Distributed Scheduling of Parallel I/O in the Presence of Data Replication

Jan-Jan Wu

Institute of Information Science

Academia Sinica

Nankang, Taipei, R.O.C.

wuj@iis.sinica.edu.tw

Pangfeng Liu

Department of Computer Science

National Taiwan University

Taipei, Taiwan, R.O.C.

pangfeng@csie.ntu.edu.tw

Abstract

This paper studies distributed scheduling of parallel I/O data transfers on systems that provide data replication. In our previous work, we proposed a centralized algorithm for solving this problem in systems where data transfer information is centrally available. This algorithm finds the optimal scheduling by constructing augmenting paths in the data transfer bipartite graph, requiring $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$ time, with n nodes and m edges in the bipartite graph.

In this paper, we investigate this scheduling problem in distributed systems where data transfer information may not be centrally available. We propose a distributed scheduling algorithm, *Highest Degree Lowest Workload First* (`HDLWF`), which approximates the augmenting path algorithm in distributed environments. `HDLWF` is based on a distributed, two-step scheme that determines appropriate execution order of data requests through a small number of rounds of bidding between clients and I/O servers. Our experimental results indicate that `HDLWF` yields schedules close to the centralized optimal solution, and in some cases within 3% of the optimal solution.

1 Introduction

While the speed, memory size, and disk capacity of parallel computers continue to grow rapidly, the rate at which disk drives read and write data is improving at a much slower pace. As a result, the performance of carefully tuned parallel programs can slow down dramatically when they read or write files. Parallel I/O techniques can help solve this problem by creating multiple data paths between memory and disks. Over the past few years, significant research efforts have been devoted to devising methodologies for enabling parallel I/O, ranging from low-level solutions (such as disk striping [22] and disk-directed I/O [15]), through operating system support (parallel file systems [17, 11, 20, 12, 3, 4] and compiler/library

support ([23, 24, 26, 2, 19]), to high-level algorithmic design for out-of-core parallel computation ([27]).

The performance of parallel I/O is dominated by how fast data can be transfered between processing nodes and disks. The data transfer time can be reduced in several ways. For instance, we may reduce the transfer time by choosing proper placement of I/O servers in the network to reduce the amount of remote data transfers [6], by prefetching or caching disk data to overlap computation with I/O operations, or by prescheduling I/O requests to eliminate contention on the resources (also referred to as parallel I/O scheduling). In this paper, we will focus on parallel I/O scheduling.

Data replication is commonly used for computation-intensive or data-intensive applications on distributed systems, both for reliability and performance reasons. It is typical for an application to take a long period of time to complete its execution. Failure of any disk will cause loss of data and thus faults in program execution. Data replication is necessary to ensure availability of data. Furthermore, data replication is also frequently used for better performance of distributed systems. For example, many web servers, multimedia servers and scientific databases use mirrored sites with replicated data to avoid hot spots in data transfers so as to increase performance. To obtain better parallel I/O performance, a scheduling algorithm should take data replication into consideration. However, data replication increases the complexity of scheduling, because in addition to deciding the execution order of data transfers, we also need to decide which copy of each data to be used.

In our previous work [16] we proposed a fast algorithm that finds optimal schedule for parallel I/O on systems that provide multiple copies of data. The algorithm first finds an optimal selection of data copy for all the data transfer requests by constructing augmenting paths in the data transfer bipartite graph. The selected set of data transfers with specific data copy represent an "optimal data transfer pattern". The algorithm then determines the execution order of the requests in the optimal data transfer pattern by edge coloring. We shown that our augmenting path algorithm finds an optimal data transfers pattern in $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$ time, with n nodes and m edges in the bipartite graph. Our augmenting-path approach is a centralized one, assuming global information about parallel I/O requests are centrally available. This is usually not the case in distributed systems, however, due to lack of shared memory or communication support between processing nodes in a distributed system.

In this paper, we propose and evaluate a distributed algorithm to preschedule I/O requests in the presence of data replication. Our algorithm is based on a distributed, two-step scheme that determines appropriate execution order of data requests through a small number of rounds of bidding between clients and servers. Our experimental results indicate that our distributed algorithm yields schedules close to the centralized optimal solution, in some cases within 3% of the optimal.

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 our distributed algorithm, HDLWF, that approximates the augmenting path algorithm in distributed environments. Section 4 reports our experimental results. Section 5 reviews related works, and Section 6 gives some concluding remarks.

2 Parallel I/O Scheduling

We make the following assumptions for the specific I/O scheduling problem we will consider in this paper:

- data transfers can be of arbitrary length but take place in units of fixed-size blocks and preemption is permitted at block boundaries.

- the transfers may occur in any order,

- each client has a queue of data transfer requests each destined for a specified server,

- a client or a server can handle at most one data transfer at any given time, and

- each client can communicate with each I/O server via a direct link.

Given a batch of pending I/O data transfers, our goal is to decide a schedule for performing these transfers whose total length is minimum.

2.1 Edge Coloring for I/O Scheduling

The scheduling problem can be modeled by a bipartite graph in which the vertices on the left represent clients (denoted by C_i) and those on the right represent servers (denoted by S_j). An edge is placed between C_i and S_j if a data in S_j is requested by the client C_i. There is no time dependence among the requests.
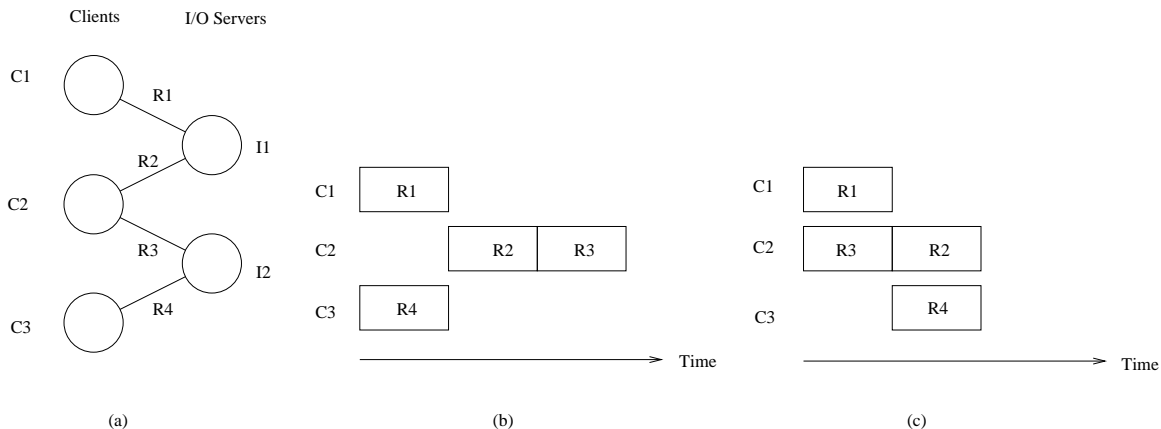

Figure 1: An example of data request pattern and two possible I/O schedules.

Figure 1(a) illustrates a system with three clients and two servers. Client C_2 has two requests and C_1 and C_3 each has one request so that there are totally four data transfers, represented by four edges R_1, R_2, R_3 and R_4. A conflict exists at client C_2, which is the client of both R_2 and R_3. Similarly, a conflict exists at server I_1 and I_2 respectively. Figure 1(b) shows a possible schedule for this bipartite graph. R_1 and R_4 are both scheduled to start simultaneously at the beginning. This is legal because R_1 and R_4 share neither client nor server. R_2 cannot start until R_1 finishes since they both share server I_1. Similarly, R_3 can only start after R_2 finishes

3

because they share the same client C_2. This schedule results in a total time of 3 time steps (Figure 1(b)). A better schedule can be obtained by scheduling R_1 and R_3 first and then R_2 and R_4, as shown in Figure 1(c), in which the total time is reduced to 2 time steps.

It is known that scheduling data transfers can be viewed as an edge coloring problem, where data transfers scheduled in the same time slot form matching in the bipartite graph. It is shown that d colors are necessary and sufficient to edge color a bipartite graph with maximum degree d [14]. Efficient algorithms to obtain optimal edge coloring can be found in [5, 13, 14, 18]. Some of these results show that when the parallel I/O request pattern is known to the algorithm, it can improve parallel I/O performance by 30% to 40% [13, 14].

2.2 Scheduling in the Presence of Data Replication

For ease of presentation, we use a tripartite graph to incorporate the factor of data replication into the basic bipartite graph. In a tripartite graph $G = (V, E)$, the vertex set V consists of three subsets C, D, IO, where C represents the set of *client nodes*, the set D is the set of *data*, and the set IO is the set of *server nodes*. Clients access data, which are duplicated at various server nodes. The edge set E consists of two subsets R and S. An edge in R connects a client c to a data d, which means that client c requests for data d. An edge in S connects a data d to a server i, which indicates that server i stores a copy of data d. Since the same data can be duplicated in many servers, a data d may be connected to more than one server via edges in S. Figure 2 illustrates a tripartite graph with three clients, four data requests, and three servers.
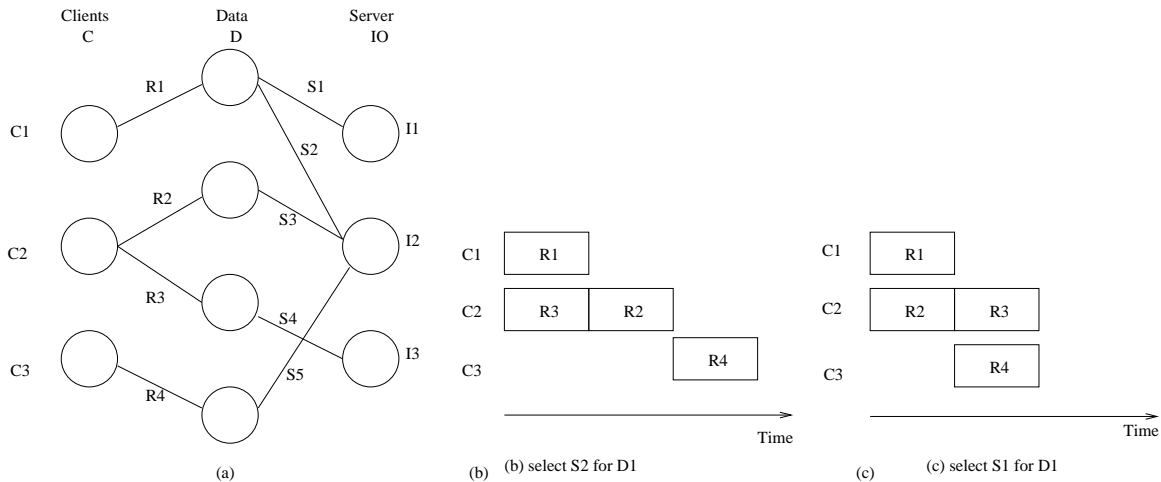

Figure 2: A tripartite graph with 3 clients, 4 data requests, and 3 servers

In Figure 2(a), data request R_1 can be satisfied by either data copy S_1 in server I_1 or copy S_2 in server I_2. Figure 2(b) and 2(c) show the two schedules when S_1 and S_2 are selected respectively. If S_2 is selected, then requests R_1, R_2 and R_4 have to be processed sequentially in three stages because they joint at server I_2 (Figure

2(b)). By choosing S_1 instead, requests R_1 and R_2 can be scheduled at the same time slot, followed by R_3 and R_4 at the next time slot (Figure 2(c)).

Since the data are duplicated on different servers, we must assign a server for each data where it can be found by its requesting client. 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 server i. After this assignment is completed, the tripartite 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 $m(d) = io$.

After the mapping function m is determined, the tripartite graph is reduced to a bipartite graph, which in turn can be edge-colored to determine the schedule of the data transfers in the bipartite graph. The scheduling problem is therefore reduced to finding a mapping function m from data to servers so that the reduced tripartite graph minimizes the maximum degree among all servers.

3   Distributed Scheduling of Parallel I/O

In many distributed environments, global information about I/O data transfers is not centrally available and clients (and similarly servers) do not have shared memory or hardware support for fast communication between them. We propose a distributed algorithm for scheduling parallel data transfers using only a small number of rounds of bidding between clients and servers.

The distributed algorithm is based on a distributed, two-step bidding process. During the first step, each client selects one of its pending transfers and sends a bidding proposal to the associated server. In the second step, each server resolves conflicts by selecting one of the bidding proposals it receives and sending back an acceptance message. Bidding proposals from other clients are rejected. When every bidding client has received an acknowledgment message (acceptance or rejection) from the associated server, the algorithm proceeds to the next round of bidding. The same process repeats until all the pending transfers have been scheduled. After the scheduling phase completes, the clients send out data transfer requests one by one as planned in their ordered lists.

The algorithm we propose, *Highest Degree Lowest Workload First* (HDLWF), is a heuristic that aims to approximate our augmenting path based algorithm in a distributed environment. First we introduce some data structures that are used in HDLWF. Each server j maintains a client set C_j that will request data from j. Each client i maintains a set of pending data requests along with their replicated copies, denoted by R_i. The function $server\_of(R_i)$ gives the server ids of the pending requests. In addition, each client i maintains a list of *current workload* (CW[1,m]) of the servers that i will request data from. CW(k) represents the number of data transfer requests that have currently been accepted by server k. CW(k) are all set to zero initially and are updated upon receiving an acceptance/rejection message on the client.

The distributed algorithm HDLWF has two steps in each round of bidding. During the first step, each client i selects a server j with the smallest current workload (smallest CW value) from R_i, and sends a bidding proposal to server j. The proposal contains both the request and the degree of client i. In the second step, each

server resolves conflicts by selecting the client that has the highest current degree. It is known that to obtain an optimal edge coloring, every matching must be a critical matching; i.e., must include an edge adjacent to the highest degree node. By favoring high-degree clients on the server side, HDLWF increases the probability of obtaining a critical matching. We also show in our previous paper [16] that to obtain an optimal selection of data transfer pattern, the maximum degree of the servers must be minimized. By favoring low-workload servers, HDLWF also increases the probability of obtaining minimum degree.

Once the server has made its choice of client, it appends the selected data request to its ordered list, increases its current workload by one, and sends back its new value of current workload in an acceptance (rejection) message to the chosen (rejected) clients. Upon receiving the acceptance message, the selected client appends the data transfer request to its ordered list and updates its current workload entry $CW(j)$ accordingly. The selected client i also removes the data request and its replicated copies from the set of pending data transfers R_i. Note that the current degree of a client and the current workload of a server are exchanged between clients and servers through the bidding proposals and the acceptance/rejection messages, and thus they do not require additional message passing. Figure 3 gives the pseudo code of HDLWF.

4 Experimental Results

We conducted simulations to evaluate the effectiveness of our algorithms. The simulation parameters include network latency and bandwidth, disk latency and bandwidth, synchronization cost, and buffer size. These parameters are obtained experimentally from a 32-node Pentium-III cluster with Myrinet interconnects and IDE disks. In all of the experiments, we fix the number of clients to be 256. For our study, we investigate the impact of the following factors on the performance of different scheduling algorithms: number of I/O servers, number of data transfers, number of replicated copies of data, and data transfer patterns (uniform vs. non-uniform or hot spots).

The most important factor is data transfer pattern. Although "uniform" data transfers are frequently seen in scientific applications (where each client requests roughly the same amount of data which are evenly distributed among the I/O servers), there are also many applications that exhibit hot spots in specific parts of the data and therefore in specific I/O servers. Thus, the actual effect of scheduling algorithms depends very much on the workload that is applied in the system. Since there are very few studies on workload analysis of parallel I/O and we have had difficulty obtaining real parallel I/O trace data, we generate synthesis workload in the following way. We do a one-to-one mapping of a geometric sequence of m items to the m I/O servers, with common ratio r, where $0 < r <= 1.0$ (that is, the first item is 1.0, the second r, the third r^2, and so on.) Let the sum of the geometric sequence be S. Then we divide each item with S. Let the resulting sequence be $p_1, p_2, \ldots, p_m$. It is clear that for all p_i, $0 < p_i <= 1.0$, and $\sum_1^m p_i = 1$. p_i represents the probability that a data transfer will be assigned to I/O server i. For each data transfer, we choose its I/O server by picking a random number between 0 and 1 and

```
Algorithm HDLWF:

On each client i:
Repeat until all R_i become empty
    /* choose the pending request whose server has the lowest current workload */
    Select data request j from R_i that has the smallest CW(server_of(j)).
    Send bidding proposal (i, j) to server_of(j)
    wait for acceptance/rejection message + new_CW.
    if proposal accepted
       append (i, j,server_of(j))) to schedule
       remove all_copies_of(j) from R_i.
    end if
    CW(server_of(j)) <- new_CW    /* update current workload of the server */
End repeat


On each server j:
schedule_done = false
Repeat until schedule_done
    Wait for all bidding proposals, B, until time out
    if B is not empty
    then choose B(i) such that client_of(B(i)) has the highest current degree
         append (B(i),client_of(B(i)),j) to schedule
         CW <- CW + 1
         Send acceptance message and CW to client_of(B(i)).
         /* send reject message to all the other bidding clients */
         Send rejection message and CW to clients_of(B) - client_of(B(i))
    else schedule_done = true
    end if
End repeat
```

Figure 3: The pseudo code of Highest Degree Lowest Workload First (HDLWF) algorithm.

then deciding its location by comparing the random number with the prefix sums of the probabilities sequence. With large number of sampling, our synthetic workload generation emulates a normal distribution function. The advantage of this method is that by choosing different common ratio r, we are able to experiment with a wide range of workload, ranging from uniform workload ($r = 1.0$) to workload with hot spots (with a very small r). Multiple hot spots can also be generated by mapping multiple sets of geometric sequences to the I/O servers.

In this section, we compare our distributed algorithm, HDLWF, against the centralized augmenting path algorithm, AP, which always finds the optimal schedule. In addition, we also use a randomized scheduling algorithm as a basis for comparison. We show that despite of lack of global information on data transfers between clients and I/O servers, HDLWF consistently produces better schedules than the randomized strategy. This shows the importance of optimization. Furthermore, under the same simulation configuration, HDLWF yields parallel performance within 15% of AP in almost all cases, and in some cases within 3% of AP.
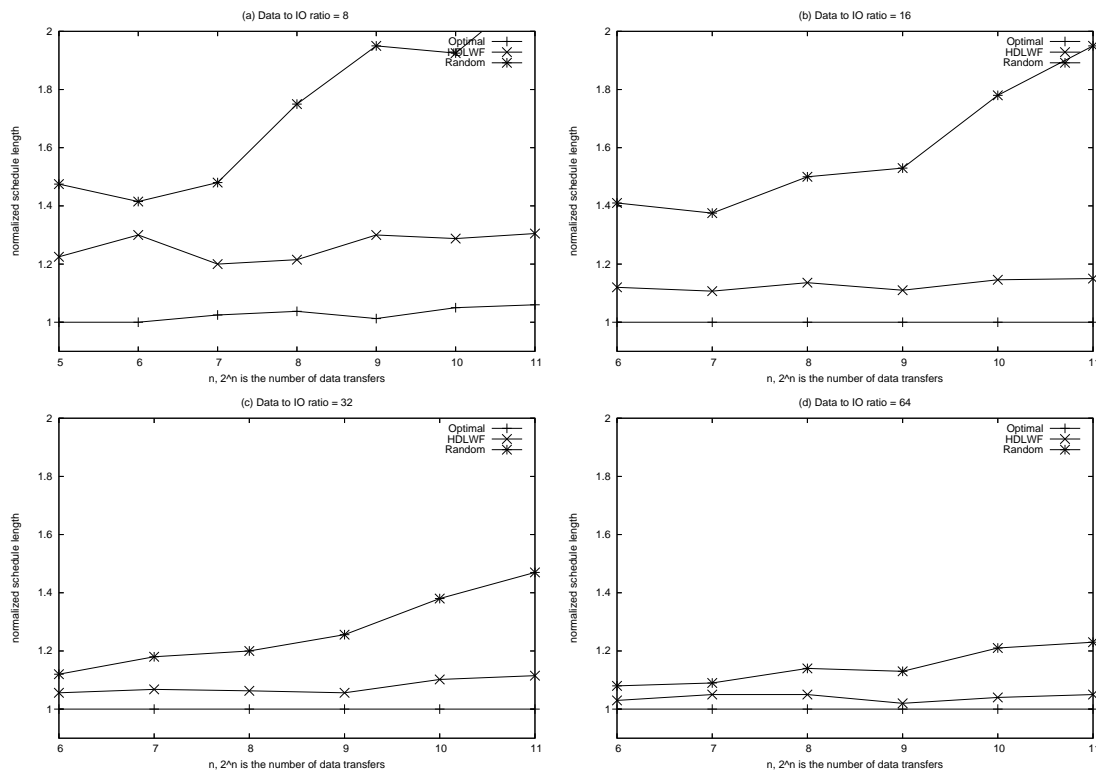
4.1 Effect of System Size

Figure 4: Effect of the number of data transfers on the schedule lengths of HDLWF, AP (Augmenting Path, the optimal solution), and Random.

In this set of experiments, we fix the data transfer distribution ratio $r = 1.0$, meaning that the I/O servers have even workload. Each data has two copies stored on different servers.

8

We use *normalized* schedule length as the metric for performance comparison. A schedule length is *normalized* by dividing it by the *shortest* schedule length found by the `AP` algorithm from *all* combination of simulation parameters in this experiment. As a result the `AP` algorithm under a particular configuration may have a normalized schedule length greater than 1.

Figure 4 compares the normalized schedule lengths from the three algorithms, `HDLWF`, `AP` and `Random` under different data-to-IO-server ratios (8,16,32 and 64 respectively). `HDLWF` is superior to `Random` in all cases. This indicates the importance of optimization. There is very small discrepancy in the normalized schedule lengths generated by `HDLWF` and `AP` when the number of data transfers increases (ranging from 2^5 to 2^{11} data transfers). This is an indication that `HDLWF` is quite stable with the increase in data transfer traffic. Furthermore, as the data-to-IO-server ratio increases, the gap between `HDLWF` and `AP` becomes smaller. That is, the heavier the workload on the servers, the closer the schedule generated by `HDLWF` is to the optimal one. This indicates good scalability of `HDLWF`.

4.2   Effect of Data Transfer Patterns

In this set of experiments, we fixed the number of I/O servers to be 16 and varied the common ratio in data transfer distribution. (ranging from $r = 0.25$ to $r = 1.0$). The four figures in Figure 5 compare the three algorithms under different number of data transfers respectively ((a)128, (b)256, (c)1024 and (d)2048). In each figure, the common ratio r ranges from 0.25 to 1.0. The lower the value of r, the less uniform the data transfer distribution. When $r = 1.0$, the servers have even workload, while when $r = 0.25$, there is one hotspot server which receives almost 75% of the total data transfer requests.

As expected, the schedule lengths all increase when the common ratio r becomes smaller, because when there is a hotspot in data transfers, the overall schedule length tends to be dominated by the hotspot server. Furthermore, `HDLWF` is superior to `Random` in almost all cases except for small number of data transfers with a hotspot (for example, when $r = 0.2$ in Figure 5(a) and (b)). When the data transfer traffic becomes heavier, `HDLWF` performs consistently better than `Random`. When the data transfer traffic is lighter (Figure 5(a) and (b)), `HDLWF` is more sensitive to the distribution of data transfers, in that the discrepancy between the performance of `HDLWF` and the optimal algorithm increases when the data transfers concentrate on a hotspot. However, when the data transfer traffic becomes heavier, `HDLWF` becomes less sensitive to the distribution of data transfers and is able to find a schedule close to the optimal one (Figure 5(c) and (d)).

Figure 6 examines the effect of hotspots on the scheduling. The total number of data transfers is fixed to be 256 and 2048 in Figure 6(a) and Figure 6(b) respectively and the number of hotspots varied from 1 to 8. In both figures, the schedule lengths decrease as the number of hotspots increases, because the data transfer traffic becomes evenly distributed over the hotspot servers. Again, when data transfer traffic is light, `HDLWF` is more sensitive to small number of hotspots. When the traffic becomes more evenly distributed, `HDLWF` is almost as good as `AP` (Figure 6(a)). When the data transfer traffic becomes heavier, `HDLWF` is less sensitive to hotspots and is able to generate near-optimal schedule (Figure 6(b)).
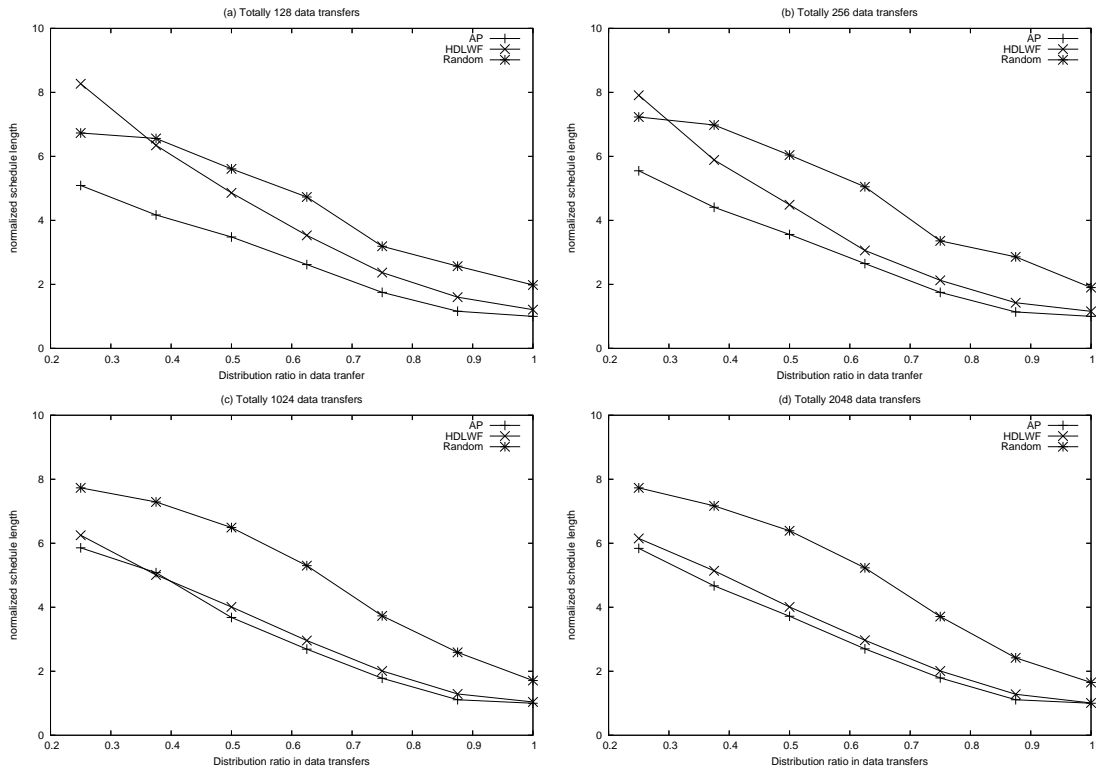
Figure 5: Effect of the distribution of data transfers on the schedule lengths of HDLWF, AP, and Random. x-axis represents the common ratio r used for the distribution of data transfers, the higher/lower the value of r, the more/less uniform the load between servers. When $r = 1.0$ every server has approximately the same workload. When $r = 0.25$, there is a hotspot server which receives almost 75% of the total data transfers.
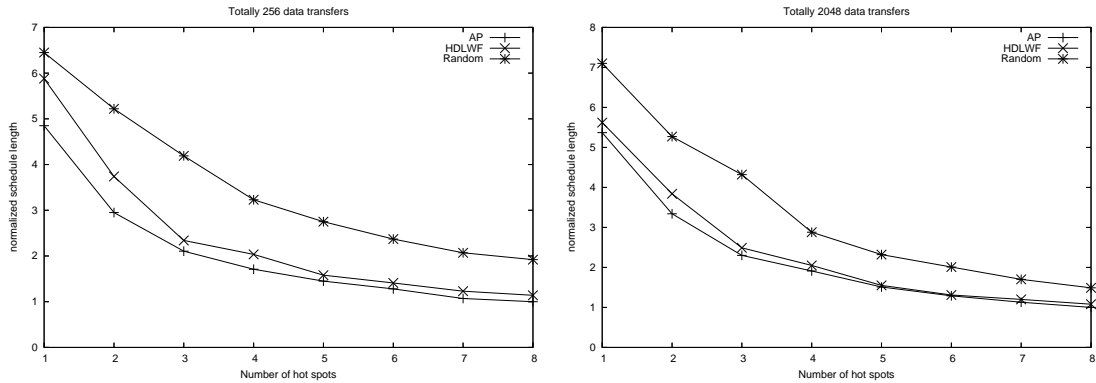

Figure 6: Effect of hotspots on the schedule lengths of HDLWF, AP and Random. A common ratio r=0.25 for distribution of data transfers is used for all experiments.

10

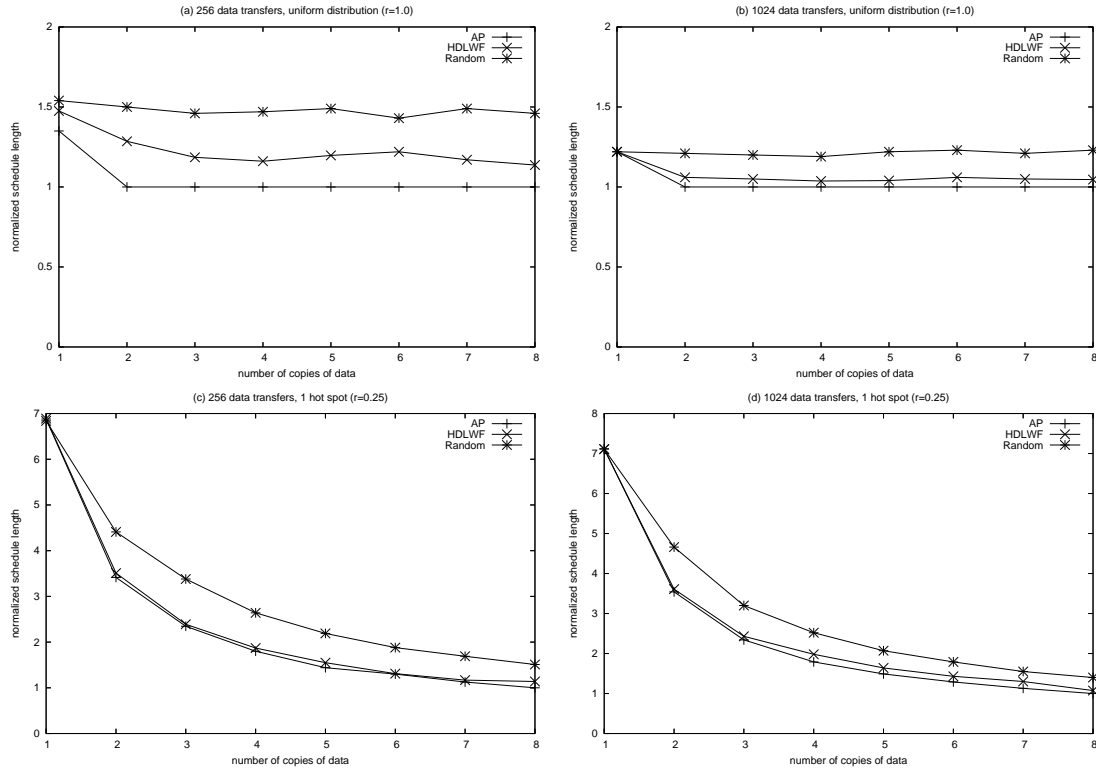## 4.3 Effect of Data Replication (Number of Data Copies)

Figure 7: Effect of data replication on the schedule lengths of HDLWF, AP, and Random. x-axis represents the number of data copies stored on the servers.

In this set of experiments, we examine the impact of data replication on the scheduling. In Figure 7(a) and Figure 7(b), data transfers are evenly distributed among the servers (i.e. the servers have even workload). In this case, the results show that only a small number of replication is needed to achieve the overall optimal schedule. For instance, in Figure 7(a), when there are two or more copies of the data, the optimal schedule length does not decrease, and AP is always able to find this overall optimal schedule. On the other hand, three copies of replication is sufficient for HDLWF and Random. When the data transfer traffic becomes heavier (Figure 7(b)), all three algorithms require only two copies of replication to achieve their best scheduling results; there is no need for keeping more copies of replication.

On the other hand, Figure 7(c) and Figure 7(d) show that when there are hotspots in data transfers, data replication seems to always help in reducing schedule length. In an idealized world where there is no cost for distributed scheduling and no cost for maintaining multiple copies of data, data replication always helps in speeding up parallel I/O.

5   Related Work

In this section, we discuss some of previous works in parallel I/O scheduling, including both centralized scheduling and distributed scheduling.

5.1   Centralized Scheduling

Jain, et. al [13] investigate parallel I/O scheduling on multi-bus systems. Their model assumes that at most k data transfers can occur simultaneously at any given time. They prove that when k is unbounded, exactly d colors are necessary and sufficient to color a bipartite graph of degree d. When k is bounded, the scheduling problem is equivalent to k-coloring of a graph, in which each color may be used to color at most k edges. It is shown that at least $p = \max(d, \lceil m/k \rceil)$ colors are necessary to k-color a bipartite graph with m edges and degree d.

To further improve the scheduling time, Jain, et. al [14] propose greedy heuristics (called HDF and HCDF) which are essentially approximation algorithms for k-coloring the edges of a bipartite graph. For each color, the algorithm attempts to color as many edges as possible with that color. They show that when k is unbounded, a greedy heuristic produces a coloring using at most $2d - 1$ colors, and that both HDF and HCDF are able to edge-color the graph with exactly $2d-1$ colors. For bounded k, it is shown that a greedy heuristic produces a coloring using at most $\lfloor m/k \rfloor + (2d-1)$ colors. HDF and HCDF are the fastest centralized algorithms known so far.

Several other researchers also proposed fast algorithms for constructing minimum length schedule for parallel I/O, including the algorithm KT [1], which takes time $O(mn(m + n))$, Somalwar's algorithm [25], which takes time $O(mn^{1.5} \log n)$, and algorithm A2 [13], which takes time $O(mn^{0.5} \log n)$.

The only work that we are aware of that schedules parallel I/O for systems providing data replication is the *Lowest Destination Degree First* (LDDF) algorithm proposed by Chen and Majumdar [5]. LDDF considers I/O nodes in ascending order of degrees, and selects a client randomly for each I/O node. Once a data is scheduled, all the edges corresponding to its duplicated copies are removed from the graph. This process repeats until all the data requests are scheduled. In each iteration, it requires $O(n \log n)$ steps to sort the I/O nodes and $O(m)$ steps to pick the client/server pair. The algorithm iterates n times, resulting in total time of $O(n^2 \log n + nm)$. Since LDDF relies on heuristics, it has no guarantee for finding the optimal solutions. Our augmenting path based algorithm always finds the optimal solution and requires $O(nm \log n + n^2 \log^{\frac{3}{2}} n)$ time. In our experiments, we found that our algorithm does not require more scheduling time than the algorithm LDDF in most cases.

5.2   Distributed Scheduling

The following distributed algorithms are shown analytically or experimentally to be efficient for various sets of data transfer traffic. However, none of them has considered data replication in its scheduling.

Panconesi and Srinavasan [21] present a distributed edge coloring algorithm (PS) for bipartite graphs. Their algorithm is based on the same two-step bidding mechanism that we used. They show analytically that their algorithm requires at

most $1.6\delta + O(\log^{1+\delta}n)$ colors to edge-color the entire graph in at most $O(\log\delta)$ phases, where *delta* is the maximum degree. However, their algorithm requires the initial degree of the graph to estimate the current degree of the graph at each stage. It would requires $O(\log\delta)$ communication to decide the degree of a distributed graph. In extensions of this work, Dubhashi, Grable and Panconesi propose faster, randomized, distributed edge coloring algorithms [10, 7]. Their algorithms require at most $O((1 + O(1))\delta)$ colorings with high probability.

Durand, Jain and Tseytlin also propose randomized, distributed edge coloring algorithms for bipartite graphs [8, 9]. Their algorithms also use a two-step bidding routine. In their algorithm, *Highest Degree First* (`HDF`), during the bidding, each server grants the request of the highest degree client, with ties broken arbitrarily. It is known that to obtain an optimal edge coloring, every matching must be a critical matching; i.e. must include an edge adjacent to the highest degree node. The intuition is that by favoring high-degree nodes, `HDF` increases the probability of obtaining a critical matching. Another algorithm, `MPASSES`, uses multiple rounds of bidding to obtain denser matchings, which in turn improves the schedule length.

6   Conclusion

This paper studies distributed scheduling of parallel I/O data transfers on systems that provide data replication. We propose a distributed scheduling algorithm, *Highest Degree Lowest Workload First* (`HDLWF`), which approximates our previous centralized augmenting path algorithm, but in a distributed environment. `HDLWF` is based on a distributed, two-step scheme that determines appropriate execution order of data requests through a small number of rounds of bidding between clients and I/O servers. Our experimental results indicate that `HDLWF` yields schedules close to the centralized optimal solution, and in some cases within 3% of the optimal solution.

Our experimental results indicate that the existence of data transfer hotspots has a significant impact on data transfer schedule length. However, when the data transfer traffic increases, the normalized schedule length of `HDLWF` actually decreases, indicating that the schedule length of `HDLWF` increases at a slower pace than the optimal one. We also found that only two or three copies of duplication is necessary to improve data transfer rate under uniform data access patterns; more duplication does not help. However, when there are data transfer hotspots, increasing copies of duplication helps in improving data transfer rate. Furthermore, under all types of data transfer patterns (uniform, non-uniform, hotspots), the performance gap between `HDLWF` and the optimal algorithm decreases as the data transfer traffic increases. This indicates good scalability of `HDLWF`.

Currently `HDLWF` use a single round of bidding in each scheduling step. Those clients that were rejected by the servers will wait for the next scheduling step. It would be interesting to allow the possibility of multiple bidding, i.e., a client may bid for several servers, or multi-stage bidding in which those rejected clients are able to compete again immediately after they received the rejection. We will continue the investigation on these alternatives, and develop bidding techniques that can improve data transfer performance in a distributed environment.

References

[1] A. Bar-Noy, R. Motwani, and J. Naor. The greedy algorithm is optimal for on-line edge coloring. *Information Processing Letters*, pages 251–253, Dec. 1992.

[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] 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.

[8] D. Durand, R. Jain, and D. Tseytlin. Distributed scheduling algorithms to improve the performance of parallel data transfers. In *in R. Jan, J. Werth, J.C. Brown (Eds.), Input/Output in Parallel and Distributed Computer Systems, Kluwer Academic Publishers*, pages 245–268, 1996.

[9] D. Durand, R. Jain, and D. Tseytlin. Parallel i/o scheduling using randomized, distributed edge coloring algorithms. *Journal of Parallel and Distributed Computing*, 63:611–618, 2003.

[10] D.A. Grable and A. Panconesi. Nearly optimal distributed edge coloring in $o(loglogn)$ rounds. In *Symposium on Discrete Algorithms*, pages 278–285, May 1997.

[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. Scheduling parallel i/o operations in multiple bus systems. *Journal of Parallel and Distributed Computing*, 16:352–362, 1992.

[14] 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.

[15] D. Kotz. Disk-Directed I/O for MIMD Multiprocessors. In Hai Jin, Tomi Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel Input/Output: Technologies and Applications*, pages 513–535. IEEE Computer Society and John Wiley & Sons, 2001.

[16] P. Liu, D.-W. Wang, and J.-J. Wu. Efficient parallel i/o scheduling in the presence of data duplication. In *Proc. of the International Conference on Parallel Processing*, pages 231–238, Oct. 2003.

[17] 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.

[18] 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.

[19] 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.

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

[21] P. Panconesi and A. Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds. *SIAM J. Computing*, 26(2):350–368, 1997.

[22] K. Salem and H. Garcia-Molina. Disk striping. In *International Conference on Data Engineering*, pages 336–342, 1986.

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

[24] 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.

[25] K. Somalwar. Data transfer scheduling. Technical Report Computer Science Report TR88-31, Department of Computer Science, Univ. of Texas at Austin, 1988.

[26] 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.

[27] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Trans. on Visualization and Computer Graphics*, 3(4):370–380, 1997.