MPIOR: A Framework to Analyze File System Performance of MPI Applications

Shankha Banerjee

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

Srinidhi Varadarajan
Calvin J. Ribbens
Eli Tilevich

Feb 17, 2012
Blacksburg, Virginia

Keywords: MPI, trace, replay, I/O

MPIOR: A Framework to Analyze File System Performance of MPI Applications

Shankha Banerjee

MPI I/O replay (MPIOR) is an I/O performance modeling and prediction tool used to trace and replay a parallel application to determine application performance under a new I/O sub system. The trace collector deduces synchronization inter-dependencies between nodes and I/O demands placed by each node on the storage subsystem. It uses a novel runtime graph traversal technique to filter and log only those MPI calls that affect I/O, thus substantially reducing both the number of runs and the size of the trace file. Unlike other such tools, MPIOR collects a valid trace in a single run and it does not rely on node sampling or I/O sampling. MPIOR's post processing engine analyzes the trace files and sets up the re-player. Due to minimal overhead for trace collection, MPIOR can be used during production runs rather than just as a debugging tool. The re-player mimics the behavior of the application across a variety of storage systems by mapping multiple processes to multiple threads running on a single node. We show average replay error for parallel applications is below 30%.

Acknowledgements

I would like to thank my advisor Dr. Srinidhi Varadarajan who has been very kind and helpful. His constant support, guidance, and patience helped me complete the thesis on time. He has been extremely patient in working with me all through my thesis, despite my mistakes. I am thankful that he provided me the opportunity to learn and experiment. I am fortunate to have the pleasure of working for him on projects that spanned several areas of computer systems and I am thankful to him for expanding my understanding of systems.

Dr. Varadarajan is an extremely polite and a humble person and that is one quality which I would like to carry with me. I am sincerely grateful to him for his guidance and his continued faith in me.

I am particularly thankful to Dr. Ribbens. He has gone out of his way to help me graduate me on time and without his constant feedback and support I would have never been able to complete my dissertation work on time.

Graduate study involves spending a lot of time in the lab. It is during those times I had the opportunity to interact with Dr. Tilevich. Dr. Tilevich has been more of a friend rather than a professor. Usually professors are difficult to get hold of due to their extremely busy and hectic work schedule. Dr. Tilevich has been an exception in this regard. His constant support and advice helped me survive in graduate school.

I thank my colleagues Vedvyas Duggirala, Hari K. Pyla, and Bharat Ramesh for their constructive feedback throughout the duration of my graduate studies.

In particular, I would like to thank Vedvyas Duggirala. Vyas has been an excellent mentor to me and I am thankful for his support and guidance during every stage of my Master's thesis. His humbleness and his patience helped me grow personally and professionally.

My respects to Almighty!

Contents

List of Figures

List of Tables

1. INTRODUCTION

1.1 Motivation

The top 10 Supercomputers in the Top 500 November 2011 list are Petascale and the road map and current trends indicate the possibility of Exascale systems in the near future. This increasing demand in computing resources stems from the growing ecology of complex scientific applications (e.g., earthquake modeling, weather simulation, nuclear stock pile simulation) and financial applications.

While computing power impacts the performance of such applications, most of these applications are I/O intensive or a combination thereof. Consequently, the performance of the file system plays a crucial role in the overall runtime of the application.

Given the plethora of high-performance file systems, it is non-trivial (even for domain experts) to identify a suitable file system that is capable of delivering good performance. To further exacerbate this task, the choice of file-systems varies with the application set. For instance, two seemingly related applications might have distinctively different file-systems that meet their performance requirements.

Evaluating an application across all possible file systems is practically infeasible. Furthermore, porting applications and their libraries to a particular file system requires a significant software development effort. Even in the presence of such an effort, issues such organizational software compliance procedures and system configuration issues preclude such an exploration technique.

Consequently, to choose a suitable file system and predict the performance of the application, software architects typically employ file system benchmarks. Unfortunately, identifying a benchmark that is representative of the application's I/0 characteristics are impractical and pose several challenges. First, there is no clear one-one or one-many mapping between existing benchmarks and the applications. Second, such selection requires sound understanding of the software system stack, often known only to a domain expert. Even in the presence of a suitable benchmark, understanding the applications performance requires a significant amount of time and effort. For instance, configuring a benchmark (typically 10-20 parameters on average), collecting statistics and making meaningful interpretation of the collected data poses additional challenges.

Trace based analysis is an alternative way to determine application behavior on a file system. In contrast to benchmarking, a trace based approach captures the application's I/O characteristics (e.g., read, write, sync, and etc. operations) during the application's runtime. Then these traces can be *replayed* on a file system of interest thus, in effect, faithfully preserving the true I/O behavior of the application. Such a technique, while relatively straight forward poses several challenges. First, we need to efficiently and accurately capture and preserve (during replay) the I/O characteristics. This includes measuring the size of each IOP (I/O operation), preserve program order of IOPs and finally preserve the invariant (computation time). Second, for large scale scientific

parallel applications the inter-node communication (e.g., send, recv) and not synchronization (e.g., barrier) dependencies across nodes in a cluster must be preserved.

Unfortunately, the existing state-of-the art techniques are either incomplete and not suitable to evaluate parallel applications, or prohibitively expensive to use in practice.

In this thesis, we propose a novel software based system called MPIOR (MPI IO replay) that is capable of efficiently and accurately determining an optimal file system. The use of MPIOR does not require any modifications to the application, nor require any re-linking or recompilation of the application. MPIOR supports both sequential and parallel applications (written in the widely prevalent MPI programming model). MPIOR is implemented as a pre-loadable shared library that overloads all I/O calls and MPI routines and simplifies collection of traces. MPIOR implements two techniques for facilitating replay on file systems – thread based replay for shared memory systems and MPI based replay for distributed systems.

MPIOR detects mapping between synchronization calls between different nodes through piggyback messages. These mappings help determine the time spent in I/O and synchronization operations. I/O and synchronization operations are collected in a trace file. The MPIOR re-player then processes the trace files, and executes the events in order to determine the file system performance. MPIOR significantly outperforms the existing state of the art and addresses several important shortcomings of trace based systems in existence today. A preliminary performance analysis of MPIOR indicates that it incurs replay error rate of no more than 30%.

1.2 Related Work

Trace based tools have found usage in a variety of areas such as analysis of locking behaviors [4] , verification of MPI [5] programs, studying garbage collection algorithms for object oriented programs [6], etc. Their primary use has been in studying storage systems through different mechanisms [7-11].

File system benchmarking has traditionally been used to predict file system performance under varying workloads and I/O patterns. Benchmarks help users to identify good choices for file systems or file system parameters and predict the performance of applications on file systems. Benchmarks are assumed to be easily portable and configurable, i.e., easier than the actual application, which may require propriety libraries, database systems, etc. File system benchmarks [1, 7-18] have traditionally been designed with I/O operations in mind and involve creation of files, reading from files and dumping data to files.

Applications have evolved over the years in terms of I/O requirements. I/O runtime can be a significant component of the overall runtime. Calibration of I/O performance can be used to make design decisions, procurement strategies, etc. Synthetic benchmarks are tuned to work for a certain class of applications. For a new application, the onus is therefore on the end-user to modify existing benchmark(s) to closely mimic the

application's needs or to look for or create a different benchmark altogether. Trace based replay is an attractive alternative to synthetic benchmarks because it fulfills the requirement to generate the correct workload, which represents the I/O characteristics of the application.

The I/O operations and characteristics that are of interest include creation and deletion of files, operations on file handles (e.g., open, close, fsync), reading from a file, writing to a file, and other seek operations. Most importantly however, a good benchmark or trace based scheme will faithfully capture the I/O issue rate of the application, which in turn depends on the time spent in each I/O operation and the time spent between I/O operations. Unless the I/O issue rate of the synthetic benchmark matches the I/O issue rate of the application when run on the same system, I/O performance of the application cannot be successfully predicted on other systems. The I/O operations should be issued at the desired frequency [7].

Trace based replay collection has several advantages over synthetic benchmarks: 1) there is no need to set up the application or benchmark; 2) the user does not need to understand the application or benchmark; 3) the user has the flexibility to capture only certain calls of interest; 4) captures the true workload of the application, including computation, I/O and communication; 5) synthetic benchmarks may not exercise some I/O operations and sequences.

There are three recent research projects that have proposed a trace–replay solution to file system benchmarking: LANL-Trace [3], Tracefs [1], and Parallel Trace [2].

LANL-Trace [3] gives users a way to define a set of I/O operations of interest and to collect tracing information on those operations for any application (distributed or shared). The user has the flexibility to decide which API's need to be stored in the trace files. A re-player is not provided with LANL-Trace.

Tracefs [1] is an attempt to provide an infrastructure for trace collection. Users are provided with the flexibility to add plugins, or "filters" as they are referred to in the paper. The plugins can have multiple functionalities such as compression of repeated information (e.g, user arguments), encryption of the trace data, flexibility to collect trace information to a file or socket, and collection of overall aggregate information. Tracefs is installed as a kernel module and can trap VFS level calls successfully.

Replayfs [18] is a re-player provided along with Tracefs. Replayfs is an in-kernel replayer, which attempts to overcome the drawbacks of user-land replayers. The disadvantages of a user-land replayer include overhead of data being copied from user buffers to kernel buffers, delaying I/O requests (e.g., due to service of a higher priority task), and eviction of pages used by the re-player by the OS. Replayfs employs several novel techniques to minimize the overhead of the re-player. It uses an in-memory table mechanism to read and store the arguments of the commands and share them across the commands. It is part of the kernel and therefore has the flexibility to pre-fetch the data as needed by the re-player. One of the advantages of Replayfs along with Tracefs is that it

tracks and replays in-memory operations (as they are part of the VFS operations) and thus improves overall replay accuracy.

The effectiveness of a replay tool depends on its ability to preserve the I/O issue rate of the I/O operations at the point of replay. In essence, the rate of issue of I/O operations (I/O issue rate) on one file-system should match with the rate if the original application was run on the same file system. For applications with high rate of I/O operations the issue rate is influenced by the performance of the file system. For applications with low rate of I/O operations, the issue rate is influenced by the amount of time the application spends within consecutive I/O operations. Applications with low rate of I/O operations are therefore not affected by the change of file system.

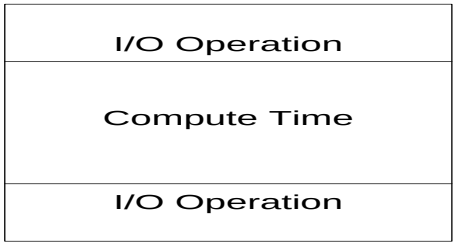

Figure 1.1: Non-parallel application (Application time = Compute Time + Time spend in I/O)

Most real world applications are somewhere in the middle, i.e., the issue rate is determined by some combination of compute time and I/O time. A re-player for those applications does need to preserve the time spent in between I/O operations. The time spent in operations that contribute to the overall runtime, but are not part of the time spent in I/O can be referred to as compute time. If you assume that system parameters other than the I/O subsystem (e.g., interconnect properties, memory hierarchy, etc.) do not change as we move to a new system, then the compute time can be assumed to remain constant. Existing re-players have relied on this assumption to correctly replay the I/O operations and replace the actual “compute” time with sleep, busy wait, etc.

This holds good for applications that execute on a single node. However, applications need to communicate among themselves. Depending whether the communication is blocking or non-blocking they may need to wait for each other.

Hence, assuming that overall runtime can simply be broken down into I/O time and compute time holds true only for sequential applications. The assumption is not true for distributed-memory parallel applications. The processes of such applications communicate among themselves and this communication induces synchronization and dependencies.

Time spent by a process in communication or synchronization operations (CSYOP) is dependent on the time at which other processes participating in CYSOP enter CSYOP.
The scenario is explained in Figure 1.2.

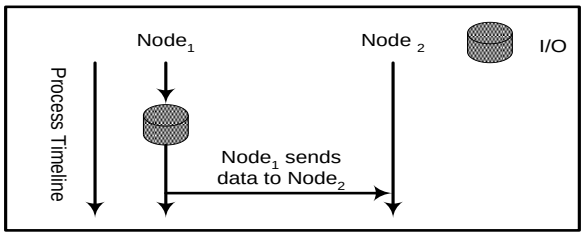

Figure 1.2: Two processes running on different nodes. Node $_2$ is waiting to receive data from $Node_1$ thus leading to dependency on $Node_1$.

Depending upon the communication medium and the communication semantics and state of the application $Node_2$ may have to wait for the data transfer to finish from $Node_1$. The time at which the process on $Node_1$ (P_1) hits the synchronization point is dependent on the time at which P_1 comes out of the I/O operation. Thus the I/O operation on $Node_1$ introduces a *direct dependency* of $Node_2$ on $Node_1$. The time spent by both nodes at the synchronization point is dependent on the time taken to finish the I/O operation.

The interdependency may not be explicit in many cases. In Figure 1.3 we show an indirect dependency between $Node_1$ and $Node_3$. The I/O operation on $Node_1$ influences the times spend by $Node_3$ in the communication operation with $Node_2$.

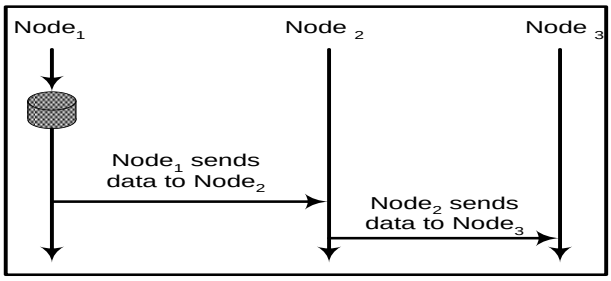

Figure 1.3: Indirect dependency between nodes.

These dependencies introduce another component other than compute time and I/O time. This component is known as the wait time [2]. For parallel applications the wait time may vary with the time spend in I/O. It is therefore not only necessary to trap I/O operations, but synchronization operations as well. This relationship between synchronization operations and I/O operations was first identified in parallel trace [2]. Experiments conducted by Mesnier et al. of parallel trace show that replay of the trace files encountered error rates ranging from 27% to 200%, if wait times of the application are ignored.

Parallel trace used a “casualty engine” to figure out node interdependencies correctly and store them in a format that can be used by the re-player. The “casualty engine” has the capability to run each node in two modes: “throttled mode” and “un-throttled mode”. The application has to be run as many times as the number of nodes to determine all the inter-dependencies. At every run a particular node is selected as the target node. The target

node for that particular run is run in “throttled mode”. Other nodes except that target node are run in “un-throttled” mode. In the throttled-mode the process is throttled at the point of an I/O operation. The “casualty engine” then checks for processes on other nodes (“un-throttled”) that do not finish within a reasonable amount of time. The “un-throttled” nodes are assumed to have a dependency on the throttled mode. A synchronization point is dumped on the trace file just after the I/O operation on the “throttled” node. On the “un-throttled” nodes a synchronization point is dumped on the trace file just before the next I/O operation.

The parallel trace paper puts forward a novel idea into of keeping track of synchronization points (note: These are “points” and “not operations”, which is altogether different from points.) along with I/O operations. The methodology to collect all the synchronization points involves at worst case throttling all the I/O operations on one node (assuming there is a synchronization operation after each I/O) and running the application as many times as the number of nodes. The authors do realize the scalability issue and provide a solution where nodes and I/O operations can be selected through a sampling mechanism. The sampling mechanism has been left as future work. For the paper the nodes and I/O operations are selected at specific intervals.

1.3 Problem Statement & Approach

The goal of this research is to enable users of MPI based I/O intensive parallel applications on a regular day to day basis; predict the performance of their applications on different file systems, without requiring huge investments in time and resources.

As HPC systems continue to climb the ladder in terms of number of floating point operations per second, the number of cores available on these large scale clusters keeps on increasing. Applications currently are lagging behind in their capabilities to use the computing power available to them. As applications start incorporating more parallelism in their inherent design and structure, the interaction among other sub-systems (I/O, memory and communication) will become critical and there will be a need to understand these interactions at such a large scale. Benchmarks designed for a non-parallel environment do not have mechanisms to study intra node interactions and thus cannot be used to predict performance of applications running in a parallel environment.

Parallel applications are currently being run on clusters built of hundreds and thousands of nodes. At this scale it is not possible to employ the mechanism suggested in [2] to collect the trace for replay. Node sampling and I/O sampling techniques unless proved to work for a certain class of applications, cannot be used to cut down the number of iterations required to collect valid trace results.

In this work we try to address the problem of capturing the relationship between I/O and communication synchronization operations online in a trace file. The trace files collected from various nodes are run through a post-tracer to set up replay files. The re-player then faithfully reproduces the I/O issue rate and synchronization operations to provide a realistic estimate of the application run time.

Faithful reproduction of I/O issue rate for a parallel application requires correct identification of synchronization points. Synchronization operations lead one node to wait on another and delay any pending I/O operations. A Delay in issue of I/O operations further leads to delay in subsequent synchronization operations. This problem will be demonstrated with the help of the following example.

The example in figure 1.4 illustrates I/O operation I_1 influencing the time at which the process on $Node_1$ (P_1) reaches B_1. This influences the time at which the process on $Node_2$ (P_2) reaches B_2. The processes at $Node_3$ (P_3) and $Node_2$ are engaged in barrier B_2. The I/O operation I_2 is delayed till the process P_3 comes out of B_2. I/O operation I_2 has a similar effect on B_3. The time at which P_1 finishes I_3 depends on the time at which P_1 comes out of I_3. I/O operation I_3 in turn determines when all the three nodes come out of B_4.

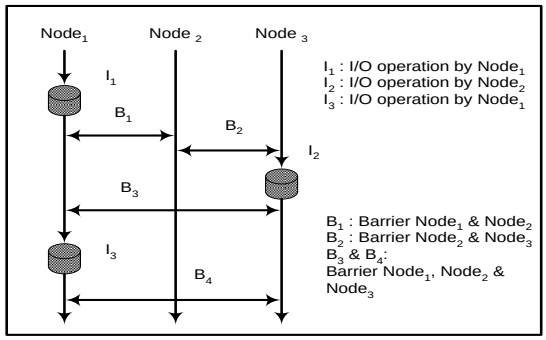

Figure 1.4: Demonstrates a scenario where I/O operations are influenced by synchronization operations and vice versa.

The example (Figure 1.4) demonstrates how the I/O operations on one node affect the issue rate of I/O operations on another node. The effect propagates through synchronization calls and it is essential to capture the effect while collecting a trace of the application.

The synchronization messages need to be logged to make sure the effect of I/O on one node is reflected as it should be on I/O of other nodes at the time of replay. Logging of messages will introduce additional overhead on the application runtime. We present an online trace pruning technique which removes extraneous synchronization messages (which won't affect I/O).

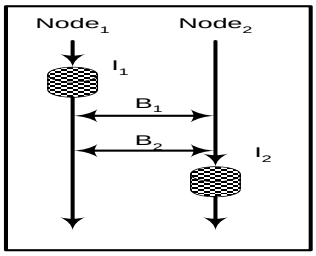

Figure 1.5: Illustrates the potential for trace pruning.

The I/O operation on $Node_1$ influences the amount of time $Node_2$ spends in B_1.The amount of time both the nodes spend in B_2 is not affected by I_1. The time spent by $Node_2$ in barrier B_2 has therefore no bearing on when I_2 is issued. The barrier B_2 therefore need not be logged.

The relationship between I/O and synchronization was captured in parallel trace [2]. The method used as explained above involves skipping I/O operations and nodes to keep the overhead down. One of the requirements of our work is to capture the relationship in a way where statistical assumptions about the number of nodes and number of I/O operations need not be taken into consideration.

Some of the applications may be running replay for weeks. Applications may not be I/O intensive and may involve a significant amount of compute time. We provide an option to the user to scale down the replay time without affecting the impact on the I/O storage subsystem. This option is currently on a trial basis as there are no current mechanisms to figure out the exact relationship between I/O and compute time and at what stage any decrease in compute time starts affecting the performance of I/O storage system. This option can be used to reduce the over-all time of replay.

2. DESIGN OVERVIEW

MPIOR (MPI I/O replay) discovers node dependencies in real time and makes a decision to log communication calls through an online graph pruning technique. MPIOR consists of three parts: a) Tracing engine b) Post processor c) Re-player. In this section we describe the design of each component.

2.1 Tracing Engine

The MPIOR tracing engine is a designed as a shared library. The application can use the library through the LD_PRELOAD mechanism. The tracing engine has the following objectives:

1) Capture all the relevant I/O and synchronization calls.
2) Determine dependency between I/O and communication calls and decide which communication calls need to be logged.
3) Categorize communication operations into groups such as synchronization operations, blocking communication calls, and non-blocking communication.

The tracing engine intercepts all the relevant MPI communication and I/O calls from the original application. The application doesn't need any modification from the user. Trap-library can be used along with any binary without requiring any re-linking.

The tracing engine on every communication call exchanges information among all the nodes involved in the communication call. The exchange messages are piggy backed on the original communication call. Every communication imposes certain kinds of dependencies among the nodes. The tracing engine takes care to not change these dependencies. Changes of these dependencies may lead to incorrect amount of time spent by nodes in these communication calls.

Communication operations and their dependencies may be categorized as follows:

1) *All to all dependencies* cause all processes participating in the operation to block on each other. Processes do not come out of the call until all of them have entered the call, e.g., MPI collective calls.
2) *One to all dependency* causes all processes participating in the operation to block on one process. Processes participating in the communication block on a single process, e.g., MPI_Bcast.
3) *One to one dependency* causes one node to wait on another node, e.g., MPI_Recv, MPI_Irecv+MPI_Wait, where the receiver waits for the sender to start the send operation. Note that MPI_Send and MPI_Isend do not lead to any dependency between the sender and receiver in the case of small messages.

2.1.1 Auxiliary Info Messages

Auxiliary info messages are sent along with regular messages. These messages have

information to decide if the participating nodes have decided to log the synchronization message or not. We could have adopted various techniques to send the auxiliary information such as appending to the original message, packing the data types, etc. We choose to send the auxiliary message as a separate message which piggybacks on the original message. The auxiliary messages are sent with a different tag to ensure that they can be differentiated from the original messages.

2.1.2 Piggyback Messages

Relationship (between I/O and communication) tracking algorithms use messages exchanged between nodes to make decisions regarding logging of messages. We use MPI_Allgather to exchange piggyback messages in case of all to all dependencies and one to all dependencies. For one to one dependencies we use MPI non blocking communications to exchange piggyback messages.

Piggyback messages for point to point communication have remained an open research problem. Various efforts have been made in these directions which have their own merits and demerits [5, 19, 20]. For this work we decided to use a separate messages technique to send piggyback information. Vo et al. [5] correctly observe that the presence of MPI "Wildcards" makes it difficult to determine the parameters of the piggyback messages. They conclude that there are too many issues related to using piggyback messages in sending data and propose a new technique based on data types. In this technique the actual piggyback data and original message are merged together in a new structure. The new structure is sent to the receiver in place of the original message. Similarly on the receive side a new structure composed of the original message and piggyback message is received in place of the original message.

In our work we propose a technique based on piggyback messages that handles issues related to MPI wildcards. Our algorithm is based on the fact that the number of piggyback messages sent by a particular node is equal to the number of send operations (blocking and non–blocking) executed on the node. A similar argument holds for the receiver as well. For each send operation an additional auxiliary info message is inserted into the runtime. (Note: A blocking send operation is broken down to a non–blocking send and wait. A blocking receive operation is broken down to a non–blocking receive and wait.) Piggyback messages use a different tag (ctag) than the original messages. The ctag helps us differentiate between application messages and piggyback messages.

The sender posts piggyback messages with tag specified as ctag and the same receiver as specified by the original message. The receiver on the other hand posts a non-blocking receive for a piggyback message with tag specified as ctag and MPI_ANY_SOURCE. The receiver stores this message in a wait queue. On successful completion of the wait call for the original message, the receiver determines the source and tag of the original message. The receiver goes through the wait queue and waits for messages one by one. If the piggyback message received is not what the receiver was looking for the message is pushed to a receive queue (recv_queue) which contains the set of received piggyback

messages. The receiver checks in the recv_queue for the appropriate message, before going through the wait queue.

2.1.3 Data Structures

In addition to the wait and receive queues, the trace library maintains two other important pieces of data used to record the dependencies between nodes:

1) The type of operation most recently seen by the trace engine, which can be either *communication* or *I/O*. I/O operations include open, fopen, read, fread, write, fwrite, sync, fsync, etc. Communication operations include message passing and synchronization operations.

2) Processes (involved in the communication operation) decide to log a communication operation only if the all the processes haven't performed a communication operation with each other after an I/O operation. This information about each process is stored in an array, called the node array. The size of the array is equal to the size of number of processes in the universe. The node array for each process is indexed through the rank of the process in the universe.

2.1.4 Localized Decision Making Process

One of the stated requirements of MPIOR is to reduce the number of communication messages that need to be logged. Communication messages are necessary as they influence the time at which I/O is issued on a particular node. They form a relationship between I/O operations on different nodes. Thus the issue rate of I/O operations on one node influences the issue rate of I/O operations on other nodes. The I/O characteristics of the application therefore cannot be studied on a per node basis. There is a need to look at the I/O operations across all the nodes to determine the I/O storage performance, and to do so in such a way that the synchronization (i.e., "happens-before" relationships) of the original application are preserved on re-play.

In the context of applications running on a single node there is a straightforward happens before relationship between various operations on a single node. The start, end and execution time of all the operations can be measured with respect to a single global clock. For applications running on single node, this happens before relationship, if captured correctly, can be used by the re-player to faithfully reproduce the sequence of events.

To figure out happens-before ordering on a cluster, a vector clock approach as mentioned in [19] could be used. This approach leads to runtime overhead. At every synchronization event the vector Lamport clock has to be exchanged between all the nodes participating in the event. The size of the message would be proportional to the number of nodes in the universe. An application with many small sends and receives would experience orders of magnitude of overhead.

In this work we therefore decided not to maintain a Lamport clock, but to make the decision to log the communication message localized. Each node individually figures out if it needs to log the call and conveys this information to nodes participating in the communication event. If any of the nodes involved in the communication message decides to log the event, all the nodes log the event. The advantage of this method is that the nodes need not maintain a Lamport clock.

2.1.5 Receiver Will Log Sender's Messages

The decision to log synchronization messages that involve two nodes (send, recv) at the receiver has been taken to avoid deadlocks. The sender can only log the message if it has received auxiliary information from the receiver. This would lead to introducing a dependency of send on recv. This is incorrect and would violate the semantics of the message passing application. Our experiments showed that MPI_Send messages of size in the range of a few bytes are not blocking on the MPI_Recv calls.

2.1.6 Algorithm

The following algorithm describes what we do to exchange auxiliary information.

Algorithm: Trace algorithm.

```
Variables:
        sz_comm_world = size of MPI_COMM_WORLD
        node_array [sz_comm_world]: An array which holds information if a
                        synchronization operation has been encountered after I/O operation.
        ctag: A unique tag used to exchange auxiliary info between nodes.
         wait_queue: A queue which hosts the auxiliary info messages posted.
        recv_queue [sz_comm_world]: An array of queues which holds auxiliary info
                                        messages.
        decision: A variable that decides if the message is to be logged

if (Operation is a I/O operation) {
        for (i = 0; i < sz_comm_world; i++)
                node_array[i] = 0
}
else if (Operation is MPI collective operation) {
         for (i = 0; i < sz_comm_world; i++)
        {
                if (!node_array[i]) {
                        decision = true
                        break
                }
        }
        Get decision information from all nodes.
```

```
        if (!decision)
        {
                if (any of the nodes decided to log)
                        decision = true
        }
        for (i = 0; i < sz_comm_world; i++)
                node_array[i] = 1

}
else if (Operation is MPI_Isend) {
        dst: MPI_Isend destination
        tag: MPI_Isend tag
        if (!node_array[dst])
                decision = true
        Send the decision through MPI_Isend with the same source and ctag.
        Call this message as auxiliary info.
        Associate the original request handle and auxiliary info request handle.
}
else if (Operation is MPI_Irecv) {
        src: MPI_Irecv src
        Post a MPI_Irecv with src as MPI_ANY_SRC and tag as ctag
        Call this message as auxiliary info.
        Associate the original request handle and auxiliary info request handle.
}
else if (Operation is MPI_Wait) {
        if (wait is for MPI_Isend)
        {
                Retrieve the auxiliary info request handle.
                MPI_Wait on the handle.
        }
        else if (wait is for MPI_Irecv)
         {
                Retrieve the original source (origsrc) and tag (origtag) from
                wait message.
                Iterate through recv_queue[origsrc]
                Check for the auxiliary info message with the tag.
                if (auxiliary info message not found) {
                        Iterate through wait_list and retrieve the message.
                        if (source == origsrc and tag == origtag) {
                                Auxiliary message has been found.
                        }
                        else
                        {
                                Push the auxiliary message into recv_queue[origsrc]
                        }
                }
```

```
                while (auxiliary info message not found) {
                        Post a MPI_Recv with src as MPI_ANY_SRC and tag as ctag
                        Call this message as auxiliary info.
                        Identify the source and tag of the message.
                        If (source == origsrc and tag == origtag) {
                                Auxiliary message has been found
                                break
                        }
                        Else
                        {
                                Push the auxiliary message into recv_queue[origsrc]
                        }
                }

                if (auxiliary info message found with tag) {
                        Retrieve the message.
                        if (sender decides to log)
                                decision = true
                        else
                                if (!node_array[origsrc])
                                        decision = true
                        node_array[origsrc] = 1
                }
        }
}
```

Operation	Start	End	Arguments
MPI_Bcast	1000	1023	1417268 (comm)
fprintf	2000	2012	21 (bytes) 0xf213 (file pointer)
MPI_Barrier	3000	3010	2345678 (comm)

Table 2.1: Format of the trace file.

2.2 Post Processing

The primary design objective of the post processing script is to construct the replay files (files read by the re-player and used for replay) in a way so as to induce minimum overhead from the re-player. We define re-play error on a per I/O event basis, as the difference between the time at which a particular event is triggered in the replay and the time at which it was supposed to be triggered as indicated in the trace file.

We do as much as possible during the post-processing step so that replay error is minimized. For example, file pointers recorded in the trace files are replaced by simple indices that will be easily mapped to new file pointers during replay. Another important example involves correctly and efficiently identifying nodes involved in a particular communication or synchronization operation. The problem arises because MPI API's can create new communicators with the help of API's such as MPI_Comm_create, MPI_Cart_create, etc. These new communicators can be created from existing communicators or from the original communicator (MPI_COMM_WORLD). However, the comm values of the new communicators may vary from one node to another even though they may denote the same communicator. It is important to determine during post processing which messages from one node are related to messages from another node.

Other tasks that are taken care of during post-processing include creating the files needed at the time of replay, ensuring that any events logged by receivers on behalf of senders are moved to the correct (sending) replay file, and computing aggregate statistics such as total number of calls logged, time spent in logging those calls, etc.

2.3 Re-player

We provide a thread based re-player with the capability to re-play events at a rate as expected. Due to the work done post-processing, we are able to keep the overhead down to a minimum.

There are a few differences between a replayed execution and the original application's execution.

1. We use the same buffer for reading and writing data from a file. If the original application used multiple buffers, our replay may have some advantages in terms of cache behavior.

2. Our threaded re-player replaces MPI calls by equivalent operations. For example:

    - MPI_Send operations are mapped to sem_post.
      Rational: MPI_Send operations do not wait for the MPI_Recv to begin.

    - MPI_Recv operations mapped to sem_wait
      Rational: MPI_Recv waits for MPI_Send to send the message.

    - MPI Collective Operations are mapped to pthread_barrier
      Rational: For MPI collective operations all nodes are waiting for each other.

3. EXPERIMENTAL RESULTS

3.1 Setup

The applications are traced and replayed on two file-systems: a) Network File System b) PANSYS file system. The storage systems are mounted on the cluster shadowfax.vbi.vt.edu [21]. Each node has 8 cores with 24G of RAM and Linux installed. Three parallel applications are used for experimental evaluation.

Pseudo [3] is a MPI - application from Los Alamos National Labs. It is designed to test file systems for MPI-IO calls. Pseudo can be configured to have various kinds of interleaving among synchronization calls, I/O calls and communication among processes. For our use we perform barrier operations after I/O operations. Preads and pwrites are used for reading and writing to and from the file. The computation time of the application was increased to reduce the variance in vanilla non-traced runs.

Lammps [22] is a MPI based application which is used to study the interactions of atoms, molecules and macroscopic particles among liquids, solids and gas. Lammps involves I/O operations by a single node (root node). At repeated intervals the root node receives data from other nodes and dumps it to stdout.

GenIDLEST [23] is a MPI based computational fluid dynamics package that solves for the velocity and temperature fields in turbulent flows.

Lammps and Pseudo are run for all configurations where the total number of processes does not exceed 8. GenIDLEST is configured to run for fewer configurations as it requires a different test case for each configuration. At the time of running the tests, test cases for the entire set of configurations were not available to us.

3.2 Evaluation

Results for our three scientific applications are summarized in Figures 3.1-3 and Tables 3.1-3 below. The “error rate” data reported in the tables corresponds to the performance of each trace run or replay run relative to the performance of the original code on the indicated file system. A positive value means that the original run actually ran slower than the trace or reply run.

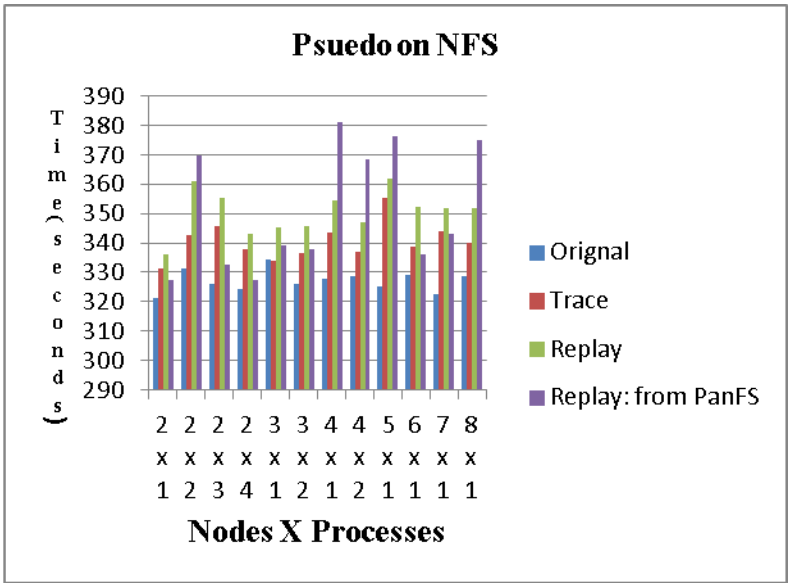

Figure 3.1: Pseudo runs on NFS.

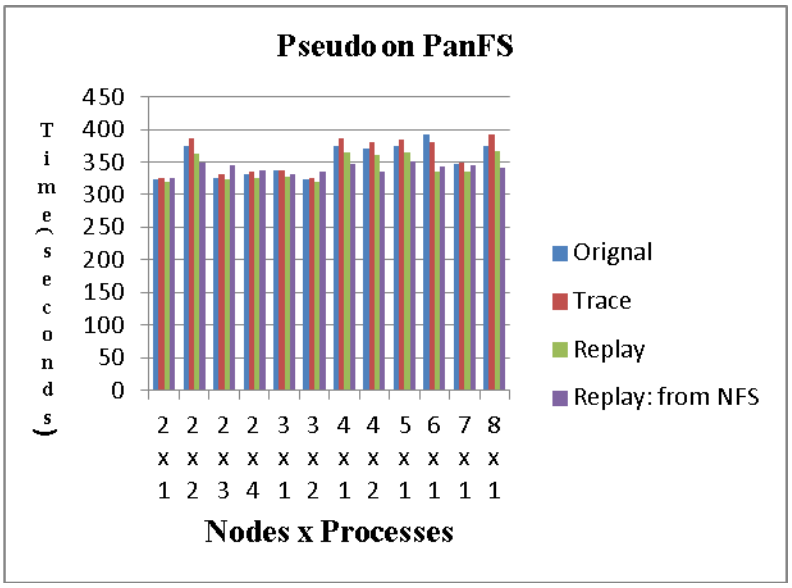

Figure 3.2: Pseudo runs on PanFS.

Table 3.1: Error rates for Pseudo's run on NFS.

	Trace Error Rate	Replay Error Rate	Replay error rate: from NFS
Max	2.86	14.32	12.48
Min	-4.43	0.48	-6.35
Average	-1.48	3.22	3.55
Standard Deveation	1.97	3.61	5.88

Table 3.2: Error rates for Pseudo's run on PanFS.

	Trace Error Rate	Replay Error rate	Replay error rate: from PanFS
Max	0.09	-3.3	-0.9
Min	-9.31	-11.26	-16.22
Average	-4.14	-7.18	-7.37
Standard Deviation	2.36	2.24	6.13

Table 3.3: Pseudo timings.

	Orignal	Trace	Replay	Replay: from NFS		Orignal	Trace	Replay	Replay: from PanFS
2 x 1	323.7	324.9	318.9	325.5	2 x 1	321.2	331.4	336.2	327.2
2 x 2	373.9	386.6	362.6	348.6	2 x 2	331.2	342.6	361.1	369.9
2 x 3	324.4	330.6	322.8	345.0	2 x 3	325.9	345.6	355.3	332.7
2 x 4	331.0	335.1	326.2	336.2	2 x 4	324.2	337.9	343.2	327.2
3 x 1	337.9	337.1	326.9	330.8	3 x 1	334.4	334.1	345.4	339.3
3 x 2	323.5	324.8	319.5	334.8	3 x 2	326.2	336.5	345.5	337.8
4 x 1	374.1	386.7	364.0	346.7	4 x 1	328.0	343.7	354.6	381.1
4 x 2	370.6	379.5	361.7	335.2	4 x 2	328.5	336.8	347.1	368.3
5 x 1	374.7	384.2	364.1	350.4	5 x 1	325.1	355.4	361.8	376.5
6 x 1	392.2	380.9	336.0	343.2	6 x 1	328.9	338.9	352.3	336.0
7 x 1	346.8	349.4	335.4	344.2	7 x 1	322.5	343.9	351.9	343.2
8 x 1	374.8	391.4	366.7	340.7	8 x 1	328.5	340.0	351.7	375.0

As expected, the overhead of collecting the trace data is very small, with an average performance penalty of less than 5%. The average replay error rate for Psuedo is within 10%. Table 3.3 lists down the overall values for time taken in seconds for each of the runs.

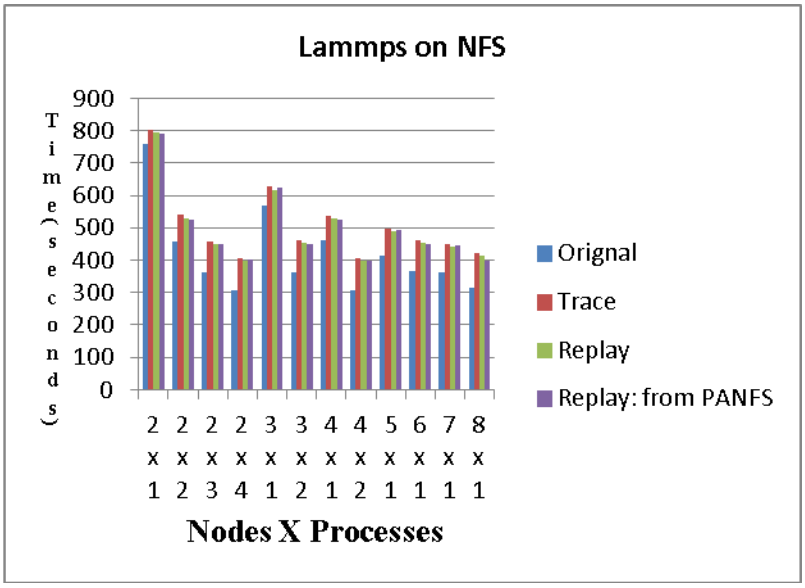

Figure 3.3: Lammps runs on NFS.

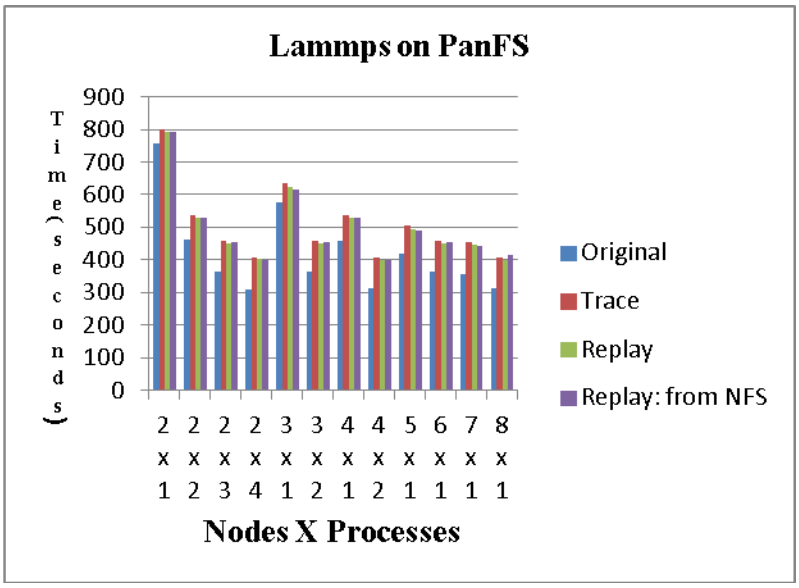

Figure 3.4: Lammps runs on PanFS..

Table 3.4: Error rates for Lammps's run on NFS.

	Trace Error Rate	Replay Error Rate	Replay error rate: from NFS
Max	-5.76	-4.33	-4.71
Min	-31.4	-29.47	-33.04
Average	-22.4	-20.38	-20.59
Standard Deviation	8.56	8.34	8.8

Table 3.5: Error rates for Lammps's run on PanFS.

	Trace Error Rate	Replay Error rate	Replay error rate: from PanFS
Max	-6.17	-4.74	-4.37
Min	-32.63	-30.49	-29.82
Average	-22.44	-20.37	-20.17
Standard Deviation	8.45	8.26	7.94

Lammps has high overhead (as much as 32.63%) at the time of trace collection. The overhead can be broken down into three components. MPI collective primitives exchange piggy back messages through MPI_Allgather. Handling of MPI collective primitives introduces a 10% overhead in trace collection. The process with Rank 0 dumps data through fprintf calls. Most of these writes are in the range of a few bytes. The logging of fprintf calls introduces another 10% overhead in trace collection. Piggy back messages are exchanged for MPI send and recv calls. Piggy back messages for send recv calls contribute the remaining 10% of overhead. High overhead at the time of trace collection leads to high overhead on relay as well. Table 3.6 lists down the overall values for time taken in seconds for each of the runs.

Table 3.6: Lammps timings.

	Original	Trace	Replay	Replay: from NFS		Orignal	Trace	Replay	Replay: from PanFS
2 x 1	757.8	801.5	790.7	793.5	2 x 1	757.1	803.7	793.0	790.2
2 x 2	460.8	535.5	527.5	527.8	2 x 2	459.6	539.4	527.3	526.9
2 x 3	363.0	459.2	451.5	452.3	2 x 3	363.2	459.4	451.5	451.1
2 x 4	310.5	408.0	402.0	398.6	2 x 4	308.7	404.7	398.0	400.8
3 x 1	575.6	633.3	623.3	617.1	3 x 1	567.1	626.6	616.7	622.9
3 x 2	362.5	458.6	450.5	453.6	3 x 2	363.1	460.8	453.0	449.5
4 x 1	459.8	536.3	527.0	530.2	4 x 1	462.1	538.1	529.6	526.6
4 x 2	310.9	407.4	400.4	398.2	4 x 2	308.1	404.5	397.2	399.6
5 x 1	419.4	504.1	495.1	489.6	5 x 1	415.0	497.5	488.9	494.5
6 x 1	362.2	459.4	451.2	453.2	6 x 1	365.2	460.5	452.5	450.4
7 x 1	356.9	452.8	445.3	443.2	7 x 1	361.3	449.6	442.4	444.3
8 x 1	311.2	407.0	400.0	414.1	8 x 1	316.8	420.2	413.4	399.8

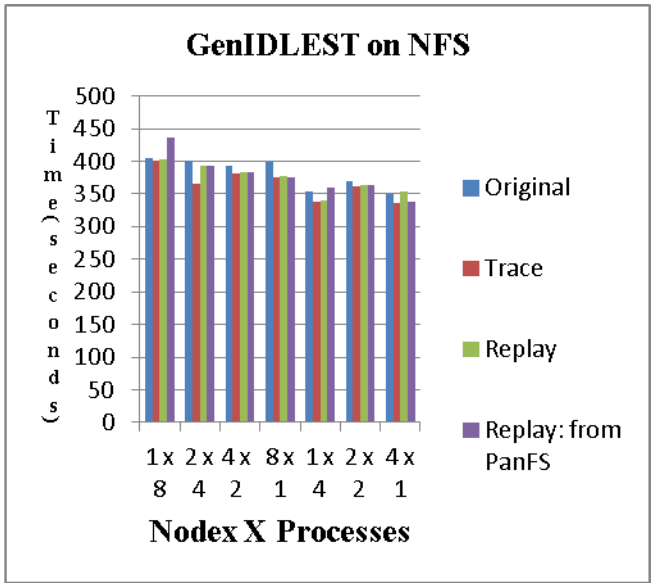

Figure 3.5: GenIDLEST runs on NFS.

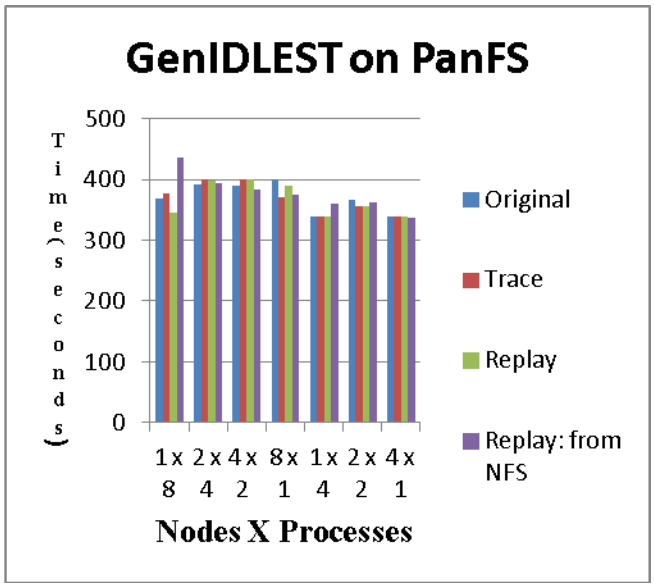

Figure 3.6: GenIDLEST runs on PanFS.

Table 3.7: Error rates for GenIDLEST's run on NFS.

	Trace Error Rate	Replay Error Rate	Replay error rate: from NFS
Max	7.66	6.19	6.23
Min	-2.38	-2.48	-18.12
Average	0.54	1.09	-2.13
Standard Deviation	3.67	3.03	7.89

Table 3.8: Error rates for GenIDLEST's run on PanFS.

	Trace Error Rate	Replay Error Rate	Replay Error rate: from PanFS
Max	8.55	5.44	7.25
Min	0.89	-0.67	-4.45
Average	4.06	2.18	-0.16
Standard Deviation	2.58	2.07	3.68

Table 3.9: GenIDLEST timings.

	Original	Trace	Replay	Replay: from NFS		Original	Trace	Replay	Replay: from PanFS
1 x 8	368.8	377.3	346.0	435.6	1 x 8	404.7	401.1	402.1	435.6
2 x 4	391.6	399.9	398.7	393.6	2 x 4	400.4	366.1	393.3	393.6
4 x 2	390.0	399.3	399.7	383.2	4 x 2	392.7	381.6	382.7	383.2
8 x 1	400.6	369.9	389.6	375.6	8 x 1	398.0	375.0	376.4	375.6
1 x 4	339.9	339.1	339.4	359.4	1 x 4	354.0	337.9	339.4	359.4
2 x 2	366.6	355.9	356.1	362.8	2 x 2	368.6	361.7	363.3	362.8
4 x 1	339.1	340.0	339.1	337.6	4 x 1	350.7	336.7	353.1	337.6

GenIDLEST performs as per our expectations. The tracing engine incurs minimal overhead at the time of trace collection. This helps the re-player to be within permissible error rates. Average error rate for GenIDLEST is below 5%.

4. FUTURE WORK

Our primary objective during trace collection is to determine the relationship between I/O and communication calls and log only those communication calls that affect I/O operations. The decision to log or not log a call is taken at runtime. This induces additional overhead, which may start affecting the overall runtime of the application.

The applications we have examined use a fixed communication and I/O pattern. Dependency analysis can be performed on applications to figure out which synchronization calls directly influence I/O calls. These calls can be annotated so that they are logged without the need for any runtime checks. Source code availability is an issue with propriety libraries. Dependency analysis may be difficult on machines in a production environment.

Although we have focused on file system performance, our piggybacking message technique could be applied to other distributed computing performance and debugging issues as well. Distributed memory parallel programming on large clusters continues to use the MPI API's for communication among nodes. MPI provides a large selection of API's, which cater to different kinds of communication requirements (blocking, non-blocking, deterministic, and non-deterministic). There are error-checking mechanisms built in to verify program correctness and behavior. However, not all programming errors (e.g., incorrect participating nodes, tags, buffers) can be caught through these mechanisms [5]. Use of wild cards (MPI_ANY_SOURCE and MPI_ANY_TAG) leads to program execution paths that are difficult to reproduce. Debugging becomes difficult and some of the bugs are manifest only when the program is run on a large number of nodes. Many of these defects occur due to incorrectly specified participating nodes. Thus there is a need to identify the mapping of MPI operations on different nodes that correspond to each other, e.g., send, receive, collective operations. One way to create this mapping is to send additional piggyback information along with the original message, as we have done in this work. The piggyback information can be used to determine ordering among operations on different nodes. In this way, our framework could be used to address challenging issues such as MPI program verification and message tracking.

Message piggybacking has limitations. Introduction of new messages increases the latency for application messages. Message piggybacking cannot be used in all scenarios. We list a scenario (Figure 4.1) where it doesn't work [24].

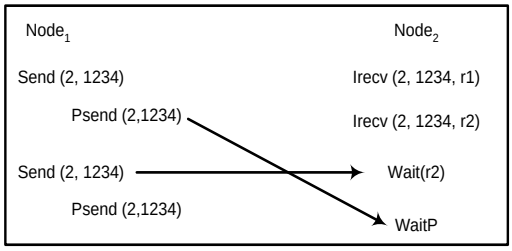

Figure 4.1 Example where message piggybacking doesn't work.

Another alternative to sending an additional message with the piggyback information is to pack this data along with the original message and send a single combined message, i.e., "datatype piggybacking." Vo et al. [24] use this approach to verify the correctness of MPI programs. In this technique piggyback information and the original message are packed in a new structure. The structure is sent in place of the original message. No additional messages are introduced during application runtime. We propose to build a new version of the MPIOR racing engine which will use "datatype piggybacking" technique in place of message piggybacking to send piggyback information.

Threaded MPI applications can be divided into two categories. The first category consists of applications that perform all MPI operations on a single thread. These applications would require keeping track of thread creation and deletion and synchronization operations. The second category of applications is much more complex in nature and involves MPI primitives being used across threads. The message piggybacking technique would be difficult to use with no guarantee of ordering of messages. Datatype piggybacking needs to be used for sending piggyback messages in case of threaded applications.

The MPIOR tracing engine currently dumps data in ascii text format. Dumping data in ascii text format is expensive and leads to an increase in runtime of the application and an increase in the size of the trace files. Dumping data in binary format would reduce the runtime significantly and reduce the size of trace data.

The re-player could be extended to provide more detailed performance information about individual I/O operations. In this way the replay system could be evaluated in a more fine-grained way (by comparing the replay performance on individual I/O calls) and it could be used to provide more detailed data on performance on other file systems.

5. LIMITATIONS

Our current implementation has limitations which hamper trace collection and thus affect replay accuracy. MPI_Bcast is handled as an MPI collective call. Auxiliary information messages are exchanged through MPI_Allgather. Thus, during trace runs, MPI_Bcast causes all the nodes to wait for the root node because the extra MPI_Allgather imposes an additional dependency of all nodes waiting on each other. Introducing these additional dependencies, which were not part of the original application, leads to an increase in overall runtime of the application.

Many applications instead of using system calls for I/O may use memory mapped I/O. The advantage of memory mapped I/O is that it has less overhead and operations can be performed with loads and stores. However, our current implementation of MPIOR does not trap memory mapped I/O.

6. CONCLUSIONS

In our work we propose a technique to determine performance of file-systems for parallel applications. Our technique is non-intrusive and can be used on a production system without any special privileges. The overhead of trace collection for two of the three applications studied is less than 5%, although a third application (Lammps) still shows 22% overhead due to inefficient logging, etra piggyback messages and additional collective calls. We plan to overcome these issues during the next iteration and bring down the overhead to 10%. The replay error rate for these applications shows similar characteristics, i.e., less than 10% for Psuedo and GenIDLEST and about 20% for Lammps.

BIBLIOGRAPHY

1. Aranya, A., C.P. Wright, and E. Zadok, *Tracefs: A File System to Trace Them All*, in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*2004, USENIX Association: San Francisco, CA. p. 129-145.
2. Mesnier, M.P., et al., *Trace: parallel trace replay with approximate causal events*, in *Proceedings of the 5th USENIX conference on File and Storage Technologies*2007, USENIX Association: San Jose, CA. p. 24-24.
3. Trace, L.; Available from: http://institute.lanl.gov/data/software/.
4. Singhal, V. and A.J. Smith, *Analysis of locking behavior in three real database systems.* The VLDB Journal, 1997. **6**(1): p. 40-52.
5. Vo, A., et al., *Formal verification of practical MPI programs.* SIGPLAN Not., 2009. **44**(4): p. 261-270.
6. Hertz, M., et al., *Error-free garbage collection traces: how to cheat and not get caught.* SIGMETRICS Perform. Eval. Rev., 2002. **30**(1): p. 140-151.
7. Anderson, E., et al., *Buttress: a toolkit for flexible and high fidelity I/O benchmarking*, in *Proceedings of the 3rd USENIX conference on File and storage technologies*2004, USENIX Association: San Francisco, CA. p. 4-4.
8. Andrerson, D., *Fstress: A flexible network file service benchmark*, 2002, Duke University.
9. Blaze, M., *NFS tracing by passive network monitoring.*, in *USENIX*1992: San Fransisco.
10. Katcher, J., *PostMark: A new filesystem benchmark*, in *Tech. Rep. TR3022*1997, Network Appliance.
11. Zhu, N., et al., *TBBT: scalable and accurate trace replay for file server evaluation.* SIGMETRICS Perform. Eval. Rev., 2005. **33**(1): p. 392-393.
12. *Transaction Processing Performance Council. www.tpc.org*. 2005.
13. *SPC. Storage performance council. www.storageperformance.org*. 2007.
14. *Iometer project*. 2004; Available from: www.iometer.org/.
15. Bray, T. *Bonnie home page. www.textuality.com/bonnie*. 1996.
16. Coker, R., *Bonnie++ home page. www.coker.com.au/bonnie++*. 2001.
17. Howard, J.H., et al., *Scale and performance in a distributed file system.* ACM Trans. Comput. Syst., 1988. **6**(1): p. 51-81.
18. Joukov, N., T. Wong, and E. Zadok, *Accurate and efficient replaying of file system traces*, in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*2005, USENIX Association: San Francisco, CA. p. 25-25.
19. Vo, A., et al., *A Scalable and Distributed Dynamic Formal Verifier for MPI Programs*, in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*2010, IEEE Computer Society. p. 1-10.
20. Vo, A., et al. *Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update*. in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. 2011.
21. Shadowfax. *http://www.vbi.vt.edu/high_performance_computing/*.
22. *Lammps*. Available from: http://lammps.sandia.gov/.

23. *GenIDLEST*. Available from: http://www.hpcfd.me.vt.edu/codes.shtml.
24. Vo, A., *Scalable Formal Dynamic Verification Of MPI Programs Through Distributed Casualty Tracking*, in *Computer Science*2011, University of Utah.