Improving Collective I/O Performance Using Threads

Phillip M. Dickens
Department of Computer Science and Applied Mathematics
Illinois Institute of Technology

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory

September 27, 1998

Abstract

Massively parallel computers are increasingly being used to solve large, I/O intensive applications in many different fields. For such applications, the I/O requirements quite often present a significant obstacle in the way of achieving good performance, and an important area of current research is the development of techniques by which these costs can be reduced. One such approach is *collective I/O*, where the processors cooperatively develop an I/O strategy that reduces the number, and increases the size, of I/O requests, making a much better use of the I/O subsystem. Collective I/O has been shown to significantly reduce the cost of performing I/O in many large, parallel applications, and for this reason serves as an important base upon which we can explore other mechanisms which can further reduce these costs. One promising approach is to use threads to perform the collective I/O *in the background* while the main thread continues with other computation in the foreground.

In this paper, we explore the issues associated with implementing collective I/O in the background using threads. The most natural approach is to simply spawn off an I/O thread to perform the collective I/O in the background while the main thread continues with other computation. However, our research demonstrates that this approach is frequently the *worst* implementation option, often performing much more poorly than just executing collective I/O completely in the foreground. To improve the performance of thread-based collective I/O, we developed 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 new technique can significantly improve the performance of thread-based collective I/O, providing up to an 80% improvement over sequential collective I/O (where there is no attempt to overlap computation with I/O). Also, we discuss one very important application of this research which is the implementation of the *split-collective* parallel I/O operations defined in MPI 2.0.

1 Introduction

Massively 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 generally 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 [5, 14]. 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 to/from the file system. 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 system. The application can make 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 [6, 19, 20, 2, 15, 18, 3]. 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.

Two significant implementation techniques for collective I/O are two-phase I/O [6, 19, 20] and disk-directed I/O [13, 16]. In disk-directed I/O, the collective I/O request is sent to the I/O processors which collectively determine and carry out the optimal I/O strategy. In the two-phase approach, the application processors collectively determine and carry out the optimized approach. In this paper, we deal only with the two-phase approach.

Consider a collective read operation. If the data is distributed across the 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 orders of magnitude faster than I/O operations, it is possible to obtain performance that approaches that of the conforming distribution.

Thus collective I/O plays a critical role in reducing the cost of I/O requirements, and it provides the basis from which we can explore other approaches to further reduce the impact of such operations. A very promising approach 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 it can significantly reduce the impact of these operations by overlapping the collective I/O with other computation.

There are many factors which can impact the performance of thread-based collective I/O. Paramount among these is the parallel architecture and the configuration of the parallel file system. Thread switching costs, incurred when the background I/O thread and the main thread compete for the CPU, also has a tremendous impact on performance. Additionally, the type of computation performed by the main thread also impacts performance. Other important factors include thread scheduling and the memory available to perform the collective I/O operation.

In this paper, we study the issue of implementing thread-based collective I/O, and do so for four important massively parallel architectures: the IBM SP2, the Intel Paragon, the HP Exemplar and the SGI Origin 2000. This research shows 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.

An important application of this research is the development of implementation techniques for the *split-collective* parallel I/O operations defined in MPI 2.0. These operations provide an implementation the opportunity to perform collective I/O in the background, but there are currently no implementations which do so. The wide-spread use of MPI, and the importance of *portable* parallel I/O operations, make this a very important and timely application of this research.

The rest of the paper is organized as follows. In Section 2, we discuss the technique of two-phase collwctive I/O and split-collective operations in more detail. In Section 3, we study the performance of various approaches to overlapping collective I/O and computation. In Section 4, we discuss related work and we provide our conclusions in Section 5.

2 Two-phase I/O

Most parallel architectures provide some form of hardware support for parallel I/O. In general, 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 platform 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 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 naive 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.

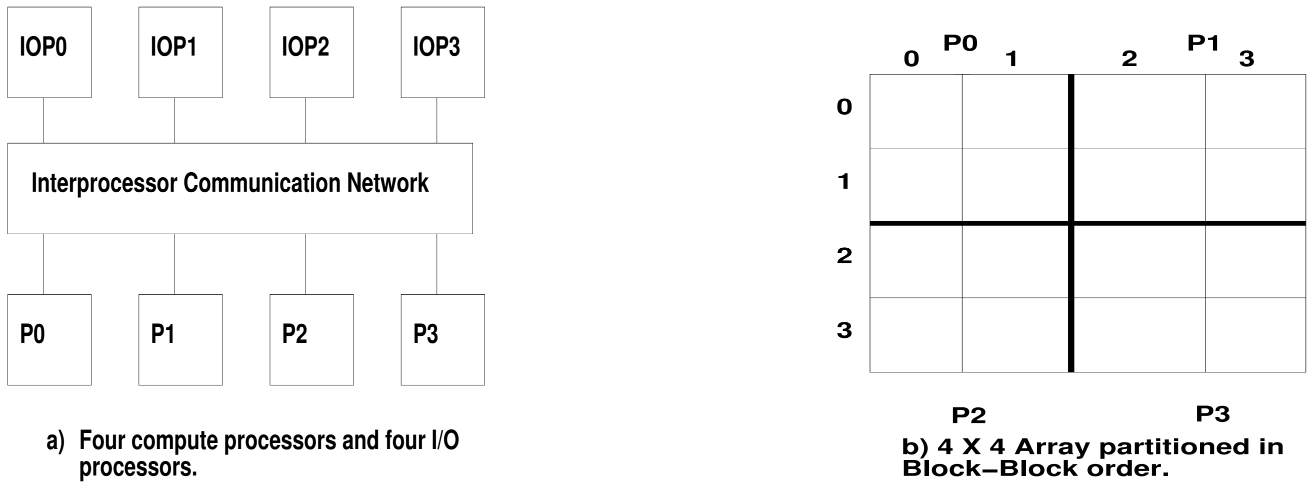

Figure 1: Example system with four compute processors, four I/O processors and a 4 X 4 array.

2.1 Split-Collective I/O Operations

The Message Passing Interface (MPI) [12] is the emerging standard by which distributed computers communicate and synchronize. As such, it provides a platform upon which portable distributed codes can be developed. The recently released MPI 2.0 standard incorporates parallel I/O into the specification. As noted above, this specification includes *split-collective* operations, which provides the implementation the opportunity to perform collective I/O in the background.

A split-collective operation has a **begin** statement, which initiates the collective I/O, and an **end** statement, which blocks the calling thread until the collective operation is completed. Between the **begin** and **end** statements, 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 a correct implementation option is to block the main thread until the collective I/O operation completes, thus executing the collective I/O and the computation sequentially. Currently, no 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, execution of the **begin** statement would spawn an I/O thread to perform the collective I/O. Immediately after execution of the **begin** statement the main thread would continue with its computation. The background thread would simply exit when it completed the collective I/O, and the main thread would block on the **end** statement 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** statement, the implementation may choose to execute all of the collective I/O routine *except* the actual write to disk in the foreground, and perform the write to disk in the background. In this case the main thread would again block on the **end** statement 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 **begin** and **end** statements would be essentially ignored and there would be no attempt to overlap computation with collective I/O. In the following sections we explore each of these alternatives.

3 Experiments

3.1 Computational Platforms

In order to make our conclusions as general as possible, we conducted all experiments on four massively parallel machines. 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 is located at Argonne National Laboratory, and consists of 80 compute nodes and 4 I/O processors. Each I/O processor controls four SSA disks, each with a 9 Gigabyte capacity. The Intel Paragon is located at the California Institute of Technology, and is configured with 381 compute nodes and 64 SCSI I/O processors. Each I/O processor controls a 4 Gigabyte seagate drive. The SGI Origin 2000, housed at Argonne National Laboratry, is configured with 128 compute processors and two fibre channel connections each of which is connected to a disk array with 110 9-Gigabyte disks. These two fibre channel connections are shared by all 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.

3.2 Experiment 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, the processors did not engage in a collective phase to map out the optimal I/O strategy and did not collect or redistribute data among the processors. Each processor simply performed one large write of four Megabytes to *different* locations on the disk. The time taken to complete the compute phase was controlled by varying the number of floating point operations performed during that phase. These experiments were performed with 8, 16, 32 and 64 processors, where the length of the compute phase was calibrated to take approximately as long as the average I/O phase with 64 processors. Since each processor wrote 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.

The metric of interest is 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. The main thread blocked until both phases were completed.

The results are shown in Figure 2. As can be seen, all of the architectures, except for the HP Exemplar, show excellent improvement in performance as the number of processors and the size of the file are simultaneous increased. With 64 processors, the SP2 showed an improvement in performance of 58%, the Paragon showed an improvement of 64%, and the SGI Origin produced a speedup of 60%. The Exemplar showed *no* improvement in performance with 64 processors, most likely because all of the file activity is funneled through a *single* hypernode creating a bottleneck. The Exemplar does show a modest improvement in performance with 8, 16 and 32 processors.

These results show 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.

3.3 Collective I/O and Threads

In this section, we investigate overlapping collective rather than non-collective I/O with computation. Thus the whole collective I/O operation is executed, including the use of inter-processor communication to collectively determine the optimal I/O strategy and to redistribute the data among the processors as needed. These experiments assume an SPMD computation, where each processor computes over a different region of a $4096X4096$ array of integers. The array is distributed among the processors in a Block-Block distribution as shown in Figure 1. We hold the size of the array constant, and measure the time required to complete both the computation and collective I/O for 8, 16, 32 and 64 processors. The collective I/O operation is a collective write. We note that due to memory constraints, we employed 16, 32, 64 and 128 processors on the Intel Paragon.

As noted above, the most natural implementation option is to simply spawn a thread to perform the collective I/O operation in the background, while the main thread continues with its computation in the foreground. In terms of implementation techniques for split-collective operations, this corresponds to spawning an I/O thread when the **begin** statement is executed, allowing the main thread to immediately enter into its compute loop, and blocking the main thread at the **end** statement until the collective I/O operation is completed. We compare this approach with performing the computation and collective I/O in sequence.

In Figure 3, we compare these two approaches, and 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 and the SGI Origin 2000, this approach actually *decreases* performance, sometimes significantly. On the Intel Paragon, the performance of both approaches is about the same. This shows that merely spawning a background thread to perform the collective I/O operation is *not*, in general, sufficient to achieve high performance.

To understand these results, it is important to differentiate between *user-level* threads, where the threads are executing in user space, versus *kernel-level* threads, where the threads are managed by the kernel. With user-level threads, there is very little context and thus the cost of thread switching 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. With kernel-level threads, only the calling thread is blocked, allowing computation and I/O (or communication) to be overlapped. However, the kernel must schedule and control these threads. While the cost of managing kernel-level threads is less than that for a heavy-weight process, this cost is still greater than the cost of implementing user-level threads. For this reason, kernel-level threads are often termed *light-weight processes.*

Now consider that only *parts* of the collective I/O algorithm can be overlapped with computation. In particular, setting up and initiating communications, setting up the disk write and copying to and from message buffers

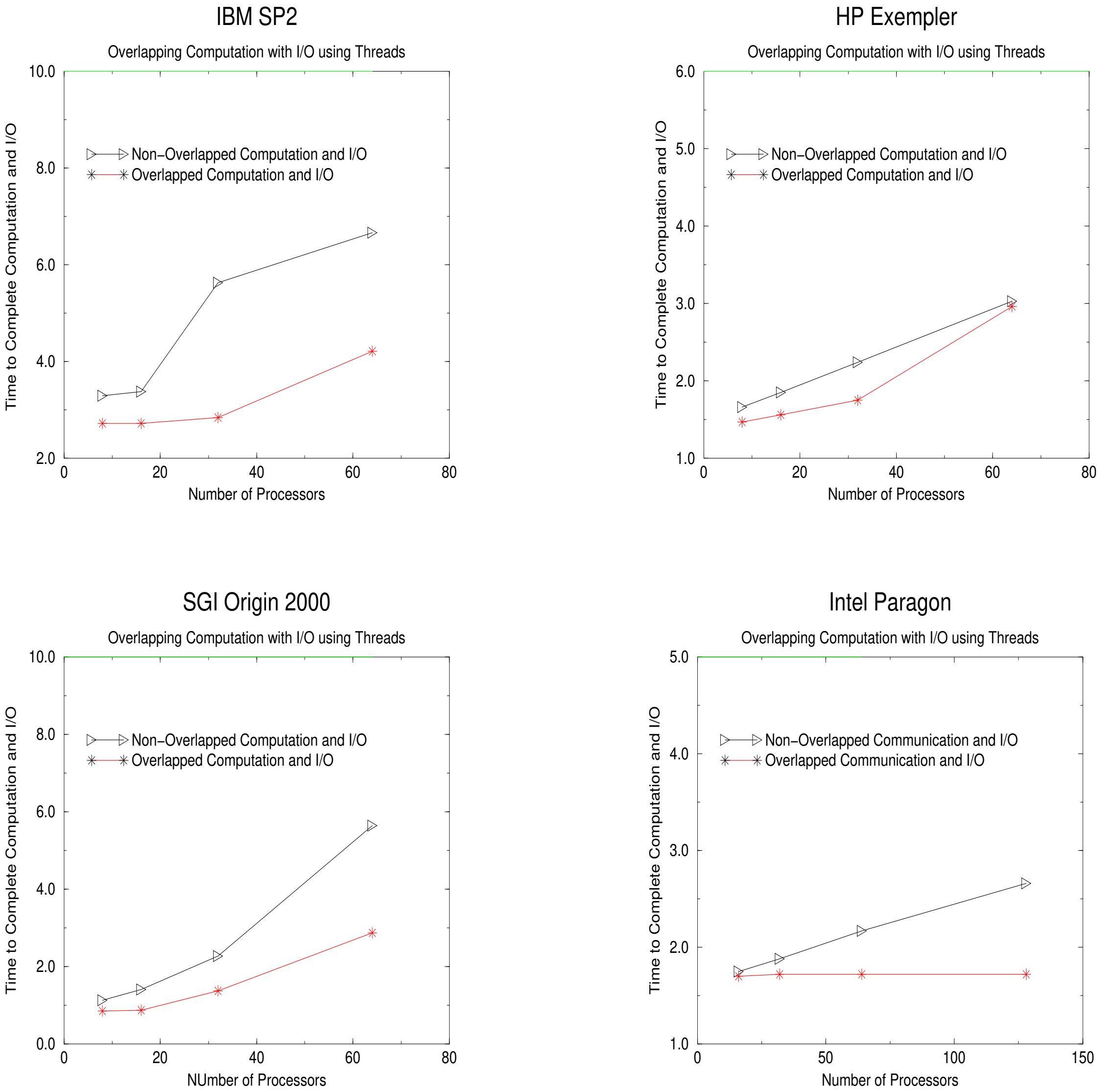

Figure 2: This figure shows the improvement in performance that is possible when (non-collective) I/O is overlapped with computation.

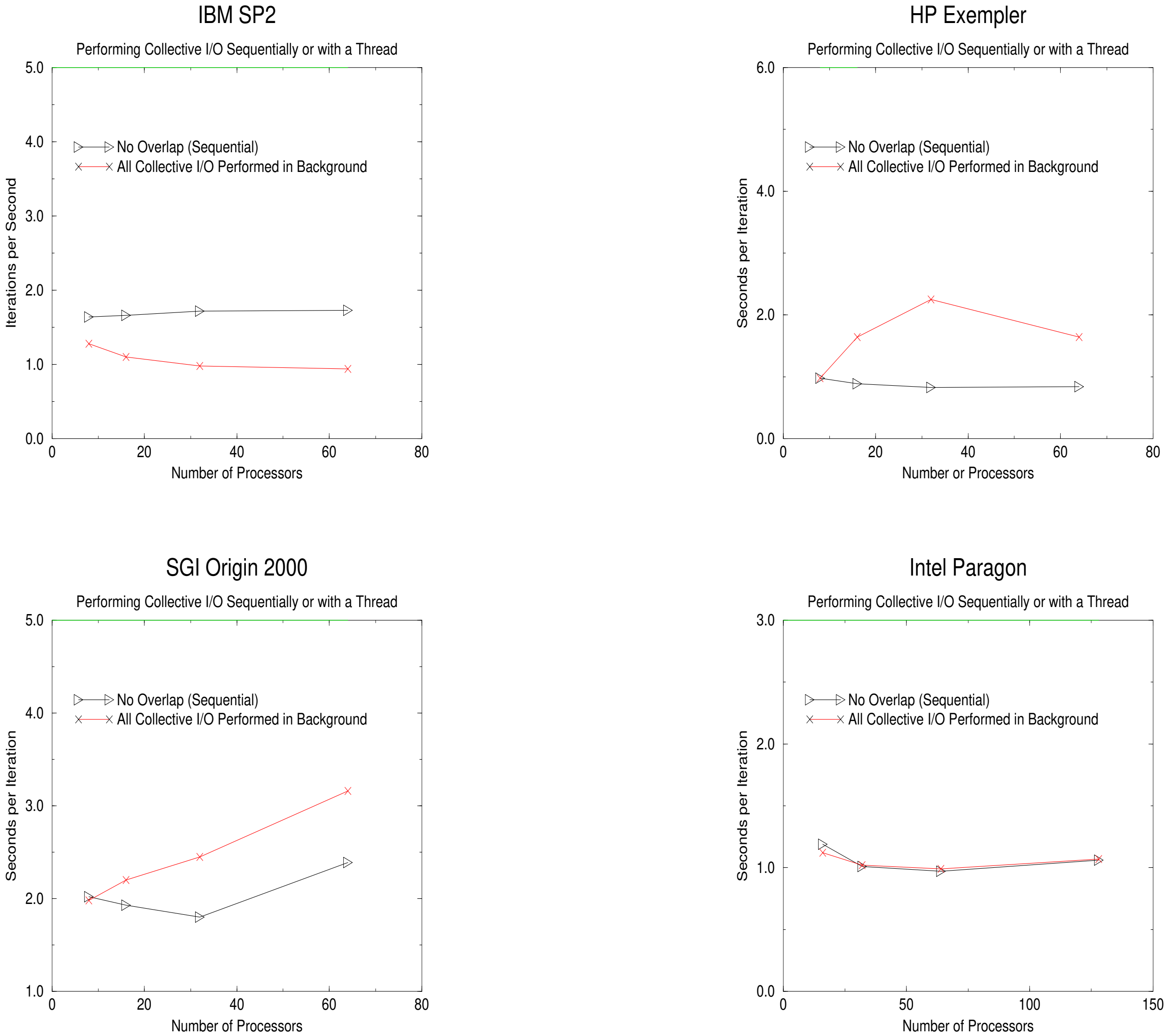

Figure 3: This figure shows the time required to complete both the computation and communication phases when the two are performed in sequence, and when a thread is spawned to execute the collective I/O routine in the background.

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. Thus there is a trade-off. When the actions taken by the I/O thread cannot be overlapped with the main thread, the two threads are competing for control of the CPU. This of course requires the multiplexing of threads on and off the CPU, incurring the relatively high costs of thread switching. As shown in this set of experiments, the performance gains obtained by overlapping (parts) of the collective I/O with computation are offset, or more than offset, by the overhead of thread scheduling and thread switching.

To reduce these costs, we modified the implementation such that the collective I/O thread 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 are performed by the *main* thread. The I/O thread is spawned to perform *only* the actual write to disk. With this approach, competition between the main thread and the I/O thread is minimized, and the overlap of computation and I/O is maximized.

Consider how this approach corresponds to the implementation of split-collective operations. When the main thread executes the **begin** statement, the initial phase of the collective I/O operation is executed by the main thread. This includes *all* of the activity required to get the data distributed among the processors 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, a thread is spawned to perform the actual write to disk and the main thread enters into its compute phase. Execution of the **end** statement 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.)

The performance of the three implementation options is shown in Figure 4. 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 provides up to a 38% improvement in performance over both of the other techniques. On the SGI Origin 2000, spawning a thread to perform only the disk write results in a 48% improvement over the sequential approach, and a 96% 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 disk write performs roughly as well as performing the two phases in sequence. However, it exhibits an 86% improvement over performing the whole collective I/O operation in the background. On the IBM SP2, both of the thread-based strategies perform at approximately the same level, and demonstrate up to an 85% 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.

3.4 Overlapping Communication with Collective I/O

Thus far, we have looked at performance when *computation* is overlapping the collective I/O operation. It is also important to consider overlapping *communication* with collective I/O. To investigate this issue, we left all of the parameters the same except for replacing the compute phase with a communication phase. In the communication phase, processor N repeatedly sends an eight kilobyte message to processor $N+1$, and receives an eight kilobyte message from processor $N-1$. All communication is synchronous. The time spent in the communication phase is again calibrated to take approximately as long as the average I/O phase with 64 processors. We are again interested in the time required to complete both the communication and collective I/O phases.

In this set of experiments, we compared the sequential implementation, where there is no attempt to overlap the communication with the collective I/O, and the implementation technique that spawns a thread to perform

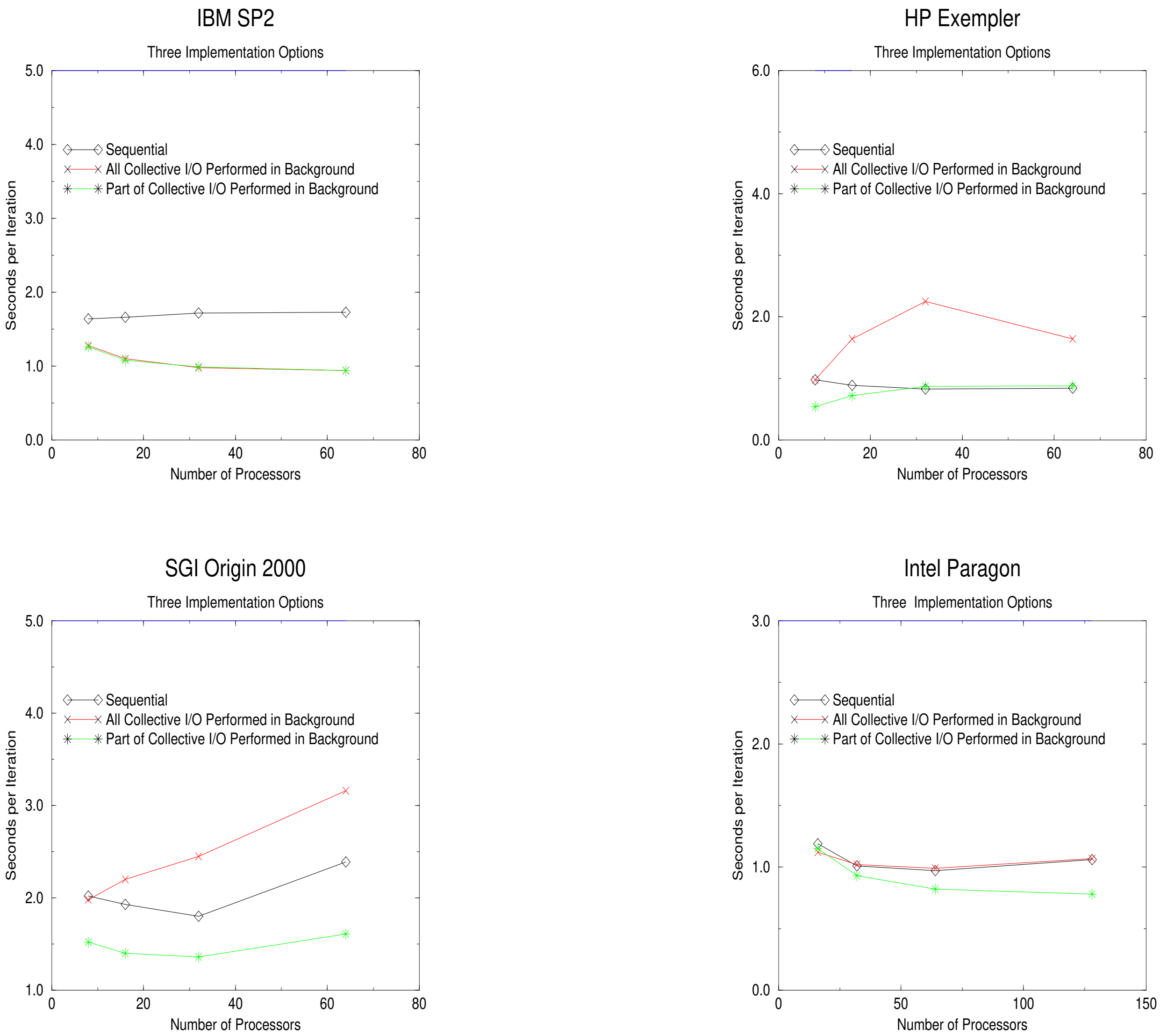

Figure 4: This figure showns the performance of the three implementation options for split-collective I/O operations.

only the actual write to disk. We note that we could not overlap the complete collective I/O operation with communication since the communication libraries on three of the four machines are not thread-safe, disallowing calls to the communication library by more than one thread.

The results of this experiment are shown in Figure 5. As can be seen, performing communication rather than computation in the main thread did have a negative impact on performance. On the SP2, the improvement in performance fell from 80% to 50%. On the SGI Origin this improvement dropped from 50% to 30%, and on the Intel Paragon the improvement in performance dropped from 38% to 20%. The reason for this decrease in performance is that the main thread and the I/O thread are competing for network resources concurrently. It is interesting to note that performance on the HP Exemplar actually improved when computation was replaced by communication, although this improvement decreases as the number of processors approaches 64. The technique of spawning a thread to perform only the write to disk appears to scale well as the number of processors, and consequently the total number of messages in the network, are simultaneously increased.

4 Related Work

The research most closely related to this project is the development of the MTIO library [17], which is a multi-threaded MPI-based 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 very similar to the results we obtained on the SP2. As noted above however, we found that the SP2 is the only architecture for which this approach performed well.

The performance of two-phase I/O on the Intel Paragon has been studied extensively by both Dickens and Thakur [8] and by Bordawekar [2]. However, neither of these studies looks at thread-based collective I/O. Also, Bordawekar [4] 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] investigate 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. The Jovian 2 I/O library is used in this study, and this library is also thread-based. However, the approach outlined in this study requires significant modifications to the application code and knowledge of future I/O requests.

There are other projects using collective I/O. For example, Passion has been extended to handle out-of-core arrays [19]. Also, a variation of the disk-directed I/O technique is used in the Panda runtime library [18]. Excellent overviews of the field of parallel I/O can be found in [9, 7, 10].

5 Discussion and Conclusions

The research presented here clearly demonstrates that it is *possible* to obtain good performance by overlapping computation with collective I/O, but it is not *automatic*. We have also shown that the most natural implementation technique, that of simply spawning a background thread to perform the whole collective I/O operation, is in general, *not* sufficient to obtain good performance. In fact, for the architectures studied here, this simple approach more often *reduces* rather than *enhances* 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

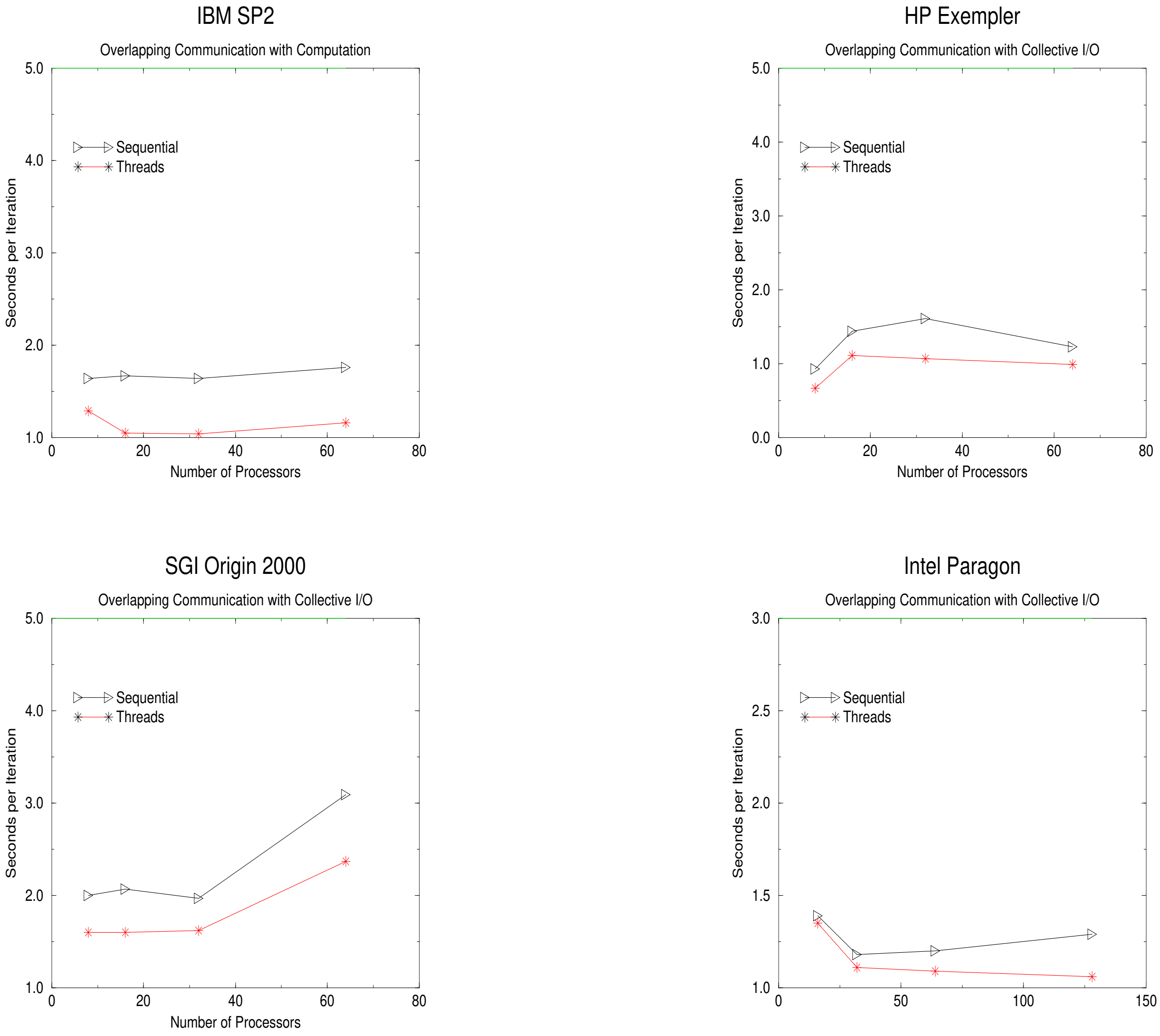

Figure 5: This figure compares the performance of executing the communication and collective I/O phases in sequence versus spawning a thread to execute the actual write to disk.

benefits of overlapping computation and I/O. However, when this competition is minimized, such as when the I/O thread performs only the actual write to disk, excellent performance gains can be obtained. It is important to note however that this whole investigation is built upon the premise that there is sufficient computation to effectively overlap computation and collective I/O. Whether this is in general true remains to be seen.

There are currently three main obstacles to this line of research. Foremost is the lack of thread-safe MPI implementations. Secondly, in most cases the user does not have the ability to set thread scheduling policies. It is possible that performing the whole collective I/O operation in the background does provide good performance *if* the priorities of the two threads can be manipulated. 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 the code between architectures to be somewhat tedious.

Current research is focusing on implementing the complete MPI 2.0 parallel I/O library, and performing this same study on important application codes.

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] R. Bordawekar. 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.

[3] Bordawekar, R., del Rosario, J. and Alok 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.

[4] 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.

[5] 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.

[6] DelRosario, J., Bordawekar, R. and Alok 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.

[7] DelRasario, J. and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. IEEE Computer, 27(3):59-68, March 1994.

[8] 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.

[9] 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.

[10] Feitelson, D., Corbett, P., Baylor, S. and Yarson Hsu. Parallel I/O subsystems in massively parallel supercomputers. In *IEEE Parallel and Distributed Technology*, 3(3):33-47, Fall 1995.

[11] 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.

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

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

[14] Kotz, D. and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Supercomputing '94* pages 640-649, November 1994.

[15] Kotz, D. Disk-directed I/O for MIMD multiprocessors. it ACM Transactions on Computer Systems, 15(1):41-74, February 1997.

[16] 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.

[17] 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.

[18] Seamons, K., Chen, Y., Jones, P., Jozwial, 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.

[19] 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.

[20] Thakur, R., Choudhary, A., More, S and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70-78, June 1996.

[21] Toledo, S. and F. Gustavson. The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28-40, Philadelphia, May 1996. ACM Press.