Efficient Parallel I/O Scheduling in the Presence of Data Duplication

Pangfeng Liu
Department of Computer Science
National Taiwan University
Taipei, Taiwan, R.O.C.
pangfeng@csie.ntu.edu.tw

Da-Wei Wang Jan-Jan Wu
Institute of Information Science
Academia Sinica
Nankang, Taipei, R.O.C.
wdw@iis.sinica.edu.tw

Abstract

This paper investigates the problem of scheduling parallel I/O operations on systems that provide data replication. The objective is to direct each compute node to access data from an I/O node where the data is duplicated, in such a way that requests for data are evenly distributed among I/O nodes. We identify a necessary and sufficient condition on whether the current data request pattern can be improved, in terms of the maximum number of data requests on any I/O node. We propose an augmenting path algorithm that examines this necessary and sufficient condition, and adjusts the current data request pattern accordingly. Using network flow technique, we show that the augmenting path algorithm finds an optimal assignment in $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$ *time.*

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 becomes larger, the performance bottleneck is likely to get worse.

Parallel input/output techniques can help solve this problem by creating multiple data paths between memory and disks, that is, exploiting parallelism in the I/O system. One active research area in parallel I/O is parallel file systems. PIOUS [12], VIP-FS [8], Galley [14], PPFS [9] and VIPIOS [2], 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. Although more recent parallel file systems (such as PVFS [3, 16]) and parallel I/O libraries (such as Panda [17, 18] and PASSION [19]) that are designed for network of workstations/PCs have provided collective I/O [19, 18], they have not addressed the performance issue sufficiently.

The performance of a parallel I/O operation is dominated by how fast data transfers between processing nodes and disks are performed. Several optimizations for reduc-

ing data transfer time for parallel I/O have been proposed in the past few years. The two-phase I/O optimization [15] 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 [5]. Parallel prefetching and caching strategies were proposed in [11, 20] 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 [10]. 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. [6] and Durand, et. al. [7] proposed various bipartite graph edge-coloring algorithms for solving the scheduling problems. Jain, et. al. [10] 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 [13] 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 [4]. 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` model.

This paper investigates the problem of scheduling parallel I/O operations on systems that provide data replication. The objective is to direct each compute node to access data from an I/O nodes where the data is duplicated, in such a way that requests for data are evenly distributed among I/O nodes. We identify a necessary and sufficient condition on whether the current data request pattern can be improved, in terms of the maximum number of data requests on an I/O node. We propose an augmenting path algorithm that examines this necessary and sufficient condition, and adjusts the current data request pattern accordingly. Using network flow technique, we show that the augmenting path algorithm finds an optimal assignment in $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$ time.

The rest of the paper is organized as follows. Section 2 describes our model of parallel I/O and the scheduling problem. Section 3 presents the algorithm that finds the optimal solution. Section 4 gives some concluding remarks.

2 Communication 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 computation node is allowed to simultaneously access at most one data, and similarly an I/O node can supply one data at a time. When an I/O node has multiple data to send, it per-

forms these send operations one after another. An I/O node can transfer data in any order, and each transfer requires a specified compute node and I/O node.

We define a *duplicated data access pattern graph* $G = (V, E)$ as follows: The vertex set V consists of three subsets C, D, IO, where C represents the set of *compute nodes*, the set D is the set of *data*, and the set IO is the set of *I/O nodes*. Compute nodes access data, which are duplicated at various I/O nodes. The edge set E consists of two subsets A and S. An edge in A connects a compute node c to a data d, which means that compute node c needs to access data d. An edge in S connects a data d to an I/O node io, which indicates that I/O node io stores a copy of data d. Since the same data can be duplicated in many I/O nodes, a data d may be connected to more than one I/O node via edges in S (Figure 1).

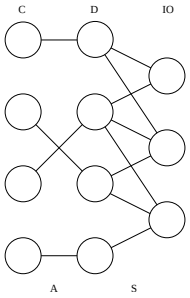

Figure 1: A duplicated data access pattern graph with 4 compute nodes, 4 data, and 3 I/O nodes.

We now formulate our parallel I/O scheduling problem for accessing duplicated data. For ease of discussion we will assume that each data is requested by a single compute node. The general case of compute nodes sharing data will be discussed in Section 3.3. Since the data are duplicated on different I/O nodes, we must assign an I/O node for each data where it can be found by its requesting compute node. Formally we define this mapping as a function m from D to IO so that $m(d) = io$ indicates that data d will be provided by I/O node io. After this assignment is completed, the duplicated data access pattern graph is reduced to a bipartite graph $G'(G, m) = (D \cup IO, M)$, where an edge (d, io) is in M if and only if $m(d) = io$. The reduced graph of G from Figure 1 can be found in Figure 2.

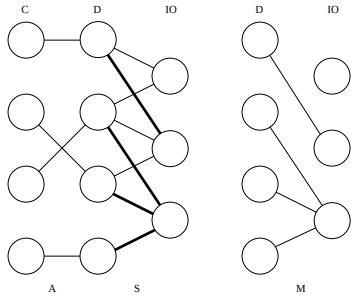

Figure 2: A reduced duplicated data access pattern graph with 4 compute nodes, 4 data, and 3 I/O nodes, after the mapping function m is chosen.

After the mapping function m is determined, the original duplicated data access pattern graph is reduced to a bipartite graph. Since in practice a communication between an I/O node and a compute node very often requires dedicated resources, an I/O node cannot send different data to multiple compute nodes simultaneously. We adopt the communication requirement that the communication between I/O nodes and compute nodes must be performed in stages. During each stage an I/O node can only send data to a compute node. It is well-known that the edges of a bipartite graph can be colored with at most d colors where d is the maximum degree, so that no edges of the same color are adjacent, therefore the communication can finish in d stages. Our scheduling problem is therefore reduced to finding a mapping function m from data to I/O nodes so that the reduced data access pattern graph minimizes the maximum degree among all I/O nodes. Note that we do not consider the maximum degree of nodes in C since the mapping between C and D is fixed a priori, and the only thing we can schedule is to assign an I/O node responsible for each data.

3 Augmenting Path Method

This section describes our algorithm for assigning data to I/O nodes so that the loads on I/O nodes are evenly distributed. Given a duplicated data access pattern graph $G = (C \cup D \cup IO, A \cup S)$ (refer to Figure 1 for an illustration), we consider only the bipartite graph $G' = (D \cup IO, S)$ since the communication pattern between C and D is independent of how we choose I/O nodes for data, and the maximum degree of nodes in C is fixed a priori.

3.1 Augmenting Path

The algorithm starts with an arbitrary assignment m, that is, for any data d we pick an arbitrary I/O node where it is available. Formally, we pick an arbitrary edge for each node d in D, and assign the other endpoint as the function value of $m(d)$. For ease of explanation we assign a *direction* to each edge in S. All edges chosen by m, e.g., $(m(d), d)$ for all d in D, will have the direction from I/O node $m(d)$ to data d. All the other edges in S will have the direction from data in D to I/O nodes in IO, as shown in Figure 3(a).

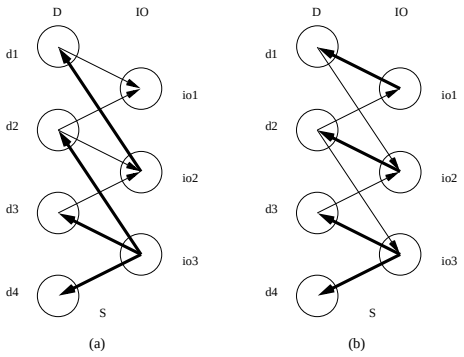

Figure 3: Adding direction for each edge in S after a mapping function m is chosen. Those edges selected by m (from IO to D) are indicated by thicker edges.

Let $deg(m)$ be the maximum number of edges adjacent to an I/O node chosen by an assignment m. All these edges will be "outgoing" from an I/O node.

$$deg(m) = \max_{io \in IO} |\{d | d \in D, m(d) = io\}|$$

We consider two types of I/O nodes – those that are adjacent to $deg(m)$ edges chosen by m (hence with outgoing degree $deg(m)$), and those that are adjacent to less than $deg(m)-1$ edges chosen by m (hence with outgoing degree less than $deg(m)-1$). These two sets are denoted by H and L respectively. Figure 3(a) shows that io_1 and io_2 are in L and io_3 is in H.

After we pick an arbitrary mapping function m and set direction for each edge in S, we derive a new directed bipartite graph. Then we try to find a path from an I/O node in H to any I/O node in L. That is, we want to locate a directed path from an I/O node with the maximum outgoing degree $deg(m)$ to any I/O node with outgoing degree $deg(m) - 2$ or less. We will refer to such a path as an *augmenting path*. If we successfully locate such a path $(io_1, d_1, io_2, d_2, ...io_k)$, where io_i is an I/O node and d_i is a data, we make the following adjustment in m: We reverse the direction of all the edges along this path, that is, we change $m(d_1)$ from io_1 to io_2, $m(d_2)$ from io_2 to io_3, and so on. As a result, the outgoing degree of I/O node io_1 will decrease by one, the outgoing degree of I/O node io_k will increase by one, and the outgoing degrees of those I/O nodes in between will remain the same. For example in Figure 3(a) there exists a path $(io_3, d_2, io_2, d_1, io_1)$. After changing $m(d_2)$ from io_3 to io_2, and $m(d_1)$ from io_2 to io_1, the maximum degree reduces from 3 to 2 (Figure 3(b)).

By finding possible directed paths from H to L, and augmenting them accordingly as described above, we will stop at a bipartite graph without any augmenting path. The following theorem states that this bipartite graph indeed has the minimum possible $deg(m)$ for all possible m.

Theorem 1 *Consider a bipartite graph $G' = (D \cup IO, S)$ induced from a duplicated data access pattern graph. A mapping function m gives the minimum $deg(m)$ if and only if there is no augmenting path.*

Proof. The only part can be verified by the "direction-reversing" process should an augmenting path is located. We only need to show the "if" part.

We prove the theorem by contradiction. Suppose the algorithm proceeds and stops at a mapping function m which does not minimize the maximum degree, there must exist another mapping function m' such that $deg(m') < deg(m)$. We will show that we can find an augmenting path by considering the edges in m and m' – a contradiction to the assumption that there is no augmenting path for m. First we define an undirected edge set for each of these two functions respectively. Let $S(m)$ be the set of edges from S chosen by m, that is, $S(m) = \{(d, m(d))|\ \forall d \in D\}$. Similarly we define $S(m') = \{(d, m'(d))|\ \forall d \in D\}$. Now we define the *difference* of $S(m')$ and $S(m)$ to be those edges appearing either in $S(m')$ or $S(m)$, but not both. Also depending on whether the edge appears in $S(m')$ or $S(m)$, we assign a direction to this edge. Formally we have the following definition:

$$
\begin{aligned}
&S(m) - S(m') = \\
&\quad \{(m(d), d)|(d, m(d)) \in S(m) - S(m')\} \\
&\quad \cup \{(d, m(d))|(d, m(d)) \in S(m') - S(m)\}
\end{aligned}
$$

Note that the edges in $S(m) - S(m')$ are directed – those in $S(m)$ only are from I/O nodes to data, and those in $S(m')$ only are from data to I/O nodes. Now we consider the directed graph $P = (D \cup IO, S(m) - S(m'))$. Consider a node io_1 in IO that has outgoing degree $deg(m)$ in m. Since $deg(m)$ is at least $deg(m') + 1$, there exists at least one edge in $S(m) - S(m')$ that goes from io_1 to a data d_1 in D (see Figure 4 for an illustration). Now

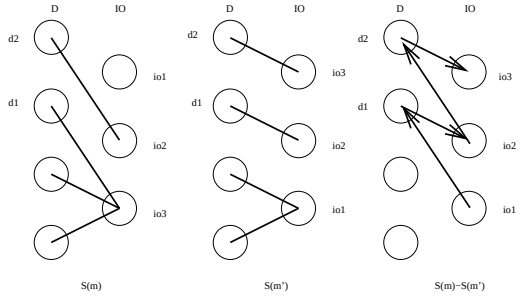

Figure 4: An illustration of $S(m)$, $S(m')$, and $S(m) - S(m')$.

we consider the data d_1. We know that $m(d_1)$ is io_1, and since (io_1, d_1) is in $S(m) - S(m')$, $m'(d_1)$ could not possibly be io_1, therefore $m'(d_1)$ is another I/O node io_2.

Now we consider two cases: If io_2 has outgoing degree $deg(m) - 2$ or less in m, we found an augmenting path, which is contrary to the fact that the algorithm could not find such a path. As a result we conclude that io_2 must be adjacent to at least $deg(m) - 1$ edges chosen by m.

Now we consider the I/O node io_2. The number of edges in S chosen by m that are adjacent to io_2 is at least $deg(m) - 1$, and we know that this number does not include the edge (d_1, io_2). On the other hand, the number edges in S chosen by m' that are adjacent to io_2 is at most $deg(m) - 1$, and it *includes* the edge (d_1, io_2) since $m'(d_1) = io_2$. Consequently, there exists an edge going from io_2 to some other node in D, that is, we can find an edge in $S(m) - S(m')$ that leads us to a new data in D. By repeating this process, eventually we either ended up at an I/O node with degree at most $deg(m) - 2$ in m, in which case we are done, or comes back to an I/O node that has a outgoing degree at least $deg(m) - 1$ in m.

Since the incoming degrees induced by m' is at most $deg(m) - 1$ for I/O nodes, we conclude that whenever this tracing goes into an I/O node with outgoing degree at least $deg(m) - 1$ via an edge from $S(m')$, it will be able to get out by an edge from $S(m)$. In addition, when-

ever a data node is visited it will not be visited again since it could be adjacent to at most two I/O nodes (from m and m' respectively). The tracing eventually ends at an I/O node with degree at most $deg(m) - 2$ in m because if there is no such I/O node, there could not be a mapping function m' that could map at most $deg(m') = deg(m) - 1$ data to *every* I/O node, considering the fact that there exists an I/O node io_1 that has degree $deg(m)$. ■

3.2 Time Complexity

We now analyze the time complexity of our augmenting path algorithm. A simple implementation involves a breadth-first-search from all I/O nodes with the maximum degree D (denoted by set H), and the search ends when it finds any node with degree $D - 2$ or less (denoted as set L). Assuming that the bipartite graph has m edges and n vertices. For each I/O node v we consider the quantity $d(v) - d^*$, where $d(v)$ is the outgoing degree of v, and d^* is the maximum degree of I/O nodes in an optimal solution. The sum of all $d(v) - d^*$ is at most m since the summation of all $d(v)$ is at most m. However, the sum of all $d(v) - d^*$ decreases by at least 1 after each breadth-first-search, and the number of rounds is at most m, therefore the total execution time of the augmenting path algorithm is bounded by $O(m^2)$.

Suppose that we want to know if there is a mapping with maximum degree less than or equal to d^*. We can use a bipartite network flow to find out if there exists a set of augmenting paths which will transform the current mapping to a target mapping with maximum degree d^*. We add a source s and a sink t into the directed bipartite graph. For every vertex v with degree d, if $d > d^*$ then add an edge from s to v with a capacity $d - d^*$; if $d < d^*$ then add an edge from v to t with a capacity $d^* - d$. Let f denote the sum of the capacity of out going edges of s. It can be shown that the maximum flow of the network is f if and only if there is a mapping with maximum degree no greater than d^*. Perform a binary search on d^* we can find the optimal mapping. Let $TBMF(n, m)$ denote the time complexity for computing the maximum flow of a bipartite graph, where n, m denote the number of vertices and edges respectively. Our algorithm needs time $O(TBMF(n, m) \log n)$. Since the maximum capacity of the above network is bounded by n, by using the wave scaling technique [1] the maximum flow problem can be solved in $O(nm + n^2 \log^{\frac{1}{2}} U)$, where U is the maximum capacity in the network, therefore time complexity of the proposed algorithm is $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$.

3.3 Shared Data

In the previous section we made the assumption that the compute nodes do not share data. This section describes the general case where a data could be shared by different compute nodes, and how our augmenting path algorithm can apply to these cases as well.

Due to the fact that we do not have any information on how the data will be shared by different compute nodes, we assume that the order by which the data is accessed is irrelevant, and the computation can proceed as long as the data is received by all requesting compute nodes. Consequently, we assume that we can "duplicate" the shared data, with each copy earmarked for a particular compute node, as shown in Figure 5.

Formally we duplicate each shared data d as follows: For each shared d we make k copies of it, where k is the number of compute nodes sharing d. Then we add k edge from these k compute nodes to these newly added data, one edge for each pair of compute node and data duplication. Then we duplicate edges from the data copies to the I/O nodes where they could reside. This results in a new duplicated data access pattern graph without data sharing among compute nodes, therefore the augmenting path algorithm can compute the minimum number of communication stages re-

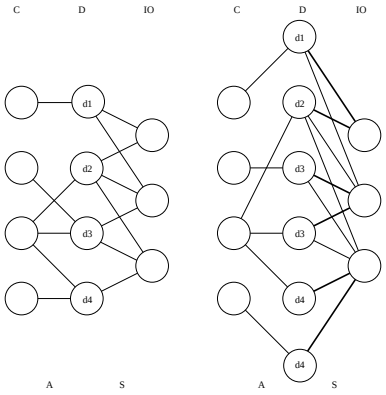

Figure 5: By duplicating the shared data, the augmenting path algorithm can also apply to the general cases where different compute node share data.

quired.

An important observation on this transformation is that although extra copies of data are duplicated, an I/O node could still only provide one data during any stage, hence the transformation does not invalid the restriction in the original communication model.

4 Conclusion

This paper investigates the problem of scheduling parallel I/O operations on systems that provide data replication. We identify a necessary and sufficient condition whether the current data access pattern can be improved, in terms of the maximum number of data requests on any I/O node, and propose an augmenting path algorithm that examines this necessary and sufficient condition, and adjusts the current data request pattern accordingly. Using network flow technique we design an algorithm runs in $O(TBMF(n,m)\log n)$, where $TBMF(n,m)$ is the time complexity for solving maximum flow problem in a bipartite graph with n vertices and m edges. Plug in the best time for $TBMF$ we derive an $O(nm\log n + n^2\log^{\frac{3}{2}} n)$ time algorithm that produces an optimal data request pattern which minimizes the maximum number of data requests on I/O nodes.

Another future work would to be to measure the time to schedule the assignment, and more importantly, the actual communication time. From our preliminary scheduling experiments we do not find that the augmenting path algorithm requires much more time than LDDF, since LDDF requires sorting procedure among the degrees of all I/O nodes. Also the augmenting path algorithm starts with a random mapping, and it is not likely we will need a tremendous number of rounds for the algorithm to complete. On the other hand, we would expect to see better I/O nodes utilization since the number of rounds is optimized in the augmenting path approach. The combined timing results from both scheduling and communication would be an interesting quantity to measure and optimize.

Finally, it will be interesting to compare our algorithm with LDDF in non-random graphs. We may introduce "hot" spots and see if two algorithms can distribute the workload evenly. We will report and compare the scheduling quality from these two algorithms.

References

[1] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18:9039–954, 1989.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.