Efficient Parallel I/O Scheduling For Clusters

*** submitted to Workshop on Computer Systems ***

Lien-Wu Chen Jan-Jan Wu

Institute of Information Science
Academia Sinica
Taipei 115
Taiwan, R.O.C.
E-mail: {lwchen,wuj}@iis.sinica.edu.tw
phone: (02)2788 3799
fax: (02) 2782 4814

Abstract

This paper studies scheduling of parallel I/O operations in cluster environments. Due to dynamic nature of clusters, fault tolerance has been a crucial issue in executing data-intensive applications on clusters of workstations/PCs. One key strategy for enhancing fault tolerance is data replication, that is, replicating multiple copies of a data file and striping them over multiple disks. Parallel I/O scheduling aims to reducing finish time of a batch of parallel I/O operations by finding an optimal sequence in executing these I/O oprations. Most existing works in parallel I/O scheduling, however, have not taken data replication into consideration.

In this paper, we study parallel I/O scheduling for systems that provide data replication. The main contribution of this paper is a complexity analysis of the scheduling problem and a set of scheduling algorithms. We show that finding an optimal schedule for systems that provide data replication is NP-complete, and then we propose a set of heuristic algorithms. We evaluate the I/O performance of these algorithms using a software simulator and report our experimental results.

Keywords: parallel I/O, scheduling, data replication, clusters, heuristic algorithms.

1 Introduction

Parallel processing has been an effective vehicle for solving large scale, computationally intensive problems. In the past decades, significant research efforts have been devoted to exploiting parallelism and effective mapping of computation problems to parallel computing platforms so as to maximize performance of the parallel programs. However, while the speed, memory size, and disk capacity of parallel computers continue to grow rapidly, the rate at which disk drives can read and write data is improving much more slowly. As a result, the performance of carefully tuned parallel programs can slow down dramatically when they read or write files. As the gap between improvement of processor speed and that of disk drive's becomes larger, the performance bottleneck is likely to get worse.

Parallel input and output techniques can help solve this problem by creating multiple data paths between memory and disks, that is, exploiting parallelism in the input and output system. To develop promising solutions for parallel I/O requires novel technologies in hardware design, understanding of application program's I/O behavior, and most importantly, software techniques for management of I/O resources and efficient implementations of parallel I/O operations. Numerous efforts in attempting to provide suitable hardware support have emerged in a number of standard storage technologies (e.g. RAID [7, 22], distributed RAID [15], network attached storages [16, 17], active disks [1, 23], and cluster parallel I/O system). Sophisticated tools for analysing program's I/O behavior have also become available in the past few years (e.g. Vesta [2], Charisma [20]). Research in software support and optimization strategies for efficient implemenations of parallel I/O operations have not received much attention until recently.

One active research area in software support for parallel I/O is parallel file systems. PIOUS [18], VIP-FS [11], Galley [21], PPFS [12] and VIPIOS [3], to name a few, are popular parallel file systems. However, each of these lacks one or more of the features desired for parallel applications running on cluster parallel systems: collective I/O, special consideration for slow message passing, and minimized data transfer over the network. More recent parallel file systems (such as PVFS [4, 25]) and parallel I/O libraries (such as Panda [26, 27] and PASSION [28]) that are designed for network of workstations/PCs have provided collective I/O [28, 27].

The performance of collective I/O and general parallel I/O operations is dominated by how fast data transfers between processing nodes and disks are performed. Serveral optimizations for reducing data transfer time for parallel I/O have been proposed in the past few years. The two-phase I/O optimization [24] reduces disk access time by breaking an I/O operation into two phases: inter-processor data exchange through the network, and bulk accesses to the disks. The Panda I/O library exploits data locality by choosing proper placement of I/O servers [6]. Parallel prefetching and caching strategies were proposed in [14, 29] to improve I/O performance. Several algorithms were proposed for scheduling parallel I/O operations to minimize the completion time of a batch of I/O operations [13]. In this paper, we focus on the parallel I/O scheduling problems.

In prior works, the I/O scheduling problem was modeled by a bipartite graph. Dubhashi, et. al. [8] and Durand, et. al. [9] proposed various bipartite graph edge-coloring algorithms for solving the scheduling problems. Jain, et. al. [13] proposed edge-coloring-based approximation algorithms for scheduling I/O transfers for systems that only allow at most k transfers at a time. Narahari, et. al [19] investigated network contention in parallel I/O transfers on mesh networks.

All prior works mentioned above do not take data replication into consideration. Data replication is commonly used in executing data-intensive applications in cluster environments for two reasons. First, it is typical for a data-intensive application to take a long period of time to complete its execution. Failure of any disk will cause lost of data and thus faults in program execution. Data replication is necessary to ensure fault tolerance. Secondly, clusters usually lack dedicated I/O servers. Instead, a subset of processing nodes are chosen to do part-time I/O services (that is, these nodes switch between computing and I/O). Since cluster environments are usually highly dynamic, some processing nodes (including part-time I/O nodes) may leave during execution of an application program due to heavy load demands from other jobs. Data replication is an effective way to ensure availability of data.

The only work we have noticed that takes data replication into consideration is by Chen and Majumdar [5]. The authors proposed the *Lowest Destination Degree First* (`LDDF`) heuristic algorithm for scheduling a batch of I/O operations. Their model only allows data transfers with uniform costs, which we refer to as `UniIO`.

In this paper, we study the problem of scheduling parallel I/O operations on systems that provide data replication. We consider a more general model where data transfers may have non-uniform costs, which we refer to as `VarIO`. The main contribution of this paper is a complexity analysis of the scheduling problem and a set of scheduling algorithms. We show that finding an optimal schedule for systems that provide data replication is NP-complete, and then we propose a heuristic algorithm, *Lowest Combined Degree First* (`LCDF`), for `UniIO`, and a set of heuristic algorithms for `VarIO`. We compare the I/O performance of these algorithms using a software simulator. We show that our proposed algorithm `LCDF` compliments the existing algorithm `LDDF` in different cases and outperforms `LDDF` on systems with heavy data transfer traffic. Our proposed algorithm *Earliest Available Time First* for `VarIO` is shown to be consistently superior to the other algorithms.

The rest of the paper is organized as follows. Section 2 describes our model of parallel I/O, the scheduling problem, and its complexity analysis. Section 3 presents the set of heuristic algorithms we propose. Section 4 uses an example to illustrate the schedules generated by these algorithms. Section 5 reports the experimental results. Section 6 gives some concluding remarks.

2 Parallel I/O Scheduling

We first state the parallel I/O scheduling problem we will consider in this paper, and give an analysis of its complexity.

2.1 Parallel I/O Model

We consider I/O intensive applications in an architecture where the processors are connected by a complete network where every compute node can communication with each I/O node. Our model also assumes that a node is allowed to simultaneously participate in at most one send and one receive operation. When a node has multiple send operations to do, it performs these send operations one after another. If multiple nodes simultaneously send to any node P_j, these data transfers are received one after the other at P_j. The data transfers may occur in any order, and each transfer requires a specified compute node and I/O node.

Our approach involves a scheduling stage, during which I/O requests are assigned to time slots, followed by a data transfer stage, during which the data transfers are executed according to the schedule. The scheduling stage requires exchange of request information among computing nodes. We make the assumption that these request messages are much shorter than the actual data transfer, so their costs is amortized by the reduction in the time required to complete the data transfers. This assumption is appropriate for data intensive applications.

Each compute node has a queue of data transfer requests destined for various I/O servers. Since the data volume to be transferred in an I/O operation is large, communication overhead is dominated by the time for transmitting the data through the network, making start-up cost negligible. Therefore, we represent the time for data transfer between any pair of compute node and I/O node (P_i, P_j) using m_{ij}, the amount of data transmitted between them.

2.2 The Scheduling Problem

The scheduling problem in parallel I/O can be modeled by a bipartite graph in which the vertices on the left rerpesent compute nodes (denoted by P_i) and those on the right repesent I/O nodes (denoted by I_j). An edge is placed between P_i and I_j if a data transfer request from P_i can be served by I_j. There are no dependence between the requests. Note that multiple edges can exist between a compute node and I/O nodes due to data replication, since the same I/O request can be served by multiple I/O nodes. Note that although a request can be served by multiple I/O nodes, the goal is to choose one from them so as to shorten the finish time of the batch of requests. This goal holds for both read and write operations, assuming that the multiple writes for maintaining data consistency in data replication can be done by background jobs and can fully overlap with other I/O operations.

Figure 1 illustrates a system with four I/O nodes. An array `A` is replicated on these four I/O servers in the following way. A master copy of array `A` is striped (with block size four) over the four servers in a wrap-around fashion, such that the first server stores A[1..4], the second server stores A[5..8], and so on. A replicated copy is striped over the servers with the same block size but with an offset of four elements. Suppose the compute node P_1 is to read data elements A[5..8] and A[25..28] from disks. The read operation can be satisfied by two data transfers from two distinct disks, one for A[5:8] and one for A[25..28]. The request for A[5..8] can be satisfied by either I_2 or I_3, and the request for A[25..28] can be satisfied by either I_3 or I_4. The scenarior can be represented by the bipartite graph in Figure 1, in which two edges exist between P_1 and I_3, while one edge exist between P_1 and I_2 and between P_1 and I_4 respectively. In this particular example, all the edges have the same weight of four.

A Parallel I/O operation usually involves multiple compute nodes performing data transfers concurrently. An example of parallel I/O is illustrated in Figure 2. P_1 requests for A[5..8] and A[25..28], while P_4 requests for A[9..12], which can be satisfied by either I_3 or I_4 as shown in the bipartite graph in Figure 2. Our goal is to schedule these requests in proper time slots in order to minimize the completion time. We use the notion of timing diagram to help describe the scheduling problem.

Timing Diagrams We use timing diagrams to represent data transfer schedules for a given request pattern. Figure 5 shows an example of timing diagram for the request pattern depicted in Figure 4(a). The diagram consists of two `I` columns (i.e. the I/O nodes) and four `P` columns (i.e. the compute nodes). The vertical axis represents time. The data transfers in column j of

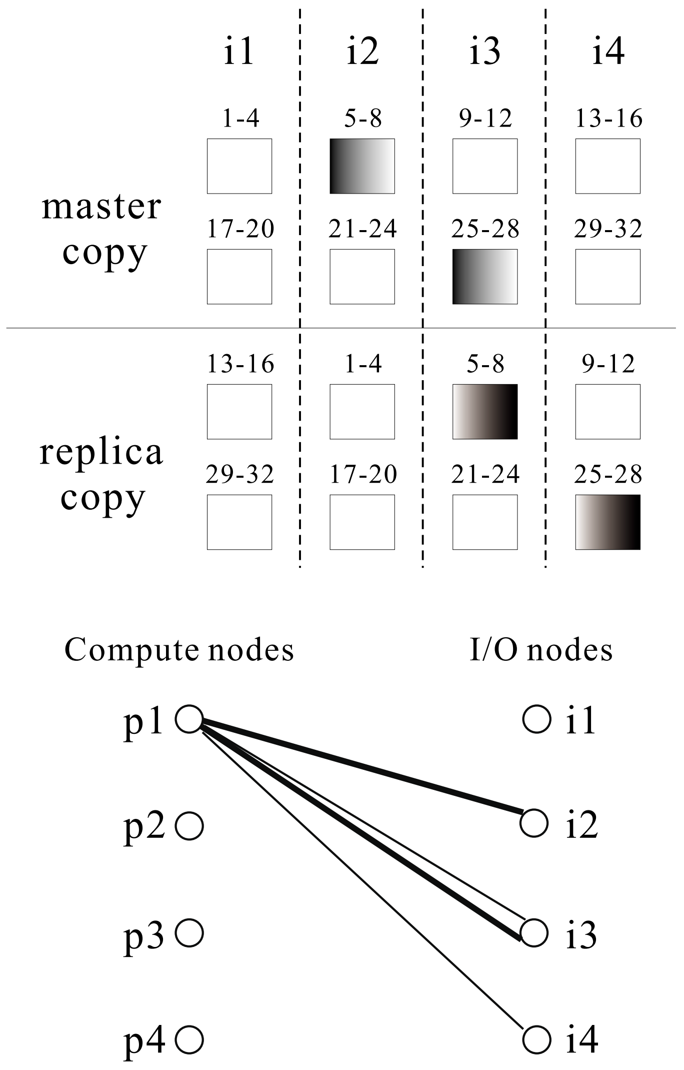

Figure 1: An example of data replication and data transfer.

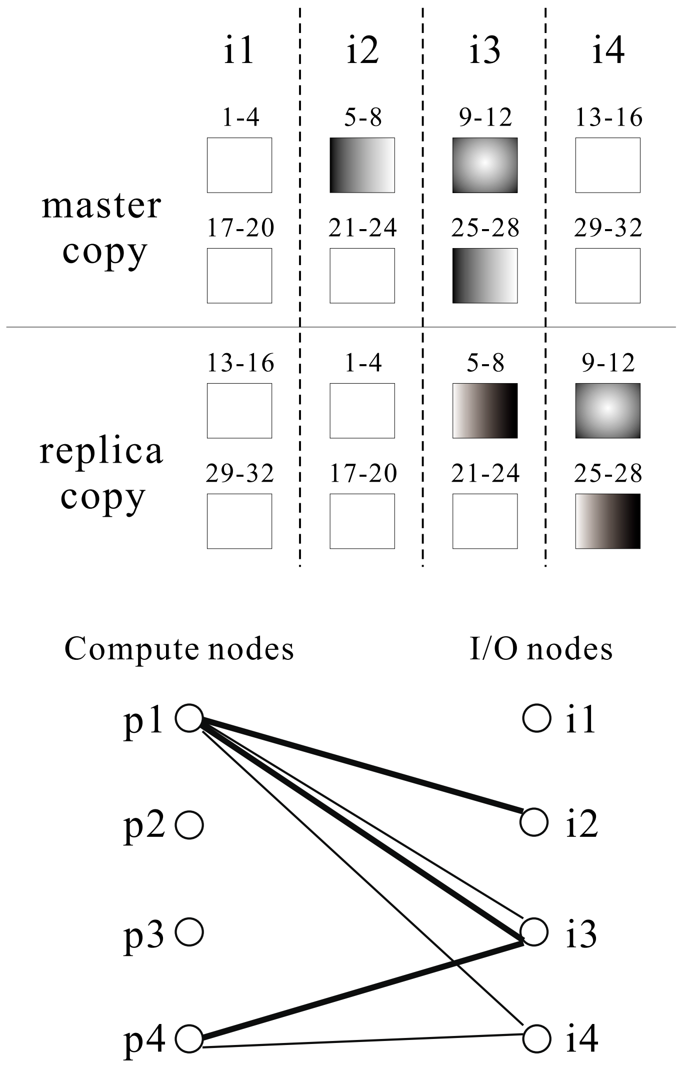

Figure 2: An example of data replication and parallel I/O.

I columns represent the messages sent from I/O node I_j. The rectangle labeled i in column j represents the message sent from I_j to P_i. The height of the rectangle represent the time for the data transfer, which is determined by the size of the message. Similarly, the data transfer in column j of P columns represent the message receive by P_j. The rectangle labeled i in column j represents the message sent from I_i to P_j. The height of the rectangle represent the time for the data transfer.

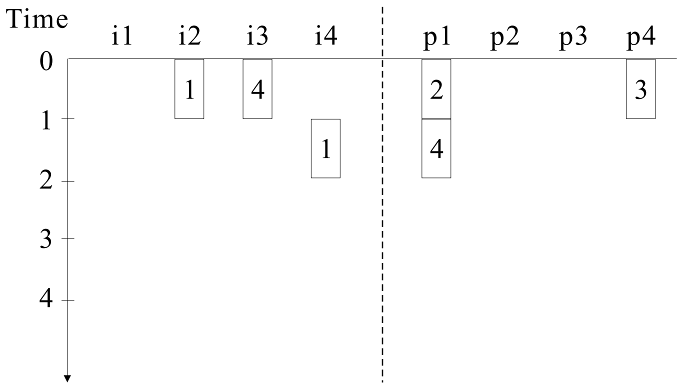

Figure 3: An example of time diagram

The parallel I/O scheduling problem is to determine the position of the individual data transfer events in the timing diagram so that the completion time is minimized. A valid schedule must satisfy the following rules. Since a node can only send/receive one message at a time, none of the rectangles in a column can overlap in time, and all the rectangles with the same label must have mutually disjoint time intervals. To analyze the complexity of this scheduling problem, we first consider the special case where only one copy of data is stored in the disks (i.e. number of replication is zero).

PIO_NO_REPLICA: Given n compute nodes $(P_1, ..., P_n)$, m I/O nodes $(I_1, ..., I_m)$, a deadline T, and a $n \times m$ matrix C, where $C_{i,j}$ is the time for the data transfer between P_i and I_j. Is there a schedule with completion time less than T?

Theorem 1 PIO_NO_REPLICA *is NP-Complete for* $m > 2$.

Proof. The theorem can be proved by reducing *PIO_NO_REPLICA* to the shop scheduling problem [10]. The problem consists of m machines and n jobs. Each job has m tasks. There is no precedence constraint on the tasks. Each machine i performs task $t_{j,i}$ of job j. The execution time of $t_{j,i}$ is given in an $n \times m$ matrix. Each machine can work on only one job at a time, and each job can be processed by only one machine at a time. The goal is to schedule the tasks on the machines so as to minimize the completion time. The problem is known to be NP-Complete for $m > 2$. ■

Since the parallel I/O scheduling problem for systems without data replication (*PIO_NO_REPLICA*) is NP-Complete, parallel I/O scheduling for systems with data replication is also NP-Complete.

3 Scheduling Algorithms

3.1 Algorithms for Uniform-Length Data Transfers

This section presents the two algorithms, *Lowest Combined Degree First* (LCDF) and *Lowest Destination Source Degree First* (LDSDF), we propose for scheduling uniform-length I/O operations for systems with data replication. The set of heuristic algorithms are based on the degrees of the nodes. The degree of a node in the bipartite graph is equal to the number of edges incident on it. Each edge has a source and a destination degree. The source degree is the degree of the compute nodes it is coming from and the destination degree is the degree of the I/O node it is going into. The combined degree of an edge is the sum of its source and destination degrees.

Lowest Destination Degree First (LDDF). This algorithm is proposed by Chen and Majumdar [5]. For self-content of the paper, we give an overview of this algorithm. At the beginning of each iteration, the destination nodes are ordered in increasing order of degree. An inner loop is executed next. The I/O node j, with the lowest degree is picked and any one of the requests for this I/O node is mapped to j. The next I/O node from the sorted set is then chosen, in the inner loop, and an attempt is made to match it with a request coming from a processor that has not been assigned an I/O node in the current (main) iteration. The algorithm stops when all requests are scheduled. In each iteration, it requires O(MlogM+M) steps to sort the destination nodes and pick the pair. The algorithm iterates N times. Therefore, the total time for LDDF algorithm is $O(N(MlogM + M))$.

Lowest Combined Degree First (LCDF). An intuitive explanation for this algorithm is given. The assignment of a request to a node can result in removal of multiple edges from the graph due to data replication. The pair of compute node and I/O node with a smaller combined degree has a small number of requests to choose from. If this pair is not considered earlier and the only requests they can handle are assigned to other pairs, this pair of nodes will idle while other nodes may have a long queue of waiting requests. LCDF works iteratively as LDDF. Instead of the I/O nodes, the edges are sorted in increasing order of the combined degree of their source and destination. In each iteration of the inner loop, the unscheduled requests are mapped from each processor to a different I/O node by traversing the sorted list of edges in order. In each iteration, it requires $O(NMlogNM + M)$ steps to sort the edges and pick the pair. The algorithm iterates N times. Therefore, the total time for LCDF algorithm is $O(N(NMlogNM + M))$.

```
Algorithm LCDF:
Repeat until all requests are scheduled
    Mark all source nodes (compute nodes) as unchosen.
    Sort the edges (requests) in increasing order of combined degree
            and store in the list SL.
    Repeat until all source nodes as chosen
        Map the unchosen source node i to a different destinstion node j
```

```
                by traversing the sorted list SL in order.
        Reduce the degree of i by 1.
        Reduce the degree of j by 1.
        Mark the source node i as chosen.
    end repeat
end repeat
```

Lowest Destination Source Degree First (LDSDF). The prior work, `LDDF`, assumes that a data request can be served entirely by an I/O node. Given the fixed number of replication, it suffice to choose the I/O nodes with the lowest degree. In a more realistic setting, a data request may spread over multiple I/O nodes, which would require that the degree of compute nodes be taken into consideration too. The `LDSDF` strategy is that in each iteration we choose I/O node j that has the lowest degree, then choose the lowest-degree compute node i from the set of compute nodes that are connected with j. In each iteration, it requires $O(MlogM + MN)$ steps to sort the destination nodes, source nodes, and pick the pair. The algorithm iterates N times. Therefore, the total time for `LDSDF` algorithm is $O(N(MlogM + MN))$.

```
Algorithm LDSDF:
Repeat until all requests are scheduled
    Sort the destination nodes (I/O nodes) in increasing order of
           degree and store in the list SL.
    Repeat until the sorted list SL become empty
        Pick the edge (request) which is connected to the lowest
           degree destination node j in the list SL and the source
           node i (compute node) that has the lowest degree.
        Reduce the degree of i by 1.
        Reduce the degree of j by 1.
        Remove j from SL.
    end repeat
end repeat
```

3.2 Algorithms for Variable-Length Data Transfers

This section presents the three algorithms, `Shortest Job First` (`SJF`), `Earliest Available Time First` (`EATF`), and `Random Selection` (`Random`), that we propose for scheduling non-uniform-length I/O operations for systems with data replication.

Shortest Job First (SJF). This algorithm schedules the data transfer requests according to their data transfer time. In each iteration, each I/O node chooses the request that has the shortest data transfer time (edge weight) as the next request to serve. It requires $O(NMlogNM)$

steps to sort the edges. The algorithm iterates NM times. Therefore, the total time for SJF algorithm is $O(NMlogNM + NM)$.

```
Algorithm SJF:
Repeat until all requests are scheduled
    Mark all destination nodes (I/O nodes) as unchosen.
    Repeat until all destination nodes are chosen
        Pick the lowest weight (time) edge k which is connected to the
               unchosen destination node j.
        Mark the destination nodes j as chosen
        Remove the edge k.
    end repeat
end repeat
```

Earliest Avaliable Time First (EATF). The SJF algorithm only consider data transfer time of a request regardless of whether the compute node and I/O node of that request is ready to serve the request. Therefore, SJF may cause unnecessary holes in the timing diagram, which will delay the completion of the batch of requests. The algorithm EATF attemps to solve this problem by taking the available times of compute nodes and I/O nodes into consideration.

Some data structures are used in EATF. Each I/O node i maintains a receiver set R_i, and each compute node j maintains a sender set S_j. Initially, R_i contains all the compute nodes that are connected to I/O node i and S_j contains all the I/O nodes that are connected to compute node j. Each I/O node i (compute node j) maintain an available time $SAvail(i)$ ($RAvail(j)$). Initially $SAvail(i)$ and $RAvail(j)$ are all zeros.

The algorithm proceeds as follows. In each iteration, the algorithm chooses the earliest available I/O node and then chooses the earliest available compute node that has requests to be served by that I/O node. In each iteration, it requires $O(M + N)$ steps to search the earliest avaliable pair. The algorithm iterates $N * M$ times. Therefore, the total time for EATF is $O(NM(M + N))$.

```
Algorithm EATF:
Repeat until all R_i become empty
    choose the earliest available I/O node i from the set S.
    Search R_i for compute node j that has the earliest available time.
    The new pair (i,j) will be scheduled a time t, where
       t = max(SAvail(i), RAvail(j))
    !! The available time of node i and node j should be updated accordingly as:
       SAvail(i) = t + w(i,j,m)
       RAvail(j) = t + w(i,j,m)
    !! Remove P_j from R_i
       R_i -= {P_j}
end repeat
```

Random Selection (R). In this strategy, an I/O request is selected randomly from the pending requests. The total time for Random algorithm is $O(NM)$.

```
Algorithm Random:
Repeat until all requests are scheduled
    Mark all destination nodes (I/O nodes) as unchosen.
    Repeat until all destination nodes are chosen
        Pick randomly any edge k which is connected to the
              unchosen destination node j.
        Mark the destination nodes j as chosen
        Remove the edge k.
    end repeat
end repeat
```

4 An Example

We use two examples (Figure 4(a) and Figure 4(b)) to illustrate these algorithms. In Figure 4(a), each compute node P_i initiates a request which can be satisfied by either one of the I/O server. P_i (j) means that compute node P_i performs a data transfer of size j.

All of LCDF, LDSDF, and LDDF generates the same schedule which transfers the data in two iterations. In the first iteration, data are transfered from i_1 to p_1, and i_2 to p_2 concurrently. It takes two time units to complete the first iteration. In the second iteration, data are transfered from i_1 to p_3 and i_2 to p_4 concurrently. It also takes two time units to complete the second iteration. Totally, it takes four time units to complete all the data transfers.

EATF and SJF also generates the same schedule which completes the data transfer in two iterations. In the first iteration, data are transfered from i_1 to p_2 and from i_2 to p_4 concurrently. In the second iteration, data are transfered from i_1 to p_1 and from i_2 to p_3 concurrently. Totally, it takes three time units to complete all data transfers.

When the available times of compute and I/O nodes are taken into account, EATF will be superior to SJF. For the example graph shown in Figure 4(b), let the available times of i_1, i_2, and p_2 be at five time units and the available time of p_1 be at 15 time units. SJF will generate the schedule: i_1 to p_1 and i_2 to p_2 concurrently. The generated schedule will take 16 time units to serve the two requests. EATF will result in two iterations. In the first iteration, data is transfered from i_2 to p_1. In the next iteration, data is transfered from i_2 to p_2. The generated schedule will only take eight time units to serve all requests.

5 Experimental Results

To evaluate the effectiveness of the scheduling algorithms, we have developed a software simulator for cluster parallel I/O systems. In the simulated cluster, the processing nodes are interconnected by wormhole switches, and each processing node is equipped with local disks so that every pro-

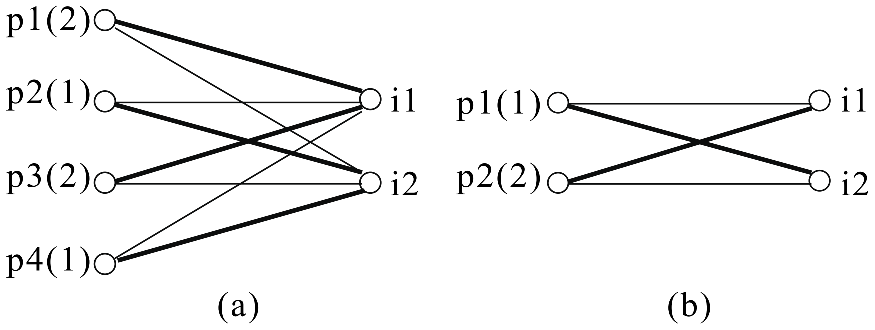

Figure 4: An example of parallel data transfer and scheduling.

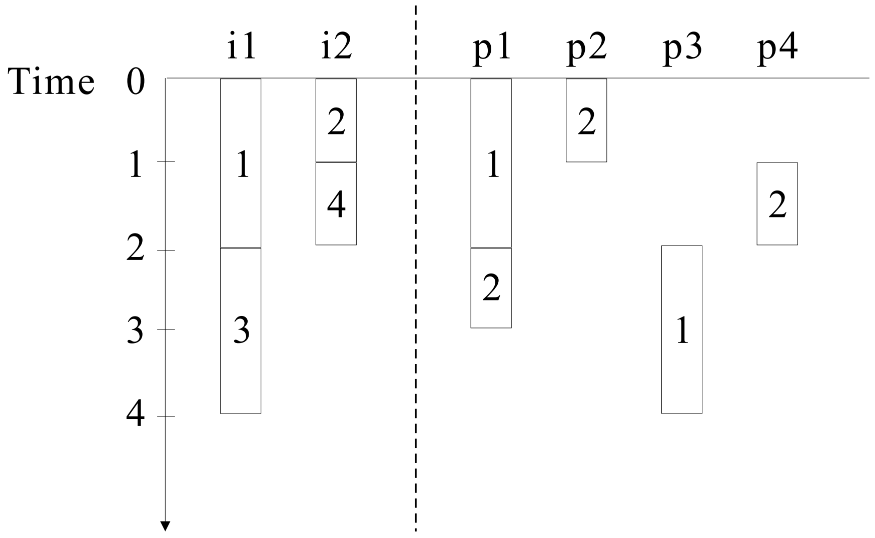

Figure 5: Schedule generated by LCDF, LDSDF, and LDD.

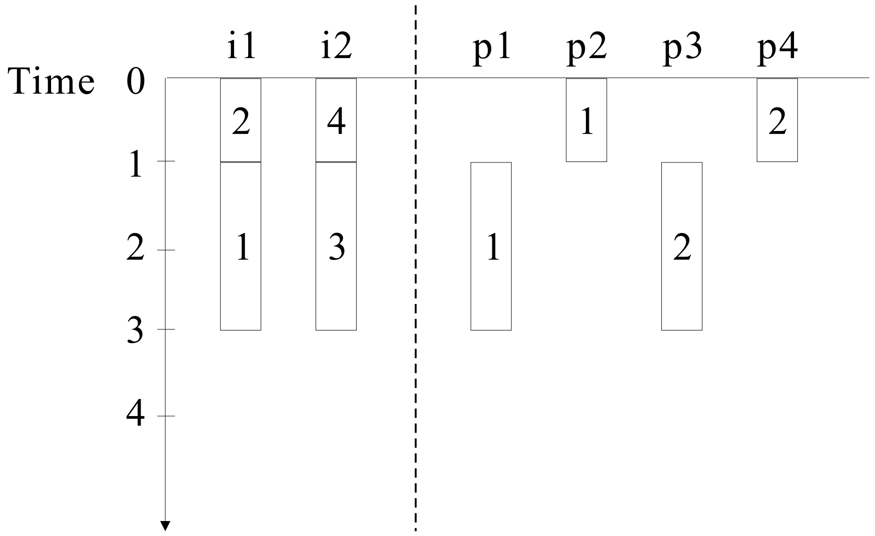

Figure 6: Schedule generated by EATF and SJF.

cessing node can perform the job of a compute node and an I/O node. This gives us flexibility in experimenting with different numbers of compute nodes and I/O nodes.

The simulator takes the following parameters as inputs: the number of I/O nodes and compute nodes, the number of data elements transferred in each I/O request (referred to as message lengths), and the data transfer patterns. Disk and network bandwidth are chosen based on the current hardward configuration of the small cluster we have built: an average disk bandwidth of 22.77 MB/s is observed on the IDE disks we have recently installed, and an average network bandwidth of 10.10 MB/s is observed on the Fast Ethernet network that is used in our small cluster. We chose message length (data size in a data transfer) of 1MB for UniIO (uniform-length data transfers), and message range of 1MB to 1GB for VarIO (non-uniform-length data transfers). Number of data replication is limited to two; that is, a data transfer request can be served by two I/O nodes.

In this paper, we report the results of read operations. Results of write operations are similiar. In each experiment, the finish time of I/O operations is measured by the average time of one hundred runs. We compare the finish times of the schedules produced by the set of algorithms (LDDF, LCDF, LDSDF, SJF, EATF, and Random). Since the algorithm EATF is consistently superior to all the other algorithms, we use EATF as the base for comparison. All the timing results are presented as the ratio of the algorithm's finish time against that of the EATF.

Several factors are taken into consideration in our experiments: the size of the cluster, the ratio between compute nodes and I/O nodes, and the data transfer patterns. Two kinds of data transfer patterns are experimented: random requests and all-to-all requests. In random requests, data transfers between compute nodes and I/O nodes (the edges in the bipartite graph) are generated randomly. In all-to-all requests, a data transfer request exists between each pair of compute node and I/O node, which is the case with heaviest data transfer traffic. Figure 7 and Figure 8 show the timing ratios of the algorithms under different system configurations, with random requests and all-to-all requests respectively.

Results of Random Requests

We varied the number of I/O nodes from 4 to 20 in different ratio of compute nodes vs. I/O nodes (1:1, 2:1, 4:1). Figure 7 shows that, for UniIO, our proposed algorithm `LCDF` is superior to the `LDDF` algorithm on systems that have larger number of I/O nodes. For VarIO, our proposed algorithm `EATF` outperforms all the other algorithms. This is because that `EATF` takes both message lengths and the available time of compute nodes and I/O nodes into consideration when scheduling the I/O requests. We also observed that `EATF` improves the finish time more for VarIO than for UniIO. This is because that message length is an important factor in effecting the available time of each node.

Results of All-to-All Requests

In Figure 8, we also varied the number of I/O nodes from 4 to 20 in different ratio of compute nodes vs. I/O nodes (1:1, 2:1, 4:1). Comparing with the results in Figure 7, we can see that the

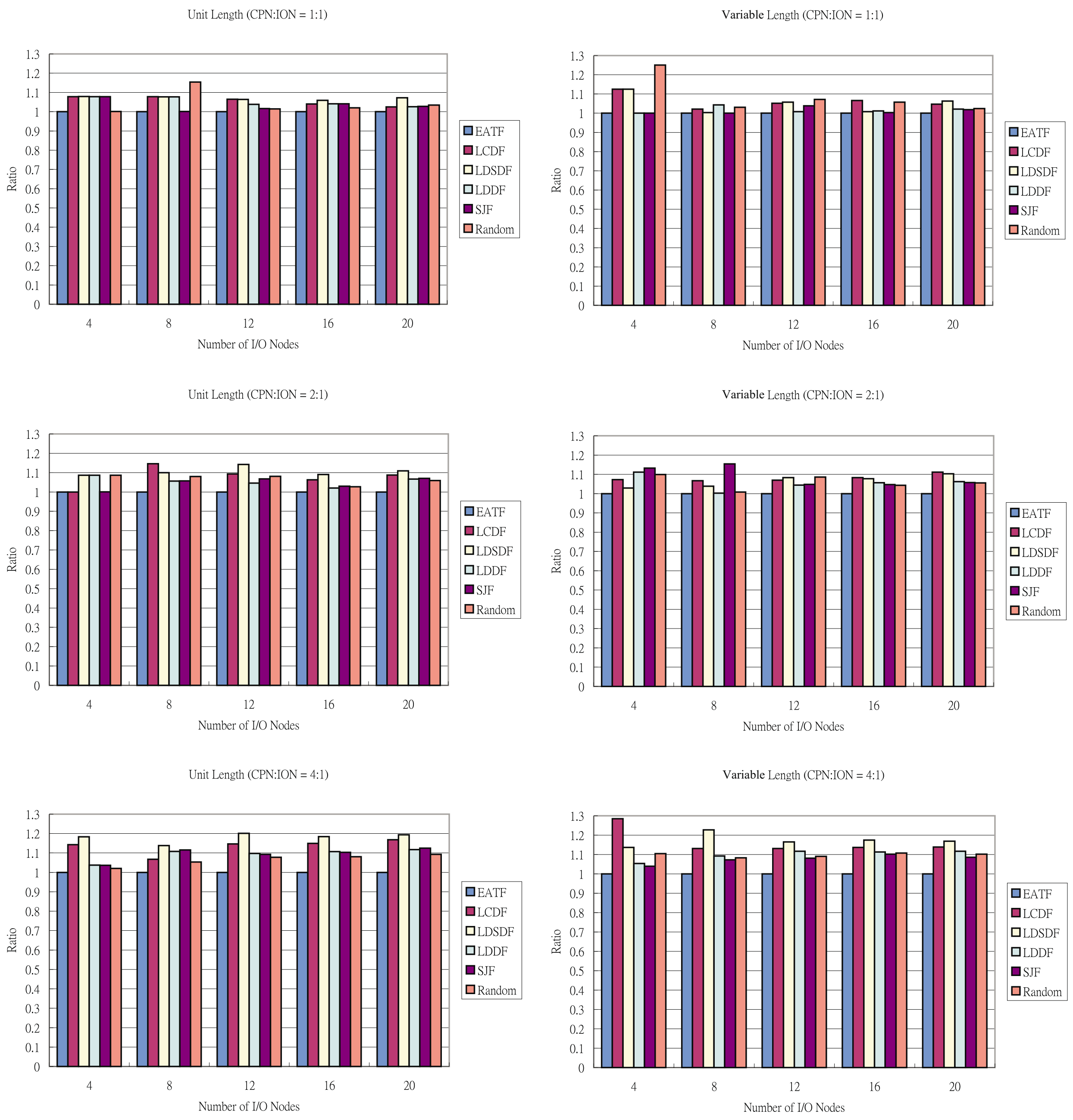

Figure 7: Performance comparison of the algorithms. The data transfer requests are generated randomly.

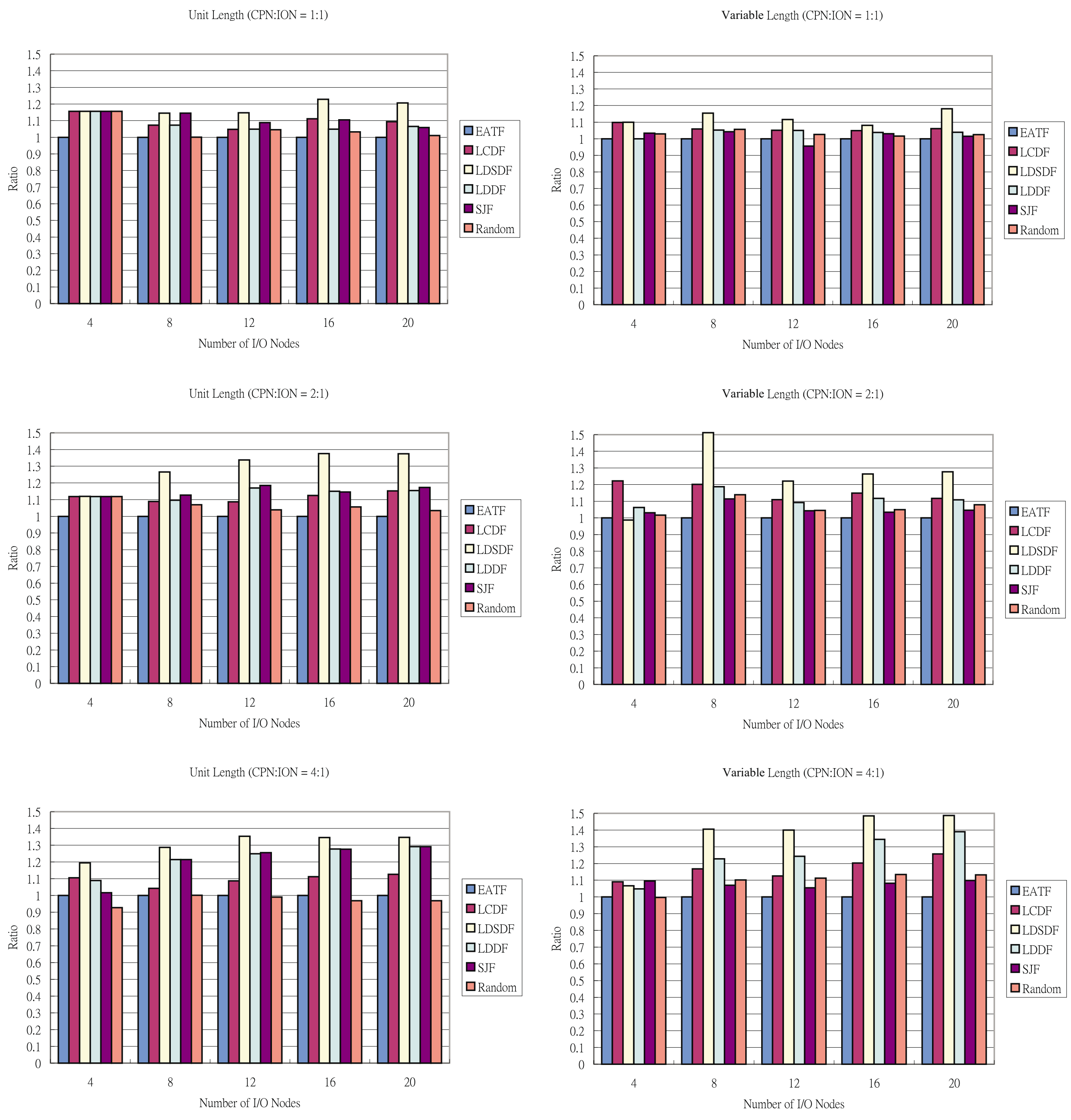

Figure 8: Performance comparison of the algorithms. Each compute node initiates one data transfer request to each of the I/O nodes.

I/O performance improvement by `LCDF` and `EATF` becomes more significant on systems with heavy data transfer traffic. `EATF` performs best in both request patterns: a performance improvement of up to 30% for random requests and up to 50% for all-to-all requests.

6 Conclusion

In this paper, we have studied parallel I/O scheduling problem for cluster-based parallel systems that provide data replication. We have shown that finding an optimal schedule for such systems is NP-Complete. We have also proposed a set of heuristic algorithms for solving this problem.

Our proposed algorithm *Lowest Combined Degree First* (`LCDF`) compliments the existing algorithm *Lowest Destination Degree* (`LDDF`) in different cases. `LCDF` is superior to `LDDF` on systems with heavier data tranfer traffic or with larger number of I/O nodes. This indicates that both the source degree and destination degree of a data transfer request should be considered when scheduling I/O operations. Our algorithm *Earliest Available Time First* (`EATF`) is superior to the *Shortest Job First* (`SJF`) algorithm in all cases. This implies that considering data transfer time alone may not generate good schedules. Available time of the compute nodes and I/O nodes is a crucial factor in scheduling I/O operations.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proc. Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 1998.

[2] S. J. Baylor and C. E. Wu. *Input Output in Parallel and Distributed Computing Systems*, chapter Chapter 7: Parallel I/O workload characteristics using Vest. The Kluwer International Series in Engineering and Computer Science. Kluwer Academics, 1996.

[3] P. Brezany, T. A Mueck, and E. Schikuta. A software architecture for massively parallel input-output. In *Proc. 3rd International Workshop PARA'96, LNCS Springer Verlag*, 1996.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. Pvfs: A parallel file system for linux clusters. In *Proc. 4th Annual Linux Showcase and COnference*, pages 317–327, 2000.

[5] F. Chen and S. Majumdar. Performance of parallel i/o scheduling strategies on a network of workstations. In *Proc. IEEE International Conference on Parallel and Distributed Systems*, pages 157–164, 2001.

[6] Y. Cho, M. Winslett, M. Subramaniam, Y. Chen, S. W. Kuo, and K. E. Seamons. Exploiting local data in parallel array i/o on a practical network of workstations. In *Proc. fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, 1997.

[7] drapeau:94. RAID-II: A high bandwidth network file server. In *Proc. International Symposium on Computer Architecture*, pages 234–244, 1994.

[8] D. Dubhashi, D. A. Grable, and A. Panconesi. Near-optimal distributed edge coloring via the nibble method. In *Proc. of the 3rd European Symposium on Algorithms*, 1998.

[9] D. Durand, R. Jain, and D. Tseytlin. Applying randomized edge coloring algorithms to distributed communication: An example study. In *ACM Symposium of Parallel Algorithms and Architectures*, 1995.

[10] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23(4):665–679, October 1976.

[11] M. Harry, J. Rosario, and A. Choudhary. Vipfs: A virtual parallel file system for high performance parallel anddistributed computing. In *Proc. 9th International Parallel Processing Symposium*, 1995.

[12] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. Ppfs: A high performance portable parallel file system. In *Proc. 9th ACM International Conference on Supercomputing*, pages 485–394, 1995.

[13] R. Jain, K. Somalwar, J. Werth, and J. C. Brown. Heuristics for scheduling i/o operations. *Proc. IEEE Trans. On Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[14] T. Kimbrel and A. R. Karlin. Near-optimal parallel prefetching and caching. In *Proc. of the IEEE Symposium on Foundations of Computer Science*, 1996.

[15] D.D.E. Long, B.R. Montague, and L. Cabrera. Swift/RAID: A Distributed RAID System. *Computing Systems*, 7(3):333–359, 1994.

[16] G. Ma, A. Khaleel, and A.L. Narasimha Reddy. Performance Evaluation of Storage Systems Based on Network-Attached Disks. *IEEE Trans. Parallel and Distributed Systems*, 11(9):956–967, 2000.

[17] R. Van Meter. A Brief Survey of Current Work on Network Attached Peripherals. *Operating Systems Review*, 30(1), 1996.

[18] S. Moyer and V. Sunderam. Pious: A scalable parallel i/o system for distributed computing environments. Technical Report Computer Science Report CSTR-940302, Department of Math and Computer Science, Emory University, 1994.

[19] B. Narahari, S. Subramanya, S. Shende, and R. Simba. Routing and scheduling i/o transfers on wormhole-routed mesh networks. *Journal of Parallel and Distributed Computing*, 57(1), April 1999.

[20] N. Nienwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File access characteristics of parallel scientific workloads. *IEEE Trans. Parallel and Distributed Systems*, 7(10):1075–1088, 1996.

[21] Nils Nieuwejaar. *Galley: A New Parallel File System for Scientific Workload.* PhD thesis, Dept. of Computer Science, Dartmouth College, 1996.

[22] D. A Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. SIGMOD*, pages 109–116, 1988.

[23] E. Riedel, G. Gibson, and C. Faloustos. Active storage for large-scale data mining and multimedia. In *Proc. 24th VLDB Conference*, 1998.

[24] J. M. Del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel i/o via two-phase run-time access strategy. *ACM Computer Architecture News*, 21(5):31–38, 1993.

[25] R. B. Ross. Providing parallel i/o on linux clusters. In *Proc. Annual Linux Storage Management Workshop*, 2000.

[26] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proc. of Supercomputing*, 1995.

[27] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. *Reading in Disk Array and Parallel I/O*, chapter Server-directed collective I/O in Panda. IEEE Computer Society Press, 2001.

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

[29] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed multi-process prefetching. In *Proc. of the ACM Interanational Conference on Measurement and Modeling of Computer Systems*, June 1997.