Evaluation of Collective I/O Implementations on Parallel Architectures

Phillip M. Dickens* Rajeev Thakur†

Abstract

In this paper, we evaluate the impact on performance of various implementation techniques for collective I/O operations, and do so across four important parallel architectires. We show that a naive implementation of collective I/O does *not* result in significant performance gains for *any* of the architectures, but that an optimized implementation does provide excellent performance across *all* of the platforms under study. Furthermore, we demonstrate that there exists a single implementation strategy that provides the best performance for all four computational platforms. Next, we evaluate implementation techniques for thread-based collective I/O operations. We show that that the most obvious implementation technique, which is to spawn a thread to execute the *whole* collective I/O operation in the background, frequently provides the *worst* performance, often performing much worse than just executing the collective I/O routine entirely in the foreground. To improve performance, we explore an alternate approach where *part* of the collective I/O operation is performed in the background, and part is performed in the foreground. We demonstrate that this implementation technique can provide significant performance gains, offering up to a 50% improvement over implementations that do not attempt to overlap collective I/O and computation.

*Department of Computer Science, Illinois Institute of Technology
†Mathematics and Computer Science Division, Argonne National Laboratory

1 Introduction

Parallel computers are increasingly being used to solve large, I/O-intensive applications in several different disciplines. However, in many such applications the I/O subsystem performs poorly and represents a significant obstacle to achieving good performance. The problem is often not with the hardware; many parallel I/O subsystems offer excellent performance. Rather, the problem arises from other factors, primarily the I/O patterns exhibited by many parallel scientific applications [9, 18, 2, 3, 22, 24, 25, 28]. In particular, each processor tends to make a large number of small I/O requests, incurring the high cost of I/O on each such request. One reason for this access pattern is that parallel scientific codes frequently involve large arrays distributed across the processor's local memory. After a processor performs some computation on its local array, it will often need to read/write its portion of the array from a common file. If the processor's local portion of the array is not stored in a logically contiguous fashion, the processor will be forced to make a series of disjointed I/O requests to complete the operation. While each processor may need to perform several, disjointed requests, it is often the case that *in the aggregate* the whole array is being written to or read from the file. The application can use this knowledge to significantly improve its I/O performance.

The technique of *collective I/O* has been developed to better utilize the parallel I/O subsystem [10, 26, 27, 4, 17, 23, 5, 8]. In this approach, the processors exchange information about their individual I/O requests to develop a picture of the *aggregate* I/O request. Based on this global knowledge, I/O requests are combined and submitted in their proper order, making a much more efficient use of the I/O subsystem.

There are three approaches to collective I/O: two-phase I/O [10, 26, 27], disk-directed I/O [17, 19], and server-directed I/O [7, 23] The primary distinction between these approaches is the level at which the optimal I/O strategy is derived and carried out. In disk-directed I/O, the collective I/O request is sent to the disk controllers which collectively determine and carry out the optimal I/O strategy. In server-directed I/O, the I/O servers collectively determine and carry out the optimal strategy, and in two-phase I/O, the application processors collectively determine and carry out the optimized approach. In this paper, we deal only with two-phase collective I/O.

To help understand how two-phase I/O can improve performance, consider a collective read operation. If the data is distributed across processors in a way that conforms to the way it is stored on disk, each processor can read its local array in one large I/O request. This distribution is termed the *conforming* distribution and represents the optimal I/O performance. Assume the array is *not* distributed across the processors in a conforming manner. The processors can still perform the read operation *assuming* the conforming distribution, and then use interprocessor communication to redistribute the data to the desired distribution. Since interprocessor communication is, in general, orders of magnitude faster than I/O operations, it is possible to obtain performance that approaches that of the conforming distribution. Our research shows up to a 1000% improvement in performance when using a collective I/O approach rather than a non-collective approach.

There have been several studies in the literature that document excellent performance using two-phase I/O [4, 12, 27]. However, two important issues have not yet been addressed. The first issue has to do with the sensitivity of the two-phase I/O algorithm to various implementation approaches. In a previous study [12], it was shown that for at least one architecture, the Intel Paragon, a "naive" implementation of two-phase I/O provided a bandwidth of 78 Megabytes per second (MB/sec), but that an optimized two-phase I/O implementation provided a bandwidth of 440 MB/sec. In this paper, we extend these results to three other computational platforms and demonstrate that there exists one implementation approach that provides the best performance across all four architectures.

The next issue has to do with mechanisms by which the impact of large I/O requirements can be *further* reduced. It can be argued that basic collective I/O techniques are maturing, and that other approaches must be developed to further reduce the impact of I/O requirements on large, scientific computations. A very promising approach for obtaining high performance is to use threads to execute the collective I/O *in the background* while continuing with other computation in the foreground. This approach does not reduce the cost of collective I/O per se, but can significantly reduce the *impact* of collective I/O by overlapping its execution with other computation.

We study the potential benefits of thread-based collective I/O by taking the best collective I/O implementation found in this study, and attempting to overlap its execution with computation occurring in the foreground. We show that the most natural implementation choice, to simply spawn off a thread to perform the whole collective I/O routine in the background, is quite often the *worst* implementation option. We demonstrate that this approach improves performance on only one architecture and produces significantly *worse* performance on two of the four architectures. To overcome this poor performance, we developed a technique where part, but not all, of the collective I/O is performed in the background. We demonstrate that this modified technique, in general, is a much better implementation option than either performing the whole collective I/O operation in the foreground or performing the whole operation in the background.

There are three important contributions of this research. First, it serves as a guide for implementation techniques for two-phase I/O on a wide range of parallel architectures. Secondly, it provides significant insight into the development of implementation techniques for the *split-collective* parallel I/O operations defined in MPI-2 [21]. These operations provide an implementation the opportunity to perform collective I/O in the background, but there are currently no implementations that do so. The wide-spread use of MPI, and the importance of developing *portable* parallel I/O operations, make this a very important and timely application of this research. Thirdly, this research sheds new light on the use of threads for collective I/O and disproves the commonly held belief that simply spawning a thread to perform work in the background leads to significant performance gains.

The rest of the paper is organized as follows. In Section 2, we discuss the technique of two-phase collective I/O in more detail. In Section 3, we outline various implementation techniques for two-phase I/O, and we study their performance in Section 4. In Section 5, we discuss the split-collective parallel I/O operations defined in MPI-2, and we investigate approaches to their implementation in Section 6. In Section 7 we discuss the implications of this research for real-world applications and benchmarks. In section 8 we discuss related

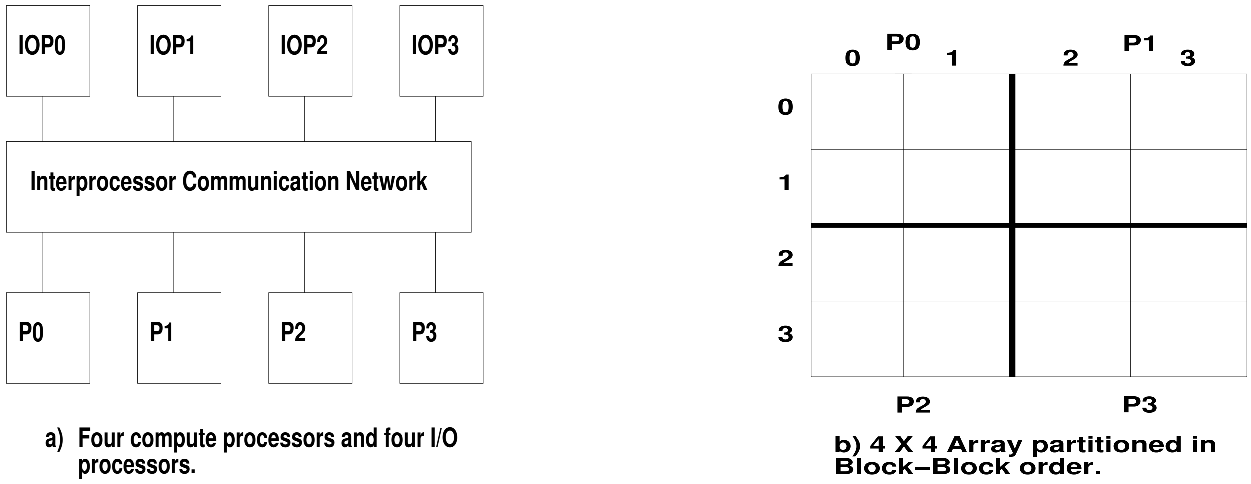

Figure 1: Example system with four compute processors, four I/O processors and a 4 X 4 array.

work, and we provide our conclusions and future work in Section 9.

2 Two-Phase I/O

In this section, we provide a simple example to demonstrate the two-phase I/O technique and show how it can significantly improve performance. The architecture chosen for this discussion is based on the two distributed memory architectures studied, the Intel Paragon and the IBM SP2 (we discuss all of the architectures in the next section).

Most parallel architectures provide some form of hardware support for parallel I/O. In the case of the two distributed memory architectures studied, this support consists of a set of I/O processors, each of which controls some number of disks. The data in a file is divided into a set of *striping units*, each of which represents a logically contiguous portion of the file data. These striping units are (generally) distributed among the disks in a round robin fashion and are contiguous on a disk. Performance is enhanced because concurrent requests to *different* positions within the same file can be serviced in parallel by the I/O subsystem. To illustrate how a two-phase algorithm can exploit this hardware consider the following simple example.

Assume an SPMD computation where each processor computes over a different region of a two dimensional array. Further, assume there are four compute nodes, four I/O processors and that a 4 X 4 array of integers is distributed across the local memories of the processors (see Figure 1). The array is stored on the disks in row-major order with a striping unit equal to one row of the array. The array is distributed among the processors in a block-block distribution as shown in Figure 1.

Assume the processors are all ready to write their data to disk. In the two-phase I/O approach, the processors first communicate with each other to determine the aggregate I/O request and the optimal strategy for performing this request. In this example, the whole array is being written to disk. Since the conforming distribution is row major, the optimal I/O strategy is for each processor to write out one full row of the array to disk in one request. Assume the derived strategy is for P0 to write row 0 of the array to disk, for processor P1 to write row 1 to disk and so forth. Then the processors must use interprocessor communication to permute the data to match the conforming distribution. In particular, processor 0 must send data elements (1,0) and (1,1) to P1, P1 must send to P0 data elements (0,2) and (0,3), P2 must send to P3 elements (3,0) and (3,1), and P3 must send to P2 data elements (2,2) and (2,3). After this exchange is completed, each processor is able to write out its row in one I/O operation.

Contrast this with the I/O activity required in the non-collective I/O implementation. Processor P0 would have to make two separate I/O calls, one to write array elements (0,0) and (0,1), and another to write elements (1,0) and (1,1). Similarly, each of the other processors would have to make two I/O requests to write their data to disk resulting in eight small I/O operations. Performance is degraded even more since processors P0 and P1 contend for I/O processors IOP0 and IOP1, and processors P2 and P3 contend for I/O processors IOP2 and IOP3.

As can be seen, using a collective I/O strategy can significantly reduce the (very expensive) calls to the I/O subsystem. The next task is to determine the best collective I/O implementation for each architecture and to determine if there exists a single implementation that provides the best performance across *all* of the architectures under study. We begin with a more detailed description of the four parallel machines used in this study.

3 Experimental Design

3.1 Computational Platforms

Of the four architectures studied, two of the machines, the IBM SP2 and the Intel Paragon, are distributed memory architectures. The other two, the SGI Origin 2000 and the HP Exemplar, are distributed shared memory (DSM) architectures. The IBM SP2 used in these experiments is located at Argonne National Laboratory, and consists of 80 compute nodes and 4 I/O processors. Each I/O processor controls 4 SSA disks, each with a 9 Gigabyte capacity. The Intel Paragon used is located at the California Institute of Technology, and is configured with 381 compute nodes and 64 I/O processors. Each I/O processor controls a 4-Gigabyte Seagate drive.

The SGI Origin2000 used in these experiments, is also housed at Argonne National Laboratory, and was configured (at the time of these experiments) with ten fiber channel connections to an array of 90 9-Gigabyte disks. The Origin had 128 processors that were configured as three separate machines, the largest of which (and the one used in this study) contained 96 compute processors. The fiber channels are shared by all of the processors, and the I/O traffic shares the same routers and Cray links as does the memory traffic. However, the I/O traffic does not interact directly with any of the compute processors.

The HP Exemplar, located at the California Institute of Technology, is configured with 256 compute processors grouped in clusters termed *hypernodes*. Each hypernode consists of

16 compute processors and 4 Gigabytes of shared random-access memory connected through a non-blocking $8X8$ cross-bar switch. Each hypernode has its own local file system, and a file system cannot span more than one hypernode. In general, the file systems consist of eight disks with a total of 35 Gigabytes of storage although there is some variation from hypernode to hypernode. Since a file system cannot span more than one hypernode, parallel access by more than 16 processors must all flow through the same hypernode on which the file system is located. As with the SGI Origin2000 however, the I/O traffic flows directly into the file system buffer and and does not interact with the host processors.

3.2 Application

The goal of this research was to determine the best implementation strategy for collective I/O, and to use this implementation as the basis for thread-based collective I/O. Thus we were interested in the performance characteristics of various implementation options for the collective I/O routine itself, and for different implementation techniques for thread-based collective I/O. For this reason, we used a simple test application which did nothing but make repeated calls to the various implementations of the collective I/O and the thread-based collective I/O operations. Another constraint on the application is that the number of processors must be a power of two, and that the array being written must be two dimensional, where the number of rows is equal to the number of columns. However, the principles apply to more general cases as well. Also, the collective I/O operation called by the application was a collective write operation. We focused on write operations because it is usually more difficult to obtain high performance for parallel writes than for parallel reads.

3.3 Implementation Options for Two-Phase I/O

The motivation for this phase of the research arose while implementing a set of collective I/O routines on the Intel Paragon. We had developed a rather simple implementation that provided much better performance than non-collective I/O operations, but did not achieve anywhere close to the available bandwidth. We then experimented with several implementation options to determine their impact on the performance of two-phase I/O and to quantify the improvement in performance due to various approaches. In this section, we describe the steps we took to achieve high performance collective I/O operations on the Intel Paragon and show that these same techniques can improve performance over a wide range of architectures.

3.3.1 Initial Implementation

The initial implementation of the two-phase I/O algorithm used in these experiments is quite simple. First, the processors exchange information related to their individual I/O requirements to determine the aggregate I/O requirement. Next, each processor goes through a series of sending and receiving messages to redistribute the data into the conforming distribution. When a processor receives a portion of its data it performs a simple byte for byte copy into its write buffer. When a processor sends a portion of its data, it performs a byte for byte copy from its local array into the send buffer (this is not always necessary as discussed below). After all data has been exchanged, each processor performs its write operation in one large request. This initial implementation uses both blocking sends and blocking receives. We refer to this implementation option as Step1. The pseudo code for Step1 is shown below.

```
Begin Two-phase I/O
      exchange information regarding write requests ;
      Use this information to determine collective strategy;
      for (i = 1 ; i < numProcs; i++)
         {
          source  =  (mynode - i + numProcs) % numProcs ;
          dest    =  (mynode + i) % numProcs
          do I need to send dest some of my data ?

          if (yes)
             {
              copy portion of local array into send buffer ;
              Send to dest ;
              }

         does source have data to send me ?
         if (yes)
              {
               wait for message from source ;
               copy into write buffer ;
              }

         do I need to copy some of my data into write buffer?
         if (yes)
             {
              copy portion of my local array into write buffer ;
             }
         }
      Write my write buffer to disk in a conforming manner ;
End Two-phase I/O
```

It is important to note that the copy from the local array into a send buffer is not necessary when the data is distributed among the processors in a block-block distribution and the conforming distribution is row major (as assumed in these experiments). This is because in such cases the data to be redistributed is contiguous within the local array. In the general case however, such as when the data is distributed in a block-cyclic manner or when the local array has a ghost area, this copying would be required. To make our results as general as possible we include the cost of such copying in our algorithm. Also note that the copy from the receive buffer into the write buffer is necessary in our example because the received data cannot be stored contiguously in the write buffer.

3.3.2 Step2: Reducing Copy Costs

As can be seen, the two-phase I/O operation requires a significant amount of copying to and from various buffers. For this reason, it would be expected that modifications to the implementation which reduced the cost of copying data would have a significant impact on performance. The next step then was to change all of the copy routines to use the `memcpy()` library call whenever possible. We term this implementation approach Step2.

3.3.3 Step3: Asynchronous Communication

The next step (Step3) investigated the use of asynchronous rather than synchronous communication. In this approach, the processor first posts all of its sends (using `MPI_Isend`) and then posts all of its receives (using `MPI_Irecv`). After posting its communications, the processor copies any of its own data that it will write to disk from its local array into its write buffer. The processor then goes into a loop polling for messages, and copies the messages into its write buffer as they arrive. Finally, it waits for all of the send operations to complete and then performs the write to disk. This approach can increase paging costs since the application will allocate memory for many communication buffers, but should decrease waiting time since the runtime system can complete *any* message when it arrives rather than waiting for a *particular* message.

3.3.4 Step4: Reversing the Order of Asynchronous Communication

The next approach (Step4) reverses the order of the asynchronous communications. Thus a processor first posts all of its receives, and then posts all of its sends. The idea behind this optimization is that communication in MPI is faster if the application has posted a buffer into which a message may be received, rather than first receiving the message into a system buffer and then copying it into the application buffer [16]. As noted above however, pre-allocating all of the communication buffers can result in increased paging costs.

3.3.5 Step5: Combining Synchronous with Asynchronous Communication

The final optimization (Step5) combined asynchronous receives with synchronous sends. The idea behind this optimization is to reduce communication costs by posting asynchronous receives and to reduce paging costs by having only one send buffer allocated at any given time.

3.4 Experiments

We compared the performance of each approach across all computational platforms, where the metric of interest was the bandwidth achieved by each implementation. In this set of experiments, a 64 Megabyte array of integers was distributed across the processors in a block-block distribution as shown in Figure 1. On the IBM SP2 and the SGI Origin2000, we used 4, 8, 16, 32, and 64 processors. On the HP Exemplar we also used 128 processors. Due to memory constraints, we employed 16, 32, 64 and 128 processors on the Intel Paragon. All experiments on the Intel Paragon, the HP Exemplar and the SGI Origin2000 were performed with the machine in dedicated mode. On the SP2, we used the average of the results of thirty separate trials, taken at the same time of day, over a course of three weeks.

Another important issue we wanted to investigate was how each approach scaled as the size of the array and the number of processors were simultaneously increased. To test this, we employed the largest number of processors with which the application could be configured on the given machine, and varied the size of the file from four Megabytes to one Gigabyte. We used 64 processors on the Intel Paragon and SGI Origin2000, 128 processors on the HP Exemplar, and 256 processors on the Intel Paragon. We tested each approach with 4, 16, 64, 256 and 1024 Megabyte files.

Lastly, we wanted to investigate the impact of executing the collective I/O operation in the background while the main thread continues with other computation in the foreground. We provide a detailed description of these experiments in Section 5.

4 Experimental Results

The results of these experiments are shown in Figure 2 and Figure 3. To reduce complexity, we show only the results of the non-collective approach, Step1, Step2 and Step5. This is because across all architectures, the results for Step3 and Step4 generally fell between the results for Step2 and Step5.

The power of collective I/O techniques can be seen from the fact that there was a significant increase in performance when moving from the non-collective I/O approach to the unoptimized collective approach for all architectures. These results also clearly demonstrate that the chosen implementation strategy for the collective I/O algorithm does have a significant impact on performance. In particular, Step5, which combines asynchronous receives with synchronous sends, provided the best performance across all architectures. As noted above, this approach balances reduced paging costs (only the receive buffers are pre-allocated), and reduced waiting costs (the system can receive *any* message as it becomes available).

It is helpful to consider the results obtained for each architecture in more detail.

4.1 IBM SP2

The importance of both software techniques *and* the I/O subsystem hardware is clearly demonstrated by the results obtained for the IBM SP2 used in these experiments. The optimized two-phase I/O algorithm (the software technique) resulted in a bandwidth of 68 MB/sec compared to 5 MB/sec for the non-collective approach (with 64 processors). However, the under-configured I/O subsystem (only four I/O processors) resulted in a maximum bandwidth of 79 MB/sec (obtained using 32 processors), which was the smallest maximum bandwidth observed across all architectures studied.

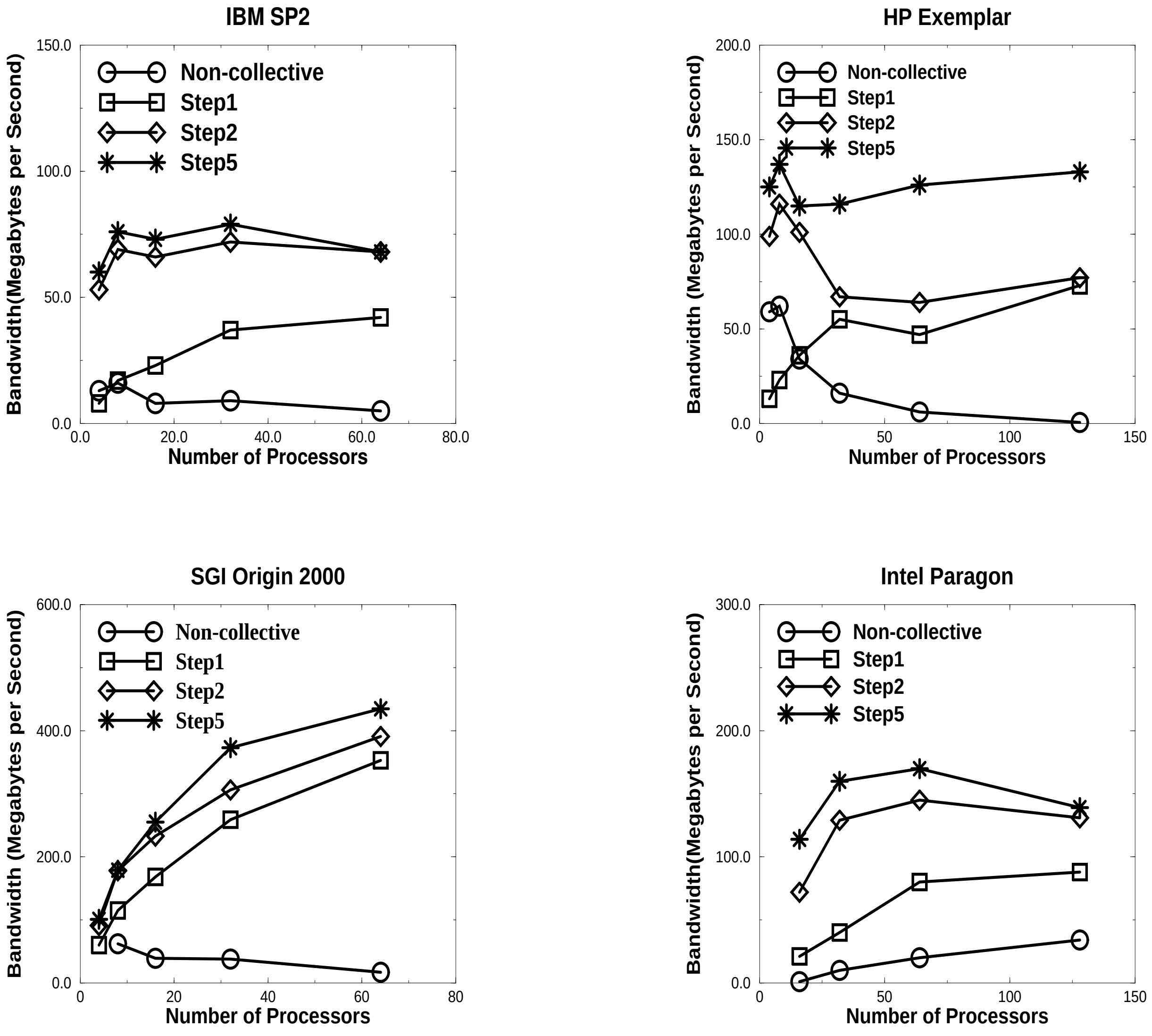

Figure 2: This figure shows the bandwidth achieved (in Megabytes per second) with non-collective I/O and three different implementation options for collective I/O (Step1, Step2, and Step5).

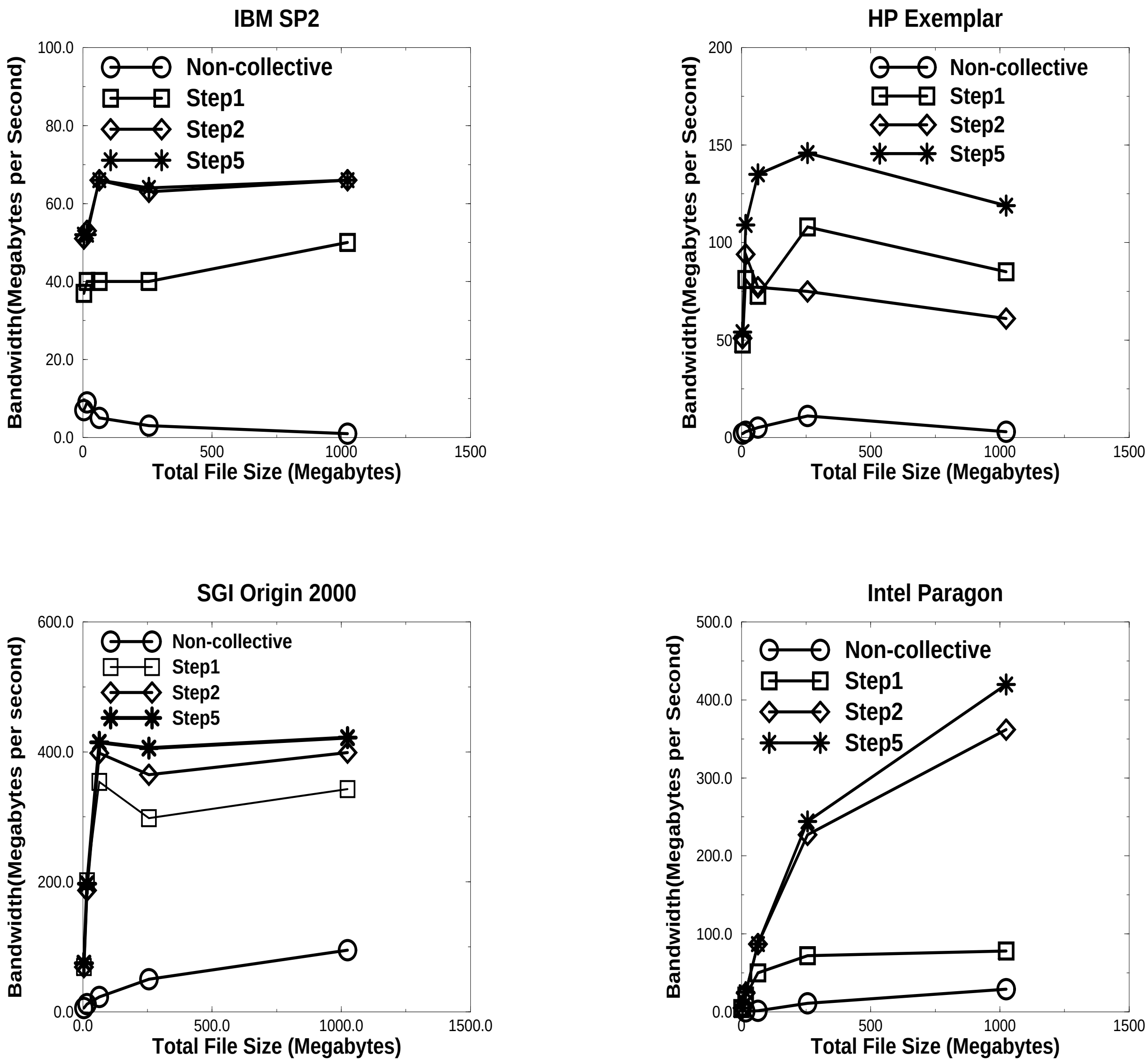

Figure 3: This figure shows performance of non-collective I/O and three collective I/O implementations as the number of processors is held constant and the size of the file is varied between four Megabytes and one Gigabyte.

The impact of the under-configured I/O subsystem is also evident when comparing the results obtained for 32 and 64 processors. With 32 processors, there was an increase in bandwidth of 310% when going from the non-collective approach to the initial collective I/O implementation, a 95% increase between Step1 and Step2, and a further 10% increase between Step2 and Step5. With 64 processors however, there was a greater increase between the non-collective approach and Step1 (740%, showing the non-collective approach does not scale), a reduced improvement of 62% between Step1 and Step2, and *no* improvement in performance between Step2 to Step5. This decrease in relative performance is due to a significant *increase* in contention for the four I/O processors with 64 application processors. This increased contention makes the write to disk much more expensive than the cost of data permutation, and thus techniques that modify only the data permutation costs (which include all of the implementation techniques outlined here) have *less* of an impact on performance as the actual write becomes the dominant cost. The impact of the contention for the I/O processors was also demonstrated by the fact that there was a *decrease* in bandwidth as the number of processors was increased from 32 to 64.

It is interesting to note that the difference in performance between Step1, which uses a byte for byte copy, and Step2, which uses `memcpy` whenever possible, also began to shrink as the number of processors increased from 32 to 64. This also was due to the fact that the cost of permuting the data, which included the cost of copying, became less important as the number of application processors contending for the I/O processors became large.

These same trends are seen in Figure 3 where we held the number of processors at 64 and increased the file size from 4 MB to 1024 MB. Again we observed no difference in performance between Step2 and Step5 because the cost of performing the actual write to disk was the dominant cost. Also, the poor scalability of the non collective approach becomes even more apparent as the file is increased to 1 Gigabyte.

4.1.1 HP Exemplar

The HP Exemplar used in these experiments also demonstrated the impact of both hardware and software design on the performance of the file system. In particular, the defining feature of the Exemplar file system is that a file cannot span more than one hypernode. Thus whenever more than 16 processors access the same file, all of the file activity is being funneled into one particular hypernode (or more specifically one particular file system buffer) creating a significant bottleneck. This bottleneck is clearly demonstrated by the fact that the highest observed bandwidth (140 MB/sec using Step5) was obtained with *eight* processors.

Consider the results for Step5 in more detail. It is interesting that as the number of processors increased from 8 to 16, the bandwidth actually decreased even though all 16 processors resided on the same hypernode. What is happening is that system processes, such as the file system daemon, must execute on one of the 16 processors in a hypernode, and thus on that node(s) there was competition between the collective I/O process and the system process(es) thereby reducing performance. It is also interesting to note that there was a modest increase in performance as the size of the file remained constant and the number of processors was increased from 16 to 128. However, as shown in Figure 3, when the number of processors was held at 128 and the size of the file grew to one Gigabyte, the cost of all 128 processors performing their write into the same file system buffer became prohibitive and overall performance was reduced (by approximately 20%). The Exemplar was the only architecture for which such a reduction in performance was observed.

Now consider the results of Step1 in more detail. With four and eight processors, the non-collective approach outperformed the collective approach by a rather significant margin (62 MB/sec versus 23 MB/sec with eight processors). What was happening was that the cost of data permutation was much more expensive than the cost of writing to disk. In fact, our measurements indicated that with eight processors it took on the order of twice as long to permute the data as it did to perform the actual write to disk. This somewhat surprising observation again had to do with contention. In this case however the contention was for memory bandwidth as the processors, which all shared the same region of memory, exchanged information and data. However, as the number of processors increased to 16 and beyond, the cost of data permutation began to decrease, and the cost of many processors performing small, independent writes to disk, began to increase. With 128 processors, the non-collective approach degraded to a bandwidth of less than 1 MB/sec, and the unoptimized collective I/O approach achieved a bandwidth of 72 MB/sec.

Another interesting result is that as the number of processors increased, the difference in performance between Step1 and Step2 virtually disappeared. In fact in the limit, as shown in Figure 3, Step1 significantly outperformed Step2. This is certainly counter-intuitive since the only difference between the two implementations is that Step2 used `memcpy`, rather than a byte for byte copy, whenever possible. The explanation for this result is as follows.

Consider the results for a 256 MB file with a 128 processors shown in Figure 3. With this configuration, the data permutation cost associated with Step1 was approximately three times as great as that for Step2 (as would be expected). However, the cost of performing the actual write to disk was less than one half the cost incurred by Step2. Given that the write to disk is very expensive with 128 processors, the net result is that Step1 achieved a higher bandwidth than Step2 (108 MB/sec versus 75 MB/sec). The most reasonable explanation for this significant difference in the cost of performing the write to disk is that performing a byte for byte copy helped to separate (in time) the concurrent writes, thereby reducing I/O contention. We conducted a simple experiment to test this hypothesis.

In this experiment, we removed the impact of data permutation and focused on the actual write to disk. We conducted the experiment with 128 processors, all writing into a 256 MB file *assuming* the conforming distribution. In this experiment we observed a bandwidth of 94 MB/sec. We then inserted a very simple delay mechanism before the write, forcing some of the processors to delay their writes for a short period of time. This very simple modification increased the bandwidth to 140 MB/sec, suggesting that when a large number of processors are writing into the same file on the same hypernode, that some staggering of the writes can significantly improve performance. We did not attempt to study the optimal delay to insert between writes as that activity is beyond the scope of this study.

4.1.2 SGI Origin2000

The SGI Origin2000 used in these experiments had a very powerful I/O system with 10 fiber channel connections connected to a total of 100 9-Gigabyte disks. Additionally, the file system (XFS) on this platform has a software optimization that allows an application to write directly from user space to disk (assuming the data to be written falls within a given set of constraints), thus avoiding the write from user space to kernel space. This software optimization, which uses the `O_DIRECT` flag in the `open` call, made a tremendous difference in performance, more than doubling the bandwidth for all of the collective implementation options studied. This optimization is particularly effective for concurrent writes to a common file (as done in these experiments) on a well-configured I/O system. The results shown in Figure 2 and Figure 3 all used this optimization.

There are two striking results that can be seen from Figure 2. First, the maximum bandwidth obtained (435 MB/sec with 64 processors) is significantly higher than that achieved by any of the other architectures. This was a direct result of the powerful hardware configuration and the software optimization. Secondly, the difference between the non collective approach (17 MB/sec with 64 processors) and Step1 (353 MB/sec with 64 processors) was the largest across all architectures studied. This result is a powerful demonstration of the fact that a high performance I/O subsystem, even when it incorporates software optimizations such as a direct write to disk, cannot be effectively utilized *unless* some sort of *global* I/O strategy is employed.

Given the very powerful I/O system (both hardware and software), we were interested in whether the data redistribution costs, or the actual write to disk, would dominate the costs of the collective I/O operation. To answer this question, we inserted simple timers into the code and monitored the amount of time spent in each of the two phases. Our measurements showed that it was the *data permutation* costs, rather than the write to disk, that was the dominant cost (because of the fast I/O system on this machine). In fact, with 16 processors the time spent in the data permutation phase represented 90% of the total cost of the collective operation (using Step5). With 32 processors, the data permutation phase represented 75% of the total cost, and with 64 processors, the data permutation phase took approximately 55% of the total cost (again using Step5). This is particularly noteworthy because, for all other architectures using 64 processors and Step5, the write to disk represented *at least* 90% of the cost of the collective I/O operation. However, when the number of processors on the Origin was held constant at 64 and the file size was increased to 1 Gigabyte, the percentage of time spent in the write to disk did begin to increase. In the limit, the data redistribution phase accounted for around 35% of the total cost of the collective I/O operation. The fact that the relative time spent in the write increased to 65% is reflected in Figure 3 where, as can be seen, the performance of Step1, Step2 and Step5 begin to converge as the file size approaches 1 Gigabyte.

4.1.3 Intel Paragon

The Intel Paragon used in these experiments also had a very powerful I/O subsystem consisting of 64 I/O processors. However, the power of this hardware support is not evident in

Figure 2. In particular, the maximum bandwidth observed was 170 MB/secs with 64 processors using Step5, and there was a decrease in performance when the number of processors was increased from 64 to 128. Given the powerful I/O subsystem, it is somewhat surprising that bandwidth actually decreased as the number of processors was increased to 128. The reason for this decrease in performance was a *combination* of two factors: increased contention for the I/O processors and a file size too small to use the I/O subsystem efficiently. This explanation was validated by the results shown in Figure 3, where we observe a bandwidth of 420 MB/sec using 256 processors and a 1 Gigabyte file.

It is interesting to note that the bandwidth was still increasing when the file size was increased from 256 MB to 1 Gigabyte, suggesting that even higher bandwidths are possible with an increased file size. The Intel Paragon was the only architecture studied where performance was still increasing at the limits of these experiments.

5 Executing Collective I/O in the Background

In the previous section, we showed that a non-collective I/O algorithm cannot make efficient use of even a very powerful I/O subsystem. Further, we showed that a naive implementation of two-phase I/O does not, in general, provide high performance, but that an optimized algorithm can offer excellent performance. However, it can be argued that basic collective I/O techniques are maturing, and that we must look to other approaches to make a further *significant* impact on the performance of applications with large I/O requirements. One promising approach is to use threads to execute the collective I/O operation in the background while the main thread continues with other computation in the foreground. This approach does not reduce the cost of I/O per se, but can reduce the *impact* of I/O on the performance of the application.

As noted above, MPI-2 [21] defines just such operations, termed *split-collective* operations, where the collective I/O routine may be executed in the background while other computation/communication continues in the foreground. Given the importance of MPI as a standard for message passing applications, and given its definition of a set of *portable* collective I/O operations, it is worthwhile to frame our discussion in terms of implementation techniques for split-collective I/O operations.

5.1 Split-Collective I/O Operations

A split-collective operation has a *begin* function, which initiates the collective I/O, and an *end* function, which blocks the calling thread until the collective operation is completed. Between calls to the begin and end functions, the implementation *may* allow the main thread to continue with its computation while the collective I/O operation is carried out in the background, overlapping the two operations. We say *may* because an implementation is allowed to perform the entire collective I/O in the begin function (in the main thread), thus executing the collective I/O and the computation sequentially. Currently, no published implementation of MPI parallel I/O overlaps computation in the main thread with collective I/O in the background.

There are essentially three implementation options for split-collective I/O operations: to perform all of the collective I/O in the background, to perform part of the collective I/O in the background and part in the foreground, and to perform none of the collective I/O in the background. In the first case, the begin function would spawn a thread to perform the collective I/O and then immediately release the main thread allowing it to continue its computation. The background thread would simply exit when it completed the collective I/O, and the end function would ensure that the main thread blocks until the background thread did exit. In the second option, part, but not all, of the collective I/O would be performed in the background. Consider a collective write request. When the main thread executes the begin function, the implementation may choose to execute all of the collective I/O routine *except* the actual write to disk in the begin function, then spawn a thread to perform the write to disk and immediately release the main thread. In this case the end function would again block the main thread until the I/O thread exited. This sequence would be reversed in the case of a collective read operation. In the final implementation option, the entire collective I/O operation would be performed in the begin function and there would be no attempt to overlap computation with collective I/O. In the following sections we explore each of these alternatives.

6 Experiments with Thread-Based Collective I/O

6.1 Non-collective I/O and Threads

The first set of experiments was designed to provide an estimate of the maximum speedup obtainable by overlapping computation with I/O operations. The application program simply repeated the execution of a compute phase followed by an I/O phase. The compute phase consisted of performing some number of floating point multiplications, and the I/O phase consisted of each processor writing a four Megabyte section of an array to disk *assuming* the conforming distribution. That is to say, each processor wrote its section of the array to different locations on the disk, but the processors did not engage in a collective phase to map out the optimal I/O strategy, nor did they collect and redistribute data among the processors. The time taken to complete the compute phase was controlled by varying the number of floating point operations. We used 8, 16, 32 and 64 processors, and calibrated the compute phase to take approximately as long as the average I/O phase with 64 processors. Since each processor writes four Megabytes to the file, the total number of bytes written was 32 Megabytes with 8 processors, 64 Megabytes with 16 processors, 128 Megabytes with 32 processors and 256 Megabytes with 64 processors. It is important to note that we calibrated the length of the compute phase independently for each architecture. That is to say, the length of the compute phase for a given architecture was dependent only on the time required for *that* particular architecture to write 256 Megabytes to disk.

The metric of interest was the time required to complete both the computation phase and the I/O phase. In the first approach, the application performed the compute phase and the I/O phase sequentially (i.e. there was no overlap of computation and I/O). In the second approach, the application spawned a thread to perform the I/O in the background and then immediately entered into its compute phase. Once the compute phase was completed, the main thread blocked (if necessary) until the I/O phase was completed.

The results are shown in Figure 4. We note that in this figure there is not (necessarily) any correlation in the time scales of the four architectures studied as the time required to write a 256 Megabyte file to disk and the corresponding amount of computation performed by each machine are architecture dependent. However, what *can* be compared is the percentage by which performance was improved due to executing the write to disk in the background.

As can be seen, using a background thread to perform the write did have a significant impact on performance as the number of processors and the size of the file were simultaneously increased. With 64 processors, the SP2 showed an improvement in performance of 46%, the Paragon showed an improvement of 35%, and the SGI Origin produced an improvement of 38%. The Exemplar showed a 26% improvement in performance with 32 processors, but only a 9% improvement with 64 processors. This decrease reflects some inefficiency in the thread library which exerts itself when there are a large number of threads (in this case 128), and all of the threads are attempting to write into the same region of memory (the file system cache). As can be seen, the Exemplar is the only architecture for which such a decrease in performance was observed.

6.2 Collective I/O and Threads

In the previous section, it was shown that spawning a background thread to overlap I/O with computation can result in significant performance gains, at least when the I/O thread does nothing but perform a single, large write to disk. In this section, we seek to determine if similar benefits can be obtained when the whole collective I/O operation is executed in the background.

In these experiments, the processors executed the whole collective I/O operation in the background, including the use of inter-processor communication to collectively determine the optimal I/O strategy and the redistribution the data. These experiments modeled an SPMD computation, where each processor operated on a different region of a 64-Megabyte array. The array was distributed among the local memories of the processors in a block-block distribution as shown in Figure 1. We held the size of the array constant and measured the time required to complete both the computation and collective I/O for 8, 16, 32 and 64 processors. The collective I/O operation was a collective write. We note that, due to memory constraints, we employed 16, 32, 64 and 128 processors on the Intel Paragon. The best implementation of collective I/O (Step5) was used in all experiments.

The most natural implementation option is to simply spawn a thread to perform the whole collective I/O operation in the background, while the main thread continues with other computation in the foreground. In terms of implementation techniques for split-collective operations, this corresponds to spawning an I/O thread in the begin function and then immediately returning control to the main thread allowing it to continue with its computation. The main thread would then be blocked in the end function until the collective I/O operation is completed. We compared this approach to performing the collective I/O and computation in sequence.

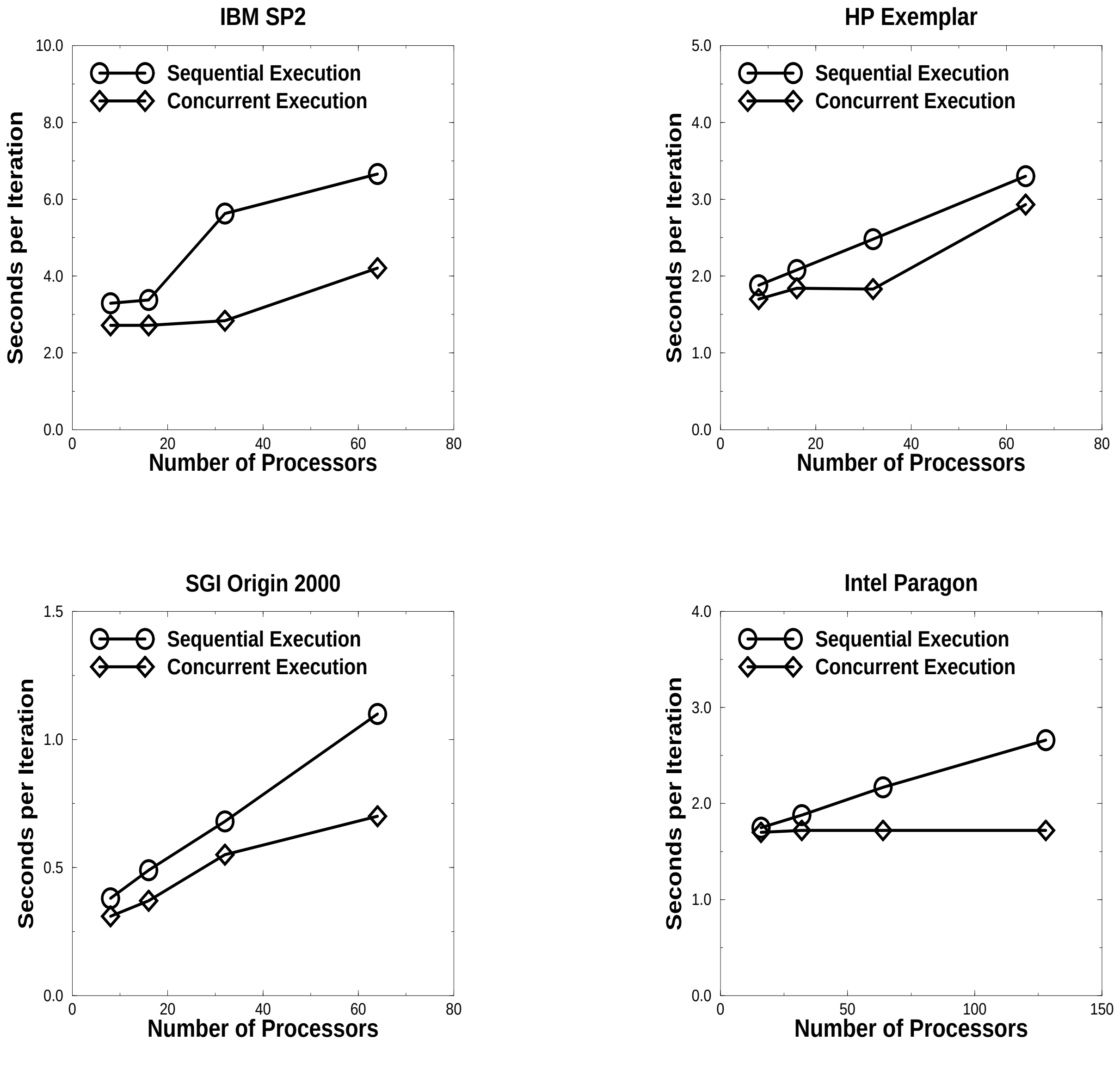

Figure 4: This figure shows the improvement in performance that is possible when computation is overlapped with a large write to disk (assuming the conforming distribution).

In Figure 5 we compare these two approaches. In this figure, we measure the time required to complete one iteration of a compute phase followed by (or overlapped with) a collective I/O phase (labeled Seconds per Iteration). As can be seen, the results are quite disappointing. The use of a background thread to overlap computation with collective I/O resulted in little, if any, improvement in performance for *any* architecture other than the IBM SP2. On the HP Exemplar (with more than 8 processors), and the SGI Origin2000, this approach actually *decreased* performance, and this decrease became more significant as the number of processors was increased. On the Intel Paragon, the performance of both approaches was about the same. This clearly demonstrates that merely spawning a thread to perform the collective I/O operation in the background is *not*, in general, sufficient to achieve high performance.

To understand these results, it is important to note that only *parts* of the collective I/O algorithm can actually be overlapped with computation. In particular, the computation required to determine the optimal I/O strategy, the copying to and from the various message buffers, the setting up and initiation of communications, and the initiation of the disk write *cannot* be overlapped with computation. The time spent waiting for messages to arrive and waiting for a write to disk to complete *can* be overlapped with computation. Thus there is a trade-off. When the actions taken by the I/O thread cannot be overlapped with computation in the main thread, the two threads are *competing* for control of the CPU. This requires the multiplexing of the two threads on and off the CPU, incurring the additional costs of thread switching and thread scheduling. In this set of experiments, the performance gains obtained by overlapping (parts) of the collective I/O with computation were offset, or more than offset, by the overhead of thread scheduling and thread switching.

These results are perhaps non-intuitive, and can best be explained by differentiating between *user-level* threads, where the threads are executing in user space, and *kernel-level* threads, where the threads are managed by the kernel. With user-level threads, there is very little context and the cost of performing a thread switch is quite low. The trade-off however is that when a user-level thread blocks, such as when it performs a write to disk, the *whole* process is blocked, not just the calling thread. This of course precludes the process from overlapping computation with IO or communication. In the case of kernel-level threads, only the calling thread itself is blocked, allowing the process to overlap computation with I/O or communication. The trade-off however is that the kernel must maintain, schedule, and multiplex these threads. While the cost of managing kernel-level threads is less than that of managing heavy-weight processes, it can still be significant, and can have a negative impact on performance. Kernel-level threads were used in all of these experiments, and thus the relatively high cost of thread switching was incurred.

To test our hypothesis we modified the implementation of the collective I/O routine to (virtually) eliminate the cost of thread-switching altogether. We accomplished this by implementing the I/O thread such that it performed part, but not all, of the collective I/O algorithm. In particular, all of the copying and interprocessor communication required by the collective I/O algorithm were performed by the *main* thread, and the I/O thread was spawned to perform *only* the actual write to disk. With this approach, virtually all of the competition for the CPU between the two threads is eliminated since the I/O thread does

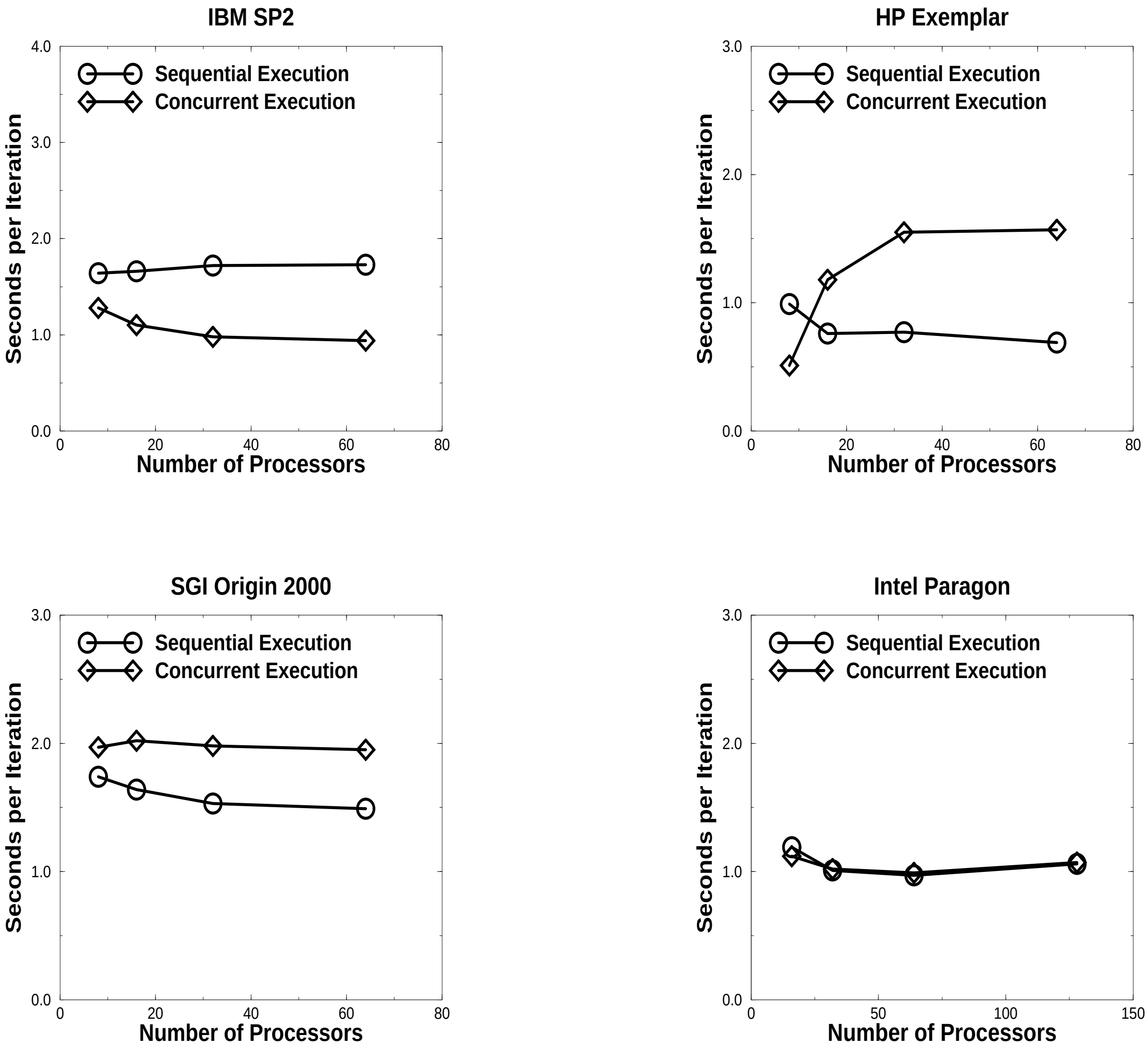

Figure 5: This figure shows the time required to complete one iteration of a compute phase and one iteration of a collective I/O phase. In the first case, the computation and collective I/O are executed sequentially. In the second case, a thread is spawned to execute the collective I/O in the background while the computation phase continues in the foreground
.

nothing but make the I/O call and wait for its completion.

Now consider how this approach corresponds to the implementation of MPI split-collective operations. In this case the initial part of the two-phase I/O routine is executed in the begin function. This includes *all* of the activity required to redistribute the data in such a way that each processor can perform its write to disk assuming the conforming distribution. Once this part of the collective I/O routine is complete, the begin function spawns a thread to perform the actual write to disk and then immediately returns to the main thread allowing it to enter into its computation. Execution of the end function blocks the main thread until the write to disk is complete. (Again note the order of these events would be reversed in the case of a collective read operation.)

In Figure 6, the performance of the three implementation options is shown. Consider the results of each architecture when 64 processors are used. As can be seen, executing only the actual write to disk in the background can provide significant performance benefits. On the Intel Paragon, this strategy provided up to a 26% improvement in performance over both of the other techniques. On the SGI Origin2000, spawning a thread to perform only the disk write resulted in a 25% improvement over the sequential approach, and a 43% improvement when compared to executing the whole collective I/O operation in the background. On the HP Exemplar, spawning a thread to perform only the write to disk improved performance by 12% over the sequential approach, and by 61% when compared to performing the whole collective I/O routine in the background. On the IBM SP2, both of the thread-based strategies performed at approximately the same level and provided up to a 45% improvement over executing the two phases in sequence.

Before leaving this section it is important to note that performing one large I/O request in the background *assumes* there is enough memory to buffer all of the data that will be written to disk. If the buffer provided by the application is not large enough to hold all of the data, then the implementation is forced to perform the write to disk iteratively. This would certainly have a negative impact on performance, and in such cases the best technique may be to perform the whole collective I/O operation in the foreground.

6.2.1 SGI Origin2000 Using the `O_DIRECT` Option

The results shown in Figure 6 for the SGI Origin2000 were obtained *without* using the `O_DIRECT` option. That is, the file system on the Origin did not copy data directly from the application buffer to disk, but rather to the kernel buffer cache first and then to disk. We were interested in knowing whether the `O_DIRECT` option that writes directly from application buffer to disk would significantly modify the results shown in Figure 6. The outcome of this set of experiments is shown in Figure 7.

Again we observe that executing the whole collective I/O operation in the background exhibits the worst performance of the three implementation options. What is interesting however is that executing only part of the collective I/O operation in the background did not provide *any* improvement over the sequential approach. This result has to do with the very powerful I/O subsystem and the ability of the file system to bypass the kernel buffer cache. In particular, the write to disk was so fast that the time required to complete the

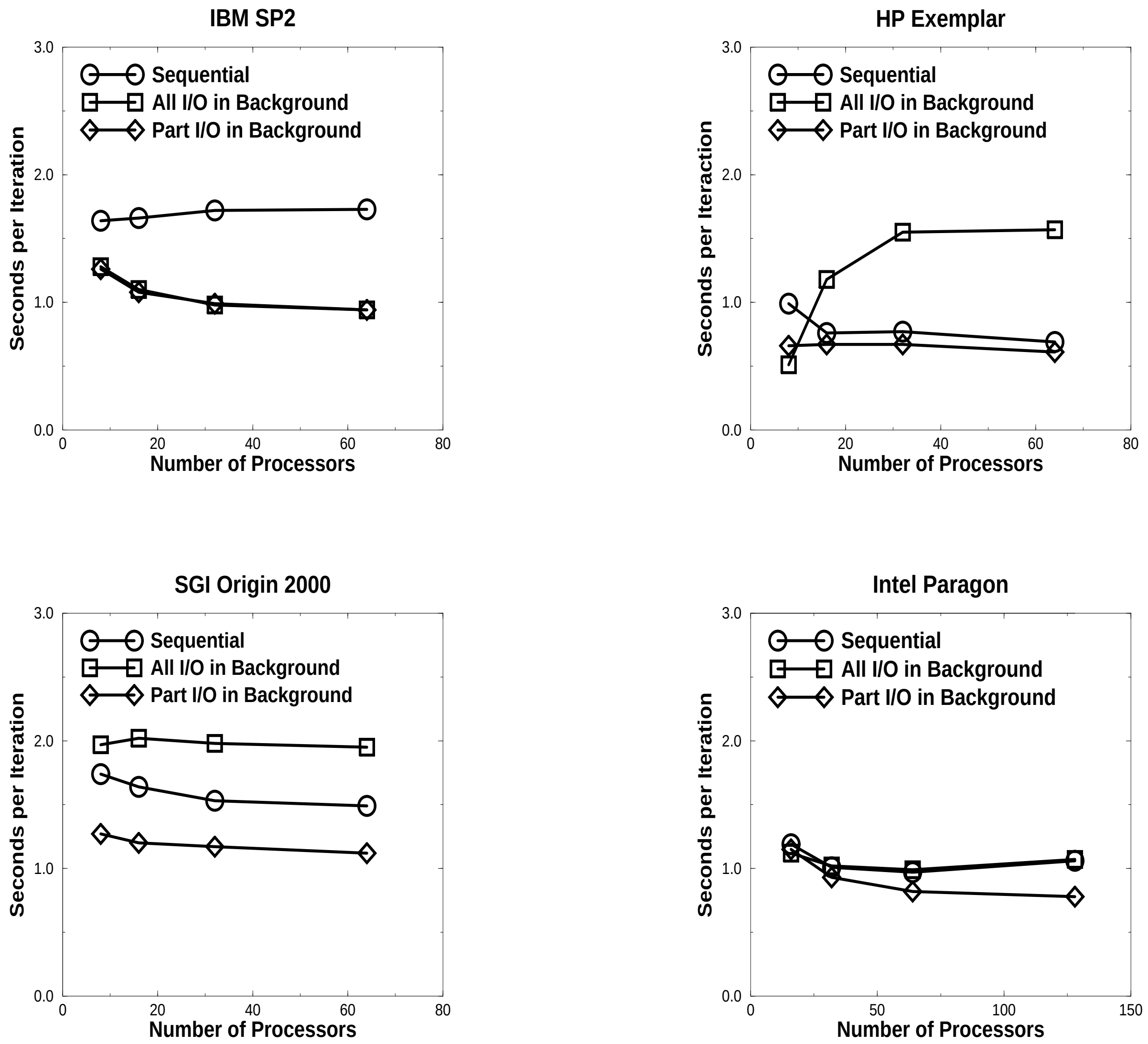

Figure 6: This figure shows the performance of the three implementation options for split-collective I/O operations. In the first approach, the whole collective I/O operation is performed in the main thread. In the second approach, a thread is spawned to execute the whole collective I/O operation in the background. In the third approach, all of the computation and communication associated with data redistribution is performed in the foreground, and a thread is spawned to perform only the actual write to disk

.

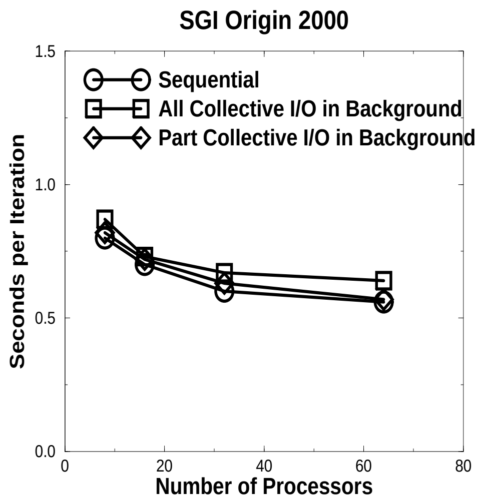

Figure 7: This figure shows the relative performance of thread-based collective I/O when the application can write directly to disk.

write was *less* than the time required to perform the interprocessor communication and data redistribution (45% for the write, 55% for the data permutation). Since the proportion of time spent in the disk operation was relatively small, and since this was the only time a thread was active, it makes sense that there would be little improvement in performance.

7 Implications for Real-World Applications

A fair question to ask is whether the results obtained in this research are directly applicable to real-world applications. This question can perhaps best be answered by considering the basic structure and execution of the experimental applications. First, the experimental applications employed large (two dimensional) arrays distributed across the processors using a block-block data distribution pattern. Such data structures and data distribution patterns are quite common in parallel scientific applications. (It is also worth noting that we experimented with several other data distribution patterns and obtained similar results.) Secondly, the basic structure of the experimental applications, where each processor performs some computation on its segment of the distributed array and then writes a set of results to disk, is also common in parallel scientific applications. This model of application behavior is also quite relevant to large applications that periodically check-point their results to disk, where the write to disk can be overlapped with computation in the main thread.

Perhaps a better question to ask is what characteristics must a parallel scientific application possess to obtain results similar to those reported here. Our experience and experimentation suggests there are three such critical requirements. First, the I/O and computational requirements must be reasonably balanced. If the computational requirements are significantly greater than the cost of performing the collective write to disk, then the techniques presented here will have little impact on overall performance. Secondly, the application must be structured such that the collective I/O operation can be overlapped with computation in the main thread. Finally, the communication library itself must be thread safe.

8 Related Work

The research most closely related to this project is the development of the MTIO library [20], which is a multi-threaded parallel I/O library. MTIO supports the overlap of computation with collective I/O by spawning an I/O thread to complete the whole collective routine in the background. The MTIO library is implemented on the IBM SP2. The authors report up to an 80% overlap of computation and I/O, which is similar to the results we obtained for the SP2. As noted above, however, we found that the SP2 is the only architecture for which this approach performed well; on other architectures we obtained better performance by doing only the actual write to disk in the background and the rest of the collective operation in the main thread.

The performance of two-phase I/O on the Intel Paragon has been studied extensively by both Dickens and Thakur [12] and by Bordawekar [4]. However, neither of these studies looks at thread-based collective I/O. Also, Bordawekar [6] provides an excellent discussion of the I/O characteristics of the HP Exemplar.

Two-phase I/O is not the only approach that can significantly improve performance of I/O intensive applications. Acharya et al. [1] investigated code restructuring and other optimizations to improve the performance of I/O bound computations and reported excellent performance without the use of collective I/O. However, the approach outlined in their study requires significant modifications to the application code and knowledge of future I/O requests. Also, the Vesta file system has been shown to enhance performance by using prefetching and caching without two-phase I/O [13].

There are other projects using collective I/O. For example, Passion has been extended to handle out-of-core arrays [26]. Also, a variation of disk-directed I/O is used in the Panda runtime library [23]. Excellent overviews of the field of parallel I/O can be found in [11, 14].

9 Discussion and Conclusions

Obtaining high performance collective I/O is critical to large, I/O intensive scientific computations. The research presented here demonstrates that it is *possible* to obtain such high performance using two-phase I/O and thread-based collective operations, but it is not *automatic*. To demonstrate the importance to performance of the implementation technique, we developed a series of optimizations to the basic two-phase I/O algorithm and studied the impact on performance of each such optimization. One powerful example of this impact was observed on the Intel Paragon, where the initial collective I/O implementation resulted in a bandwidth of 70 MB/sec, and the fully optimized implementation resulted in a bandwidth of 440 MB/sec. In fact, *all* of the architectures studied showed a significant increase in performance between the non-collective I/O approach and the naive implementation of collective

I/O (Step1), and between the Step1 implementation and the fully tuned implementation (Step5) of collective I/O.

Also, this research clearly demonstrates that it is possible to reduce the impact of collective I/O operations by executing such operations in the background while the main thread continues with other computation in the foreground. We showed that the most natural implementation technique, to simply spawn a thread to perform the whole collective I/O operation in the background, is in general, *not* sufficient to obtain high performance. In fact, for some of the architectures studied here, this simple approach more often *reduced* rather than *enhanced* performance. The reason is simple: If a thread can block without blocking the whole process, then the threads are being managed at the kernel level. This makes thread switching expensive, and, when there is a lot of competition between the main thread and the I/O thread, can negate the benefits of overlapping computation and I/O. We did show however that when this competition is minimized, such as when the I/O thread performs only the actual write to disk, that (in general) excellent performance gains can be obtained.

It is important to note however that this whole investigation of thread-based collective I/O is built upon the premise that there is sufficient computation in the main program to effectively overlap computation with collective I/O. Whether this is in general true remains to be seen.

There are currently three main obstacles to the investigation of thread-based collective I/O. Foremost is the lack of thread-safe implementations of MPI. Secondly, there is no way to directly observe the behavior of the threads and how they interact with the rest of the system. Rather, using threads is like using a black box which, from time to time, exhibits completely non-intuitive behavior. Finally, although the threads package on each machine studied is based on the POSIX standard, there are still enough differences between the libraries to make porting of code between the architectures somewhat tedious.

Current research is focusing on implementing the complete MPI-2 parallel I/O library, and performing this same study on important application codes.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Acharya, A., Uysal, M., Bennett, R., Mendelson, A., Beynon, M., Hollingsworth, K., Saltz, J. and Alan Sussman. Tuning the Performance of I/O Intensive Parallel Applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, Philadelphia, May 1996. ACM Press.

[2] Baylor, S. and C. Wu. Parallel I/O Workload Characteristics Using Vesta. *Input/Output in Parallel and Distributed Computer Systems*. Chapter 7, pages 167–185, 1996. R. Jain, J. Werth, and J. Browne editors. Kluwer Academic Publishers.

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

[4] Bordawekar, R. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proceedings of the 11th ACM International Conference on Supercomputing.* ACM Press, July 1997.

[5] Bordawekar, R., del Rosario, J. and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[6] Bordawekar, R. Quantitative Characterization and Analysis of the I/O Behavior of a Commercial Distributed-shared-memory Machine. *California Institute of Technology Technical Report CACR TR-157* March 1998.

[7] Cho, Y., Winslett, M., Subramaniam, M., Chen, Y., Wen Kuo, S. and Seamons, K. Exploiting Local Data in Parallel Array I/O on a Practical Network of Workstations. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems.*, ACM Press, November 1997.

[8] Choudhary, A., Bordawekar, R., Harry, M., Krishnaiyer, R., Ponnusamy, R., Singh, T. and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report number SCCS–636, NPAC, Syracuse University, 1994.

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

[10] DelRosario, J., Bordawekar, R. and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems* pages 56–70, Newport Beach, CA, 1993.

[11] DelRasario, J. and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. IEEE Computer, 27(3):59–68, March 1994.

[12] Dickens, P. and R. Thakur. A Performance Study of Two-phase I/O. In *4th International Euro-Par Conference Proceedings*. In Lecture Notes of Computer Science, 1470. Published by Springer, D. Pritchard and J Reev Eds., pages 959–965.

[13] Feitelson, D., Corbett, P., Hsu, Y. and J. Prost. Parallel I/O Systems and Interfaces for Parallel Computers. In *Topics in Modern Operating Systems.* IEEE Computer Society Press, 1997.

[14] Feitelson, D., Corbett, P., Baylor, S. and Y. Hsu. Parallel I/O Subsystems in Massively Parallel Supercomputers. In *IEEE Parallel and Distributed Technology*, 3(3):33–47, Fall 1995.

[15] Feitelson, D., Corbett, P. and J. Prost. Performance of the Vesta Parallel File System. In *Technical Report RC 19760*, IBM Watson Research Center, Yorktown Heights, N.Y., September, 1994.

[16] Gropp, W., Lusk, E. and A. Skjellum. Using MPI. Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Cambridge, Massachusetts. 1996.

[17] Kotz, D. Disk-Directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[18] Kotz, D. and N. Nieuwejaar. Dynamic File-Access Characteristics of a Production Parallel Scientific Workload. In *Supercomputing '94* pages 640–649, November 1994.

[19] Kotz, D. Expanding the Potential for Disk-Directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*. Pages 490–495, IEEE Computer Society Press.

[20] More, S., Choudhary, A., Foster, I. and M. Xu. MTIO a Multi-Threaded Parallel I/O System. In *Proceedings of the Eleventh International Parallel Processing Symposium*, April 1997.

[21] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. URL: http://www.mpi-forum.org/docs/docs.html.

[22] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C. and M. Best. File-Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems*, volume 7, number 10, pages 1075–1089, October 1996.

[23] Seamons, K., Chen, Y., Jones, P., Jozwiak, J. and M. Winslett. Server-Directed Collective I/O in Panda. In *In Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Science press.

[24] Smirni, E., Aydt, R., Chien, A., and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, IEEE Computer Society Press, 1996.

[25] Smirni, E. and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. In *Performance Evaluation: An International Journal*, Volume 33, Number 1, pages 27–44, June, 1998.

[26] Thakur, R. and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming* 5(4):301–317, Winter 1996.

[27] Thakur, R., Choudhary, A., More, S and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *IEEE Computer*, 29(6):70–78, June 1996.

[28] Thakur, R., Gropp, W. and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, Lecture Notes in Computer Science 1127. Springer-Verlag, pages 24–35, September, 1996.