ABSTRACT

SREENIVASA, ABHISHEK B. I/O Coordination for Co-Running Scientific Applications to Improve Parallel I/O Performance. (Under the direction of Xiaosong Ma.)

Despite advancements in the I/O subsystem, there is an ever increasing performance gap between computation and I/O in high-performance computing (HPC) systems making I/O highly bottleneck prone. Although applications have dedicated access to compute nodes, the file system is typically shared by all applications running on the system. Without coordination, applications interfere with each other while accessing the file system, resulting in significantly degraded application and file system performance.

In our work, we focus on improving the I/O performance of scientific applications that perform checkpoint I/O by reducing I/O contention between co-existing jobs. We approach this by proposing two different mechanisms—backoff and locking—that allow I/O coordination between applications that contend for file system resources. The aim of I/O coordination is to prevent multiple applications from performing checkpoint I/O simultaneously. The benefits of coordination are better files system throughput and application performance. With *backoff*, each application writes a small amount of dummy data to the file system before starting a checkpoint operation. Based on the perceived throughput of this probing operation, the application decides to either back off and probe again, or perform the checkpoint operation. With *locking*, each application acquires a lock from a centralized server before performing the checkpoint operation. The centralized server services applications in FIFO order, allowing only one application to acquire the lock at any given time.

We evaluate our proposed approaches with pseudo applications generated with the IOR benchmark and tested on the NCSU ARC cluster. We compare the results using

the baseline system (without any I/O coordination scheme), backoff, and locking. Our experiments showed an average of 19.22% and 81.38% reduction in I/O completion time with backoff and locking, respectively, over base case with a system running 10 jobs simultaneously. We noticed a significant reduction in makespan time of the application set with our proposed mechanisms. The percentage variance in checkpoint duration reduces with locking. Backoff, however, shows an increased percentage variance than baseline as it relies on randomized back off time, but with a lower mean than baseline.

I/O Coordination for Co-Running Scientific Applications to Improve Parallel I/O Performance

by
Abhishek B. Sreenivasa

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

Frank Mueller	Rudra Dutta
Zhe Zhang	Xiaosong Ma Chair of Advisory Committee

DEDICATION

To my parents and teachers.

BIOGRAPHY

Abhishek B. Sreenivasa was born in Bangalore, India on February 15th, 1985. He received Bachelor of Engineering degree in Electronics and Communication in June 2007 from R.V. College of Engineering. After graduating, he worked as a software development engineer at Cisco Systems, Bangalore. He began his Master of Science in Computer Science at North Carolina State University in August 2009. After graduation, he will join Amazon.com, Inc. as a software development engineer.

ACKNOWLEDGEMENTS

I'm very thankful for my advisor, Dr. Xiaosong Ma, for guiding me in my graduate studies for the past two years. It's not often that one gets an advisor who can motivate, be inspirational, show confidence in your work, allow you to work on cutting-edge technology, and be cool. I'm grateful for her for being all that.

I'd like to thank Dr. Zhe Zhang for being my mentor and guiding me through my research work. I would also to thank the committee members, Dr. Frank Mueller and Dr. Rudra Dutta, for their guidance. Our work would not have been possible without NCSU's ARC cluster. I'd like to thank Dr. Frank Mueller again for setting up this system.

It's been a pleasure working with PALM research group for the last two years. I'm thankful for interesting interesting discussions and help that I've received over these years.

I had the opportunity of working with some of the great minds at Oak Ridge National Laboratory. I'd like to thank Galen Shipman for giving me an opportunity to intern in Technology Integration group. I'd like to express my gratitude to Kenneth Matney, Sr. and Scott Klasky for guiding me.

Last but not the least, I'd like to thank all friends, roommates, and family, especially my parents, for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

High-performance computing (HPC) systems are used to solve a variety of scientific problems. They comprise of hundreds to thousands of compute nodes, fast interconnects, one (or few) file system(s), and specialized software.

The computational performance of high-performance computing (HPC) systems has increased tremendously over the years with the current state-of-the-art systems reaching several petaflops [9]. Among other things, the scalability of compute nodes, with top-of-the-line systems comprising of several thousand nodes, is responsible for the increase in computational power. Despite advances in I/O subsystem, the growth of compute power has outpaced that of the I/O throughput, making I/O increasingly bottleneck prone [23].

In this work, we focus on reducing the I/O resource contention between parallel applications that execute simultaneously. Note that most scientific applications do not scale up to utilize all the compute nodes available on a system for a number of reasons. Applications are tuned to strike the right balance between the size of the data set involved in simulation and the number of compute nodes. Increasing the number of compute nodes to scale up the system size in a simulation may not necessarily improve the performance of

the application as it may increase the communication and I/O overhead. With imbalance in performance of components in HPC system, applications are configured to utilize only a part of the compute nodes available on the system. To improve the utilization of the system's compute resources, more than one application, each having exclusive access to a set of compute nodes allocated to it is allowed to run on the system simultaneously. Compute resource managers and schedulers are responsible for scheduling applications. A variety of scheduling algorithms are available [18] that improve compute node utilization.

HPC systems are typically equipped with one (or few) shared file system(s). Unlike compute nodes that are exclusive to a single application, file system is shared by all applications running on a cluster or supercomputer. In addition, file system also handles requests from interactive commands from the user. Jaguar [12], the leadership-class supercomputer managed by Oak Ridge National Laboratory, TN, comprises of 18,688 nodes and uses a Lustre file system called Spider. The scale of applications running on Jaguar is such that typically, several dozen applications run simultaneously, and all applications share the Spider file system. This facility manages several other supercomputers. In order to prevent islands of data spread across several file systems, all systems mount Spider [33].

Scientific applications comprise of several iterations of compute and checkpoint phases. In a checkpoint phase, the state of the simulation is written to the file system. These checkpoint files are used for restarting the simulation from a given state, analyzing intermediate data from a running scientific simulation, or stitching the data to form a movie to analyze the objects of simulation over a period of time. In this work, we use the terms scientific application and checkpoint-based application interchangeably. Unless otherwise specified, I/O or I/O operation refers to checkpoint I/O operation. Examples of scientific applications that perform checkpoint-based I/O are Gyrokinetic Toroidal Code, Parallel

Table 1.1: Variation in checkpoint I/O completion time of S3D

Description	Min. time (s)	Max. time (s)
Original S3D	0.22	46.53
S3D with Darshan	0.79	10.22
S3D with IOTA	0.43	98.1
S3D with Strace	1.75	135.28

Ocean Program, S3D, Community Climate Simulation Model, etc.

With a shared file system and more than one application allowed to run on the system, the I/O phases of applications may overlap. The consequences of overlapped I/O are multi-fold. Firstly, simultaneous I/O requests from several applications result in increased disk seeks, so the peak I/O throughput of file system is not achieved. Secondly, the caches close to the file system will be shared by several applications, forcing smaller data transfer into the caches. Finally, I/O completion rate reduces as I/O cycles are split between applications resulting in increased idle time on compute nodes. All these contribute to increased and highly varying application runtime. From an economic perspective, increased application time results in increased costs as applications are billed on the number of CPU-hours of compute nodes they utilize. So checkpoint operation is becoming an expensive operation [27], forcing application developers to reduce the frequency of such operations.

During a study of performance of I/O tracing tools that we conducted on Jaguar, we noticed huge variance in I/O performance when the same application was run several times. We used S3D [29] application configured to perform a collective checkpoint write of 2.02GB. Table 1.1 shows minimum and maximum time to finish the checkpoint I/O under various tracing tools. The performance is largely dependent on the number of applications

running on the system and their checkpoint I/O pattern. Generally, applications running during off-peak hours, when there are fewer job running simultaneously, have high I/O throughput.

1.1 Research Overview

To address the I/O contention and performance problem discussed earlier, we present two I/O coordination mechanisms. In the first mechanism called backoff, an application writes a small amount of dummy data into the file system to determine its throughput just before performing a checkpoint operation. If the perceived throughput is less than a specified threshold, the application backs off for a random duration of time before it starts with probing all over again. If the perceived bandwidth is above the specified threshold, the application proceeds with checkpoint I/O operation. In order to prevent starvation, applications are allowed to perform checkpoint I/O operation when it reaches a certain number of unsuccessful probings.

The second I/O coordination mechanism is called locking. In this approach, only one application is allowed to perform checkpoint I/O at a time. This is enforced by requiring each application to request a lock to use the file system before performing checkpoint I/O. A central locking server is used to manage such a resource lock. The applications communicate with this server to acquire and release a lock. Fairness is ensured and starvation is avoided by serving the applications in FIFO order.

As a proof of concept, we use the NFS file system [14] in our experimental evaluation. However, we propose ways to handle parallel file systems with multiple file servers for both the mechanisms.

In summary, the major contributions of this work are:

- We confirm the need for I/O coordination in checkpoint-based applications, with large-scale experiments on a leadership-class supercomputer.
- We propose two I/O coordination mechanisms and present their design and implementation.
- We perform preliminary evaluation of our proposal with pseudo-applications based on I/O benchmarks.

1.2 Thesis Outline

The rest of the thesis work is organized as follows. In chapter 2 we present background information. Chapter 3 covers design and implementation of probing and locking mechanisms. In chapter 4, we evaluate our work through experimentation. Chapter 5 presents an overview of related research. Finally, we provide our conclusion along with future work in chapter 6.

Chapter 2

Background

2.1 IOR Benchmark

Interleaved or Random (IOR) is a benchmark used for analyzing I/O performance of parallel applications [3]. It allows users to perform I/O at various levels of abstraction, namely POSIX, MPI I/O, HDF5, and parallel netCDF.

IOR supports both read and write operations. With write operation, IOR can be configured to check the integrity of the written data. When supported by the I/O abstraction, collective operation can be configured for both read and write.

A diverse set of workloads, such as file-per-process POSIX I/O, single task POSIX I/O, file-per-process MPI I/O, collective MPI I/O, etc., can be generated using configurable parameters [32]. These parameters can be specified in a script file or as command line parameters to IOR command.

Because of its versatility, we chose IOR benchmark to generate different workloads with checkpoint operation. We created a script to automatically create IOR configurations with a fair degree of randomness in configuration to prevent experimental bias. To

```
int MPI_xxx (...)
{
    int return_value;

    /* Pre-call logic */

    /* Standard MPI call */
    return_value = PMPI_xxx (...);

    /* Post-call logic */

    return return_value;
}
```

Figure 2.1: Extending MPI using PMPI

obtain detailed information about the various phases of the workload, we instrumented the IOR code to generate additional information in its report. This is discussed in detail in section 4.1.2.

2.2 PMPI

PMPI is the profiling layer of MPI that allows profiling, tracing, and extending the functionality of MPI calls [5]. With this interface, MPI calls can be intercepted and their logic can be modified by wrapping standard MPI calls between user-defined code. This is supported in C, C++, and Fortran.

An implementation of MPI that supports PMPI, such as Open MPI, provides the MPI APIs as weak links. This allows users to define MPI functions without link-time conflicts. The standard implementation of MPI can then be called using PMPI_ prefix instead of MPI_ prefix. Figure 2.1 illustrates how a standard MPI call is extended using PMPI.

Open MPI implementation of MPI is implemented using C regardless of the language

in which the MPI APIs are used [7]. So PMPI can be extended for all languages by just providing profiling wrappers in C.

In this work, we use PMPI layer to setup TCP connection between the client processes and the locking server during MPI initialization (MPI_Init). At the termination of the application, we tear down the TCP connection ($MPI\_Finalize$). In future we plan to use PMPI to embed backoff and locking calls inside MPI I/O calls and free the user from modifying the application source code to instrument I/O coordination APIs.

Chapter 3

I/O Coordination Mechanisms

3.1 Backoff

To address the performance issues arising from lack of I/O coordination, we propose backoff. It is an I/O coordination mechanism in which an application probes the file system throughput with a small dummy data write and decides whether to back off and probe again, or proceed with checkpoint I/O. It is a self-regulation mechanism since it's the applications that decide whether to back off rather than a central server deciding as in locking. Since probing is closely associated with backoff, we use these terms interchangeably to refer to these mechanisms.

Without any form of coordination among applications, their checkpoint I/O phases may overlap resulting in degradation in performance of the file system and the I/O completion rate. It is also possible for other forms of file system activities, such as heavy interactive commands, to overlap with checkpoint I/O. By backing off when file system activity is high, the applications are idle for a while. However, this ensures that file system operates at maximum throughput resulting in higher I/O completion rate.

The inspiration for our proposed backoff scheme is drawn from binary exponential backoff and truncated binary exponential backoff [10] schemes in computer networking. In networking, the shared resource is the network medium. A node that wants to transmit data senses the channel before transmitting. If the medium is being used, the node backs off for a duration that is a multiple of a fixed time and a random value between 0 and $2^x - 1$, where x is incremented by 1 after every unsuccessful transmission attempt. The truncated binary exponential backoff places a ceiling on the value of x.

3.1.1 Design

Algorithm 1 Backoff pesudocode

```
1:  iter ← 0
2:  exp ← STARTING_EXPONENT
3:  while true do
4:      iter ← iter + 1
5:      beginTime ← getTime()
6:      writeDummyDataToFile(DATA_SIZE)
7:      endTime ← getTime()
8:      time ← endTime − beginTime
9:      rate ← DATA_SIZE/time
10:     if (rate ≥ THRESHOLD) or (iter = MAX_COUNT) then
11:         break
12:     end if
13:     sleepTime ← random() mod BASE^exp
14:     sleep(sleepTime × BACKOFF_TIME_CONST + FIXED_SLEEP_TIME)
15:     exp ← exp + 1
16: end while
```

Algorithm 1 shows the basic principle behind backoff mechanism. In lines 5–9, the file system is probed by writing dummy data of a configurable size to a file. Lines 10–12 is used to decide whether to break out of the while loop and perform checkpoint I/O,

or back off and repeat the probing. We use only write in probing as checkpoint I/O is predominantly a write operation.

There are a number of parameters that can be tuned to vary the performance of the prober. They are STARTING_EXPONENT, DATA_SIZE, THRESHOLD, MAX_-COUNT, BASE, and FIXED_SLEEP_TIME.

Determining File System Activity

The file system throughput is determined by determining the time it takes to complete writeDummyDataToFile(). The writeDummyDataToFile() takes write size as a parameter. Each time this function is invoked, a new file is created, DATA_SIZE bytes of dummy data is written to the disk, the buffers are flushed, and the file is closed.

DATA_SIZE is an important parameter that we determine heuristically. In order to reduce the overhead of probing on the file system, the DATA_SIZE must be as small as possible. However, a small data size is susceptible to caching effects resulting in both false positives and false negatives. So it is critical to choose the right data size.

The THRESHOLD constant is the parameter that decides whether an application should repeat probing or continue with checkpoint I/O. The maximum throughput for a given DATA_SIZE can be determined by timing the writeDummyDataToFile() during off-peak hours when the file system activity is low. The THRESHOLD is set to 80–90% of this maximum throughput. A leeway of 10–20% is provided to account for inaccuracies in throughput performance measurement and to filter out small file system activities from determining the outcome of probing.

Backoff Parameters

The MAX_COUNT parameter allows applications to break out the loop after a number of consecutive probings and back offs. Without this option, the applications could starve if they happen to probe at a time when file system is busy or if the throughput of the file system decreases. Again, the value of this parameter is set heuristically.

The duration of back off is controlled by STARTING_EXPONENT, BASE, FIXED_-SLEEPING_TIME parameters. The actual back off time is determined by the sum of sleepTime and FIXED_SLEEP_TIME, where sleepTime is a random variable with uniform distribution taking values between 0 and $BASE^{exp} - 1$.

Design on Parallel File System

Parallel file systems are used in HPC systems to achieve high performance by striping the data across multiple file servers. Commonly used parallel file systems in HPC systems are Lustre [4], GPFS [30], PVFS [15], etc.

Although our work is a proof of concept focusing on single file server on NFSv3 [14], we explain how backoff and locking mechanisms can be extended to parallel file systems with multiple file servers.

A file that is created on a parallel file system is striped across multiple file servers. File systems provide APIs to get or set the file servers across which a file is striped. In order to probe the correct file servers, the prober needs the list of of file servers across which a checkpoint file is striped. This requires prober APIs to be called after the checkpoint file is opened or created and accept file handle as a parameter. Each file server associated with the checkpoint file is probed by writing to a file with dummy data striped across these disks. If all the file servers do not pass the threshold, the application backs off and

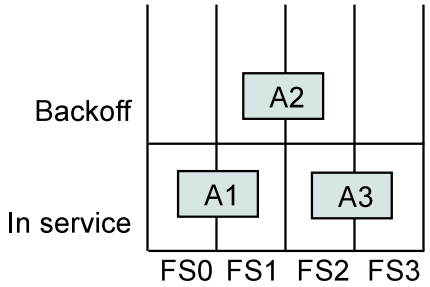

Figure 3.1: Probing on parallel file system

probes again. If not, the application performs the checkpoint operation.

Figure 3.1 illustrates probing with three applications (A1, A2, and A3) and four file servers (FS0, FS1, FS2, and FS3). *In service* area refers to the applications that are performing checkpoint I/O on the file servers. *Backoff* area refers to applications that have backed off after unsuccessful probing. Let us assume that A1, A2, and A3 have overlapping I/O phases in time and are striped across two file servers each as show in in the figure. Let checkpoint I/O of A1 begin before A2 and A2 begin before A3. A1 probes FS0 and FS1 and proceeds with checkpoint as there is no other application performing I/O on these servers. A2 probes FS1 and FS2, does not pass the threshold in FS1 and backs off. A3 also performs the checkpoint operation as FS2 and FS3 are not serving other applications. We notice out-of-order checkpoint I/O operation with applications A2 and A3.

MAX_COUNT prevents applications from starving by setting a ceiling on number unsuccessful probings. Once the application reaches this limit, it is allowed to perform checkpoint operation.

Since the granularity of back off applies to the entire file, the application can not write data into file servers that pass the threshold if the overall proving fails. This could lead to underutilization of file servers. The application itself may not see any performance

```
/*
 * Called by MPI applications before doing checkpoint I/O
 */
void MPI_prober ()
{
    if (rank == 0) {
        prober();
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Figure 3.2: MPI prober

benefit if it is allowed to write checkpoint data into file servers that pass the threshold as I/O is considered complete only when checkpoint data is written to file servers that do not pass the threshold. The file system performance may degrade due to underutilization, but we believe performance benefits from probing will overcome this.

3.1.2 Implementation

Our backoff API library is implemented in C and is dependent on MPI library. Figure 3.2 shows how the prober that is described in algorithm 1 is extended for MPI-based applications. Only a single MPI task is required to perform probing. In the figure, it is rank 0. The rest of the MPI tasks synchronize with rank 0 using a barrier.

The success of this mechanism is dependent on the accuracy of determining the file system throughput. Since each file system type behaves differently, certain file system specific changes are made to the prober peseudocode to improve its accuracy.

Priming

We noticed the distribution of maximum throughput of file system during off-peak hours was bimodal. When two writeDummyDataToFile() are called one after the other, the

throughput of the second call is usually higher. This affected the accuracy since prober calls that immediately followed another I/O call showed higher throughput than the ones that didn't. In order to reduce this type of inaccuracy, we introduced priming where a small amount of data is written just before calling writeDummyDataToFile(). From trial and error method of experimentation, we found 256KB of priming data is sufficient to fully throttle the writeDummyDataToFile() to maximum throughput in a quiet file system.

Min. Prober

In section 3.1.1, we described the impact of DATA_SIZE on probing accuracy. With small DATA_SIZE, the accuracy of probing is susceptible to caching effects. During experimentation, we observed that accuracy can be increased significantly by probing several times with small DATA_SIZE and reporting the smallest throughput value. A small delay is also introduced between successive probings to spread them apart.

3.1.3 APIs

- **void MPI_prober()**: Place this function just before a checkpint I/O operation. This function should be visible to all the tasks during execution.

3.2 Locking

Locking is an I/O coordination mechanism in which the file system is treated as a lockable resource for checkpoint operations. Applications request a centralized server, called locking server, to acquire the lock, perform the checkpoint operation, and release the lock by sending a message back to the locking server. The single, central locking server handles the requests from applications and manages the lock.

In order to improve the performance of applications and the file system, locking mechanism allows only one application to acquire the lock at a time. The locking server maintains a queue and serves application in FCFS order. Any application that wishes to acquire the lock is blocked until all applications ahead of it in the queue have been served.

3.2.1 Design

Message Types

Compute nodes and the locking server exchange three types of messages over the TCP stream:

- **REQ**: Lock request messages are sent by compute nodes to the locking server to indicate that the application is ready to perform checkpoint I/O.
- **ACK**: The acknowledgement is sent from the locking server to indicate that the application has acquired the lock and can proceed with checkpoint operation. The application is blocked until the receipt of this message.
- **REL**: When an application acquires the lock and finishes the checkpoint operation, it sends this message to the server to indicate that it is releasing the lock.

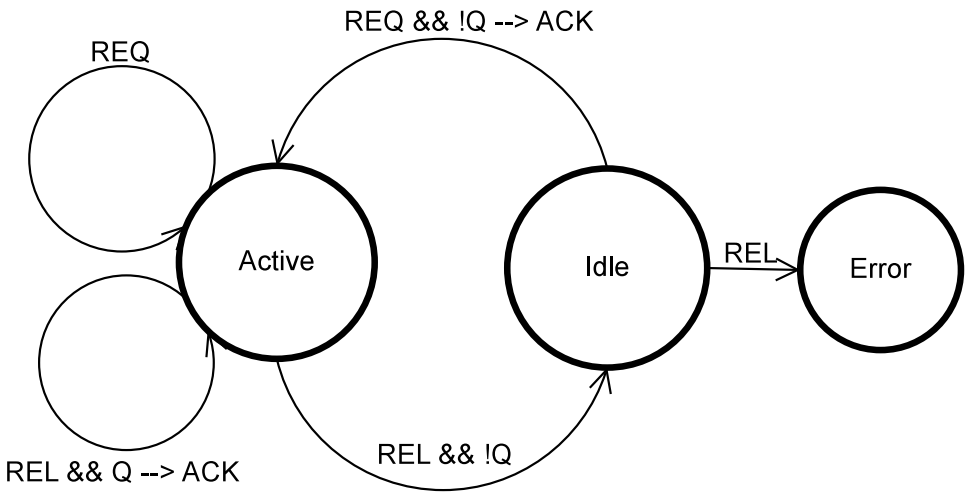

Figure 3.3: Lock state in locking server

Locking Server

The locking server is a multi-threaded server that handles several asynchronous requests from applications. For each application, it maintains a separate thread that listens to incoming messages. When a message arrives, it is registered as an event and enqueued in an event queue which is common to all the applications. A separate thread dequeues the event queue and handles the message from the applications.

The locking server maintains another queue, called service queue, which is used to the serve the applications in FCFS order. Applications whose lock request arrives when another application has acquired the lock are enqueued in this queue. When the application owning the lock releases it, the first element in the service queue is dequeued and the lock is owned by the application associated with that element.

Figure 3.3 illustrates that state diagram of the lock in the locking server. A lock in in idle state when it is not acquired by any application and active state when it is acquired. There is a third state to identify errors if applications do not handle the locks in a proper

way. The state transition occurs on REQ and REL events. The variable Q indicates the state of the service queue. Q evaluates to false if there are no elements in the queue, otherwise true. An ACK during the transition indicates that an ACK message is sent to the compute node during transition.

On a supercomputer the size of Jaguar with over 200,000 cores, there are not more than a few dozen applications running on the system simultaneously. Our locking server, despite being implemented in an interpreted language like Python, did not show any scalability issues when over a dozen applications were running simultaneously. So we believe scalability would not be a challenge on large-scale systems.

Application

The application is responsible for requesting and releasing the lock, i.e., sending REQ and REL messages. A single task from the application sends the request and the rest of the tasks are blocked by barrier synchronization until the node that sent the REQ gets an ACK message from the locking server.

Interaction Between Locking Server and Applications

Figure 3.4 illustrates the interactions between locking server and the applications. Application 1 acquires the lock before performing checkpoint I/O operation and the server receives another request from application 2. Application 2 waits for application 1 to finish the checkpoint operation and release the lock. Once the lock is released, the locking server allows application 2 to proceed with checkpoint operation.

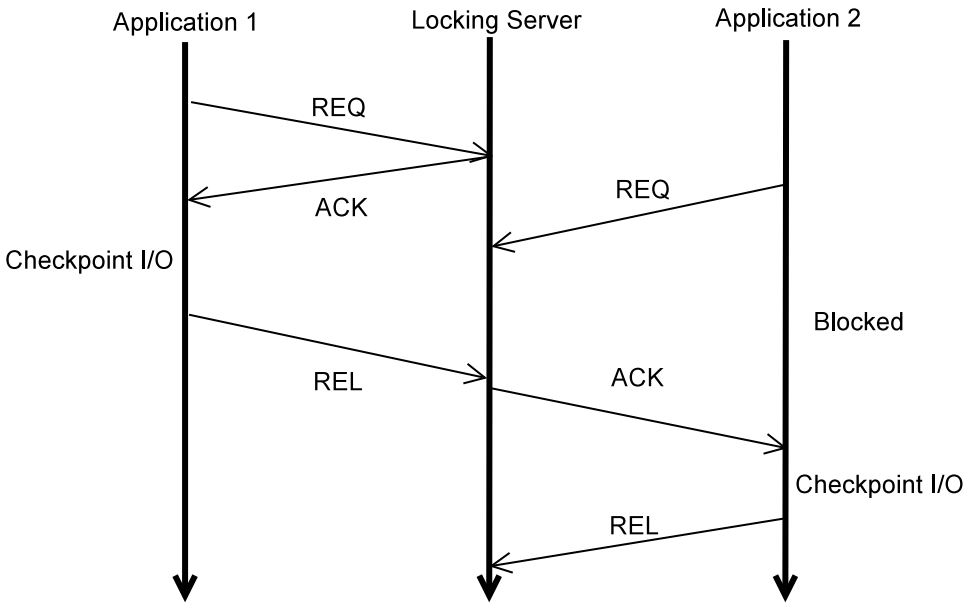

Figure 3.4: Timing diagram of interaction between applications and locking server

Design on Parallel File System

When multiple file servers are involved, each file server is treated as a lockable resource. An application is allowed to perform checkpoint operation only if all the file servers across which the checkpoint file is striped are unlocked. The application locks all the relevant file servers before performing the checkpoint I/O.

We present two locking mechanisms for parallel file systems: locking with and without order preservation. In lockint with order preservation, an application can not perform checkpoint I/O when the file server across which the checkpoint file is striped is not locked and there are applications ahead of it in the locking queue. In figure 3.5, let us assume that A1 requests lock before A2 and A2 requests before A3. Figure 3.5 (a) illustrates locking with order preservation where A3 can not proceed with checkpoint I/O although FS2 and FS3 are unlocked as A2 is ahead of A3 in FS2 queue. This could lead to underutilization of file system resources. So we propose another strategy called locking

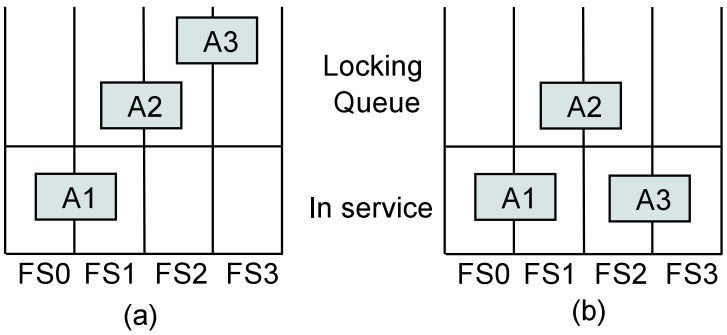

Figure 3.5: Locking with and without order preservation on parallel file system

without order preservation as illustrated in figure 3.5 (*b*). In this case, A3 is allowed to acquire locks on FS2 and FS3 before A2 does. A2 could starve if requests similar to A1 and A3 keep arriving. To prevent starvation, we set an upper limit on blocking duration. Applications that are blocked for this duration are allowed to perform checkpoint I/O.

3.2.2 Implementation

In our implementation, login node serves as the locking server. A server daemon is launched on the login node to listen to applications on a specific port. The port number and IP address of the locking server is provided to the applications during compilation.

Locking Server

The server is implemented using Python. It is a multi-threaded service with one thread for each application for listening to messages and an additional thread to manage the lock. The event queue and service queue are Queue objects from Python's Queue module. Objects from this module are thread safe [8]. Hence, explicit synchronization is not necessary.

It must be noted that applications are serviced in FCFS order based on entry into

```
void acquire_lock ()
{
    if (rank == 0) {
        send_REQ();
        wait_ACK();
    }
    MPI_Barrier(MPI_COMM_WORLD);

}
```

Figure 3.6: Overview of acquire_lock() API

```
void release_lock ()
{
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        send_REL();
    }

}
```

Figure 3.7: Overview of release_lock() API

event queue. It is possible for network latency and server's thread scheduling to alter the order of requests that originate close to each other in time.

Application

Just before a performing a checkpoint operation, the application requests a lock by sending REQ message to the server. This is done by calling the acquire_lock() API. After performing the checkpoint operation, the application calls release_lock() API. An overview of these APIs are provided in figures 3.6 and 3.7, respectively.

3.2.3 APIs

- **void acquire_lock()**: Call this function just before a checkpint I/O operation. This function should be visible to all the tasks during execution.

- **void acquire_lock()**: Call this function just after the checkpint I/O operation. This function should be visible to all the tasks during execution.

For our proof of concept work, we backoff/block at the MPI_COMM_WORLD communicator or application-level granularity. In future, we plan to make backoff and locking APIs available at MPI communicator-level granularity.

The APIs described here and in section 3.1.3 require manual invocation in the application code. Going forward, we plan to embed the backoff and locking logic in MPI-IO and high-level I/O libraries. Commonly used high-level libraries in scientific applications such as ADIOS[25], HDF-5[2], and NetCDF[6] make it easy to identify checkpoint I/O from other forms of I/O in scientific applications. This would allow embedding of I/O coordination mechanisms when applications are linked with these libraries.

Chapter 4

Experimental Evaluation

4.1 Experimental Setup

4.1.1 ARC Cluster

Our experiments were conducted on North Carolina State University's A Root Cluster (ARC) [1]. The cluster comprises of 108 compute nodes, a login node, and a head node. Each node is a dual socket board hosting two 8-core, 64-bit AMD Opetron 6128 processors for a total of 16 cores per node. All nodes have 32GB RAM. The system has a total of 1728 cores and 3456GB of memory. All nodes are equipped with Mellanox ConnectX-2 VPI InfiniBand Adapter Card and 1GE ethernet ports. The head node acts as the NFS server comprising of 12TB HDDs.

The cluster is setup with Rocks Cluster Distribution v5.3. The operating system is CentOS 5.7 running Linux kernel version 2.6.32. The cluster uses Open MPI v1.5.4 implementation of MPI. Jobs are submitted to the system using Torque/Maui using PBS script interface. NFSv3 is used in the system.

4.1.2 Scripts

FCFS-based Backfilling Scheduler

Without administrative access to Torque/Maui, it is difficult to have tight control over the order in which jobs are executed, especially when jobs are submitted by other users too. In order to overcome this difficulty, we developed a script that implements FCFS-based backfilling scheduler described in [26].

In order to run the experiments, a set of N nodes are requested from the system through Torque/Maui. Once the requested resources are available, our script takes control of the N nodes, treating them as a single HPC system with N nodes. All jobs from our experiments are then managed and executed by the script in FCFS order. To improve system utilization, backfilling is allowed. The jobs are submitted to our custom scheduler by listing them in a file along with their start time relative to the start time of the scheduler script and the job's PBS script file.

Our scheduler script collects information about a job from the job's PBS file. This includes the number of nodes requested and the wall time of the script. The scheduler script is also capable of terminating a job if it exceeds the wall clock time. The scheduler script also gathers statistical information about the jobs and the system.

Timeline Visualization

To clearly understand the behavior of applications with and without I/O coordination mechanisms, it was necessary to visualize the different phases on a timeline. To generate such a visualization, we instrument IOR to log the beginning and duration of each phase of the application into a file. Each application generates a few kilobytes of data from these events. This is negligible compared to the size of checkpoint operation, so we believe our

instrumentation neither impacts the performance of the application nor the accuracy of the measurements. Once the events are recorded, we process them offline using Python scripts to generate the timeline graphs of the application.

Job Configuration Generator

In order to test the performance of I/O coordination mechanisms, we had to generate configurations for IOR to represent different workloads. To ensure a fair degree of randomness in choosing the workloads, we developed a script that generates workloads with their parameters treated a random variables.

The parameters of the application that are treated as random variables are compute time, checkpoint I/O size, number of checkpoint operations, number of cores, and application launch time relative the start of execution of custom scheduler. All random variables conform to uniform distribution. We have control over the minimum and maximum value of the random variable. The random variables are time variant, i.e., they do not necessarily take the same values on different executions of the job generation script.

4.2 Results

Identical Workload

In this experiment, we run two instances of the same application and analyze their interaction with and without I/O coordination mechanisms. When the two identical applications are launched simultaneously, their I/O phases coincide resulting in maximum I/O performance degradation.

Although this appears to be an extreme case with maximum overlap, such a scenario

is quite common. From our observation of jobs submitted on ORNL's Jaguar supercomputer, several instances of the same application operating on different data sets are submitted together and the scheduler runs them simultaneously.

The two jobs that run simultaneously are configured with the same parameters. The jobs request 16 MPI tasks each. Each application has a compute phase of 90s followed by 1024MB write in I/O phase per iteration. There is an explicit synchronization between MPI tasks after each I/O. The real world applications may or may not have explicit synchronization, but scientific applications are tightly coupled and make use of MPI commands that implicitly synchronize tasks. There are three iterations in each job.

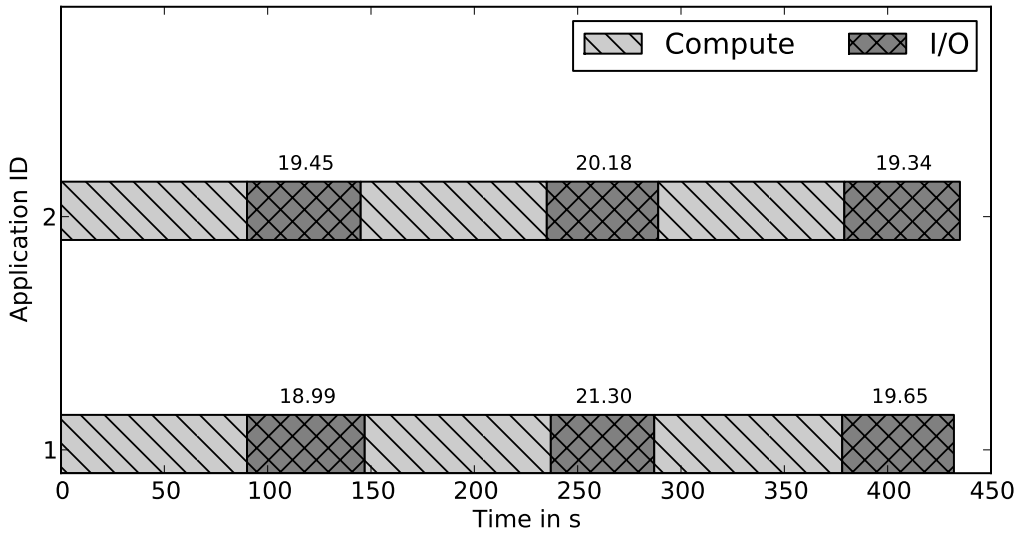

Figure 4.1: Timeline graph of identical workloads without I/O coordination

Figure 4.1 shows the different phases of the applications on a timeline graph without I/O coordination. Both applications 1 and 2 have identical configurations, but operate on different data sets and write to different checkpoint I/O files. The peak throughput

achieved for any checkpoint operation is 21.30MBps.

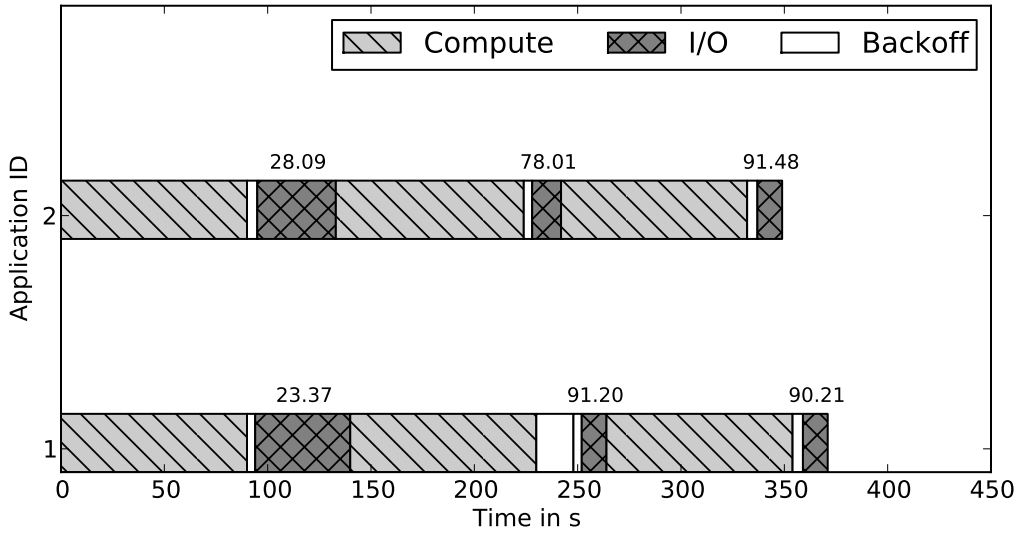

Figure 4.2: Timeline graph of identical workloads with backoff

Figure 4.2 is the timeline graph with backoff. The peak throughput achieved using backoff is 91.48MBps seen in the last iteration of application 2. The first iteration in both the applications have overlapping I/O resulting is low throughput. As discussed earlier, prober is susceptible to multi-layered caches resulting in false negatives. The prober successfully detects the overlap in 2nd iteration and backs off resulting in a higher peak application throughput. There is no overlap in iteration 3, and the overhead of backoff can be observed.

Figure 4.3 is the timeline graph with locking. As a result of locking, there is no overlap between I/O phases of applications. In 1st iteration of application 1, we notice that application waits for a considerable time to acquire the lock. Once the I/O phases no longer overlap, we see an improvement in I/O throughput compared to base case.

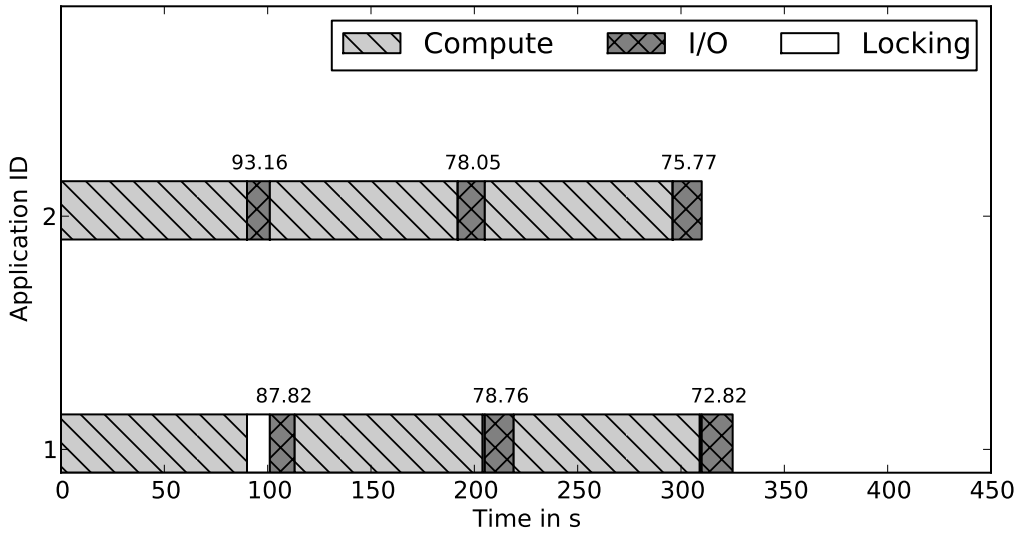

Figure 4.3: Timeline graph of identical workloads with locking

Table 4.1: IOR configuration of applications

Job ID	Number of iterations	I/O size (MB)	Compute time (s)
1	3	1392	140
2	5	1312	132
3	3	1392	140
4	3	2192	220

The total I/O time, including I/O coordination overhead if any, from both the applications is 325.95s, 178.87s, and 92.16s for base, backoff, and locking, respectively. Backoff and locking have 45.12% and 71.72% improvement over the base case, respectively.

Mixed Workload

In this experiment, we gather results from a mix of different workloads. We run 4 applications, 2 of which have identical configurations. The configurations for the applications

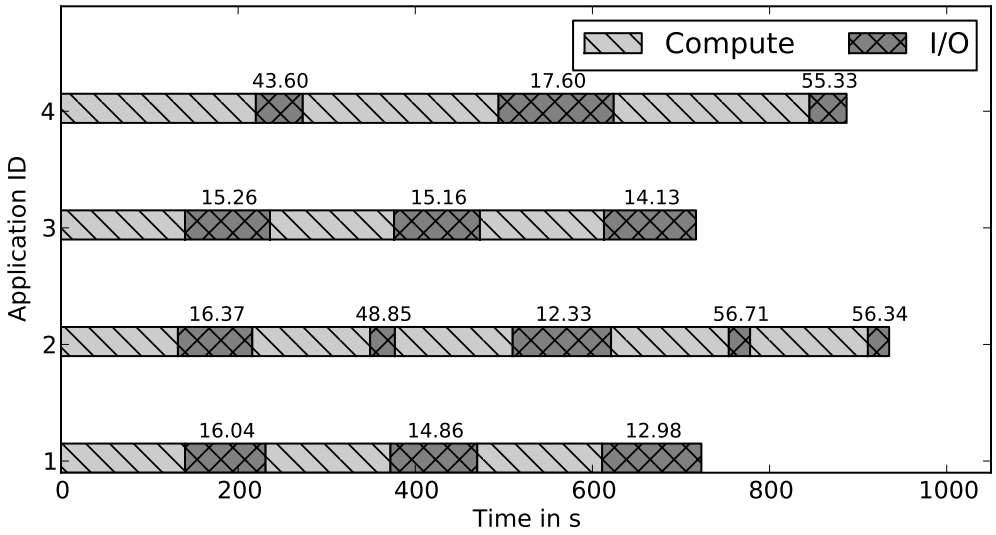

Figure 4.4: Timeline graph of mixed workloads without I/O coordination

were generated randomly using the job generator script described in section 4.1.2. The configuration of the jobs are tabulated in table 4.1

In figure 4.4, we notice the timeline graph of the 4 applications without I/O coordination. The number over the I/O phase indicates the I/O throughput of the checkpoint operation in MBps. From this figure it is clear that when I/O phases coincide, the applications' throughput decreases. The I/O throughput reaches the maximum value when I/O phases do not overlap. This can be observed in 4th and 5th iterations of application 2.

Figure 4.5 shows the timeline graph for backoff. In the figure, each uncolored block indicates backoff. Multiple successive uncolored blocks indicate that application had backed off and probed again. In the figure, we notice a total of 21 probings. Since each probing generates 9MB of dummy data, a total of 189MB of data was generated from all 4 applications. We notice 4th iteration of application 2 and 2nd iteration of application

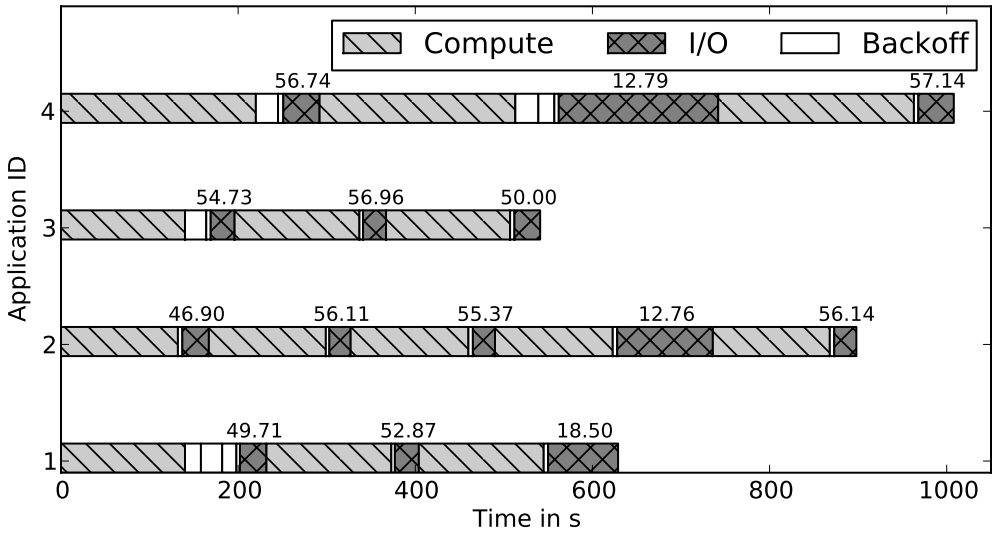

Figure 4.5: Timeline graph of mixed workloads with backoff

4 overlap with 3rd checkpoint operation application 1. This is a false negative resulting from caching effects.

Figure 4.6 shows the timeline graph with locking. The uncolored blocks indicate the period during which the application is blocked. It can be noticed that all the iterations run at maximum throughput.

We also calculate the CPU core-hours saved from I/O coordination schemes with respect to the base case. Backoff resulted in savings of 0.82 CPU core-hours and locking resulted in 2.39 CPU core-hours.

The average I/O (which includes I/O coordination overhead) completion time is 275.50s, 228.98s, and 140.96s for base, backoff, and locking respectively. This shows 16.88% and 48.83% improvement in I/O completion time with backoff and locking over the base case.

We again consider another set of experiments with mixed workloads with 2, 6, and 10

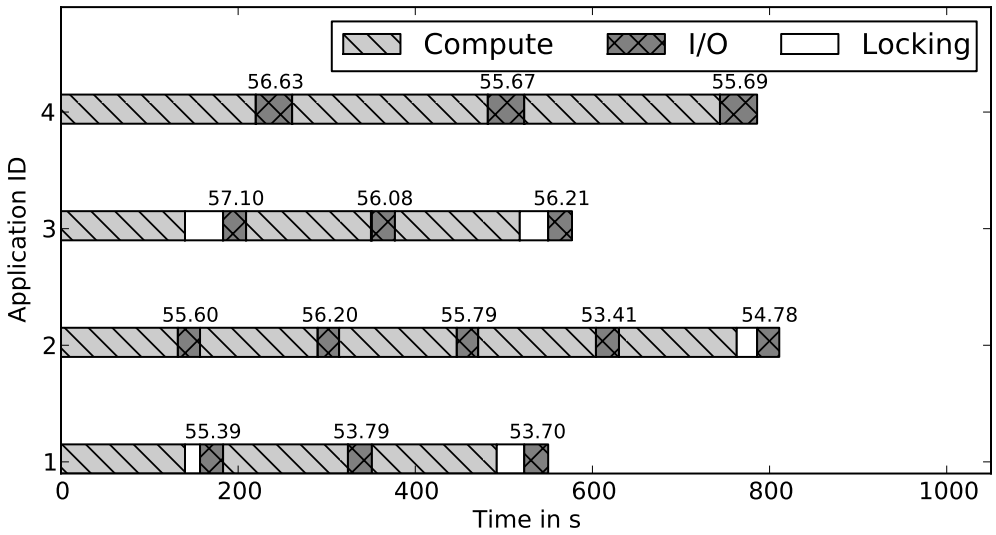

Figure 4.6: Timeline graph of mixed workloads with locking

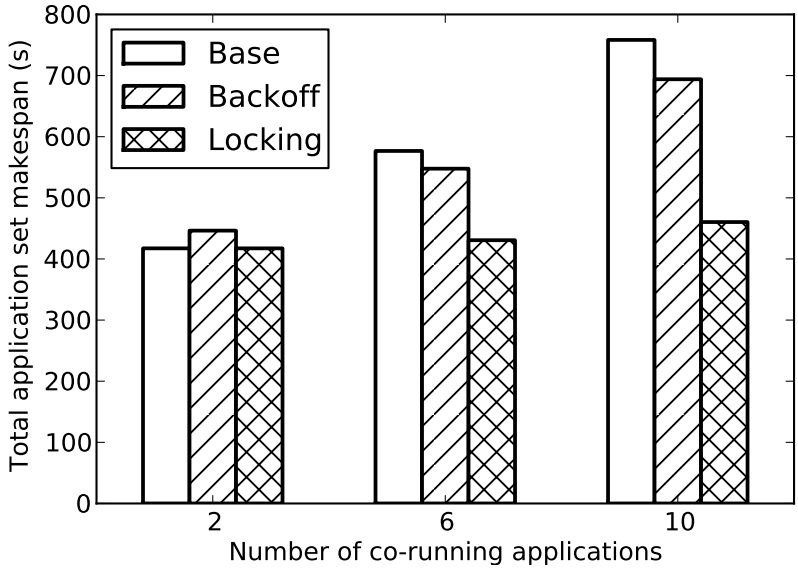

Figure 4.7: Application set makespan time versus number of co-running applications

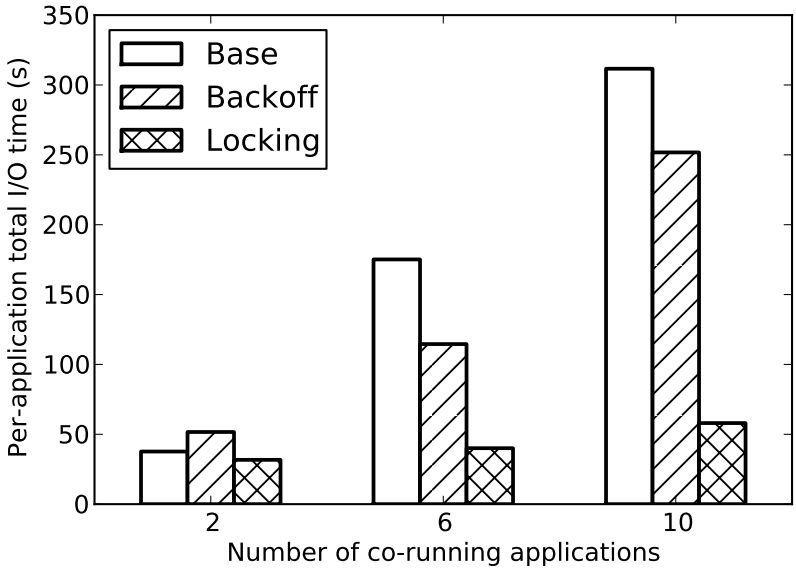

Figure 4.8: Per-application I/O completion time versus number of co-running applications

co-running jobs and analyze their performance. The job configuration generator is used to create jobs with random configurations. The total computation time of the experiment is set to take a uniformly distributed value between 7 and 9 minutes. The number of iterations is uniformly distributed between 4 and 10. Each application writes an average of 2.4GB between the iterations. We conduct the experiment 3 times with a different job configuration in each experiment.

In figure 4.7, we notice an increase in average application set makespan as the number of co-running applications in the system increase. This is a result of increased I/O contention between applications. In figure 4.8, we observe the time spent by an application in performing checkpoint operation spread across all the iterations. With 2 co-running applications, we notice that backoff mechanism takes longer than the base case. This is a consequence of using backoff parameters that are tuned for a busy system with several

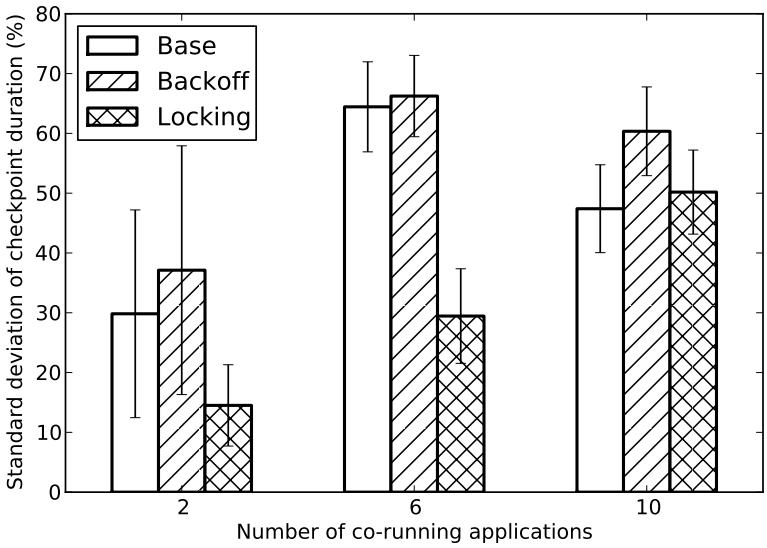

Figure 4.9: Standard deviation of application checkpoint duration versus number of co-running applications

jobs running simultaneously. In all other cases, the performance of locking and backoff is better than the base case. With a loaded system comprising of 10 simultaneous jobs, we notice 19.22% and 81.38% reduction in application I/O time with backoff and locking, respectively.

Figure 4.9 shows the percentage standard deviation of checkpoint operations, obtained by averaging the percentage standard deviation of checkpoint duration across all checkpoint iterations of each application within a group. With 2 co-running jobs, the variance is less compared to 6 and 10 co-running jobs for all three cases as there are fewer overlaps in I/O phases. The variance of 10 co-running jobs is less than 6 co-running jobs as the increased load causes most of the iterations to suffer from overlapped I/O resulting in a uniformly poor performance. Backoff mechanism exhibits the highest variance as the backoff period is randomized.

Overhead and Side Effects of I/O Coordination

In this section, we experimentally determine the overhead of backoff and locking. In backoff mechanism, the overhead is a result of writing the dummy data to probe the file system. This also includes data written to file system for priming and the sleep calls between probings in Min. Prober. The side effect of backoff is writing dummy files to the file system being probed. In locking, the overhead is dependent on messages exchanged between the application and the locking server, and the response time of the server. There is no side effect with locking.

We collect the desired metrics by running backoff and locking instrumented application on a quiet system. The application is configured to run 3 iterations. The time between compute and I/O phase gives the average overhead with I/O coordination.

With backoff, for the configuration used in all our experiments, the average overhead per iteration is 4.03s. Each probing also creates 3 dummy files of 3MB each and a 500K file generated from priming. With locking, the average overhead is 90.67μs.

Chapter 5

Related Work

With I/O being a bottleneck, several techniques have been proposed to improve the performance of I/O. I/O subsystem itself includes several subcomponents such as client, server, networking elements, and storage disks, and papers focus on one or more of these components.

The server-side I/O coordination work presented in [34] is closely related to our work. Their work groups the requests from an application that arrive within a time window and services them together by order of application ID. By grouping the requests from applications together, the disk seeks are reduced and the I/O completion rate increases. The size of the window plays a critical role in deciding which type of I/O pattern benefits from this approach. With small time window, checkpoint operations that output large amount data with multiple synchronous I/O requests will be spread across different time windows. So the file server will interleave the I/O service with requests from other applications that originate in these time windows. Setting the time window high would turn this into a priority queue, favoring applications with low job ID. Checkpoint operations with synchronous I/O requests would still be interleaved between requests. In our work,

we focus on improving checkpoint operation which is significant than other forms of I/O in scientific applications.

Other server-side I/O performance improvement techniques aim to improve performance at file system server. Data sieving [17] is one such technique where non contiguous data segments are accessed as contiguous data segments by including holes that do not belong to the requested I/O. Two-phase I/O [13] is another technique that aims to improve I/O performance in parallel I/O systems by accessing data from the disks into intermediate nodes and then distributing the data to the required nodes based on request mapping. Disk-directed I/O [22] is a combination of several techniques, including data sieving and two-phase I/O. The information from a collective I/O is used to access the disks in a optimal way to improve the performance. Server-directed I/O [31] is an extension of disk-directed I/O that provides high-level interface such as multi-dimensional data sets. These applications aim at improving I/O in general, but our work is focused on checkpoint I/O operations.

The work presented in [24] also acknowledges the problem of variability and not realizing peak I/O performance in HPC systems. Their efforts focus on monitoring the file server performance and balancing the load by distributing the workload to underutilized file servers. They also focus on staggering metadata requests to improve metadata access performance. The load balancing technique described here could complement our I/O coordination mechanisms to further improve the performance by reducing backoff and locking time.

Log-based file systems presented in [11] and [28] improves the performance of checkpoint operations. However, this improvement in write performance comes at the cost of reduced read performance.

Quality of Service (QoS) allows a certain degree of performance guarantee to applica-

tions. The work presented in [16] describes a technique to provide statistical guarantee to applications that have time-critical needs such as online transactions. These techniques are not applicable to scientific applications as they are not time critical. A technique for proportionally allocating file system resources for applications is presented in [19]. This work also probes the shared resource, but uses this information to regulate the I/O requests. [20] presents multi-dimensional parameters, such as latency, space, etc., for providing differentiated service for applications, but it is not suitable for scientific applications. [21] presents a feedback-based approach for providing QoS to applications by regulating the service to applications when there is more than one application performing I/O. [36] introduces a delay-based approach for ensuring QoS in applications. [37] presents a mechanism for guaranteeing minimum QoS for when there is contention for I/O from several applications. Performance improvement and predictability for applications that share file system is discussed in [35], but they still service requests in an interleaved manner in checkpoint operations. All these techniques try to regulate I/O to allow several applications to perform I/O operations simultaneously. Scientific applications that perform checkpoint I/O do not benefit from it as they mostly address performance issues in enterprise workloads.

Chapter 6

Conclusion

A common trend in the development of HPC systems over the years has been the increasing gap in computational and I/O performance. With thousands of cores available in today's high-end HPC systems, it is possible to run large-scale applications. However, with I/O being a bottleneck, application developers are forced to reconsider the scalability of their applications.

The architecture of today's HPC systems is aimed at providing dedicated computing resources to facilitate tightly-coupled scientific applications. However, all applications share the same file system. Without I/O coordination between applications, the application and file system performance takes a hit as a result of increased disk seeks, shared buffers on file servers, and low I/O completion rate.

In this paper, we present two I/O coordination mechanisms to improve the performance of applications. In backoff, the application writes a dummy data to a file to determine the write throughput. If the file system is busy, the application backs off and probes again, otherwise performs a checkpoint operation. In locking, the applications must acquire a lock from a central server before performing checkpoint I/O operation.

The application is blocked until the lock is acquired.

Our results show 19.22% and 81.38% improvement in average I/O completion time with backoff and locking, respectively with 10 simultaneously running applications. The percentage variance of checkpoint duration decreases with locking compared to the base case, but the value for backoff increases due to the randomized backoff time after an unsuccessful probing. The makespan time of the application set is lower than base case in both our mechanisms when there are reasonable number of jobs running on the system. The overhead of these mechanisms is found to be negligible compared to the size of the checkpoint operation.

Future Work

The success of locking is dependent on instrumenting all applications running on the HPC system with calls from locking API. Despite this, it is possible for heavy interactive user commands to interfere with checkpoint phase of applications. In backoff, applications can detect high I/O activity in the file system even when it originates from interactive commands or from applications that have not instrumented probing. A possible research direction is to stack backoff on top of locking where the central server has the additional responsibility of probing the file system before letting an application acquire a lock.

In the backoff mechanism, we use dummy data to probe the file system. This introduces additional load to the system. In future, we plan to use part of applications' checkpoint data to probe to file system.

REFERENCES

[1] ARC cluster. http://moss.csc.ncsu.edu/~mueller/cluster/arc/.

[2] HDF5. http://www.hdfgroup.org/HDF5/.

[3] IOR HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[4] Lustre. http://www.lustre.org/.

[5] MPI profiling layer. http://www.mpi-forum.org/docs/mpi-20-html/node233.htm.

[6] NetCDF. http://www.unidata.ucar.edu/software/netcdf/.

[7] Open MPI profiling layer. http://www.open-mpi.org/faq/?category=perftools.

[8] Queue a synchronized queue class. http://docs.python.org/library/queue.html.

[9] TOP500 supercomputing sites. http://www.top500.org/.

[10] IEEE Standards Association et al. *IEEE 802.3 LAN/MAN CSMA/CD Access Method*. 2008.

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21, 2009.

[12] A.S. Bland, R.A. Kendall, D.B. Kothe, J.H. Rogers, and G.M. Shipman. Jaguar: The worlds most powerful computer. *Memory (TB)*, 300(62):362, 2009.

[13] R. Bordawekar, J.M. del Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, page 452461, 1993.

[14] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: NFS version 3 protocol specification, june 1995. *See also RFC1094 [Sun89]. Status: Informational*.

[15] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*, page 2828, 2000.

[16] D.D. Chambliss, G.A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T.P. Lee. Performance virtualization for large-scale storage systems. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, page 109118, 2003.

[17] A. Choudhary, R. Bordawekar, M. Harry, and R. Krishnaiyer. PASSION: parallel and scalable software for input-output. 1994.

[18] Y. Etsion and D. Tsafrir. A short survey of commercial cluster batch schedulers. *School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, Reporte tcnico*, 13:2005, 2005.

[19] A. Gulati, I. Ahmad, and C.A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, page 8598, 2009.

[20] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, page 1424, 2004.

[21] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage (TOS)*, 1(4):457480, 2005.

[22] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(1):4174, 1997.

[23] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 40, 2009.

[24] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the IO performance of petascale storage systems. pages 1–12. IEEE, November 2010.

[25] J.F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, page 1524, 2008.

[26] A.W. Mu'alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529543, 2001.

[27] I. Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[28] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D.K. Uppugandla. Fast log-based concurrent writing of checkpoints. In *Petascale Data Storage Workshop, 2008. PDSW'08. 3rd*, page 14, 2008.

[29] R. Sankaran, E.R. Hawkes, J.H. Chen, T. Lu, and C.K. Law. Direct numerical simulations of turbulent lean premixed combustion. In *Journal of Physics: conference series*, volume 46, page 38, 2006.

[30] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, page 231244, 2002.

[31] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in panda. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, page 5757, 1995.

[32] H. Shan and J. Shalf. Using ior to analyze the i/o performance for hpc platforms. 2007.

[33] G. Shipman, D. Dillow, S. Oral, and F. Wang. The spider center wide file system: From concept to reality. In *Proceedings, Cray User Group (CUG) Conference, Atlanta, GA*, 2009.

[34] H. Song, Y. Yin, X.H. Sun, R. Thakur, and S. Lang. Server-Side I/O coordination for parallel file systems. 2011.

[35] M. Wachs, M. Abd-El-Malek, E. Thereska, and G.R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, page 55, 2007.

[36] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, page 4760, 2007.

[37] T.M. Wong, R.A. Golding, C. Lin, and R.A. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, page 125134, 2006.