Efficient Distributed Algorithms for Parallel I/O Scheduling

Jan-Jan Wu[1]
Institute of Information Science[1]
Academia Sinica
Taipei, Taiwan, R.O.C.
{wuj,ice}@iis.sinica.edu.tw

Yih-Fang Lin[1,2]

Pangfeng Liu[2]
Dept. Computer Science and Information Engineering[2]
National Taiwan University
Taipei, Taiwan. R.O.C.
pangfeng@csie.ntu.edu.tw

Abstract

In distributed systems, the lack of global information about data transfer between clients and servers makes implementation of parallel I/O a challenging task. In this paper, we propose two distributed *algorithms for scheduling data transfer in parallel I/O with non-uniform data sizes, the* Maximum-Size/Maximum-Load *(*`MS/ML`*) algorithm and the* Minimum-Size/Earliest-Completion-First *(*`MS/ECF`*) algorithm. Experimental results indicate that both algorithms achieve good performance, compared with the results achieved by their centralized counterparts. Both algorithms yielded parallel performances within 6% of the centralized solutions.*

We also compare the performance of our algorithms with a distributed Highest Degree First *(*`HDF`*) method, which handles non-uniform data transfers by dividing them into units of fixed-sized blocks which are then scheduled and transferred one at a time. Experimental results show that our algorithms require less scheduling and data transfer time, resulting in better overall parallel I/O performance. Our simulations also show that* `MS/ML` *is more suitable for parallel I/O with lighter data transfer traffic, while* `MS/ECF` *is more suitable for parallel I/O with heavy data transfer traffic.*

1. Introduction

Over the past few years, significant research efforts have been devoted to devising methodologies for enabling parallel I/O, including low-level solutions, such as disk striping and disk-directed I/O, operating system support, such as parallel file systems [15, 12, 2, 3], and compiler and library support [19, 20].

Traditional parallel systems use a smaller number of I/O servers and disks in parallel to increase I/O bandwidth. These systems can suffer I/O bottlenecks since many processors share a relatively small number of disks. On the other hand, cluster computers, and recently the Grids, can provide a larger data storage capacity since usually each node has at least one disk attached. The main drawback is that data has to be accessed through the cluster network (or the Wide Area Network that connects the sites of a Grid) that is typically slower than the I/O bus. Therefore, the performance of parallel I/O in distributed systems 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 transfer [5], by prefetching or caching disk data to overlap computation with I/O operations, or by careful scheduling of parallel I/O requests (i.e. parallel I/O scheduling). In this paper, we focus on parallel I/O scheduling.

Data transfers in parallel I/O can be modeled by a bipartite graph, where the clients and the servers form the two sets of vertices, and the edges represent data transfers between them. When the data sizes are uniform (all the edges have the same weight), scheduling data transfers can be viewed as an edge coloring problem, where data transfers scheduled in the same time slot form a 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 [1].

Efficient centralized algorithms to obtain optimal edge coloring can be found in [4, 13, 14, 16]. Some of these results show that when the parallel I/O request pattern is known to a centralized algorithm, it can improve parallel I/O performance by 30% to 40% [13, 14]. However, in many distributed systems, global information about parallel I/O requests may not be centrally available due to lack of shared memory, and it is too costly to maintain such global information in the distributed memories of the processors via message passing through the network. Over the past few years, a number of distributed algorithms were proposed for the edge-coloring problem [6, 7, 8, 11, 17, 18]. Many of these algorithms are based on randomization or re-

Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)
0-7695-2281-5/05 $20.00 © 2005 IEEE

cursive schemes such as divide-and-conquer. Previous analytical and simulation results show that some of these distributed algorithms can achieve good performance close to the optimal solution obtained from centralized algorithms.

The aforementioned algorithms are all targeted for parallel I/O with *uniform* data sizes. Many applications require parallel I/O with non-uniform data transfer sizes, for instance, scientific computation with unstructured or dynamic data structures. Directly applying existing distributed algorithms to this class of applications requires that the data transfers be divided into units of fixed-sized blocks and be scheduled and transmitted one block at a time [9], resulting in a large number of messages passing and synchronizations between clients and servers. In a distributed memory system, there is typically a non-trivial communications latency or startup cost, therefore the communication overheads of this fixed-size scheduling approach can easily become the bottleneck of the overall parallel I/O performance.

To demonstrate this I/O bottleneck we compared the parallel I/O performance of two scheduling methods – RAND and UAR. Both methods randomly pick source/destination pairs for communication, with the only difference being that UAR divides data transfers into fixed-size blocks, while RAND does not. Figure 1 illustrates their differences in parallel I/O performance on a 16-node PC cluster with Myrinet interconnect, which has peak network bandwidth of 1.28 Gbps. Each node of the cluster is a Pentium-III hooked with an IDE disk with disk capacity of 20GB and average disk bandwidth of 32.96 Mbps. Four out of the 16 nodes are used as I/O servers. As Figure 1 indicates, the UAR method is consistently inferior to RAND due to higher communication overheads from cutting messages into unit sized blocks. The gap between the two methods decreases as the block size increases and the number of blocks decreases. This experimental result convinced us that new algorithms are necessary for scheduling non-uniform data transfers.

In this paper, we first show that finding an optimal schedule for parallel I/O with non-uniform data sizes is NP-complete. Then we propose two centralized algorithms to find near-optimal solutions: a maximum-matching-based algorithm called MWM, and a greedy algorithm based on the virtual time of clients and servers, called *Earliest Completion First* (ECF). Our experimental results show that MWM and ECF achieve parallel I/O performance close to the optimal, and within 4% of the optimal for certain parameter choices.

We also propose two distributed algorithms, called MS/ML and MS/ECF that approximate MWM and ECF in distributed environments. These two distributed algorithms are 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. We compare our algorithms

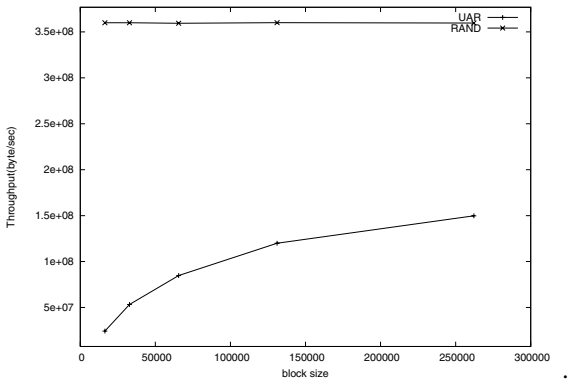

Figure 1. The comparison of throughput for algorithm RAND and UAR

with the best distributed algorithm reported in the literature, *Highest Degree First* method (HDF), for scheduling non-uniform parallel I/O. Our experimental results indicate that MS/ML and MS/ECF require less scheduling time than HDF and yield better parallel I/O performance.

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 algorithms. Section 4 reports the experimental results, and Section 5 gives concluding remarks.

2. Parallel I/O Scheduling

We consider parallel systems in which processors are connected by a complete network. There are two kinds of processors: clients and servers, and every client can communicate with every server. Each client has a queue of data transfer requests (reads or writes) destined for the servers. A client or a server can handle at most one data transfer at any given time. The data transfers can be of arbitrary lengths and can be scheduled in any order. The goal is to process all requests as fast as possible without violating the "one data transfer at a time" constraint.

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 2(a) illustrates a system with two clients and three servers. Client C_1 and C_2 each has two requests so that there are four data transfers, represented by four edges T_1, T_2, T_3 and T_4, with data transfer time 30, 10, 10, and 20 respectively. A conflict exists at server S_2 which is the target of

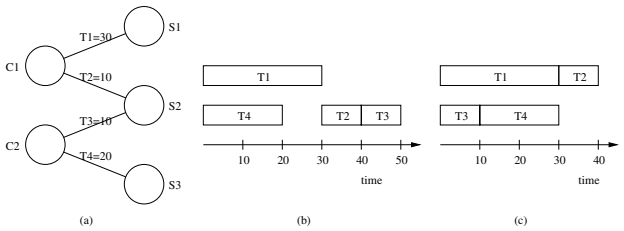

Figure 2. An example of data request pattern and two possible I/O schedules.

both T_2 and T_3. Figure 2(b) shows a possible schedule for this bipartite graph. T_1 and T_4 are both scheduled to start simultaneously at the beginning. This is legal because T_1 and T_4 share neither client nor server. T_2 cannot start until T_1 finishes since they are from the same client C_1. Similarly, T_3 can only start after T_2 finishes because they access the same server S_2. This schedule results in a total time of 50 (Figure 2(b)). A better schedule can be obtained by scheduling T_1 and T_3 first and then T_2 and T_4, as shown in Figure 2(c), in which the total time is reduced to 40.

The parallel I/O scheduling problem is to schedule data transfer events so as to minimize the overall time, under the "one data transfer at a time on each node" constraint. This problem is shown to be NP-complete.

PIO_SCHEDULE*:* Given n clients $(C_1, ..., C_n)$, m servers $(S_1, ..., S_m)$, a deadline T, and a $n \times m$ matrix W, where $W_{i,j}$ is the time for the data transfer between C_i and S_j. Is there a schedule with completion time less than or equal to T?

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

Proof. The theorem can be proved by showing *PIO_SCHEDULE* and the open shop scheduling problem [10] are equivalently difficult. The open shop problem consists of m machines and n jobs. Each job has m independent tasks to be executed by each of the m machines, that is, machine i performs task $t_{j,i}$ of job j, for $1 \leq j \leq n$ and $1 \leq i \leq m$. There is no precedence constraint among the tasks. However, each machine can work on only one job at a time, and each job can be processed by only one machine at a time. Given the execution time $t_{j,i}$ for all i and j, the goal is to schedule the tasks on the machines so as to minimize the completion time. This problem is known to be NP-Complete for $m > 2$ [10]. The *PIO_SCHEDULE* problem can be modeled as an open shop problem if we set the time to 0 for those client-server pair between which there is no data transfer. ■

3. Scheduling Algorithms

3.1. Centralized Scheduling Algorithms

Our solutions to the *PIO_SCHEDULE* are based on weighted bipartite matching and greedy heuristics.

3.1.1. Maximum Weight Matching (MWM) A matching in a bipartite graph represents a valid parallel data transfer step. A maximum matching (i.e. a matching with maximum cardinality) represents a valid parallel data transfer step with highest parallelism. A matching-based scheduling algorithm finds a maximum matching in the graph, deletes the matching from the graph, and then repeats the process until all the edges are removed.

The Maximum Weight Matching (`MWM`) algorithm starts with a matching with maximum sum of weights. The likely result is that data transfer requests with larger weights will be scheduled earlier than the ones with smaller weights. Also, by maximizing the sum of weights, maximum amount of data transfers will be grouped together, thus reducing idle cycles.

3.1.2. Earliest Completion First (ECF) The `MWM` algorithm only considers data transfer time of a request, regardless of whether the client and server of that request is ready to serve the request. Therefore, `MWM` may cause unnecessary "holes" in the schedule, which will delay the completion of a batch of requests. We propose a new algorithm, *Earliest Completion First* (`ECF`), that takes both the available time of clients and servers and the completion time of I/O requests into consideration. Available time and completion time are defined as follows.

Available time. The *available time* of a processor p is the earliest time at which processor p can execute a new data transfer. Each client i maintains an available time $CAvail(i)$ and each server j maintains an available time $SAvail(j)$. Initially all $CAvail(i)$ and $SAvail(j)$ are set to zeros.

Completion time. The completion time, $CompleteTime(i, j, w)$, of data transfer of length w, between client i and server j, is the earliest time the data transfer can complete.

$$CompleteTime(i, j, w) = \max\{CAvail(i), SAvail(j)\} + w \quad (1)$$

During `ECF` scheduling each client i maintains a set of servers that need to transfer data to i (denoted by s_i). Initially, s_i contains all the servers that need to transfer data to client i. The algorithm proceeds as follows. In each iteration, the algorithm chooses the pending data transfer request (i, j, w) with the *earliest* completion time as the next new task. It then updates the available time of client i and

server j, and removes j from the server set s_i. This process repeats until all s_i's become empty. Figure 3 gives the pseudo code of ECF.

```
Algorithm ECF:
Repeat until all s_i become empty
    Find client i and server j such that data transfer
         (i,j,w) has the earlieast completion time.
    The new pair (i,j) will be scheduled at time t,
       where t = max(CAvail(i), SAvail(j))
// Update the available time of client i and server j
    CAvail(i) = t + w
    SAvail(j) = t + w
    Remove j from s_i
end repeat
```

Figure 3. The pseudo code of Earliest Completion First (ECF) algorithm.

3.1.3. Effectiveness of Centralized Scheduling Algorithms We implement MWM and ECF algorithms and conduct experiments on a Pentium-III PC cluster that has Myrinet interconnect and a IDE disk attached to each node. Both algorithms improve parallel I/O performance by 35% to 40%, compared against a random selection algorithm. Both MWM and ECF yield parallel I/O performance close to the optimal result achieved by an exhausted search; within 4% for certain parameter choices. In this paper, we will use MWM and ECF (both use centralized control) as the comparison basis for our *distributed* scheduling algorithms.

3.2. Distributed Scheduling of Parallel I/O

In many distributed environments, global information about I/O requests is not centrally available and it is too costly for clients (and similarly servers) to share or exchange information between them. To eliminate the need of global information for scheduling purposes, we propose a distributed load balancing algorithm and a distributed greedy algorithm for scheduling I/O requests using only a small number of rounds of bidding between clients and servers.

The distributed algorithms are based on a distributed, two-step bidding process. During the first step, each client selects one of its requests 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 received and sending back an acceptance message. Bidding proposals from other clients are rejected. The accepted I/O request is appended to an ordered list on the associated client and the server respectively. 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 requests have been scheduled. After the scheduling phase completes, the clients send out data transfer requests one by one as planned in their ordered lists.

3.2.1. Distributed Matching Algorithm The algorithm, *Maximum-Size/Maximum-Load* (MS/ML), approximates the centralized algorithm *Maximum Weight Matching* (MWM) in a distributed environment.

Each client maintains a "current workload", which is the total amount of data in the pending requests in that client. Initially, the current workload is the sum of all the data requests in that client. Each time when a data request of a client is accepted by a server, the current workload of that client is decreased by the data size of that request.

During the first step of the bidding, each client selects a pending request that has the largest data size, and sends a bidding proposal to the associated server. The proposal contains both the request and the current workload of that client. In the second step, each server resolves conflicts by selecting the bidding proposal from which the client has the largest workload, and it then appends the selected data request to its ordered list and sends back an acceptance (rejection) message to the chosen (rejected) clients. Upon receiving the acceptance message, the selected client appends the data request to its ordered list and updates its current workload accordingly. This *maximum size* selection on the client side and *maximum load* selection on the server side increase the chance that larger data transfers will be scheduled earlier than smaller ones, and that the data transfers with similar sizes will likely be scheduled simultaneously, thus reducing idle cycles.

3.2.2. Distributed Greedy Algorithm The algorithm, *Minimum-Size/Earliest-Completion-First* (MS/ECF), approximates the centralized algorithm *Earliest Completion First* (ECF) in a distributed environment. Central to the algorithm is an efficient and distributed scheme for maintaining the available times that allows fast selection of requests that could complete at the earliest time.

Similar to the centralized ECF algorithm, each client i maintains an available time *CAvail(i)* and each server j an available time *SAvail(j)*. Initially, all *CAvail(i)* and *SAvail(j)* are set to zeros. In addition, each server j maintains a client set C_j that will request data from j, and each client i maintains a server set S_i that i will request data from.

The distributed MS/ECF algorithm also has two steps for each bidding round. During the first step, each client i selects a server j from s_i from which i will request the smallest amount of data, and sends a bidding proposal to server j. The proposal contains both the request and the available time of client i. In the second step, each server resolves con-

flicts by selecting the bidding proposal that has the earliest completion time. The completion time of a bidding client can be computed by Equation 1 since the available time and data size of the client are known to the server once the proposal is received. The server then appends the selected data request to its ordered list of data transfer, updates its available time *SAvail(j)*, and sends back its available time in an acceptance (rejection) message to the chosen (rejected) clients. Upon receiving the acceptance message, the selected client appends the data request to its ordered list of data transfers and updates its available time *CAvail(i)* accordingly. Note that the available times are exchanged between clients and servers through the bidding proposals and the acceptance/rejection messages, thus the `MS/ECF` algorithm does not require additional message passing rounds. Figure 4 gives the pseudo code of `MS/ECF`.

```
Algorithm MS/ECF:

On each client i:
Repeat until all S_i become empty

    Select data transfer (i,j,w) that has
           the smallest size w.
    Send bidding proposal (i,j,w) to server j
           and wait for accept/reject message.
    if proposal accepted
       then the new pair (i,j) will be scheduled
       at time t = max(CAvail(i),SAvail_from(j))

    Remove j from S_i.
// update client available tim
    CAvail(i) = t + w
end repeat

On each server j:
Repeat until all C_j become empty
    Wait for |C_j| bidding proposals and choose
         client i such that data transfer (i,j,w)
         has the earliest completion time.
    The communication between i and j will be
         scheduled at time t,  where
         t = max(CAvail_from(i), SAvail(j)).

    Send accept message and SAvail(j) to client i.
    Send reject message and SAvail(j) to clients
         in C_j - {i}

    Remove i from C_j.
//  update server available time
    SAvail(j) = t + w

end repeat
```

Figure 4. The pseudo code of Minimum-Size/Earliest-Completion-First (MS/ECF) algorithm.

4. Experimental Results

We conducted simulations to compare the performance of our distributed algorithms with the best distributed algorithm, *Highest Degree First* (`HDF`), developed by Durand et. al. [9], for scheduling parallel I/O with non-uniform data transfer sizes.

`HDF` [9] is a distributed algorithm based on edge coloring. Data transfers are divided into units of fixed-size blocks and are handled one block at a time. Each client selects one of its requests uniformly at random and sends its id and current degree to the associated server. Each server grants the request of the highest degree client, with ties broken arbitrarily.

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 number of servers, the ratio of busy servers and the ratio of large-size data transfers on the performance of different scheduling algorithms.

We study the impact of biased I/O on the performance of the scheduling algorithms by varying the ratio of busy servers (RB) from 0% to 50%. In our experiments a non-busy server receives an average of 10 requests and a busy server received 40 to 50 times more data transfer requests than a non-busy server. The busy servers are chosen randomly from the set of all servers. When the ratio of busy server is set to 0, there will be no hot spot and all the servers receive approximately equal number of requests. As the ratio of busy server increases, the total number of data requests also increases.

We also study the impact of data size on the performance of the scheduling algorithms. In our experiments, data transfers are a blend of small and large data requests. Large data sizes are chosen based on a uniform distribution function in the range of 0.95*32MB to 1.05 *32MB, and small sizes are chosen from the range of 0.95*128KB to 1.05*128KB.

We use throughput instead of execution time as the measurement metric of performance since the data size, the number of data requests, and the number of servers vary in our experiments. The throughput is defined by M/T, where M is the total data size of the requests and T is the parallel completion time of all requests, including scheduling and data transfer time. Each data point in the performance figures is the average of 100 independent runs.

4.1. Effects of the Number of Servers

We vary the number of servers (NS) from 8 to 128 while the values of other parameters remain fixed. The block size

(PS) used for HDF in dividing data requests is 32k. Figure 5 shows that MS/ML and MS/ECF outperform HDF by a factor of 200% to 500%. The improvement increases as the number of servers increases. Furthermore, MS/ML performs the best when the ratio of large data transfers is low (e.g. RL=0.1, as in Figure 5(a) and (c)), while MS/ECF yields the best performance when the ratio of large data transfers is high (e.g. RL=0.8, as in Figure 5(b) and (d)).

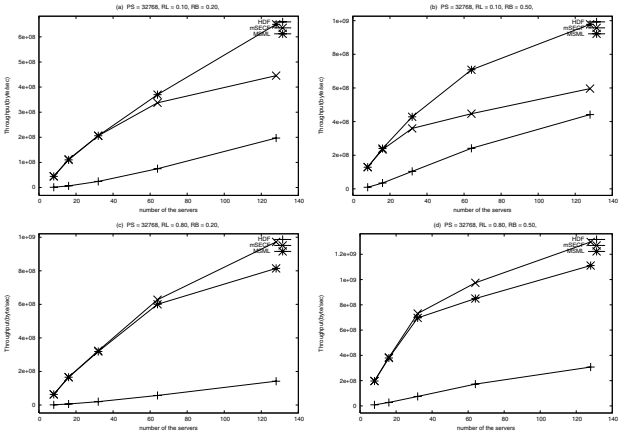

Figure 5. Parallel I/O throughput under different number of servers.

4.2. Effects of the Ratio of Busy Servers

In our simulation we vary the ratio of busy servers from 0.1 to 0.5 and fix the value of other parameters. As Figure 6 indicates, both MS/ML and MS/ECF are superior to HDF in all cases. The improvement increases when the ratio of busy servers increases. Furthermore, when the number of servers is small (e.g. NS=32, as in Figure 6(a) and Figure 6(c)), MS/ML and MS/ECF are competitive to each other. In systems with more servers (e.g. NS=128), MS/ML performs better than MS/ECF when there are fewer large data requests (e.g. RL= 0.1, as in Figure 6(b)), while MS/ECF is better than MS/ML when there are more large data requests (e.g. RL = 0.8, as in Figure 6(d)).

4.3. Effects of the Ratio of Large Data Transfers

We also vary the ratio of large data transfers and fix the values of other parameters. Figure 7 shows that both MS/ML and MS/ECF perform better than HDF by a factor of 150% to 800%. The speed-up factor increases when the ratio of large data transfers increases. A possible reason is

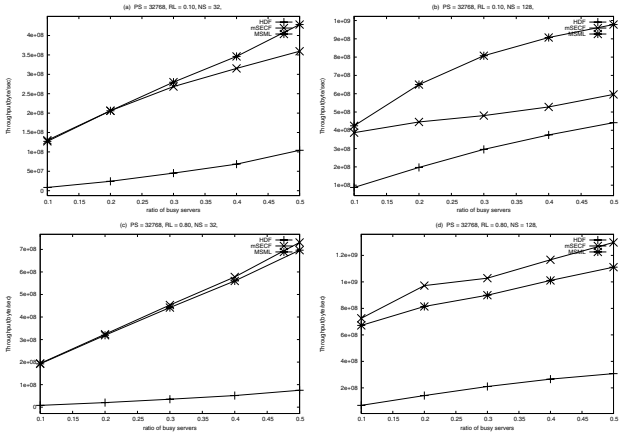

Figure 6. Parallel I/O throughput under different ratio of busy servers.

that when there are more large data transfers, HDF generates more bidding requests and thus requires more rounds of bidding, transmission, and synchronization, which in turn increase communication overheads. Furthermore, MS/ML performs better than MS/ECF until the ratio of large data transfers (RL) reaches a threshold value, and beyond that point MS/ECF becomes superior to MS/ML. This is consistent with those results shown in Figure 5 and Figure 6.

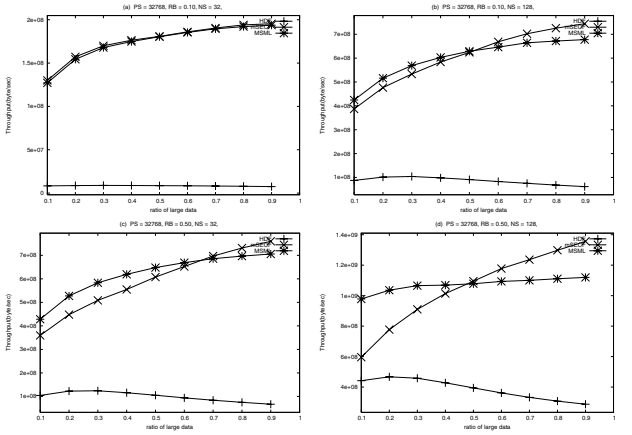

Figure 7. Parallel I/O performance under different ratio of large requests

4.4. Summary of Results

The three sets of experiments show that both MS/ML and MS/ECF yield better performance than HDF in all cases.

MS/ML and MS/ECF also have good scalability in large systems (Figure 5), and are stable in non-uniform data transfer patterns (Figure 6). Furthermore, MS/ECF performs better when most of the requests are of large data sizes, while MS/ML is more suitable for data transfers with smaller sizes (Figure 7).

5. Conclusion

In this paper we present two distributed algorithms to schedule parallel data transfers of non-uniform sizes. The *Maximum-Size/Maximum-Load* (MS/ML) heuristic favors processors with heavy I/O load and the *Minimum-Size/Earliest-Completion-First* (MS/ECF) takes available time of the processors into consideration in order to reduce idle cycles in data transfers. We measure the completion time (including scheduling time and data transfer time) required to complete a set of data transfers with these algorithms, and evaluate these two approaches. Our experimental results show that both algorithms achieve good performance. When compared with the results achieved by their centralized counterparts (the *Maximum Weight Matching* (MWM) algorithm and the *Earliest Completion First* (ECF) algorithm), both MS/ML and MS/ECF yield parallel performances within 6% of the centralized solutions.

We also compare the performance of our algorithms with the *Highest Degree First* (HDF) algorithm, which handles non-uniform data transfers by dividing them into blocks of fixed-size, which are then scheduled and transferred one at a time. Experimental results show that our algorithms require both less scheduling and data transfer time, resulting in a better overall parallel I/O performance. Our simulations also show that MS/ML is more suitable for parallel I/O with lighter data transfer traffic, while MS/ECF is more suitable for parallel I/O with heavy data transfer traffic.

References

[1] C. Berge. *Graphs*. North Holland, Amsterdam, 1985.

[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. L. 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. *Theoretical Computer Science*, 203(2):225–251, 1998.

[7] D. Durand, R. Jain, and D. Tseytlin. Applying randomized edge coloring algorithms to distributed communication: An experimental study. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 264–274, July 1995.

[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*, 63:611–617, 2003.

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

[11] 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.

[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. *IEEE Trans. On Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[15] 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.

[16] 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.

[17] A. Panconesi and A. Srinivasan. Fast randomized algorithms for distributed edge coloring. In *Proc. 1992 ACM Symposium on Parallel and Distributed Computing*, pages 251–262, 1992.

[18] 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.

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

[20] 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.