This is the published version (version of record) of:


Available from Deakin Research Online:

http://hdl.handle.net/10536/DRO/DU:30006072

Copyright : ©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Parallel I/O Scheduling in the Presence of Data Duplication on Multiprogrammed Cluster Computing Systems

J. H. Abawajy
Deakin University,
School of Information Technology,
Geelong, VIC., 3217, Australia.

Abstract

The widespread adoption of cluster computing as a high performance computing platform has seen the growth of data intensive scientific, engineering and commercial applications such as digital libraries, climate modeling, computational chemistry, computational fluid dynamics and image repositories. However, I/O subsystem performance has not been keeping pace with processor and memory performance, and is fast becoming the dominant factor in overall system performance. Thus, parallel I/O has become a necessity in the face of performance improvements in other areas of computing systems. This paper addresses the problem of parallel I/O scheduling on cluster computing systems in the presence of data replication. We propose two new I/O scheduling algorithms and evaluate the relative performance of the proposed policies against two existing approaches. Simulation results show that the proposed policies perform substantially better than the baseline policies.

Keyword: I/O Scheduling, Cluster computing, Parallel I/O, Performance analysis.

1. Introduction

Recent advances in low-latency, high-speed network technology coupled with inexpensive commodity processors have lead to the emergence of cost-effective parallel computing platform commonly known as a cluster computing [4]. As the ubiquity of cluster computing has grown in the past decade, so did the execution of data intensive scientific and engineering applications such as climate modeling, computational chemistry, computational fluid dynamics and image repositories on clusters. To date, significant research efforts have been directed to parallelization of computation as well as finding effective strategies for mapping of parallel tasks to cluster machines [26].

Although both parallelization and mapping strategies of parallel jobs have improved system utilization and response time to certain extent, obtaining maximum performance of the parallel programs have remained elusive [3]. This is because many scientific and engineering applications have large I/O requirements, in terms of both the size of data and the number of files or data sets [20]. Moreover, regardless of the fact that disk technology has been advancing rapidly especially with respect to the storage density, capacity and bandwidth, I/O subsystem continues to be a major bottleneck in many parallel applications [22] [24] [3]. Therefore, in addition to computation parallelism, the parallelization of I/O operations is required for achieving optimum application performance [3].

Although, several parallel file systems (e.g., [19], [12], [5]) as well as optimizations techniques (e.g., [24], [9], [14]) have been developed to ease the I/O bottlenecks, I/O in cluster computing still remains an area requiring significant performance improvement [23]. This is because most of these previous work were tailored for the commercial supercomputers environment where the network in the parallel machines typically has much lower lat-
tency and much higher throughput than the storage system [22].

These observations suggest that, with the adoption of cluster computing as a high performance computing platform, some additional supporting technologies are required. Parallel I/O [6] is one such supporting technology that is capable of providing high speed data storage in cluster computing environments [22]. Parallel I/O can improve I/O performance by combining large numbers of storage devices and providing the system software to utilize them in concert. Data is distributed among the disks to enable simultaneous parallel access. Individual applications can potentially speed up their I/O by fetching data blocks in parallel from multiple disks, and buffering them in memory until required. Thus, parallel I/O systems have emerged and are beginning to see use in the mainstream [22].

In this paper, we address the problem of parallel I/O scheduling for multiprogrammed cluster computing environments. The motivation for studying this problem is that scheduling is necessary when several concurrent applications are simultaneously sharing the I/O system [2]. Also, it has been shown that the performance of carefully tuned parallel programs can slow down dramatically when they read or write files in cluster computing systems [16]. Although parallel I/O has extended the range of problems that may be solved on high performance cluster computing platforms, it is shown that peak performance is rarely attained from these coordinated storage devices [23] [27] and research into optimizing these systems is still an open area [22]. Thus, we believe that scheduling parallel I/O operations will become increasingly attractive and can potentially provide substantial performance benefits. However it is a challenge to schedule and coordinate the I/Os of myriad concurrent devices to meet the resource constraints and timing demands of the applications.

In this paper, we present a set of new parallel I/O scheduling algorithms for multiprogrammed cluster computing environments running parallel I/O workloads. Although it was noted that scheduling does have noticeable effects on performance in a single-application environment [22] [7] [2] [15], there is a lack of research that investigates the effectiveness of parallel I/O scheduling strategies for multiprogrammed cluster computing environments [7] [3]. In the multi-application environment, the parallel I/O system is a shared resource which usually impacts the I/O performance delivered to the simultaneously running applications. In addition, when file data in a parallel I/O system are spread across several nodes, failure of a node will make the data stored in that node unavailable. This is especially catastrophic for long-running scientific computations. Intuitively, with duplicated resources in a cluster, failure of a node may be masked or tolerated with perhaps a slight degradation of the service. But, so far this potential capability in data availability has not been fully exploited. We present simulation results showing that the proposed I/O scheduling algorithms can produce a substantial improvement over previous I/O scheduling algorithms.

The rest of the paper is organized as follows. Section 2 is an overview of the system of interest and related work. Section 3, discusses the proposed parallel I/O scheduling algorithms. The performance analysis of the proposed scheduling policies is discussed in Section 4. The results and discussions of the experiments are presented in Section 5. The conclusions and future directions are given in Section 6.

2. Background

Parallel I/O has recently drawn increasing attention as a promising approach to alleviating I/O bottlenecks in cluster computing. In this section, we will present the system model assumed in this paper. The scheduling problem and related work are also briefly described.

2.1. System Model

Figure 1 shows the architecture of high performance cluster computing system of interest. Such parallel I/O architectures consisting of multiple disks connected with high bandwidth interconnect are the norm in high-performance data centers and
The system consists of a set of M independent processors, \( P = \{P_1, P_2, \ldots, P_M\} \), and a set of N independent disks, \( S = \{S_1, S_2, \ldots, S_N\} \), that are connected by a fast interconnection network. This hardware configuration is currently supported by networks of workstations (NOWs) and by multicomputers (e.g., the IBM SP) and is likely to remain popular in the future because of the economic and physical attributes of such architectures [3].

Data is distributed among the disks to enable simultaneous parallel access. Data is stored on the disks in units of blocks; a block is the unit of access from a disk. In each parallel I/O operation a set of up to N blocks, one from each disk, can be accessed. The blocks fetched from the disks may be buffered in an internal memory buffer until they are required.

To achieve a high degree of data availability in parallel I/O systems, two basic approaches are currently being used. In the first strategy, data are replicated and multiple copies of the same data are stored on different disks. When one copy fails, the other copies can continue to be used, if they are updated synchronously. The failure is thus transparent to the users and no interruption of the services will occur. In the second strategy, data are spread across an array of disk drives along with the redundant error detection/correction information, e.g., parity bytes. When errors are discovered, the redundant information can be used to restore the data and application programs can continue using the data. In this paper, we use the first strategy.

In this paper, we assume that a variety of workload types generated by a community of users, which include parallel applications and a stream of local jobs compete for system resources. The parallel workloads of interest to us are those characterized as the CPU I/O bound applications [16] [15]. These applications demand a great deal from underlying storage systems and software, and both high-performance distributed storage and high level interfaces have been developed to fill these needs. However, it is a challenge to schedule and coordinate the I/Os of myriad concurrent devices to meet the resource constraints and timing demands of the applications. The following subsection briefly introduces the scheduling problem and related work.

All I/O requests from a task is sent to the storage resource manager (i.e., data scheduler) in the I/O subsystem, which coordinates the I/O request on multiple I/O nodes for greater efficiency. In this manner, the parallelism of the user application can be preserved when performing I/O. It is easy to imagine that for a large parallel application, a large number of I/O requests might be in service on an I/O Scheduler at one time. This large number of I/O requests provide us with an opportunity to optimize by selecting the order in which jobs will be serviced.

A critical but often ignored component of system performance in cluster computing is the I/O subsystem [23]. In these environments it is desirable to provide QoS-based allocation of disk bandwidth to different applications sharing the I/O system. Thus, two issues of concern in multiprogrammed cluster computing are reducing the performance gap and providing fault-tolerance for long running applications. Even when considered separately the two problems are challenging. In the following section, we discuss the scheduling problem addressed in this paper.

### 2.2. Problem Statement

When several concurrent applications are simultaneously sharing the I/O system, the scheduling of the I/O requests becomes more complicated.
Parallel I/O scheduling is concerned with scheduling of parallel I/O operations with the goal of minimizing the overall I/O response times. The parallel I/O scheduling problem can be formulated as shown in Figure 2.

**Given**

1. a system that consists of a set of M independent processors, \( P = \{P_1, P_2, ..., P_M\} \), and a set of N independent servers (i.e., disks), \( S = \{S_1, S_2, ..., S_N\} \), that are connected by a fast interconnection network (see Figure 1).

2. a set of M independent applications or tasks concurrently accessing the I/O system. Each task is abstracted by a reference string \( R_i \) where \( 1 \leq R_i \leq M \); \( R_i \) is the ordered sequence of blocks that is required by that application.

**Scheduling Problem:** The problem is to find an assignment of disks to reference strings such that utilization of the disks is maximized while at the same time application response time is minimized.

---

**Figure 2. Parallel I/O Scheduling Problem**

There are two objectives of the above parallel scheduling problem: maximizing the utilization of the disks and minimizing the application response time. Two important factors commonly considered in I/O scheduling is the *data access time* (DAT) and the *data transfer time* (DTT). Here we focus on strategies that minimize the DTT overhead as the data transfer time is much greater than the access overhead in system we considered. Even when considered separately, the problems are challenging and can be shown to be in a class of NP-complete problems [11] [13]. Therefore, heuristics for I/O scheduling must be used [13].

A number of I/O scheduling heuristics have been described in the literature [25] [15] [1] [8] [11] [10] [16] [2] [26]. Most of this algorithms consider only an environment where data is not replicated and do not address the multiprogramming environments. In prior works, the I/O scheduling problem was modeled by a bipartite graph. For example, Durand, et. al. [10] proposed various bipartite graph edge-coloring algorithms for solving the scheduling problems. An edge-coloring-based approximation algorithms for scheduling I/O transfers for systems that only allow at most k transfers at a time is discussed in [13].

Two approaches referred to as *Highest Destination Degree First* (HDDF) and the *Lowest Destination Degree First* (LDDF) are discussed in [8]. A scheduling algorithm called *Highest Degree Lowest Workload First* (HDLWF) is proposed in [26]. 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. HDLWF is a minor modification of an algorithm discussed in [16] by the same authors. A class of decentralized parallel I/O scheduling algorithms based on edge-coloring and matching of bipartite graphs is discussed in [10].

Data replication is commonly used for computation-intensive or data-intensive applications on distributed systems, both for reliability and performance reasons [26]. Data replication is necessary to ensure availability of data. Furthermore, data replication is also frequently used for better performance of distributed systems. 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 [26].

Parallel I/O scheduling in the presence of data replication has been studied in [15] [26]. The scheduling strategies studied in [15] are based on various knowledge of I/O patterns such as the sum of the service demands of jobs. The problem with the work of [15] is that it is difficult if not impossible to obtain application I/O *a priori*. The work of [15] can be said at best to have theoretical values since the policies proposed are based on *a priori* knowledge of the job execution time. Unfortunately, it is very difficult if not impossible to know the pending I/O service demands of a job.
Our work differs from existing approaches in that the existing parallel I/O scheduling policies focus on uniprogrammed systems that run single jobs at a time in isolation. In some cases, these techniques have also been extended to multiprogrammed environments that execute multiple parallel jobs simultaneously [15]. Therefore, most studies have not investigated the potential performance problems of handling large outstanding I/O requests in a multi-workload environment.

3. Parallel I/O Scheduling Strategies

Parallel I/O is a necessary component of data-intensive applications such as scientific simulations. However, it is a challenge to schedule and coordinate the I/Os of myriad concurrent devices to meet the resource constraints and timing demands of the applications. To make best use of available parallelism and locality in I/O accesses, it is necessary to design and implement scheduling algorithms that schedule I/O requests intelligently. To this end, we propose two new adaptive parallel I/O scheduling policies by extending the work reported in [3]. The extension is to direct each application 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.

3.1. Equi-Partition Policy

The Equi-Partition (EQUI) algorithm has two main parts. The first part is to determine a target size (i.e., the number of I/O requests) that can be assigned to an I/O server at scheduling point. Let \( S = \{S_1, S_2, ..., S_N\} \) be a set of independent servers. Let \( D \subseteq S \) denotes the number of servers that can service the outstanding I/O requests. At each scheduling iteration, the algorithm first determines the average number of I/O requests (i.e., \( \text{Avg} \)) to be handed to each I/O server as follows:

\[
\text{Avg} = \frac{R_{\text{pending}}}{D}
\]

where \( R_{\text{pending}} \) is the number of outstanding I/O requests.

The second part of EQUI is to select appropriate I/O server and assign the requests to it. Let \( OR_k \) be a subset of outstanding I/O requests (i.e., \( OR_k \subseteq R_{\text{pending}} \)) that can be serviced by an I/O server \( D_k \). While not all pending I/O requests are assigned (i.e., \( D > 0 \)), the algorithm selects an I/O server \( D_i \in S \) with \( OR_k > 0 \) and the \( OR_k \) is the lowest among the I/O servers. It then removes a set of \( \text{min}(\text{Avg}, OR_k) \) unassigned I/O requests from the I/O request pending queue and forwards it to server \( D_i \). This process repeats until all pending I/O requests are assigned to the I/O servers.

3.2. Adaptive Equi-Partition

In the Adaptive Equi-Partition (AEQU) policy, at each scheduling point, the AEQU first determines the average number of I/O requests (i.e., \( \text{Avg} \)) to be allocated to each I/O server as follows:

\[
\text{Avg} = \frac{R_{\text{total}}}{D}
\]

where \( S = \{S_1, S_2, ..., S_N\} \) be a set of independent servers and \( D \subseteq S \) denotes the number of servers that can service the outstanding I/O requests. \( R_{\text{total}} \) is the total number of outstanding I/O requests and computed as follows:

\[
R_{\text{total}} = R_{\text{pending}} + R_{\text{backlog}}
\]

where \( R_{\text{pending}} \) is a subset of outstanding I/O requests; and \( R_{\text{backlog}} \) is the total number of locally backlogged (i.e., assigned to I/O servers but not yet processed) I/O requests.

The I/O servers are then sorted in an increasing order based on the total number of locally backlogged I/O requests. The algorithm then assigns pending I/O requests to under loaded I/O servers while re-scheduling I/O requests from overloaded I/O servers onto the under loaded servers.

3.3. Baseline Scheduling Policy

We use the Highest Destination Degree First (HDDF) and the Lowest Destination Degree First (LDDF) [8] as the baseline scheduling policies.

In LDDF policy, I/O requests are mapped to the I/O servers in a round robin fashion. The algorithm first sorts the I/O servers in an increasing
order of the pending I/O requests that they can possibly service. Starting with an I/O server that has the lowest number of pending I/O requests, the algorithm removes a pending I/O request for the I/O servers from the queue and assigns it to the server. It then picks the next I/O server, removes a pending I/O request for the I/O servers from the queue and assigns it to the server. This process continues until there is no more pending I/O requests.

The HDDF is similar to the LDDF policy except the algorithm sorts the I/O servers in a decreasing order of the pending I/O requests that they can possibly service. It also chooses the I/O server with the highest number of pending I/O requests first.

4. Performance Analysis

This section evaluates the performance of the proposed algorithms and compare them with the baseline policies using simulation. We used normalized mean response time (MRT) as a performance metric. We implemented the proposed algorithms using event-driven simulator written in C. In the following subsections, we describe the simulation environment and the workloads in detail.

The simulator employs 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. The file sizes range from 1 MB to 16 MB. Files were written and read by making from 1 to 128 request per file. A delay was placed between each request ranging from zero to one second. This delay represents the time for which a real application would perform computation or wait for some reason other than I/O. For validation, a batch strategy is used to compute confidence intervals (at least 30 batch runs were used for the results reported in this paper).

We generate two competing I/O workloads synthetically. We studied two CPU-I/O bound workloads referred to as \( W_1 \) and \( W_2 \), which are the same workloads used in [15]. These two workloads are characterized by alternating CPU and I/O phases that repeat \( K \) times as observed in [21] [18]. However, they differ primarily in the amount of I/O parallelism and when the I/O is performed. The request size of the I/O requests are distributed normally with a mean of 8 blocks of 512 bytes. The ratio of reads to writes is set to two, as used in other QoS work [17]. We use a model of [15] to determine the proportion of the computation and I/O times for each task.

5. Results and Discussions

In this section, the results of the experiments are discussed. In all experiments, we used 32 nodes for computation and 3 storage nodes unless explicitly specified.

![Figure 3. Relative Performance of the Policies.](image)

Figure 3 shows the relative performance of the four I/O scheduling policies for the two workloads. The data shows that the proposed I/O scheduling policies are quite efficient with respect to the baseline policies under both workloads.

Under \( W_2 \), we observe that: (1) the Lowest Destination Degree First performs somewhat better than the Highest Destination Degree First, which confirms with the results reported in [8]; (2) the Equi-partition policy performs better than both the Lowest Destination Degree First and the Highest Destination Degree First policies. This is
because the *Equi-partition* policy distributed the I/O requests in a balanced fashion over the I/O servers whereas the other three policies do not; and (3) the *Adaptive Equi-partition* is the best as this policy allocates pending I/O requests as well as move unprocessed I/O requests from heavy loaded to lightly loaded I/O servers.

Under W1 Workload, the graph shows that the *Equi-partition* policy performs better than the *Lowest Destination Degree First* and *Highest Destination Degree First* policies. This is because the *Equi-partition* policy distributed the I/O requests in a balanced fashion over the I/O servers whereas the baseline policies do not. The *Adaptive Equi-partition* is the best as this policy allocates pending I/O requests as well as move unprocessed I/O requests from heavy loaded to lightly loaded I/O servers.

### 6. Conclusions and Future Directions

Parallel I/O is a necessary component of data-intensive applications such as scientific simulations. As a result, parallel I/O architectures are increasingly deployed for high performance computing and in shared data centers. Thus, the I/O bottleneck in cluster computing systems has recently begun receiving increasing attention. In this paper, we addressed the problem of effective management of parallel I/O in multiprogrammed cluster computing systems by using appropriate I/O scheduling strategies. We presented two new parallel I/O scheduling policies and demonstrated their effectiveness by comparing their performance with two other existing parallel I/O scheduling policies.

We are currently investigating their performance through experimentation with I/O traces taken from the scientific and non-scientific application domains, including web-servers and interactive applications. We are also looking at implementing and testing the proposed scheduling policies on. Using knowledge of the application domain, it is possible to achieve much more efficient I/O than more general solution. In this paper we assumed homogeneous cluster computing environment. We plan to study the performance of the proposed scheduling policies in heterogeneous clusters.

### Acknowledgments

Financial support is provided by Deakin University. My thanks to Maliha Omar (Shubbee) without whom I could not have completed this report.

### References


