application constitutes 90%-95% of the execution time and therefore is the dominant factor in the overall behavior of the application.

The most costly operation in the 2-D out-of-core FFT is a 2-D out-of-core local transpose performed by each processor. 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, transposing it in the local memory, and writing it in the other file. Since both files are column-major, optimizing the block dimension for one array has a negative impact on I/O performance of the other array, resulting in poor I/O performance observed in Figure 5. 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 Figure 5 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 very effective, and the optimized version outperforms the unoptimized version which uses more number of I/O nodes.

An important point about those types of layout optimizations is that they can sometimes be detected by parallelizing compilers by using suitable linear algebraic techniques. For example, reference [7] shows how the data layout optimizations can be automatized within a parallelizing compiler. 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 decides which layout to choose for each disk-resident array.

4.5 BTIO: effect of collective I/O

As mentioned earlier, the base version of the application uses MPI-2 I/O as a UNIX style interface and contains a lot of seek opera-

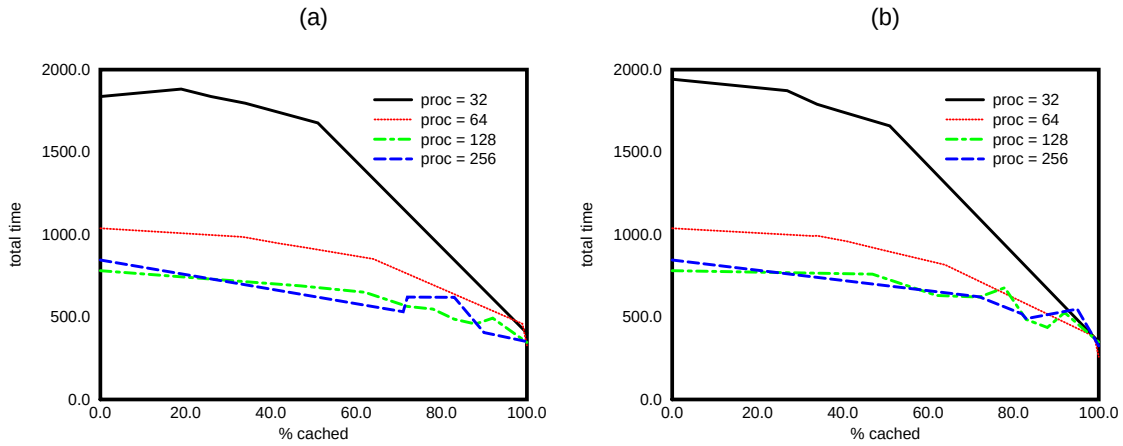

Figure 4: Performance of `SCF 3.0` for different percentages of cached integrals for `MEDIUM` input on the Intel Paragon (a) with 16 I/O nodes and (b) with 64 I/O nodes [Note that for the full recompute version (0% cached), increasing the number of processors is very effective whereas for the full disk version (100% cached) increasing the number of processors does not make a significant difference].

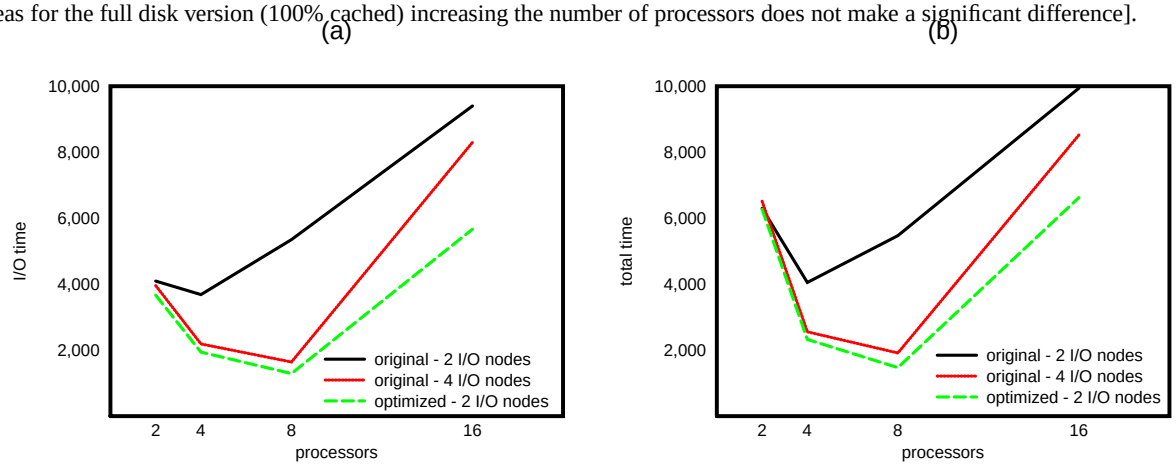

Figure 5: Performance of `FFT` on Intel Paragon [1.5GB total I/O amount].

tions. The I/O performance of `BTIO` for the input Class A is shown in Figure 6(a) and the overall performance is shown in Figure 6(b), respectively. In contrast to `FFT`, `BTIO` is not as I/O dominant. From Figure 6(a), 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 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 [10] discuss 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 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. 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, 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 [10].

The optimized version of `BTIO` uses the two-phase I/O. The solution vector is completely described by using MPI data types. Figure 6(a) show 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 Figure 6(b). As an example, with 36 and 64 processors, there is 46% and 49% reduction in the overall execution time respectively. Similar trend is observed in the Class B input and other inputs as well.

We also measure the I/O bandwidth of the original and the opti-

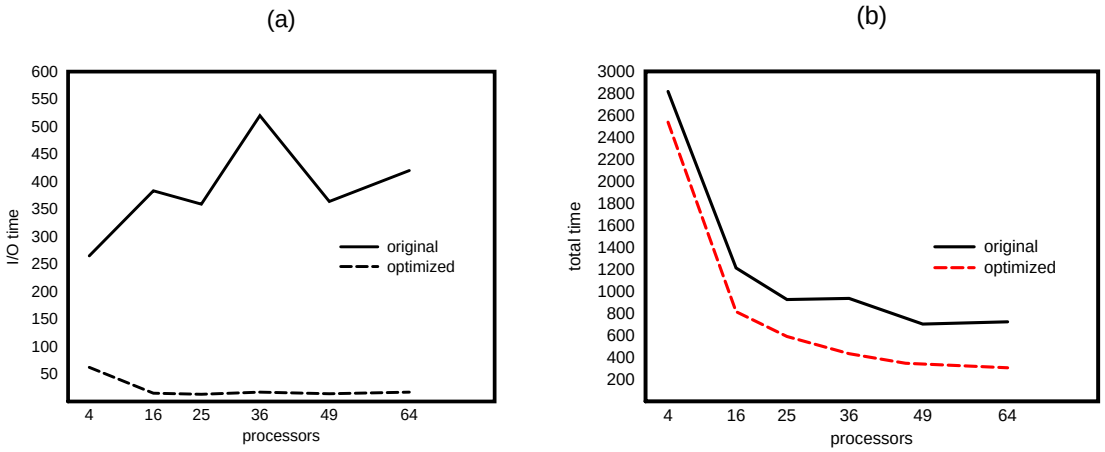

Figure 6: Performance of BTIO on IBM SP-2 for Class A input. (a) I/O time and (b) Total time. (408.9 MB total I/O amount)

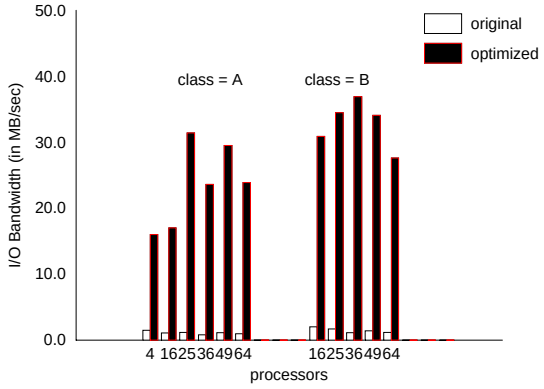

Figure 7: I/O bandwidths of the original and optimized versions of BTIO on IBM SP-2 for Class A and Class B inputs

mized versions. The results are given in Figure 7 as bar charts. The I/O bandwidth of the original program is between 0.97 MB/sec and 1.5 MB/sec while the I/O bandwidth of the optimized version is between 6.6 MB/sec and 31.4 MB/sec. In summary, BTIO is an example of a class of I/O intensive programs whose the I/O performance can be improved by software optimizations, 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.

4.6 AST: effect of collective I/O

The results of astrophysics application for a reasonably large input array size of 2K $\times$ 2K elements is presented in the Table 4 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 one 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. And data are also processed and written out for the purposes of visualization. We compare two different implementations of the code: (1) I/O done using the Chameleon library and (2) I/O done using a run-time system library performing two-phase I/O (which is a form of collective I/O: please refer to BTIO section of the paper for further details on 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 non-contiguous chunks and also has a bottleneck of all I/O performed by a single node and this adds to the I/O time. Whereas the two-phase I/O approach 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 of the I/O nodes as shown in the Table 4.

Table 4: Execution times for AST. [The I/O amount is 2.2 GBytes].

Num of procs	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

Table 5: Applications and effective optimization techniques. A tick-mark is entered in the table on the effective optimization.

Application	*Optimization techniques*				
	collective I/O	*file layout*	*efficient interface*	*prefetching*	*balanced I/O*
`SCF 1.1`			√	√	
`SCF 3.0`			√	√	√
`FFT`		√			
`BTIO`	√				
`AST`	√				

5 Related Work and Conclusions

Several different works have been published in the study of I/O intensive parallel applications. Three I/O-intensive applications from the Scalable I/O Initiative Application Suite are studied in [2]. Thakur et. al. [11] evaluate the I/O performance of the IBM SP and the Intel Paragon using an astrophysics application, and concluded that IBM SP2 is faster with read operations and Paragon for writes. In [5], Foster et. al, optimize `BTIO` on small number of processors using remote I/O. In the area of check-pointing, Chen, Plank and Li in [1], develop a library to implement rollback recovery of parallel applications.

To summarize, from our experiments we observe the following: The I/O intensive applications in our experimental suite deliver poor performance mostly due to I/O bottleneck. This bottleneck originates from both hardware and software. From the hardware, the limited number of I/O nodes in Paragon and 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 re-computes the integrals; for large number of compute nodes, they tend to use the re-compute version, as the I/O version performs very poorly. An obvious solution to this problem is to increase the number of I/O nodes. In this paper, we experimentally evaluated the impact of the increase in number of I/O nodes on the I/O as well as the overall performance of the applications.

From the software point of view, the I/O software is not easy to use and is not portable at all. 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 to the programs in our experimental suite. The results are summarized in Table 5. We have two main observations to make: First, different I/O intensive applications are amenable to different types of optimizations. The second observation is that for up to a number of compute nodes, the software techniques are competitive and sometimes more successful than merely increasing the number of I/O nodes. Beyond a certain point however, we still need to increase the number of I/O nodes for further improvement.

An important question from the software point of view is how to select a proper sequence 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, we infer the following: First, the I/O access pattern of the individual compute nodes should be improved. That is, instead of doing I/O whenever they need, each compute node should consider the I/O requirements of the other nodes as well. Here the techniques like collective I/O and buffering I/O requests are proven to be useful. For example, for `BTIO` and `AST`, the individual nodes' I/O pattern are improved by collective I/O substantially. For `SCF 1.1` and `SCF 3.0`, the individual nodes first buffer the data that they are going to write into their private files, minimizing the number of I/O accesses substantially. After a good access pattern for the individual compute nodes are obtained, then the performance can be further improved by determining the most suitable file layouts for the disk resident data. As an example, assigning different file layouts to different disk resident arrays in the `FFT` improves the performance substantially. In addition, the remaining I/O time can sometimes be hidden by prefetching one or more data chunks in advance. Both `SCF 1.1` and `SCF 3.0` benefit from prefetching substantially.

6 Acknowledgments

A modified version of `SCF 1.1` and `SCF 3.0`, as developed and distributed by Pacific Northwest National Laboratory, P. O. Box 999, Richland, Washington 99352, USA, and funded by the U. S. Department of Energy, was used to obtain some of these results. We also thank Evgenia Smirni for her help in using the Pablo instrumentation library. We would like to thank Rajeev Thakur for his assistance in running `BTIO`. We are grateful to Caltech and Argonne National Laboratory for allowing us to use their parallel machines for conducting our experiments.

References

[1] Y. Chen, J. Plank, and K. Li CLIP: A Check-pointing Tool for Message-Passing Parallel Programs (1997) In *Proceedings of SuperComputing'97*, San Jose, California.

[2] P. E. Crandall, R. A. Aydt, A. A. Chien and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proc. Supercomputing'95*.

[3] 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 Journal*, 34(2):222-248, January 1995

[4] S. Fineberg. Implementing the NHT-1 application I/O benchmark. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 37-55, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 23-30.

[5] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: fast Access to Distant Storage. In *Proc. Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 14–25, San Jose, CA, Nov 1997.

[6] Meenakshi A. Kandaswamy. Design and Evaluation of Optimizations in I/O-Intensive Applications, *Ph.D. Thesis, EECS Department*, Syracuse University, May 1998.

[7] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the Performance of Out-of-Core Computations. In *Proc. 1997 International Conference on Parallel Processing*, pages 128–136, Bloomingdale, IL, August 1997.

[8] NWChem, a computational chemistry package for parallel computers, version 1.1, 1995. *High Performance Computational Chemistry Group*, Pacific Northwest Laboratory, Richland, Washington 99352, USA.

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

[10] R. Thakur, A. Choudhary, R. Bordawekar, S. More and S. Kuditipudi, PASSION: optimized I/O for parallel applications, In *IEEE Computer*, IEEE Computer Society, June 1996.

[11] 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. In *Proc. the Third International Conference of the Austrian Center for Parallel Computation (ACPC)*, pages 24-35, September 1996.