A Performance Study of Two-Phase I/O

Phillip M. Dickens
Department of Computer Science
Illinois Institute of Technology

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory

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 subsystem represents a significant obstacle in the way of achieving good performance. While massively parallel architectures do, in general, provide parallel I/O hardware, this alone is not sufficient to guarantee good performance. The problem is that in many applications each processor initiates many small I/O requests rather than fewer larger requests, resulting in significant performance penalties due to the high latency associated with I/O. However, it is often the case that *in the aggregate* the I/O requests are significantly fewer and larger. Two-phase I/O is a technique that captures and exploits this aggregate information to recombine I/O requests such that fewer and larger requests are generated, reducing latency and improving performance. In this paper, we describe our efforts to obtain high performance using two-phase I/O. In particular, we describe our first implementation which produced a sustained bandwidth of 78 MBytes per second, and discuss the steps taken to increase this bandwidth to 420 MBytes per second.

1 Introduction

Massively parallel computers are increasingly 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 [1, 5] 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.

The technique of *collective I/O* has been developed to better utilize the parallel I/O subsystem [2, 7, 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.

Two significant implementation techniques for collective I/O are two-phase I/O [2, 7] and disk-directed I/O [4, 6]. 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 calls, it is possible to obtain performance that approaches that of the conforming distribution.

The question then arises as to what is the best approach to implement the two-phase I/O algorithm. While much has been published regarding the performance gains using this technique, relatively little has been written about specific implementation issues, and how these issues affect performance. In this paper, we focus on the implementation issues that arise when implementing a two-phase I/O algorithm. We start by describing a very simple implementation, and go through a sequence of steps where we explore different optimizations to this basic implementation. At each step, we discuss the modification to the algorithm and discuss its impact on performance.

2 Experimental Design

We performed all experiments on the Intel Paragon located at the California Institute of Technology. This Paragon has 381 compute nodes and an I/O sub-system with 64 SCSI I/O processors, each of which controls a 4GB seagate drive.

We define the *maximum* I/O performance as each processor writing its portion of the file assuming the conforming distribution. We define the *naive* approach as each processor performing its own independent file access without any global strategy, i.e. no collective I/O. We assume an SPMD computation, where each processor operates on the portion of the global array that is located in its local memory. We study the costs of writing a two-dimensional array to disk for various two-phase I/O implementations. Our metric of interest is the percentage of the maximum bandwidth achieved by each approach. The application does nothing except make repeated calls to the two-phase I/O routine. Our experiments involved a two dimensional array of integers with dimensions 4096 X 4096 (for a total file size of 64 megabytes). We used MPI for all communications. To conserve space, we present one graph which maps the performance of each of the various approaches.

3 Experimental Results

The initial implementation of the two-phase I/O algorithm is quite simple. First, the processors exchange information related to their individual I/O requirements to determine the collective I/O requirement. Next, each processor goes through a series of sending and receiving messages to perturb the data into the conforming distribution. When a processor receives a portion of its data it performs a simple byte to 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. 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.

The cost associated with the naive implementation (no collective I/O) is the latency incurred from issuing many small I/O requests. There are four primary costs associated with the simple implementation of two-phase I/O. First is the extra buffer space required for the write buffer and the communication buffers. Second is the cost of copying data into the communication buffers, copying data from the communication buffers to the write buffer, and copying data from the local array into the write buffers. Third is interprocessor communication and fourth is the actual writing of the data to the disk.

The results for the non-collective I/O implementation is depicted in the curve labeled the *naive approach* of Figure 1. The initial implementation of the two-phase I/O algorithm is depicted in the curve labeled *Step1*. As noted, the graph depicts the percentage of the maximum bandwidth achieved by each approach given 16, 64 and 256 processors. With 16 processors, each processor must allocate a four megabyte write buffer and the messages passed between the processors are one megabyte. With these values, the time to perform the many small I/O operations is virtually the same as the extra costs associated with the two-phase approach. When we move to 64 processors however, each processor allocates a much smaller write buffer (one megabyte), and the messages between the processors are much smaller (131072 bytes). Thus its relative performance is improved. With the naive approach however the additional processors are all issuing many small I/O requests resulting in significant contention for the IOPs and communication network. With 64 processors, this very simple implementation of the two-phase I/O algorithm improves performance by a factor of 3. With 256 processors, it improves performance by a factor 40.

3.1 Step2: Reducing Copy Costs

As noted above, two-phase I/O requires a significant amount of copying. For this reason, we would expect modifications that reduced the cost of copying data would have a significant impact on performance.

The next step then is to change all of the copy routines to use the **memcpy()** library call whenever possible. This optimization, labeled as *Step2* in Figure 1, has a tremendous impact on performance. As can be seen, with the optimized copying routine this implementation of the two-phase I/O algorithm outperforms both the naive approach and the initial implementation. The improvement in performance over the initial implementation varies between a factor of 1.7 and a factor of 15. The reason for such a tremendous difference is that **memcpy()** uses a block move instruction, and requires only four assembly language instructions per *block* of memory. Performing a byte for byte copy requires 20 assembly language instructions *per byte*.

3.2 Step3: Asynchronous Communication

The next step we investigated was performing asynchronous rather than synchronous communication. In this implementation, a 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 into the write buffer. The processor then waits for all of the messages it needs to receive, and copies the data into the write buffer as they arrive. It then waits for all of the send operations to complete and performs the write.

The trade-offs in this implementation are quite interesting. In previous versions of the algorithm, a processor would alternate between sending a message and waiting to receive a message, blocking until a *specific* message arrives before posting its next send. The advantage is that the processor frees the send and receive buffers immediately, thus maintaining only one communication buffer at any time. With reduced buffering requirements the cost of paging is decreased.

There are two advantages to using asynchronous communication. First, the underlying system can complete *any* message that is ready without having to wait for a *particular* message. Second, it can perform the copying between its local array and its write buffer with the asynchronous communication going on in the background. The primary drawback of this approach is that each asynchronous request requires its own communication buffer. Thus the cost of buffering is considerably higher reducing performance.

The results are shown in the curve labeled *Step 3* in Figure 1. With 16 processors, the asynchronous approach further improves performance by approximately 20%. With 64 processors however, this improvement over the

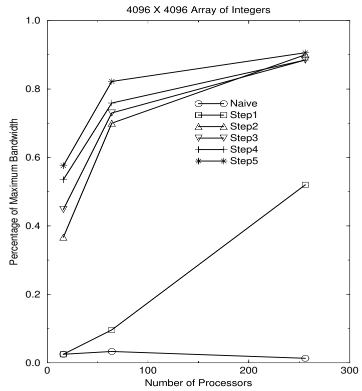

Figure 1: Improvement in performance as a function of the implementation and the number of processors.

previous implementation is reduced to approximately 5%. There is no noticeable improvement with 256 processors. The reason for the decrease in performance with 64 and 256 processors is that as the number of processors increase, the time required to perform the actual write to disk begins to dominate the costs of the algorithm. Thus using asynchronous communication is less important than it is with a smaller number of processors.

3.3 Step4: Reversing the Order of the Asynchronous Communication

The next approach is to reverse the order of the asynchronous sends and receives. Thus a processor first posts all of its asynchronous receives and then posts its asynchronous sends. The idea behind this optimization is that MPI communications are generally much faster if the receive has been posted (and thus a buffer in which to receive the message has been provided) before the message is sent [3]. Thus pre-allocating all of the receive buffers before the corresponding sends are initiated should improve the interprocessor communication costs. There is of course the same issue discussed above: pre-allocating all of the communication buffers can result in increased paging activity.

The results are given in the curve labeled *Step4* in Figure 1. With 16 processors, reversing the order of sends and receives provides a 15% improvement over the previous approach. Again this improvement in performance decreases as the number of processors is increased. With 64 processors there is an improvement of approximately 5%, and the improvement disappears with 256 processors. This is again due to the fact that the write time begins to dominate the cost of the algorithm as the number of processors increases.

3.4 Step5: Combining Synchronous with Asynchronous Communication

The final optimization we pursued was to combine asynchronous receives with synchronous sends. The idea behind this optimization is that posting all of the receive buffers will improve communication costs, and releasing the send buffer after each use will reduce the buffering costs. The results are shown in the curve labeled *step5*. This implementation results in performance gains of up to 10%.

3.5 Scalability of Two-Phase I/O

For completion, we examine the behavior of these two-phase I/O implementations as the amount of data being written increases and the number of processors remains a constant 256. We looked at total file sizes of 16

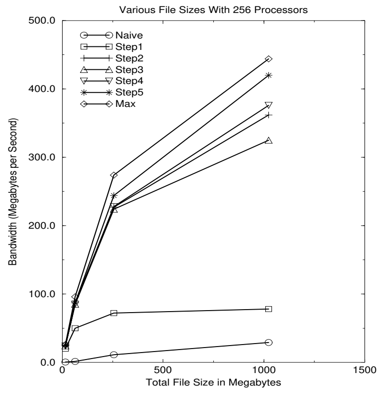

Figure 2: The bandwidth achieved by each approach as the file size is increased and the number of processors is a constant 256.

megabytes, 64 megabytes, 256 megabytes and 1 gigabyte. The results are shown in Figure 2.

In this figure, the bandwidth achieved by each approach is given as a function of the total file size. For comparison, the maximum bandwidth is also shown. As can be seen, neither the naive approach nor the first two-phase implementation scale well. It is interesting to note that step3, where all communication is asynchronous and the sends are posted before the receives, performs more poorly in the limit than does step2 where all communication is synchronous. Also in the limit there is a small improvement in performance between step2 and step4 and both approaches appear to scale relatively well. The combination of asynchronous receives and synchronous sends scales very well, and, in the limit, approaches the optimal performance.

It is important to note that in the limit the initial implementation of two-phase I/O resulted in a bandwidth of 78 megabytes per second. The final implementation resulted in a bandwidth of 420 megabytes per second.

4 Conclusions

In this paper, we investigated the impact on performance of various implementation techniques for two-phase I/O, and outlined the steps we followed to obtain high performance. We began with a very simple implementation of two-phase I/O that provided a bandwidth of 78 megabytes per second, and ended with an optimized implementation that provided 420 megabytes per second. During the course of this analysis, we provided a good look at the trade-offs involved in the implementation of a two-phase I/O algorithm. Current research is aimed at extending these results to other parallel architectures such as the IBM SP2.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

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