Implementation and Evaluation of Prefetching in the Intel Paragon Parallel File System [*]

Meenakshi Arunachalam   Alok Choudhary   Brad Rullman [†]
ECE and CIS
121 Link Hall
Syracuse University
Syracuse, NY 13244
E-mail: {meena@top.cis.syr.edu, choudhar@cat.syr.edu and brad@ssd.intel.com}

Abstract

The significant difference between the speeds of the I/O system (e.g. disks) and compute processors in parallel systems creates a bottleneck that lowers the performance of an application that does a considerable amount of disk accesses. A major portion of the compute processors' time is wasted on waiting for I/O to complete. This problem can be addressed to a certain extent, if the necessary data can be fetched from the disk before the I/O call to the disk is issued. Fetching data ahead of time, known as prefetching in a multiprocessor environment depends a great deal on the application's access pattern. The subject of this paper is implementation and performance evaluation of a prefetching prototype in a production parallel file system on the Intel Paragon. Specifically, this paper presents a) design and implementation of a prefetching strategy in the parallel file system and b) performance measurements and evaluation of the file system with and without prefetching. The prototype is designed at the operating system level for the PFS. It is implemented in the PFS subsystem of the Intel Paragon Operating System. It is observed that in many cases prefetching provides considerable performance improvements. In some other cases no improvements or some performance degradation is observed due to the overheads incurred in prefetching.

1 Introduction

Input-Output for parallel systems has drawn increasing attention in the last few years as it has become apparent that I/O performance rather than CPU or communication performance may be the limiting factor in future computing systems. A large number of applications in diverse areas such as large scale scientific computations, database and information processing, hypertext and multimedia systems, information retrieval etc. require processing very large quantities of data. For example, a typical Grand Challenge Application at present could require 1Gbyte to 4Tbytes of data per run [2]. These figures are expected to increase by orders of magnitude as teraflop machines become readily available. Unfortunately, the performance of the I/O subsystems of MPPs has not kept pace with their processing and communication capabilities. A poor I/O capability can severely degrade the performance of an entire application.

The focus of this paper is parallel file systems in parallel computers, and specifically read-ahead and prefetching. The goal of prefetching is essentially to access data from disks anticipating future use of the data. In a parallel file system a file is normally striped across a large number of I/O nodes and disks, and several processing nodes concurrently access the files. Thus, prefetching strategies that may work reasonably well for sequential files in uniprocessor environments may not be extended in a straightforward manner because access patterns seen by the I/O nodes are interleaved accesses of many compute nodes.

In [4, 5] Kotz and Ellis have concluded that employing prefetching results in a definite gain in the read throughput as seen by the user, but the issues involved in prefetching in a parallel machine have to be carefully analyzed. A two-phase access strategy used at run-time has been proposed in [1] that shows that the performance of the I/O subsystem can improve significantly if the storage distribution on the disks is decoupled from the data distribution on the computational nodes. Galbreath et al.[3] argue that the abstraction of parallel I/O routines can enhance development, portability, and performance of I/O operations in applications.

In this paper, we describe the design and implementation of a prefetching prototype on the Paragon Parallel File System (PFS). It should be noted that the implementation of the prefetching strategy is done in a real and production parallel file system by modifying the code of the file system. Thus, all the overheads that come with the production software are also present in the prefetching software, which are normally ignored in simulations and other studies. The prototype has been extensively tested. In this paper, we present initial experience and performance studies from this implementa-

[*]This work was supported by an NSF Young Investigator Award CCR-9357840 and Intel Corp.

[†]Brad Rullman is with Intel Scalable Systems Division.

tion. The results show that in many situations it is beneficial to perform prefetching, and in some situations prefetching does not provide any benefits.

The rest of the paper is organized as follows. A brief description of the architecture of the Intel Paragon system, the PFS and the PFS I/O modes is presented in Section 2. This is followed by the implementation details of the prefetching prototype in Section 3. The performance measurements and evaluation of the prototype are presented and analyzed in Section 4. Conclusions and future work are presented in Section 5.

2 Intel Paragon and the PFS

The Intel Paragon is a massively parallel supercomputer with a large number of nodes connected by a high-speed mesh interconnect network. Each node is populated with an i860 processor (SMP nodes are available with three i860 processors) and 16 MBytes or more of memory. The nodes can operate individually or as a group to run a parallel application in the MIMD fashion. The nodes are classified into service nodes, compute nodes and I/O nodes. This classification is based on the functionality of the nodes in a system. Service nodes are used for interactive processes, compute nodes run compute-intensive applications and I/O nodes manage the system's disk and tape drives, network connections, and other I/O facilities.

The PFS is designed to provide high bandwidth necessary for parallel applications. This is accomplished by striping the files across a group of regular Unix File Systems (UFS) which are located on distinct storage devices, and by optimizing access to these file systems for large transfers. Any number of PFS file systems may be mounted in the system, each with different default data striping attributes and buffering strategies. Stripe attributes describe how the file is to be laid out via parameters such as the stripe unit size (unit of data interleaving) and the stripe group (the I/O node disk partitions across which a PFS file is interleaved). Currently supported buffering strategies allow data buffering on the I/O nodes to be enabled or disabled.

When buffering is disabled, a technique called *Fast Path I/O* is used to avoid data caching and copying on large transfers. The file system buffer cache on the Paragon OS server is bypassed, as is the client-side memory mapped file support used by default in the UFS file systems. Instead, Fast Path reads data directly from the disks to the user's buffer, and writes from the user's buffer directly to the disks. Also, file system block coalescing is done on large read and write operations, which reduces the number of required disk accesses when blocks of the file are contiguous on the disk.

The Paragon PFS provides a set of file sharing modes (Figure 1) for coordinating simultaneous access to a file from multiple application processes running on multiple nodes. These modes are essentially *hints* provided by the application to the file system which indicate the type of access that will be done. These hints allow the file system to optimize the I/O accesses based on the desired file layout, the degree of parallelism, and the level of data integrity required. The I/O mode can be set when a file is opened, and the application can also set/modify the I/O mode during the course of reading or writing the file.

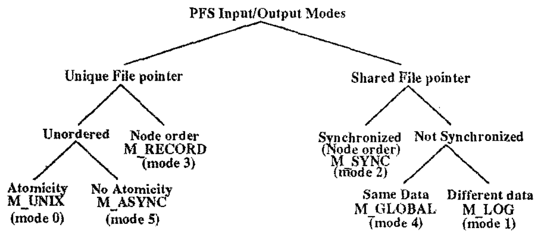
PFS Input/Output Modes
Unique File pointer
Shared File pointer
Unordered
Node order M_RECORD (mode 3)
Atomicity M_UNIX (mode 0)
No Atomicity M_ASYNC (mode 5)
Synchronized (Node order) M_SYNC (mode 2)
Not Synchronized
Same Data M_GLOBAL (mode 4)
Different data M_LOG (mode 1)

Figure 1: Paragon Parallel File System I/O modes

Figure 2 displays the read performance of most of the various PFS I/O modes supported by the PFS. These results were obtained on a Paragon with 8 compute nodes and 8 I/O nodes, with all compute nodes reading a single shared file. Each I/O node was configured with a single SCSI-8 card and RAID array; it should be noted that SCSI-16 hardware is also available that effectively quadruples the bandwidth available on each I/O node. In the graph, data for the "Separate Files" case is also presented for comparison with the I/O mode data; in this case each compute node accesses a unique file rather than opening a shared file. The prefetching prototype described in the remainder of this paper was implemented using the M_RECORD mode. This mode was chosen because it is well suited for the SPMD programming model, in which applications performing an extensive amount of I/O usually distribute the data equally among the I/O nodes for load-balancing and concurrency.

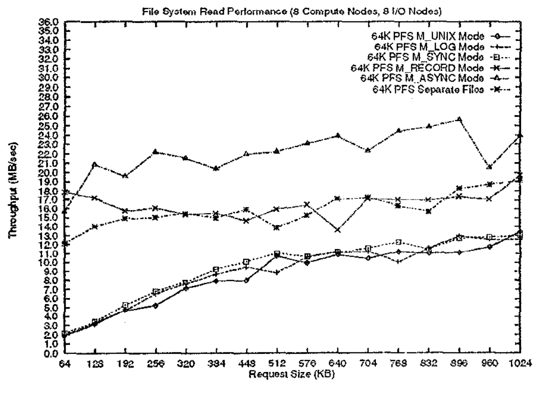

Figure 2: Read Performance of the PFS I/O Modes

3 Prefetching Prototype Implementation

A read request to a PFS file can be issued either as a blocking (synchronous) or a non-blocking (asynchronous) request. There are two main phases for any asynchronous request and they are, the *request setup* and *posting phase.* During the setup phase, the incoming request for read is allocated an internal structure for tracking the state of request during the asynchronous processing. A pointer to this structure then resides in the list of pointers maintained for active asynchronous requests issued by the user. Associated with each request structure is an asynchronous request thread (ART). The ART will concurrently post and process the user's I/O request while the user thread is performing other operations. The input parameters of the asynchronous read operation [6] are passed to the ART and the ART is initialized. Once the ART is initialized, it begins processing asynchronous requests that are queued in a FIFO manner on the active list. The read request to the disk is itself performed by the ART using the Fast Path I/O technique described earlier. The data is read directly from the disk to the user's buffer.

The prefetch requests are implemented like the asynchronous read requests making use of the existing support for asynchronous read requests in the Paragon OS. They are issued as asynchronous requests by the user thread following any read request to a PFS file. The prefetching requests are dynamic in nature and totally driven by the application's access requests. Details about when and where to prefetch is derived from the read request from the application. The prefetch request is issued in anticipation of another read request issued by the same user thread on the same file. The prototype prefetches only one block of data it anticipates will be needed for the future read request. A read prefetch request is issued from the client-side of the Paragon OS for every read request that is issued by the user. Once the asynchronous request is done, the data that has been read is stored in a buffer along with other details such as the PFS file offset, the size of the data in bytes etc. This prefetch buffer structure is part of a list of all the prefetch buffer structures of data that have been prefetched from that particular file. The prefetch buffer list is a part of the internal structure of that file. When the file is opened newly by a process, the prefetch list gets initialized and as the read requests come in, new prefetch buffers are added onto the list. Memory for the prefetch buffers is allocated in the compute node. At the time the process closes the file, all the prefetch buffers are freed and the prefetch buffer list points to a null pointer.

All the individual file pointers are required to point to the same location before a read request is issued in any of the PFS I/O modes. Before processing the read request, the Paragon OS sets the individual file pointers from the nodes to point to the starting locations of separate areas in the file. Once the reading is complete, all the node file pointers point to one common location which is the end of all the separate areas. In the case of prefetching, the data from a read request is stored in a prefetch buffer which is located in the memory of the compute node. Also, the file pointer is not changed in the process of prefetching. This is done to present a file system image that is consistent with the application's read requests.

4 Performance Measurements and Evaluation of the Prototype

The performance measurements and evaluation of the prefetching implementation was performed on a Paragon consisting of 8 compute nodes and 8 I/O nodes each with 32 MB of memory. The default block size was 64KB and default stripe factor 8.

The workload programs opened files in the M_RECORD mode. Delays were introduced between I/O accesses in this synthetic workload to simulate the computation phases of a program. To measure the performance of our prefetching prototype, the workload performed extensive I/O on large files.

The PFS file system block size is the basic unit of transfer between the file system and the storage device. Read and write request sizes that are a multiple of the block size and start at a file system block boundary provide greater throughput than those request sizes that are not a multiple of the file system block size. This is because there is a higher overhead involved in creating temporary buffers for the size of the "partial" blocks and copying only the necessary data from a transferred block.

The stripe unit size along with the stripe factor determines how a particular request gets directed to the I/O nodes as shown in Figure 3. If the request size sz is larger than the stripe unit size su, then the first of the sz/su requests go to the first I/O node and the second of the sz/su requests to the second I/O node and so on.

When an application is executed with prefetching at the system-level, the read access time for a block as observed by the application may be less than the actual disk access time for reading that block. This is due to the fact that prefetching makes the read access time appear less than it actually is by reading the block before the read request was issued. This is exactly what we would like to achieve with prefetching. Essentially, we would like to prefetch those blocks that will be used by the application in the near future. When a prefetched block is used to serve a future request from the application, we say that there is a *hit* on that block.

Although hit_ratio serves as a good measure of performance in a sequential program, in a parallel programming model, overall read bandwidth seen by an application is a better measure of performance. This is because, a collective I/O (refers to I/O by all the nodes on which an application program executes, e.g. reading one column (distinct) each of a matrix by all processors) request is considered complete

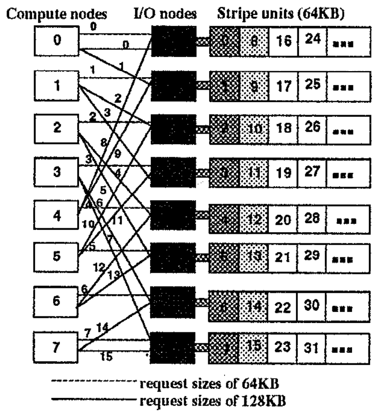

Figure 3: Declustering of Compute node requests to the I/O nodes

when the individual I/O requests of all the nodes have been satisfied. The read bandwidth is the total amount of data that can be read by all the nodes per unit time as observed by the application. For a parallel I/O mode like M_RECORD, the numerator would be the amount of data read by all the compute nodes and the time taken is the time taken by a compute node to complete all the read calls. Another important measure to consider is the amount of overlap of I/O with computation. For example, even if at the time of a read request, the data is not available in the prefetch cache (miss when the request is presented), if most of the read is already done, the performance benefits can be tremendous. Finally, the prefetching benefits should be equally distributed amongst the processors in order to see an overall benefit.

4.1 Prefetching for I/O Bound Applications

This experiment generates the I/O workload of an application which does not perform any computation between the I/O calls. An example of this would be where an application issues calls to access one block after another without any computation between these calls. This workload represents the behavior of applications that perform initial reads from files before commencing computation. As our prefetching prototype dynamically issues a prefetch request for only one block from the information of a current read request issued by the user, the benefits from prefetching in this kind of application are not significant as shown in Table 1. There are no significant differences between the read bandwidths with and without prefetching. The prefetch request aimed to serve the future I/O request call does not have a significant head start and hence the application takes the same time to complete all its read accesses and hence no change is observed in the read bandwidth. The read bandwidths for the prefetching case are comparable with the non-prefetching case in all the block sizes except for 64KB size. This is

Request size (KB) per node	File size (MB)	Read (MB/sec) (MB/sec) (no prefetching)	Read (MB/sec) (MB/sec) (prefetching)
64	128	15.7567	13.0933
128	128	16.9066	16.5867
256	128	16.0100	16.9600
512	128	15.9767	15.3334
1024	128	19.2133	19.5967
64	256	15.8700	12.5667
128	256	16.4300	16.9800
256	256	16.2334	16.6067
512	256	15.5767	18.8576
1024	256	17.9967	19.6433

Table 1: PFS Read Performance with and without Prefetching: stripeunit size=64KB stripegroup=8

due to the overhead involved in prefetching. The prefetched data is copied into the prefetch buffer present in the system and from there is copied into the user buffer which is not available until the user makes a read request. The prefetching overhead is more pronounced when the request sizes are smaller as the time to complete a read is also less for these requests.

4.2 Prefetching for Balanced Applications

To study the benefits of prefetching, we use balanced workloads that perform significant amounts of I/O interleaved between computations. In [5] such applications are termed as "balanced". To simulate computation for each block read, delays were introduced between consecutive reads. Figures 4 and 5 summarize the results for file size of 128MBytes when delays are introduced between successive read requests. The computation times between the I/O requests ranged from 0.001 second to 0.1 second. This range of delays represents from "no overlap" to "complete overlap" between computation and I/O for a subset of parameters. Figure 4(A-C) demonstrates that when overlap between I/O and computation is present, significant performance improvements can be obtained.

Thus, the closer to completion a prefetch request is, the lesser time a hit read request will have to wait for the prefetching to complete. We also notice that as the request size increases, the time taken for a prefetching request will also increase and only with a corresponding increase in computation times can we expect a proportional rise in the observed bandwidth. Table 2 gives the minimum read access times for the various request sizes. These times determine how much overlap will occur between computation and I/O. For example, for a request size of 1024KB, it takes 0.42sec to complete a read request. Thus, a delay of 0.1sec is not enough to provide any significant overlap. This fact is clearly evident from Figure 5(D and E), where the request sizes are 512KB and 1024KB per processor, and therefore, the read time itself is so large that no significant overlap

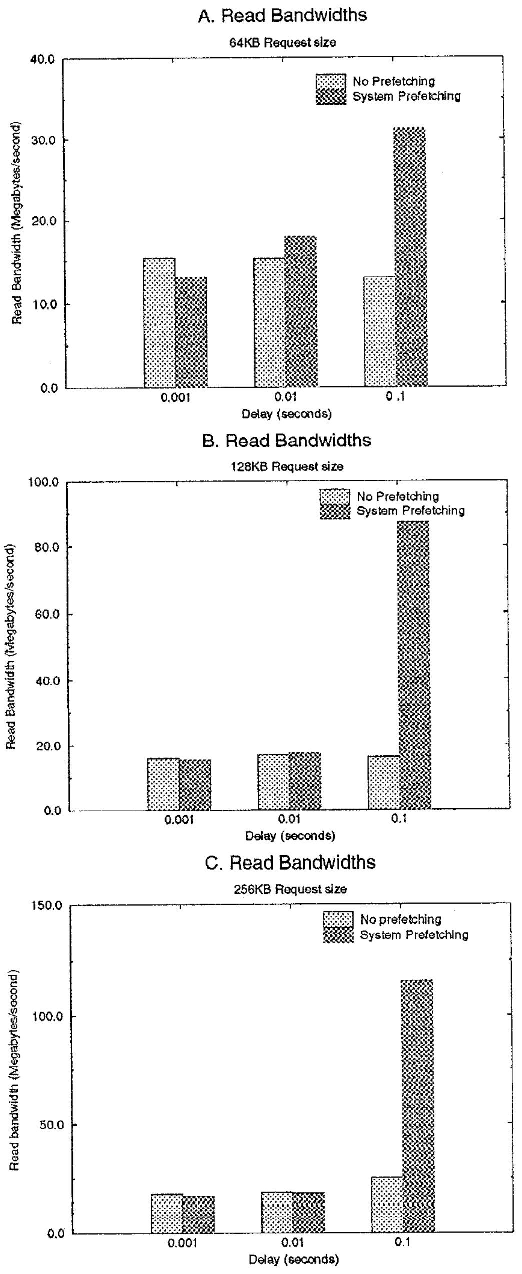

Figure 4: PFS Read Performance for Balanced Workloads for 64KB, 128KB, 256KB Request Sizes- File Size (128MB)

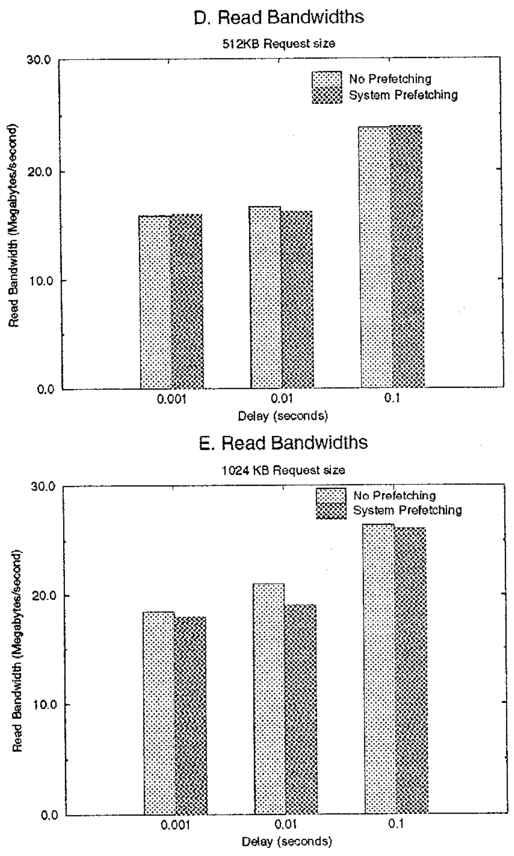

Figure 5: PFS Read Performance for Balanced Workloads for 512KB, 1024KB Request Sizes- File Size (128MB)

takes place with the computation. Thus, no performance gains are observed. The relation between the time taken to

Request Size (KB)	Read Access Time (sec)
64	0.0317
128	0.0592
256	0.1249
512	0.2504
1024	0.4164

Table 2: Read Access Times for Various Request Sizes

satisfy a prefetch read request by accessing a disk and the time spent on computation between the consecutive read requests determines what percentage of I/O can be overlapped with computation. The read request size also determines how long a prefetch read access will take and this time can be used to overlap computation with I/O in order to reduce the I/O latency.

4.3 Prefetching for Various Stripe units.

Table 3, summarizes results for varying stripe units with prefetching. Given that no delay was introduced between requests, the results are consistent with the no prefetching case. For smaller request sizes, the throughputs are less than the throughputs of the no prefetching case due to the prefetching overhead.

Request size (KB) per node	File size (MB)	Read B/W (MB/sec) su=64KB	Read B/W (MB/sec) su=256KB	Read B/W (MB/sec) su=1024KB
64	128	13.0933	13.085	13.0567
128	128	16.5867	14.6933	14.9133
256	128	16.9600	16.0967	16.3100
512	128	15.3334	15.2100	13.9034
1024	128	19.5967	13.035	16.9800
64	256	12.5667	12.5800	12.2933
128	256	16.9800	14.2433	14.2200
256	256	16.6067	14.8633	14.8633
512	256	18.8576	15.5067	14.3067
1024	256	19.6433	17.4133	17.9900

Table 3: PFS Read Performance with prefetching for different Stripe unit sizes

4.4 Prefetching for Different Stripe groups.

The measurements were obtained using two sets of stripegroups, namely striping across all 8 nodes and striping 8 ways across 1 node. The results are given in Table 4. With prefetching, we observe a maximum speedup by a factor of 7.7. Again, no delays were introduced between requests. Due to the prefetching overhead which is more pronounced when the read request sizes are small, the speedup is less than the no prefetching case for 64KB.

Request size (KB) per node	File size (MB)	Read B/W (MB/sec) R1 sgroup=1	Read B/W (MB/sec) R2 sgroup=8	Speedup R1/R2
64	128	2.1733	13.0933	6.0
128	128	2.4833	16.5867	6.7
256	128	2.5333	16.9600	6.7
512	128	2.4533	15.3334	6.3
1024	128	2.9900	19.5967	6.6
64	256	2.1767	12.5667	5.8
128	256	2.4233	16.9800	7.0
256	256	2.4467	16.6067	6.8
512	256	2.4500	18.8576	7.7
1024	256	2.9233	19.6433	6.7

Table 4: PFS Read Performance with Prefetching for different Stripe groups, Number of Nodes =8

5 Conclusions and Future Work

In this paper we presented a design and implementation of a prefetching prototype in the Intel Paragon Parallel File System. Although the PFS on the Paragon provides several access modes and file sharing mechanisms, we selected the M_RECORD mode for our implementations because it is a highly parallel mode. Furthermore, this mode seems to be a preferred choice for a large number of users because it provides consistency as well as high-performance.

We presented initial performance results for the file systems, which demonstrate that the file system performance is scalable. The access bandwidth seen by the user when using prefetching is also scalable, and given a reasonable overlap between computation and I/O, the benefits from the system prefetching can be very significant. Given that in the normal mode of operation (without prefetching), the data is directly transferred into the user's buffer, while in the system level prefetching the data is buffered, performance with prefetching is comparable when there is no overlap of I/O with computation.

In any such implementation in the system software, there are a large number of parameters that can be studied and need to be evaluated. As a part of the future work, we plan to evaluate the performance of prefetching on much larger systems and study the performance for a greater variety of workloads and access patterns. Furthermore, we plan to implement prefetching in other file I/O modes.

References

[1] Juan Miguel del Rosario, Rajesh Bordawekar, Alok Choudhary. Improved Parallel I/O via a Two-phase Run-time Access Strategy, *Workshop on Parallel I/O, International Parallel Processing Symposium*, pp.56-69, April 1993.

[2] Juan Miguel del Rosario, Alok Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects, *IEEE Computer*, March 1994.

[3] N. Galbreath, W. Gropp, D. Levine. Applications-Driven Parallel I/O,*Proceedings of Supercomputing' 94*.

[4] David Kotz, Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems,*Distributed and Parallel Databases Vol.1*, pp.33-51, 1993.

[5] David F. Kotz, Carla Schlatter Ellis. Prefetching in File Systems for MIMD Multiprocessors, *IEEE Transactions for Parallel and Distributed Systems, Vol.1*, pp.218-230, 1990.

[6] *Paragon OSF/1 User's Guide, Intel Supercomputer Systems Division.*